Patterns for
Process-Aware Information Systems:
An Approach Based on Colored Petri Nets
Mulyar, Nataliya Alexandrovna

- Eindhoven: Technische Universiteit Eindhoven, 2009. - Proefschrift. -


NUR 982

Keywords: Process-Aware Information Systems / Patterns / Business Process Management / Colored Petri Nets / Workflow patterns / Service Oriented Architecture / Flexibility

The work in this thesis has been carried out under the auspices of Beta Research School for Operations Management and Logistics.

This research was supported by the Dutch Organization for Scientific Research (NWO) under project number 612.066.407

Beta Dissertation Series D111
Patterns for
Process-Aware Information Systems:
An Approach Based on Colored Petri Nets

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 16 juni 2009 om 16.00 uur

door

Nataliya Alexandrovna Mulyar

geboren te Bratsk, Rusland
Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. W.M.P. van der Aalst

Copromotor:
dr. N.C. Russell
# Contents

1 Introduction 1

1.1 Process-Aware Information Systems 1

1.2 Historic developments and related trends 5

1.3 Research 8

1.3.1 Problem definition 8

1.3.2 Workflow Pattern Initiative 11

1.3.3 Research approach 13

1.4 Thesis outline 14

I Conceptual Foundations 17

2 Patterns 21

2.1 Types of patterns 21

2.2 Pattern format 23

2.3 Pattern identification 25

2.4 Related work 27

2.5 Summary 28

3 Colored Petri Nets Patterns 29

3.1 Main concepts of CPN 30

3.2 Catalog of CPN patterns 35

3.3 Classification of CPN patterns 102

3.3.1 Relationships between CPN patterns 102

3.3.2 Clustering of CPN patterns 105

3.4 Analysis of CPN patterns usability in practice 106

3.5 Related work 111

3.6 Summary 112

II Patterns for Process-Aware Information Systems 113

4 Workflow Control-Flow Patterns 117

4.1 Revisiting the control-flow patterns 117

4.1.1 Context assumptions 118

4.1.2 Classification of control-flow patterns 120

4.1.3 Catalog of control-flow patterns 122

4.1.4 Relationships between control-flow patterns 190

4.2 Patterns operationalization 191

4.2.1 Core Process Constructs Modeling Language 194
## 5 Service Interaction Patterns

5.1 Introduction ................................................. 227
5.2 Configurable framework for service interaction patterns .... 231
5.2.1 Pattern family: Multi-party Multi-message Request-Reply Conversation 233
5.2.2 Pattern family: Renewable Subscription ..................... 245
5.2.3 Pattern family: Message Correlation ........................ 256
5.2.4 Pattern family: Message Mediation .......................... 264
5.2.5 Pattern family: Bipartite Conversation Correlation ........... 281
5.2.6 Pattern-based service interaction design method ............. 289
5.3 Tool evaluations .............................................. 294
5.3.1 Evaluation of Oracle BPEL PM ............................ 294
5.4 Related work ................................................. 317
5.5 Summary ..................................................... 319

## 6 Process Flexibility Patterns

6.1 Taxonomy of process flexibility ................................ 323
6.1.1 Flexibility by design ........................................ 325
6.1.2 Flexibility by deviation ....................................... 326
6.1.3 Flexibility by underspecification ............................... 326
6.1.4 Flexibility by momentary change ............................... 327
6.1.5 Flexibility by permanent change ............................... 328
6.2 Catalog of process flexibility patterns ......................... 329
6.2.1 Context assumptions ......................................... 331
6.2.2 Flexible initiation ........................................... 334
6.2.3 Flexible termination .......................................... 348
6.2.4 Flexible selection ............................................ 358
6.2.5 Flexible reordering ........................................... 369
6.2.6 Flexible elimination ......................................... 377
6.2.7 Flexible extension ............................................ 383
6.2.8 Flexible concurrency ......................................... 392
6.2.9 Flexible repetition ............................................ 399
6.2.10 Discussion .................................................. 406
6.3 Tool evaluations .............................................. 408
6.4 Related work ................................................. 413
6.5 Summary ..................................................... 417

## 7 Epilogue

7.1 Contributions, limitations, and future work .................... 419
7.1.1 Colored Petri Nets .......................................... 419
7.1.2 Workflow control-flow ....................................... 420
7.1.3 Service interaction .......................................... 422
7.1.4 Process flexibility ........................................... 423
7.2 Reflection ..................................................... 424
7.3 Summary ..................................................... 428

Appendices 429
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A    Workflow Reference Model</td>
<td>429</td>
</tr>
<tr>
<td>B    Web-services stack</td>
<td>431</td>
</tr>
<tr>
<td>C    Glossary</td>
<td>433</td>
</tr>
<tr>
<td>Bibliography</td>
<td>437</td>
</tr>
<tr>
<td>Summary</td>
<td>453</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>455</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>457</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>459</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Process-Aware Information Systems (PAISs) have become increasingly popular as a technology for facilitating the specification and enactment of business processes. Contemporary organizations tend to focus on the precise definitions of relevant business processes and their automation using some form of process technology as there exists a common understanding that in order to improve operational effectiveness, business processes need to be managed in the same way as other more tangible corporate assets. The selection of a PAIS from a broad range of contemporary offerings is experienced by organizations as a complex and non-trivial task. Due to a lack of understanding of process technologies, the large diversity between functionalities offered by distinct offerings, and lack of common standards which could be used for comparing PAISs, the process of selecting a PAIS is quite chaotic and not well-defined. This thesis offers a solution to this problem by providing a rigorous foundation for PAISs in the form of a knowledge base that enhances the conceptual understanding of the various perspectives of PAISs. In doing so, it provides a reference point for the evaluation and improvement of contemporary offerings, and delivers insights into the definition of new languages and standards in the domain.

In this chapter, we introduce the research presented in this thesis. We start by introducing various kinds of PAISs in Section 1.1 in order to illustrate the plethora of different approaches to business process automation. In Section 1.2, we present an historical view on developments in the domain and relevant trends which determine the focus of investigations presented in this thesis. In Section 1.3, we define the problem addressed in this research and describe the approach selected for solving the research problem. Finally, in Section 1.4 we give an outline of the contents of this thesis.

1.1 Process-Aware Information Systems

In this section, we will focus on PAISs as these are the main topic of research in this thesis. In a generic sense, we will consider the class of information systems whose main goal is to support business processes within an organization. First, we need to understand what a business process and an information system are, and how these can be combined in order to help organizations function more effectively.

A business process is a special type of process, that can be defined as a set of tasks that need to be executed in a specific order by dedicated employees or other kinds of resources,
processing supplied input data and producing output data, with the aim of realizing one or more business goals. Typically, a business process describes an internal behavior within an organization, however it may interact with other organizations, for example, by using the resources of other organizations for accomplishing particular tasks or by providing information/products/services to them based on their requests. Business processes limited to one organization are called *intra-organizational*, whilst processes interacting with business processes in other organizations are called *inter-organizational* (the latter are also known as business processes forming process choreographies [223]).

In order to automate business processes, organizations require the support of some kind of information system. In [29], Alter defines an *information system* as “a particular type of a work system ... that processes information by performing various combinations of six types of operations: capturing, transmitting, storing, retrieving, manipulating, and displaying information”. In this definition, a *work system* is defined to be “a system in which human participants and/or machines perform a business process using information, technology, and other resources to produce products (and/or services) for internal or external customers.”

In this thesis, we consider information systems that link information technology to business processes, and which are termed as *Process-Aware Information Systems*. In [80], Dumas et al. define a PAIS as “a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models”. The term *process model* is used in this definition to represent a business process using some kind of (graphical) notation, which can be seen as “a blueprint for a set of process instances with a similar structure” [223]. In a process model, process entities and the relationships between them are explicitly defined. Various kinds of notations and languages can be used to describe business processes, e.g., Petri nets [102], EPCs [4], UML activity diagrams [44], BPMN (Business Process Modeling Notation) [165], or dedicated notations employed by specific information systems.

Unlike data-driven applications such as an e-mail client, which offers functionality for sending emails but which is unaware of the process to which email is sent, or a calculator program, which performs mathematical calculations for particular tasks in a process but which is unaware of the process to which these calculations relate, PAISs shift the focus from task to process-driven execution. The process-driven approach helps managers to keep a clear overview of the whole process in an organization, enabling them to monitor execution progress at the level of the process, rather than as a series of individual task executions, to track dependencies between execution statuses of different tasks, and to facilitate process-awareness within the organization by using process models as a means of communication and illustration.

PAISs allow processes to be enacted according to an underlying process definition. Automated enactment is a way to improve organizational efficiency by means of automatic task scheduling, information routing and time/resource optimization. The fact that PAISs are driven by underlying process models, allows the execution of a particular process instance to be adjusted as well as the original model to be optimized or even redesigned, without requiring the functionality of the application supporting process enactment to be modified [80].

The broad range of PAISs is illustrated in Figure 1. In this figure, PAISs are classified using two dimensions: the degree of adherence by a given process to an underlying process definition and nature of participants (i.e. humans or software applications) involved in interactions associated with PAISs [80]. The dimension characterizing the *nature of par-
Section 1.1 Process-Aware Information Systems

ticipants can be used to classify a PAIS as human or system-oriented. The abbreviations P2P, P2A and A2A denote Person-to-Person, Person-to-Application, and Application-to-Application processes respectively. Typical examples of P2P processes are tracking systems where a registered letter is being delivered by a postman to a person directly, project management systems where a manager evaluates the progress made by an employee using a project-tracking chart, and groupware tools which are used to enable real-time collaboration between several people such as collective writing, shared database access or electronic meetings. P2P processes consist of tasks which involve humans, i.e. they are not fully automated tasks.

The second group of PAISs are characterized by P2A processes, which involve both people and applications. The support for people in such processes is necessary in order to help a system to make a decision. A typical example of PAIS supporting a P2A process is a workflow system (i.e. a system which enacts a process based on a process model specifying who has to perform which activities and in what order). Work distribution mechanisms employed by a workflow system ensure that tasks requiring human intervention are assigned to the right people. Note that because the execution of tasks in a workflow system can be performed automatically, it may also serve as a platform for supporting A2A processes.

The third group, A2A processes, are characterized by coordinated inter-system interactions. Tasks in such processes are accomplished automatically, which is typical for transaction processing systems and enterprise application integration platforms. Although the majority of tasks in A2A processes are performed automatically, there may be some degree of human intervention required. This indicates that there is not a strict separation between P2P, P2A and A2A processes, and they may need to coexist in the context of a larger and more complex process. Note that in the end a business process always aims to serve customers (i.e. humans), however parts of the process may be fully automated.

We now move on to the classification of PAISs from the perspective of adherence to an underlying process definition. In Figure 1, the degree of adherence to an underlying process definition has four values: tightly-framed, loosely-framed, ad-hoc framed and unframed. A process is classified as unframed if there is no process model to which the execution of the process must conform. Systems where process models do not play any role (implicit or explicit) are not process-centric and thus are not considered. A tightly-framed system executes a process strictly based on a predefined process model. A typical example of this
is a workflow system such as Staffware, or service-oriented process management system such as Oracle BPEL PM. Loosely-framed systems execute processes based on a defined process model, however they allow for deviations from the prescribed flow of activities by ignoring or repeating specific tasks. Loosely-framed processes are often supported by case-handling systems such as FLOWer. Finally, ad-hoc framed processes are supported by systems which allow the execution prescribed by the process model to be changed for each specific case by adjusting or refining the process model. Such processes are often supported by ad-hoc workflow systems such as TIBCO InConcert or scientific workflows where depending on the results of tests certain steps may need to be performed less or more frequently.

The degree of adherence to an existing process definition is directly related to the degree of process flexibility supported by a particular offering. A process, whose behavior is hard to predict, may require more flexibility in adapting to changes in the operating environment than a process of the static nature. Processes of an unpredictable nature can often be observed in the medical domain. For instance, it is difficult to predefine processes of an emergency-care department as patients conditions may differ significantly and a common treatment strategy cannot be applied. Tightly-framed systems are not suitable for supporting such processes because they force the execution to strictly adhere to a process model, whereas in an emergency situation decisions often have to be made on-the-fly. In the medical domain, loosely-framed systems rely on a guidance approach, where based on encoded best practices, a guideline to prescribe one or another type of the treatment can be given to a medical assistant. A medical assistant, who uses the guidance system, has the flexibility to select one, several or even none of the treatments suggested.

Unlike loosely-framed systems, ad-hoc framed systems provide flexibility in modifying a process model during its execution. These systems are effective in supporting legislative or other rule-based processes where as a consequence of changes in laws or policies a transition from an old set of rules to a new set is required. Workflow systems and groupware are completely opposite to each other from the perspective of process flexibility. Workflow systems dictate which task needs to be executed and which resource must be assigned to execute the given task. In these systems, the decision-making related to the assignment of resources is performed centrally, resulting in very rigid and inflexible behavior. Whereas in groupware systems the decision-making is made locally, which allows resources to be assigned in a more flexible manner.

Based on the PAIS life cycle, which is visualized in Figure 2, we will characterize PAISs in terms of their design and implementation-orientation [80]. A typical PAIS life cycle consists of four phases: process design, process implementation, process enactment and diagnosis. During the process design phase, based on earlier requirements analysis, the business processes are identified, reviewed, validated and presented as process models [223]. This phase is entered initially at design-time in order to define a blueprint for future process instances. The design of process models is usually supported by business process modeling tools. These could be stand-alone process-designers or designers incorporated into a Workflow Management System (WFMS).

In the process implementation phase, a process is refined into an operational process that can be supported by a software system [80]. In this phase, the correctness of the process is verified and the process is deployed. Once the process has been deployed, it can be enacted. In the process enactment phase, a process is executed in the way prescribed by the process model. These two steps are usually supported by WFMSs and service-oriented process management systems. In WFMSs, processes are enacted by means of a
workflow engine, while in service-oriented process management systems, deployed processes are placed in a repository from where they can be initiated using a web-based interface.

Once the execution of a process has completed, execution-relevant information can be gathered and analyzed. In the diagnosis phase, any problems in the process are identified and decisions regarding possible solutions are made. For analysis purposes, the information logged by a WFMS or an audit trail produced by the web-process administrator can be used. Alternatively, dedicated project management tools can be used for this purpose. The results of analysis can be used to redesign a business process, and this step is characterized by entering the process design phase again.

In this section, we have shown the wide range of PAISs, which differ in terms of their functionality, goals, degree of rigidity/flexibility, and human/system involvement. In the next section, we take an historical view on the development of PAISs in order to show relevant trends and how these have impacted the evolution of PAISs.

1.2 Historic developments and related trends

An increasing understanding of the importance of managing business processes within and across organizational boundaries, can be characterized by changes in the enabling technologies used for information system development [80]. In this section, we will discuss the trends relevant to the development of PAISs [80].

Figure 3 shows that contemporary information systems have evolved from small operating systems with very limited functionality and tailor-made applications built on top of the operating systems, to multi-layered structures. The center of this structure is formed by the system infrastructure which makes the underlying hardware platform operational (e.g., operating systems). The second layer is formed by generic applications: these are applications that are used throughout the whole organization for general purpose applications (e.g., processing textual documentation, data management, calculations, etc.). The third layer is formed by domain specific applications: these are applications handling problems of a particular nature and usually their usage is limited to a particular department in an organization (e.g., human resource management, accounting, etc.). The fourth layer is formed by tailor-made applications developed for specific purposes.

The four outbound arrows in this figure illustrate the trend of each of these layers to increase in size by absorbing functionality from higher layers. This is a consequence of the expanding functionality provided by applications residing at each of these levels. Nowadays not only operating systems offer more functionality, but contemporary domain-specific applications also include functionality previously only found in tailor-made applications. This means that an information system with specific functionality does not need to be created
from scratch, but can be obtained simply by *configuring* currently available software systems. The ongoing trend of making various pieces of functionality available as stand-alone applications that can easily be reused, allows an application with required functionality to be obtained by *assembling* already existing components into a single system and orchestrating their behavior as required.

Figure 4 illustrates historical developments of PAISs. The development of first information systems, which appeared in the late 1960s, was driven by the need to store and retrieve data [80]. In the 1970s, office automation systems appeared in order to support everyday data processing tasks such as data calculations and document creation. These systems were mainly intended to simplify tasks related to the processing of information, e.g., text and image processing systems, spreadsheet programs for performing calculations, presentation packages, and personal database systems such as a calendar and a note-pad. Although office automation systems simplified execution of information processing tasks, they often neglected the modeling of business processes [80, 239].

With appearance of the Internet in the late 1960s and 1970s, the development of information systems focused on improving communication between people by interacting via E-mail, tele and video-conferencing and sharing information in many different forms (e.g., instant messaging or chat rooms). The scope of communication has evolved and extended since then in many ways. Groupware systems that appeared in the late 1980s help teams work together by sharing information [30]. Groupware does not dictate or guide the group work, it is aimed only at supporting the functioning of a team by facilitating messaging, E-mail, document sharing and access [90].

As the use of automation technology proved to be helpful in task planning, organization of data storage and access, processing of email and documents, organizations slowly started shifting the focus from data to processes. This shift impacted the functionality of evolving and new information systems, and triggered the development of a new discipline known as Business Process Management (BPM). BPM attempts to continuously improve processes by defining, measuring and improving various process performance indicators [95]. In order to support process definition and enactment, by mid 1980s commercial and academic workflow systems were designed. Workflow allowed tasks, process participants, and the assignment of resources to be described in a very precise way. Moreover, it offered functionality for monitoring relevant process indicators during and after execution.

The process of gathering information for monitoring purposes has evolved from recording events occurring during process execution using paper log sheets [30], after which the information gathered is sent for analysis of performance-related ratings, to the immediate
monitoring during process execution provided by workflow systems. Workflow systems offered the possibility to automatically gather data at each execution step and make the data gathered immediately available at other sources. One of the techniques related to analysis of data gathered during the process execution is process mining [174]. It allows a process model to be discovered based on the actual order of events recorded in a data log. The process model obtained can be used to check the conformance of an original process model with the actual process execution. Furthermore, on the basis of the data logged, various metrics could be analyzed and used as a source for business process redesign.

Originally, WFMSs were designed for ‘heavy imaging production applications’ [90]. Although initially the use of these systems was very restricted, by 1997 there were over 200 research and commercial systems developed [239] with a typical lifespan of 5 to 10 years. Some offerings were developed as pure workflow systems, whilst others evolved from image management systems, document management systems, relational or object database systems [62]. Each of these systems was developed independently, resulting in very diverse sets of functionality and features. In order to support the execution of specialized tasks, workflow systems offered a link to document management applications (originated from office automation systems) and other IT applications via external interfaces.

Due to the large disparity between the functionality offered by distinct workflow systems and the manner in which process modeling entities and constructs were interpreted, in 1995 the Workflow Management Coalition (WfMC) attempted to standardize the terminology in the domain and defined the Workflow Reference Model (cf. Appendix A). This model provides a functional description of necessary software components in a WFMS and the interfaces between them. The definition of interfaces between various software components aims at standardizing information exchange, thus enabling interoperability between different products [126]. The reference model introduced interfaces for interacting with external systems, however it did not provide a notation for describing the interaction between distributed processes. These gaps have been filled in by other standards (e.g., Business Process Execution Language (BPEL) [164] and Web-Services Choreography Description Language (WS-CDL) [217]). Although often criticized as ineffectual, the standardization efforts of the WfMC have had an impact on the development of workflow systems by increasing the awareness of the basic requirements workflow offerings have to satisfy and facilitating business process improvement [126].
Chapter 1 Introduction

In the early 1990s, e-commerce emerged as a way to provide and obtain services (e.g., selling or fulfilling orders) through electronic links of the World Wide Web. Any business application that has been defined using standard Web interfaces and deployed in order to communicate with other applications over a network represents a web-service [127]. Web-services are heavily based on information systems, as they require the extensive use of computers, data, and communication technologies to make a particular process available as a service and to acquire other kinds of services offered by external providers. With the appearance of web-services it has become possible to execute a business in a distributed manner. As it is possible for services to be sold or obtained from elsewhere, it has become unnecessary to physically locate a business partner, i.e. the required service could automatically be accessed via a network.

In the last five years, service-orientation started playing an important role in managing business processes. In addition to offering a basic support for process modeling and enactment, many workflow vendors nowadays promote service-orientation as a means of supporting distributed processes. Service Oriented Architecture (SOA) describes an information technology architecture that enables distributed computing environments with many different types of computing platforms and applications. It separates the functionality associated with particular processes into distinct units, and allows these processes to be accessed via network [97]. This enables reusability of processes, provides the ability to combine web-services in order to form a more complex process and support their orchestration.

Aiming at support of business processes operating in the dynamic and quickly changing environment, in the last decade, numerous academic and commercial vendors focused on the adaptability of business processes. As the majority of workflow offerings support processes in a very rigid manner, the ability to deal with unpredicted behavior remains a challenge.

In this historical view of the development of PAISs, we showed the large diversity of PAISs that have been evolving under the influence of various trends. Although PAISs provide a means of supporting business process automation, they are distinct in terms of functionality they offer for business process modeling and enactment. Because PAISs are inherently complex in nature, and different viewpoints on how they work can be taken, the interpretation of the functionality offered by distinct PAISs is not uniform. This merits further research into fundamentals of PAISs. We elaborate on the research topic in detail in the next section.

1.3 Research

We start describing the research presented in this thesis by outlining the problem definition (cf. Sub-section 1.3.1). Sub-section 1.3.2 describes the context in which the given problem is addressed. Finally, Sub-section 1.3.3 describes the research approach selected to tackling this research problem.

1.3.1 Problem definition

One of the fundamental objectives of any organization is to increase the effectiveness of their operations. In order to do so, companies need to manage their resources, control information flow, coordinate the execution of tasks and orchestrate relevant processes. Production time, use of resources and generated waste have to be minimized, while the flexibility to select suitable partners and adapt to changes in the operating environment
have to be maximized. Products supplied have to be of a high standard, moreover they need to be delivered to customers within an agreed timeframe. In many cases, these problems can be addressed by identifying relevant business processes and streamlining their operation by using appropriate Information Technology (IT). Given the wide range of commercial and non-commercial offerings that support business process enactment, the selection of a suitable information system is difficult and the process for doing so is not well-understood. Although some attempts to standardize the development of PAISs have been made (see the earlier discussion in Section 1.2), contemporary PAISs differ in terms of the functionality they provide for modeling business processes, interacting with external services and applications, and their ability to react to (unforeseen) events.

In order to reduce the complexity of PAISs analyzed, a separation of views is necessary for the analysis of relevant requirements. The separation of views has earlier been made in ARIS (Architecture of Integrated Information Systems) [199], CIMOSA (Computer Integrated Manufacturing Open Systems Architecture) [213], Zachmann framework [234], and MOBILE framework [128], which aim at providing an integrated infrastructure for execution of process models. In these frameworks, multiple perspectives have been distinguished. Although named differently, each of the frameworks identifies three common perspectives which are inherent in any business process: control-flow (function/operation/behavior), data (information), and resource (organization/network) perspectives. The control-flow perspective describes the structure of a process in terms of process modeling entities, their implementation, and interconnections between them in terms of the flow of control. The data perspective describes the kinds of data elements used in a process and the manner in which they are utilized. The resource perspective describes the manner in which tasks are assigned to resources, and the overall organizational structure.

In addition to the three fundamental perspectives, i.e. control-flow, data, and resource perspectives, in this thesis we also consider service interaction and process flexibility perspectives. The importance of the service interaction perspective can be illustrated in light of the current trend in the development of PAISs to offer a means of supporting distributed processes via service interaction (cf. Section 1.2). The availability of business processes in the form of web-services, which can be accessed via uniform interfaces, allows the functionality of these services to be easily reused by many other applications. Moreover, the ability of web-services to dynamically determine credentials of business partners allows service providers to be easily interchanged with another party at any suitable moment, if required. Whereas the control-flow, data and resource perspectives concentrate on the internal aspects of a business process, the service interaction perspective concentrates on the external behavior associated with business processes. Thus by focusing on the service interaction perspective, we aim to gain a better understanding of the requirements for PAISs in supporting inter-process communication.

In this thesis, we also focus on the process flexibility perspective as it addresses one of the biggest challenges that contemporary PAISs have to deal with, i.e. the ability to modify existing processes in order to adapt to changes in the operating environment. In order to do so, it is important not only to understand why contemporary offerings are so rigid, but also what is required in order to make them more flexible. The analysis of the process flexibility perspective has been insufficiently addressed to date [184], and we aim to develop a better understanding of the requirements relevant to PAISs for supporting processes of a highly volatile nature.

According to [239], Jablonski and Bussler identified several other perspectives (e.g., causality, integrity and failure recovery, quality, history, security and autonomy) which
could also be used for analysis. The causality perspective contains elements specifying under which conditions a process can be executed. The integrity and failure recovery perspective contains elements specifying the correct execution of a process instance and how exceptional situations need to be handled. The quality perspective is related to the establishment of a control mechanism to determine whether a process instance has been executed in an efficient manner or not. The history perspective contains elements used for monitoring the execution of a process instance on the basis of a history of executed events (recorded in the form of an audit trail or a process log). The security perspective is associated with application-based control aspects through the specification of privileges and authorizations associated with users and roles. The autonomy perspective specifies elements of remote access of work items by users and issues of continuous synchronization of a user with a worklist. These perspectives are not considered in this thesis for several reasons.

First of all, these perspectives are defined by Jablonski and Bussler as being an ‘optional enhancement’ of the mandatory perspectives addressing fundamental issues related to the control-flow, data and resources. Secondly, the majority of issues addressed by these optional perspectives are encompassed in the five perspectives considered in this thesis. As such, the context conditions associated with process execution, related to the causality perspective, can be specified by control-flow relations and data conditions associated with the control-flow and data perspectives. Issues relating to correct execution and the ability to handle unforeseen events, inherent to the integrity and failure recovery perspective, are encompassed into the process flexibility perspective which specifies various approaches to handling an unpredicted behavior. The quality perspective, which relates to determining whether a process instance is executed in an efficient manner, requires a particular corrective action to be taken when given criteria are not met. The ability to deal with unexpected events by adjusting the execution of a business process is encompassed in the process flexibility perspective. We do not consider the history perspective as it is only relevant after the process execution has completed. Any modifications that may need to be made after information gathered during process execution has been analyzed, can be seen as potential adjustments which can be encompassed by the process flexibility perspective. Issues relating to regulating task access, relevant to both the security and the autonomy perspectives, can be specified via users and roles, which are encompassed in the resource perspective. Thus, the scope of this thesis (cf. Figure 5) comprises five perspectives: the control-flow, data, resource, service interaction and process flexibility, an understanding of which is essential in order to utilize PAISs to support business process operation.

The upper part of Figure 5 represents the perspectives relevant to PAISs, whereas the bottom part of the figure represents a conceptual foundation for formalizing requirements for each of these perspectives. As Figure 5 shows, each process encompasses elements related to the control-flow, data and resource perspectives. These perspectives characterize the internal behavior of a process. The control-flow perspective specifies the structure of a process in terms of process entities, and the relationships between them describing the flow of control. The control-flow perspective essentially specifies the order of tasks in a process and can be seen as a backbone on which other perspectives reside. The data perspective augments the control-flow perspective with information that is necessary for execution of tasks. Furthermore, it provides input for data-based control-flow routing. The resource perspective describes the organizational aspects of a process, i.e. the assignment of resources to specific tasks in a process as well as their roles and responsibilities.

Although Figure 5 illustrates only two processes which interact with each other, the
The main goal of this thesis is to provide a rigorous foundation for PAISs that facilitates the conceptual understanding of the various perspectives of PAISs, provides a reference point for the evaluation and improvement of contemporary offerings, and bring new insights to the definition of languages and standards in the domain. This work should be considered in the context of Workflow Pattern Initiative, which is described in the next section.

1.3.2 Workflow Pattern Initiative

The Workflow Patterns Initiative, which commenced in 1999, aims at establishing a conceptual foundation for process technology that can be used for assessing the strengths and weaknesses of various approaches to process specification [230]. It has taken an empirical approach to identifying requirements for PAISs and documenting them in form of patterns.
The concept of pattern was introduced by Christopher Alexander who identified a series of reusable structures in an architectural context [25]. According to Alexander, a pattern is a relationship between a problem and a solution applicable in a specific context. Patterns have proven to be very successful for sharing proven and sound solutions for frequently recurring problems in various domains. Therefore the patterns approach has also been chosen by the Workflow Patterns Initiative to describe requirements for PAISs from different perspectives.

The investigation started from the control-flow perspective, which forms the basis of a process. As shown in Figure 6, in 2000 the first proposal summarizing requirements for PAISs from the control-flow perspective was published. Van der Aalst et al. empirically analyzed a selection of workflow systems available at that time and identified 20 common control-flow structures termed “workflow control-flow patterns” [9, 12]. Soon thereafter, Russell et al. investigated the data and resource perspectives. In 2005, they published 40 data patterns and 43 resource patterns, describing the various ways in which data and resources are represented and utilized in workflows respectively [191, 192]. This work was followed by the definition of a general graphical exception language with 16 primitives [190].

The patterns identified have been used to evaluate the capabilities of various workflow management systems and web-services standards. Several workflow offerings have been evaluated using the patterns identified, however at the time vendors did not show much interest in these results. In 2001, the patterns became more visible and accessible when the www.workflowpatterns.com web-site was released (this web-site is currently being visited by at least 350 people a day). Since then the patterns have been applied in product development and product evaluations. They have inspired the development of several new systems, e.g., Yet Another Workflow Language (YAWL)

\(^1\), Ivolutia Orchestration, Open-WFE, Zebra, and Alphaflow. The patterns also triggered improvements and redesign of existing systems, e.g., FLOWer 3.0, Bizagi, Staffware Process Suite, etc. Furthermore, the patterns have been extensively used in teaching and facilitating the sharing of best practice in the domain. The paper on workflow patterns [12], although less than ten years old, is already the third most cited workflow paper according to Google Scholar. The total number of references to this paper exceeds 900, which is a sign that patterns also have an impact in the academic world.

The ‘Patterns for Process-Aware Information Systems’ (P4PAIS) research project presented in this thesis started in 2004 as a continuation of the work done in the context of the Workflow Patterns Initiative with the goal of refining the control-flow patterns and

\(^1\)YAWL is a spin-off of the Workflow Patterns Initiative, whose original goal was to show how the workflow patterns can be supported in practice.
investigating requirements for perspectives which have not yet been addressed. Whilst the analysis of the data and resources perspectives was already underway, the need to revise the control-flow patterns was identified. Extensive use of the control-flow patterns for assessing workflow offerings revealed that some of the pattern definitions were ambiguous (e.g., they could be potentially interpreted and implemented in several distinct ways) and some patterns were missing. In addition to addressing these issues, the requirements for PAISs from other perspectives also merited further analysis.

The control-flow, data and resource perspectives can be used to describe business processes with a focus on the internal aspects of an organization. However this knowledge is not sufficient for supporting organizations planning to extend their boundaries by interacting with other organizations or by merging several businesses and thus requiring business process integration. In light of current trends in BPM, service-orientation is becoming increasingly important, therefore we analyze the requirements for PAISs from the perspective of service-interaction. Another very important dimension we will address in this thesis is process flexibility. Understanding what constitutes process flexibility and how it can be achieved is crucial for selecting a PAIS capable of coping with unpredicted events in a continually changing environment.

This thesis complements the earlier work on control-flow, data, resource, and exception patterns. In particular, the following patterns are provided:

- revised control-flow patterns;
- service interaction patterns;
- process flexibility patterns.

Together with the thesis of Russell [194], these patterns provide a comprehensive coverage of PAIS functionality. Note that Figure 5 shows the combined sets of patterns while highlighting the patterns presented in this thesis.

### 1.3.3 Research approach

The selection of the research approach to the problems identified above is driven by the question: “How should the requirements for PAISs be described from various perspectives (i.e. control-flow, service-interaction and process flexibility) in a systematic and precise way?”. There are several decisions that have to be taken in order to answer this question. First of all, a suitable presentation format needs to be selected that allows for describing requirements for PAISs in a structured and unified manner. Secondly, a formalism capable of describing the semantics of the requirements identified in a precise and an unambiguous way needs to be chosen:

1) In order to describe requirements for PAISs systematically, we have chosen the patterns approach previously used by the Workflow Pattern Initiative. This approach has proven to be very appealing to workflow vendors, process designers, system developers, and BPM researchers, as it facilitates the sharing of best practices within a domain in a language-independent manner. Traditionally, patterns are described in a form which motivates their usage, by providing typical examples of the pattern in practice, discussing how the pattern may be implemented, and (potential) issues that may arise as a result of their usage. We will elaborate on the topic of patterns in more detail in Chapter 2.

2) In order to describe requirements in a precise way, we specify their semantics using the Colored Petri Nets (CPNs) formalism. CPNs provide a graphical formally-defined language that has been extensively used for design, specification and simulation of dynamic systems with elements of concurrency [66]. Its application domain includes (but is not
limited to) automated production systems, workflow systems, distributed and embedded systems. CPNs are an extension of classical Petri nets with time, hierarchy and color. The extension with time enables the modeling of temporal aspects and the evaluation of system performance. The extension with hierarchy allows a model to be compactly structured by decomposing it into a series of smaller models (or pages) with well-defined interfaces. The extension with color allows data to be modeled explicitly, thus overcoming the drawback of the classical Petri nets where all data manipulations must be included directly in the net structure by means of places and transitions. Another important aspect is the tool support. CPN Tools [66] offers a good modeling and simulation environment, where CPN models can be designed, executed, and analyzed. The ability of CPN diagrams to capture dynamic behavior is a significant advantage over other formalisms as this aspect is important for describing the flow of control within a process, interaction between processes and the adaptability of processes to changes. Thus the choice of CPNs is driven by the power of the language (which allows both control-flow and data associated with it to be expressed) and very good tool support that allows models to be operationalized.

Figure 5 illustrates that for each of the three perspectives: control-flow, service-interaction and process flexibility, the requirements are described by means of patterns, and their semantics are specified using CPNs.

To design CPN models efficiently, we were hoping to gain access to the accumulated knowledge in the domain. As CPNs are heavily used in practice, similar problems can be experienced by different designers and same solutions can be used by them to solve the problems they have identified. To help developers to model efficiently without reinventing solutions, a source of knowledge describing sound and proven solutions is needed. To the best of our knowledge, a shared knowledge base satisfying the above described requirements does not exist. Due to the absence of such a knowledge base and also as a consequence of the prominent role CPNs play in our research, we decided to gather solutions commonly used for solving problems frequently occurring in CPN modeling and document them in the form of patterns.

1.4 Thesis outline

The roadmap for the research presented in this thesis is visualized in Figure 7. Part I presents conceptual foundations for the thesis:

- **Chapter 2** describes the fundamentals of the pattern-based approach to describing generic concepts in a given problem domain. The concepts of a pattern, a pattern format and a pattern language are introduced, and different types of patterns are described. Furthermore, the methods that we have used for pattern identification are summarized in this chapter.

- **Chapter 3** presents CPN patterns. The relationships between the patterns are analyzed and the patterns are divided into clusters in order to help users in selecting an appropriate pattern from the pattern catalog. Furthermore, the frequency of pattern use in practice is examined through an empirical analysis of a selection of CPN models.

The research presented in Part II of this thesis builds on the conceptual foundations presented in Part I, and addresses three perspectives of PAISs:
• **Chapter 4** presents workflow control-flow patterns. The definitions of the original patterns are revised, new patterns are added, and the semantics of all of the patterns are precisely defined in terms of CPN diagrams. In order to differentiate between different approaches to operationalizing these patterns, the formal Core Process Construct Modeling Language is presented. The use of the patterns is examined in a series of PAISs as a means of assessing the specific control-flow capabilities of individual offerings.

• **Chapter 5** presents service interaction patterns. The patterns identified address the issues of request-reply interactions involving multiple parties and multiple messages in the context of short and long-running conversations. In order to distinguish pattern variants an intuitive graphical notation is defined. Furthermore, the semantics of each of the service interaction patterns identified is described using CPN diagrams. In order to illustrate the extent of patterns support experienced in practice a selection of PAISs is analyzed.

• **Chapter 6** presents a taxonomy of process flexibility which classifies different approaches to facilitating process flexibility. The operationalization of the different process flexibility approaches identified is defined by means of process flexibility patterns. The function of these patterns is illustrated by means of a process engine expressed in terms of CPN diagrams.

Finally, this thesis concludes with **Chapter 7**, which describes limitations and proposes future work.
Part I

Conceptual Foundations
The scope of the research presented in this part of the thesis is presented in Figure 8. In particular, we concentrate on the bottom part of the figure, i.e., the conceptual foundation for PAISs. There are two main topics addressed in this part: patterns (e.g., the generic meaning of patterns, their different uses, relations, application domains, etc.) and CPNs (e.g., the formal foundation used to present the requirements for PAISs in part II of this thesis).

Figure 8: Scope of the research: Part I

Due to the prominent role CPNs play in our research, we decided to gather solutions commonly used for solving frequently occurring problems in CPN modeling and document them in the form of patterns. Before we proceed to CPN patterns, we first discuss the various characteristics and types of patterns, and their use as a means of capturing and sharing knowledge within a given domain. In Chapter 2, we concentrate on the notion of a pattern, a pattern language, a pattern format, and give a generic overview of other pattern-related work. Then in Chapter 3 we present the CPN patterns identified.
Chapter 2

Patterns

In this chapter, we describe the fundamentals of the pattern-based approach to describing generic concepts in a given problem domain. First, in Section 2.1 we introduce the concept of a pattern, a pattern language, and discuss different types of patterns. In Section 2.2, we elaborate on the pattern format used to describe different kinds of patterns. Then, in Section 2.3 we summarize the methods that we have used for pattern identification. Finally, in Section 2.4 we discuss related work.

2.1 Types of patterns

The concept of a pattern was introduced by the architect Christopher Alexander in his book “The Timeless Way of Building” [26] in 1977. Alexander defined a pattern as “a three-part rule, which expresses a relation between a certain context, a problem, and a solution”. Patterns characterize constructs, methods or techniques that have been encountered in practice repeatedly. Each pattern is intended to address an individual problem. In order for more complex problems to be solved, a number of patterns may need to be combined. By classifying different kinds of patterns and the types of relations between them, patterns can easily can be combined together. Moreover, with knowledge of the specific characteristics of individual patterns, one may choose the pattern most appropriate for a given situation. A pattern language is “a structured method of describing design practices within a field of expertise by explicitly describing the key characteristics of effective solutions for meeting some stated goal” [96]. A pattern language helps a user to move from problem to solution in a logical way, thus allowing for many alternative paths through the design process.

A pattern language is not fixed, it is built up on collected experience in a field, and as the techniques used in practice change, the pattern language may also evolve. According to Alexander, patterns which are often used in practice are “alive”, while the ones used rarely or not used any more are “dead”. In order for the pattern language to be alive, it has to consist of patterns that are actively used in practice. In order to know whether a pattern is alive, one has to observe different situations and confirm that this pattern is being repeatedly used.

Patterns are encountered by everyone in their everyday lives. Preparing meals according to their grandmother’s recipes, using proven materials for fixing electricity and water pipes, resolving common problems with electronic appliances using the manual - these are examples of the typical patterns encountered in domestic life. Patterns exist in various
fields. One can think of the health care domain where examinations and medicines are used to treat patients’ problems; the social sciences where patterns are related with selecting a leader, collaborating in order to develop solutions, approaches for resolving conflicts, etc.; the chemical industry where compositions containing various elements are used for cleaning, deodorizing or painting purposes.

Patterns represent a piece of knowledge about how to solve a particular task. While problems originate from the requirements which need to be satisfied, the actual pattern is about solutions that can be used to solve the specified problem in a given context. Figure 9 illustrates different phases of the pattern identification process.

In the analysis phase, the requirements are examined, based on which a set of problems are identified. Complex problems are decomposed into smaller parts. For each of the sub-problems in the pattern synthesis phase possible solutions are identified, which are grouped together in order to solve the original problem. In the pattern verification phase, the combined solutions are checked and tested.

Apart from the fact that patterns address different problems, depending on the nature of the domain in which the problem has been encountered, patterns differ in terms of the degree of abstraction that they demonstrate. Figure 10 illustrates the problem decomposition process, where based on the given set of requirements, a set of problems are identified. The solution for each of these problems represents a pattern. When looking for solutions to each of the problems identified, it may be necessary to decompose the problem into a series of problems at a lower level of abstraction and to solve them first. The solution to a more specific problem also represents a pattern.

![Figure 9: Different phases of pattern identification process](image)

![Figure 10: Pattern abstraction levels](image)
A typical example of a problem that can be decomposed into smaller parts, is the integration of two applications. First, an architecture needs to be defined. For this, the architecture patterns and enterprise integration patterns can be used. The components defined in the architecture need to be worked out in further detail, which requires knowledge and use of the design patterns. Finally, when a particular object needs to be implemented, typical implementation solutions are used. Problems solved at each of these levels are characterized by different degrees of problem granularity and belong to different levels of abstractions.

The pattern initiative of Alexander [26] was widely supported and triggered a set of parallel initiatives, i.e. pattern languages, in other application fields and domains. In subsequent years, the idea of patterns became popular in the object-oriented community. As an evidence of this, we refer to the 23 design patterns by Gamma et al. [99], and numerous successors such as the analysis patterns by Fowler [92], and the framework patterns by Pree [176], etc. (a more extensive overview of related patterns work follows in Section 2.4).

Patterns at different levels of abstraction describe generic or detailed solutions. The type of information specified in different solutions may be less or more specific, which needs to be reflected in the manner the patterns are described. In order to uniformly document patterns, one needs to select an appropriate pattern format. In Section 2.2, we describe different pattern formats commonly applied to systematically document patterns. Furthermore, we indicate which pattern formats are utilized in this thesis.

2.2 Pattern format

In order to communicate proven solutions for frequently recurring problems, patterns need to be documented using a systematic approach. The precise description of patterns is one of the prerequisites for these patterns to be organized in a pattern language. In [26], Alexander introduced a pattern format, which included (1) a picture graphically illustrating a problem, (2) explanation of the context in which the pattern is to be used and relation to other patterns, (3) a statement of the problem and discussion of different variants of the problem, (4) a solution to a problem described in the form of instructions, often supported by a graphical illustration, and (5) a list of patterns that are used in the solution, or patterns that need to be used in combination with the given pattern in order to solve a more complex problem.

From the moment the concept of a pattern was introduced, the pattern-based approach gained popularity and patterns have been documented in various domains. Because the nature of the problems addressed, the context conditions and methods used in different domains vary significantly, the original pattern format has been modified in order to meet the needs of specific domains, thus resulting in a variety of pattern formats. One of the most representative examples is the format introduced by Gamma et al. [99] for describing design patterns in object-oriented software development. “Each pattern is divided into sections according to the following template...

- **Pattern name and classification**: The pattern’s conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary...
- **Intent**: A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- **Also known as**: Other well-known names for the pattern, if any;
• **Motivation**: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

• **Applicability**: What are the situations in which the pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

• **Structure**: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT)...

• **Participants**: The classes and/or objects participating in the design and their responsibilities.

• **Collaborations**: How the participants collaborate to carry out their responsibilities.

• **Consequences**: How does the pattern support its objective? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

• **Implementation**: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are these language-specific issues?

• **Sample code**: Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

• **Known uses**: Examples of the pattern found in real systems.

• **Related patterns**: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?” [99].

By comparing the pattern format of Alexander and that of Gamma, one can identify numerous disparities. One can argue that the extended pattern format is needed in different domains due to the different nature of problems being addressed and higher/lower degrees of pattern granularity. While the format of Alexander is rather generic, it lacks precise semantics. The usage of this pattern format in other domains may lead to ambiguities in pattern interpretation, which is not desirable. This drawback has been addressed in Gamma’s format by using a standardized method for describing solutions and giving (programming) language-specific examples.

The approach selected for describing a pattern defines the range of readers to which the pattern will be accessible. Patterns described in a generic and language-independent way can be used in various domains, while language-specific problems need to be expressed in terms of the considered language and thus are mainly aimed at the audience acquainted with this language.

Despite numerous discussions regarding the optimal pattern format that could be used for describing patterns, no consensus has been achieved. Nevertheless, in order to be generally useful a pattern format must contain at least the pattern name, the problem description, the solution, and the consequences of applying the pattern as illustrated in Figure 11.

Such a pattern format may be sufficient for describing patterns at a more abstract level, while language-specific patterns addressing more detailed problems may require additional fields related to the pattern implementation to be added. Note that one of the ways to describe pattern variants is by describing a generic pattern problem and providing context-specific solutions for it. An example of a pattern having such a structure is a “Message Filter” defined by Hohpe en Woolf [125]:

**Pattern Name**: Message Filter.

**Problem**: How can a component avoid receiving uninteresting messages?

**Solution**: Use a special kind of Message Router, a Message Filter, to eliminate undesired
Section 2.3 Pattern identification

The process of pattern identification consists of several steps. A pattern cannot be invented, it can only be empirically identified. Identifying a new solution does not result in a new pattern being created. In order for the solution to become a pattern, it first needs to be tested (i.e., observed) in practice. Only sound solutions, which have been proven to solve particular problems effectively, can be considered. Sound solutions can be derived from expert knowledge, tutorials where common solutions are communicated to users, or empirical evaluation of products where particular features, characteristics or functionalities are frequently observed and used.

For instance, if we analyze information systems or programming tools, they offer a large set of functionality for supporting users in achieving particular results. Not only may particular features offered by the tools represent patterns, but also combinations of the steps performed by a user in order to achieve a particular goal can be considered as patterns. In order to identify patterns in a particular domain, one has to define the scope of problems to be analyzed and determine the degree of problem granularity in order to decide at which level of abstraction the patterns will operate. When the scope of the analysis domain has been clearly defined, solutions can be gathered using a bottom-up empirical approach or derived using a top-down systematic analysis method.

Using the empirical approach various products can be observed and solutions addressing a particular aspect of the problems analyzed can be identified. Figure 12 illustrates this process of pattern identification. For frequently used solutions, a corresponding problem is identified, and they both are recorded using the selected pattern format.

This approach does not guarantee the completeness of the patterns identified, because it is based on observation rather than on systematic derivation. Nevertheless, the collection
Chapter 2 Patterns

Figure 12: Pattern identification: bottom-up approach

of patterns obtained using this approach may be particularly useful for communicating solutions most commonly used in practice. The pattern collection can be extended when missing or new solutions have been identified.

An alternative to the empirical approach, is the top-down pattern derivative approach where, based on the problem domain to be analyzed, a set of dimensions central to all of the problems are identified.

Figure 13: Pattern identification: top-down approach

As illustrated in Figure 13, for each of the dimensions identified a possible set of values is defined. Then possible combinations of different values associated with distinct dimensions are analyzed, and meaningful combinations are filtered out for further analysis based on the application of domain knowledge. The set of useful combinations achieved is analyzed in terms of similarity and combinations addressing similar kinds of problems are grouped together. The groups of related patterns identified correspond to pattern groups, while the combinations within these groups represent individual patterns. To test the use of patterns derived using this method in practice, one has to analyze support for the dimensions earlier identified. Supported values for each of these dimensions define possible combinations and thus the supported patterns. Although some in the pattern community may not consider these to be patterns, it is important to underline that the main goal of this approach is to structure the knowledge in the field, set the scope and identify relevant dimensions, rather than simply to identify recurring solutions.

In this thesis, we apply both approaches to pattern identification. The CPN patterns, presented in Chapter 3, are based on the empirical approach. The goal of this chapter is to share commonly used solutions, rather than arriving at a complete collection of patterns. In
Chapter 4, patterns related to the control-flow perspective of PAISs are identified using the empirical approach, however for a selection of patterns the systematic derivative approach is applied. The patterns related to the service interaction perspective of PAISs also described in Chapter 5 using the systematic approach. Finally, in Chapter 6 patterns related to the process flexibility perspective are also systematically derived. Due to the different approaches used and the disparity in the degree of abstraction characterizing the patterns identified, we will adopt different pattern formats for each of the pattern groups.

2.4 Related work

It is impossible to give a complete overview of the different types of patterns described in the literature. As indicated before, the ideas of a “pattern” and a “pattern language” were introduced by the architect Christopher Alexander in [26] to assist people in designing own homes and communities. Since Alexander described his pattern language [25], patterns have become a popular means of capturing and sharing the co-existing knowledge in various domains. The pattern approach has heavily influenced the software development community.

The 23 design patterns by Gamma et al. [99] triggered the development of many more patterns initiatives in the object-oriented software community. Some of its successors include: the patterns for knowledge and software reuse by Sutcliffe [208], the design patterns in communication software by Rising [188], and the framework patterns by Pree [176] describing how to reuse the source code and architectural design to build software applications. Aside from generic patterns, sets of language-specific pattern languages (UML [88], Smalltalk [28], XML [229], Python [198], etc.) have also been discovered and documented (cf. www.hillside.net).

In [148], MacDonald et al. explain that design patterns are not used as generative constructs that support code re-use because design patterns describe solutions to a family of related problems and it is difficult to generate code for solving a particular problem. The authors present a parameterized approach helping users to adapt generic patterns to their specific code requirements.

Furthermore, some work has been done on formalizing organizational, process, analysis, and business-related patterns. Among them are the analysis patterns by Fowler [92] which show examples of domain models, and the framework process patterns by Carey [53] which concentrate on the resource perspective and identify means of effective communication during different phases of the framework development process. In addition, the patterns for e-business [18] which focus on business patterns, integration patterns, and application patterns; the business patterns at work [87] which use UML to model a business system; Coplien’s organizational patterns [65] which describe successful approaches for organizing and managing people involved with the software process, and Ambler’s process patterns [31] describing various stages and phases in the development process.

Other interesting patterns collections are the enterprise integration patterns by Hohpe and Woold [125] and the service interaction patterns by Barros et al. [37]. In their book [125], Hohpe and Woold propose a visual notation framework to describe large-scale integration solutions involving many technologies. It concentrates on the advantages and limitations of asynchronous messaging architectures with a particular focus on messaging systems used for message routing. The service interaction patterns presented in [37] document common problems and approaches to the design and implementation of web services. The special emphasis of these patterns is on situations where services are engaged
in concurrent and interrelated interactions (both of a bilateral and multilateral nature). Strategies for integrating applications using Microsoft technology, and patterns for data replication and synchronization are documented in [154].

Weber et al. [219] describe 18 change patterns and seven change support features that can be used for analysis of PAISs from the perspective of process change. The authors define the change patterns in terms of operations that can be applied during process execution in order to adapt a process model to new requirements. Furthermore, they address issues of version control and process instance migration, which are important means for keeping the consistency across existing process models and instances.

In addition to the large set of pattern collections, the issue of documenting patterns in a way that makes them understandable to their intended audience has been addressed by multiple authors. In [153], Meszaros and Doble present meta-patterns, i.e. best practices about pattern writing, which give insights into pattern-writing techniques. Furthermore, Jonson describes how software frameworks can be documented using patterns. In [132], Jonson considers a framework as a reusable design for a program expressed as a set of classes. To illustrate that patterns can be used not only as ways of communicating design information, in [41] Beck shows that they can also be used to derive an architecture from a problem statement (“an architecture is the way the parts work together to make the whole” [41]).

In contrast to patterns, which document solutions that have been repeatedly used in practice for solving particular problems, the concept of “anti-pattern” exists. Anti-patterns describe solutions that have been used in multiple projects which have been unsuccessful, thus providing knowledge of what does not work in practice [32]. Anti-patterns typically describe how to transform refactored solutions into a more efficient ones. Although anti-patterns are less widely studied, some of them are utilized by the software community. Moreover, to come up with a proper, sound solution, one requires both knowledge of sound solutions and typical counterexamples. This thesis mainly concentrates on patterns and does not consider anti-patterns.

The starting point for this work is the Workflow Patterns Initiative already mentioned in Section 1.3.2. To capture the functionality of PAISs in term of patterns, control-flow patterns [12], data patterns [191], resource patterns [192] and exception patterns [190] have been documented. This thesis contributes to the Workflow Pattern Initiative by redefining the control-flow patterns, defining patterns in service interaction and process flexibility, as well as CPN patterns in order to bring further understanding to problems where the control-flow and data perspectives interplay. This complements the work done by Russell [194] which focussed on the control-flow, data, resource, and exception patterns. This thesis and Russell’s thesis are intended to cover the broader domain of workflow patterns.

2.5 Summary

In this chapter, we introduced the concepts of a pattern and a pattern language, described differences in pattern formats used for documenting patterns, and explained two methods applied in this thesis for patterns identification. Patterns presented in the next chapter describe solutions frequently reoccurring when designing CPN models. Together with this chapter, the CPN patterns form a conceptual foundation for describing requirements for PAISs, which are presented in Part II of this thesis.
Chapter 3

Colored Petri Nets Patterns

In this chapter, we present a pattern language for CPNs. Although the main features of classical Petri nets are incorporated in CPN patterns, we concentrate on their extension with color. By means of color, tokens in the Petri nets can have data values associated with them, thus enabling data associated with control-flow in processes to be specified. First, in Section 3.1 we introduce the main concepts of CPNs that are necessary for understanding the patterns presented in this chapter. Then, we present our collection of 33 patterns in the form of a pattern catalog (Section 3.2). This catalog describes common solutions to modeling problems recurring during design of CPN diagrams. In order to support developers in selecting an appropriate pattern, we examine the clustering of CPN patterns into different groups in Section 3.3. Furthermore, we identify the different types of relationships between the patterns in order to facilitate easy navigation through the pattern catalog. To illustrate the use of patterns in practice, we present an empirical evaluation of a set of CPN models in Section 3.4. Finally, we discuss related work and draw conclusions in regard to the use of CPN patterns in Section 3.5 and Section 3.6 respectively.

Patterns are usually described in a standard pattern format, which includes the pattern intent, context conditions, problem description, possible solutions and relevant implementation strategies (see the discussion on this topic in Section 2.2). In order to illustrate the implementation of solutions, a tool supporting modeling in CPNs had to be selected. For this purpose, we chose CPN Tools [66] (a successor of Design/CPN [67]), a tool that offers interactive and automatic simulation facilities, by means of which models designed in CPNs can be executed. The ability to use a CPN model as a system’s description and the ability to analyze its behavior are the motivations for the extensive use and application of CPN Tools in practice. CPNs have been actively used by 700 organizations and individuals in 70 countries as illustrated by [66].

Since there are multiple views on how to document patterns and no consensus on a single pattern format has yet been achieved (as discussed in Section 2.2), we took the pattern format of Gamma [99] as a starting point, and adjusted it to fit our purposes:

- **Pattern name** This section identifies the pattern, and captures the main idea of what the pattern does.
- **Intent** This section describes in several sentences the main goal of a pattern, i.e. identifying which problem(s) it offers a solution.
- **Motivation** This section describes the actual context of the problem addressed by the pattern, and explains why the problem needs to be solved.
• **Problem description** This section presents the problem addressed by the pattern. For the sake of clarity, the problem is explained using a specific example. Examples for the majority of patterns are also illustrated by means of CPN diagrams.

• **Solution** This section describes possible solutions to the problem. Note that a single problem addressed by the pattern can be solved in several ways, depending on the requirements and/or context conditions with which the pattern is to be applied. Since multiple solutions are possible, we consider each solution separately and for each of the solutions we include an implementation subsection.

• **Implementation of Solution** This section illustrates how the approach identified in the Solution can be implemented in CPN Tools. The implementation not only graphically represents the pattern with CPNs, but also describes how to integrate this solution into the example considered in the Problem description section. A solution may have several implementations. The presented implementations may not be the only way to implement a solution correctly. An implementation alternative must be selected based on the context conditions of the pattern. Note that the correctness of an implementation is guaranteed only if CPN Tools is used for implementation purposes. The implementation may deviate from the given one if a different tool to CPN Tools is applied. However, most of the high-level Petri net tools provide similar mechanisms and it is usually easy to adapt the implementation for other tools (e.g., CPN-AMI, ExSpect, VisualPetri, SEA, Tina, etc.).

• **Consequences** This section outlines what the possible advantages/disadvantages of using the pattern are. For patterns with multiple solutions this section elaborates on the differences between them.

• **Examples** This section lists several examples which demonstrate the use of the pattern in practice.

• **Related Patterns** This section specifies the relationship of this pattern to other patterns. Specific relationship types are explained in detail in Section 3.3.

We will use this format to describe the 33 CPN patterns that have been identified. However, before presenting the pattern catalog, we first introduce main concepts required in order to understand the patterns.

### 3.1 Main concepts of CPN

A basic CPN model consists of a set of **places** and **transitions** connected by means of directed **arcs**. An example of a CPN model created in CPN Tools is shown in Figure 14. Places, depicted by ovals, represent intermediary states of the modeled process. Transitions, depicted by rectangles in Figure 14, represent tasks or actions that have to be executed. Directed arcs connect places and transitions, and denote the flow of control and data in the model. **Tokens** represent information objects, either conceptual or physical, that are stored in places. Places are typed, meaning that they may store tokens only of a specific color (i.e. data type). In Figure 14 the o1 place stores pairs of integers, while the rest of the places may store tokens of the integer (INT) type.

Transitions consume tokens from input places and produce tokens in output places. Tokens can be consumed by a transition only if the transition is enabled. The transition is enabled if sufficient tokens are present in every input place and conditions specified in the transition guard are satisfied. Figure 14 contains an example of the transition guard defined for the A transition. This guard specifies that variable i must be less than 10 in order for the transition to be enabled.
Annotations on arcs specify how many tokens can be consumed by a transition from its input places and how many tokens have to be produced to each of the output places. In Figure 14 transition \( A \) consumes a token from the \( i_1 \) input place, and produces tokens with the same value in both its output places \( p_1 \) and \( p_2 \). The \( B \) transition requires inputs from both places \( p_3 \) and \( p_4 \) in order for it to be enabled. When it fires, a pair of integers \((i,j)\) is produced in the output place \( o_1 \).

Annotations on arcs may include conditional statements and functions written using the CPN-ML-language. Conditional annotations are used in Figure 14 on the outgoing arcs from the \( C \) transition. Depending on the value of the variable \( i \), one or the other outgoing branch is taken (an \texttt{empty} value corresponds to no token being produced). An annotation associated with one of the outgoing arcs contains the \texttt{inc()} function, which is declared outside of the net body. If the condition specified in the arc inscription is satisfied, this function is called, resulting in the value \( i \) being incremented.

![Figure 14: Example of a CPN model](image)

Sub-processes defined separately from the main process can be mapped onto a particular transition (known as a substitution transition). When such a transition becomes enabled, it passes the flow of control to the sub-process associated with it. In this way, a hierarchical net structure can be created. In Figure 14 the \( B \) transition represents a substitution transition, whose behavior is unfolded to the net presented in Figure 15. The input and output places connected to the main process are marked as input and output ports.

![Figure 15: Example of a sub-net](image)

Although CPNs have the capability to represent time within a model, this feature has been not extensively used in this research and therefore is not discussed here. Having introduced the relevant concepts we now move on to the description of the CPN patterns that have been identified.
## Overview of CPN patterns presented in Section 3.2

<table>
<thead>
<tr>
<th>Type of problem addressed</th>
<th>Pattern group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-based control-flow routing</td>
<td>1 Deterministic XOR-Split (p.36)</td>
</tr>
<tr>
<td></td>
<td>2 Non-Deterministic XOR-Split (p.38)</td>
</tr>
<tr>
<td></td>
<td>3 OR-Split (p.40)</td>
</tr>
<tr>
<td>Filtering of data elements being transferred based on data</td>
<td>4 BSI Filter (p.42)</td>
</tr>
<tr>
<td>properties</td>
<td>5 BSD Filter (p.43)</td>
</tr>
<tr>
<td></td>
<td>6 NBSI Filter (p.45)</td>
</tr>
<tr>
<td></td>
<td>7 NBSD Filter (p.47)</td>
</tr>
<tr>
<td>Identity management</td>
<td>8 ID Matching (p.49)</td>
</tr>
<tr>
<td></td>
<td>9 ID Manager (p.51)</td>
</tr>
<tr>
<td>Data exchange during bilatery interactions</td>
<td>10 Asynchronous Transfer (p.54)</td>
</tr>
<tr>
<td></td>
<td>11 Synchronous Transfer (p.56)</td>
</tr>
<tr>
<td></td>
<td>12 Rendezvous (p.58)</td>
</tr>
<tr>
<td>Data exchange during one-to-many and many-to-one interactions</td>
<td>13 Asynchronous Router (p.60)</td>
</tr>
<tr>
<td></td>
<td>14 Broadcasting (p.63)</td>
</tr>
<tr>
<td></td>
<td>15 Distributed Data Processing (p.65)</td>
</tr>
<tr>
<td>Referral to single and multiple data elements and restrictions</td>
<td>16 Aggregate Objects (p.67)</td>
</tr>
<tr>
<td>associated with their storage</td>
<td>17 Deaggregate Objects (p.69)</td>
</tr>
<tr>
<td></td>
<td>18 Capacity Bounding (p.71)</td>
</tr>
<tr>
<td></td>
<td>19 Containment Testing (p.73)</td>
</tr>
<tr>
<td>Coordinating the order in which data elements aggregated into</td>
<td>20 Queue (p.75)</td>
</tr>
<tr>
<td>a collection are being utilized</td>
<td>21 FIFO Queue (p.78)</td>
</tr>
<tr>
<td></td>
<td>22 LIFO Queue (p.79)</td>
</tr>
<tr>
<td></td>
<td>23 Random Queue (p.80)</td>
</tr>
<tr>
<td></td>
<td>24 Priority Queue (p.81)</td>
</tr>
<tr>
<td></td>
<td>25 Prioritized Execution (p.85)</td>
</tr>
<tr>
<td>Content management within the specified region</td>
<td>26 Region Flush (p.86)</td>
</tr>
<tr>
<td></td>
<td>27 Content Setting (p.89)</td>
</tr>
<tr>
<td>Referral to single and multiple data elements and restrictions</td>
<td>28 Shared Database (p.90)</td>
</tr>
<tr>
<td>associated with their storage</td>
<td>29 Database Management (p.92)</td>
</tr>
<tr>
<td></td>
<td>30 Concurrent Access (p.95)</td>
</tr>
<tr>
<td></td>
<td>31 Copy Manager (p.96)</td>
</tr>
<tr>
<td></td>
<td>32 Lock Manager (p.98)</td>
</tr>
<tr>
<td></td>
<td>33 Bi-Lock Manager (p.100)</td>
</tr>
</tbody>
</table>

This table gives only a brief overview of the CPN patterns presented in Section 3.2. For classification purposes, patterns addressing the same kind of problem have been combined in a single pattern group. Pattern groups are presented in the pattern catalog in the same order as they are listed in this table. The description of problems addressed by a particular pattern group as well as the main differences between the patterns are provided in the form of a summary, preceding the actual pattern descriptions.
<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Deterministic XOR-Split</td>
<td>To allow at most one transition out of several possible transitions to execute, based on the fulfillment of conditions which mutually exclude each other.</td>
</tr>
<tr>
<td>2</td>
<td>Non-Deterministic XOR-Split</td>
<td>To allow the execution of precisely one transition out of several possible transitions that satisfy the same conditional expression.</td>
</tr>
<tr>
<td>3</td>
<td>OR-Split</td>
<td>To allow any number of transitions to be selected for execution, based on the fulfillment of a specific conditional expression.</td>
</tr>
<tr>
<td>4</td>
<td>BSI Filter</td>
<td>To prevent data, which does not conform to a specified property, from being processed.</td>
</tr>
<tr>
<td>5</td>
<td>BSD Filter</td>
<td>To prevent data being passed which does not conform to a property related to the state of an external data structure.</td>
</tr>
<tr>
<td>6</td>
<td>NBSI Filter</td>
<td>To prevent data that does not conform to a specified property from being passed, while avoiding the accumulation of non-conforming data at the filter's input place.</td>
</tr>
<tr>
<td>7</td>
<td>NBSD Filter</td>
<td>To filter-out data that does not conform to a property involving the state of an external data-structure, while avoiding accumulation of non-conforming data at the filter’s input place.</td>
</tr>
<tr>
<td>8</td>
<td>ID Matching</td>
<td>To make information objects distinguishable.</td>
</tr>
<tr>
<td>9</td>
<td>ID Manager</td>
<td>To ensure the uniqueness of identifiers used for distinguishing of identical objects.</td>
</tr>
<tr>
<td>10</td>
<td>Asynchronous Transfer</td>
<td>To allow the transfer of data from one location to another, while avoiding blocking of the sender.</td>
</tr>
</tbody>
</table>

In order to provide an overview of the CPN patterns presented in the catalog, this table describes a set of patlets consisting of the name and the intent associated with a pattern.
<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Synchronous Transfer</td>
<td>To allow the transfer of data from one location to another, ensuring that a party that posted a request is blocked until the requested information becomes available.</td>
</tr>
<tr>
<td>12</td>
<td>Rendezvous</td>
<td>To allow multiple processes to synchronously exchange data in both directions.</td>
</tr>
<tr>
<td>13</td>
<td>Asynchronous Router</td>
<td>To enable asynchronous transfer of data from a single source to a dedicated target, ensuring that old targets can be easily removed and new targets can be added without affecting the source.</td>
</tr>
<tr>
<td>14</td>
<td>Broadcasting</td>
<td>To broadcast data from a single source to multiple targets, while avoiding creating a direct dependency between them.</td>
</tr>
<tr>
<td>15</td>
<td>Distributed Data Processing</td>
<td>To decompose a data element in smaller parts in order for them to be processed in parallel, subsequently merging the processed data later on.</td>
</tr>
<tr>
<td>16</td>
<td>Aggregate Objects</td>
<td>To allow a set of information objects to be manipulated as a single entity.</td>
</tr>
<tr>
<td>17</td>
<td>Deaggregate Objects</td>
<td>To allow the manipulation of an object aggregated into a collection as an independent entity.</td>
</tr>
<tr>
<td>18</td>
<td>Capacity Bounding</td>
<td>To prevent over-accumulation of objects in a certain place.</td>
</tr>
<tr>
<td>19</td>
<td>Containment Testing</td>
<td>To allow the (non)-availability of objects with particular properties in a given location to be tested.</td>
</tr>
<tr>
<td>20</td>
<td>Queue</td>
<td>To allow the manipulation of queued objects in a strictly specified order.</td>
</tr>
<tr>
<td>21</td>
<td>FIFO Queue</td>
<td>To allow manipulation of queued objects in a strictly specified order, such that an object, which arrives first, is consumed first.</td>
</tr>
<tr>
<td>22</td>
<td>LIFO Queue</td>
<td>To allow manipulation of queued objects in a strictly specified order, such that the most recently added object is retrieved first.</td>
</tr>
<tr>
<td>23</td>
<td>Random Queue</td>
<td>To allow manipulation of queued objects such that an object is added to the collection in some specified order and an arbitrary object is consumed from it.</td>
</tr>
</tbody>
</table>
### CPN pattlets (cont.)

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>p.</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Priority Queue</td>
<td>81</td>
<td>To allow the selection of queued objects in the order of object priority.</td>
</tr>
<tr>
<td>25</td>
<td>Prioritized Execution</td>
<td>85</td>
<td>To coordinate the execution order of two tasks, such that in situations when both of them are simultaneously enabled, one of them always executes before the other.</td>
</tr>
<tr>
<td>26</td>
<td>Region Flush</td>
<td>86</td>
<td>To clear the content of all places in a particular region.</td>
</tr>
<tr>
<td>27</td>
<td>Content Setting</td>
<td>89</td>
<td>To reset the content of a particular region to another state.</td>
</tr>
<tr>
<td>28</td>
<td>Shared Database</td>
<td>90</td>
<td>To enable the centralized storage of data shared by multiple tasks with support for different levels of data visibility (i.e. local, group, or global).</td>
</tr>
<tr>
<td>29</td>
<td>Database Management</td>
<td>92</td>
<td>To specify an interface for accessing data, stored in a shared database, for read-only and modification purposes.</td>
</tr>
<tr>
<td>30</td>
<td>Concurrent Access</td>
<td>95</td>
<td>To provide concurrent access to common data elements by a series of individual tasks without implying any changes to these data elements.</td>
</tr>
<tr>
<td>31</td>
<td>Copy Manager</td>
<td>96</td>
<td>To make data, stored in a shared database, available at other locations for local use, whilst maintaining the consistency of data in all places.</td>
</tr>
<tr>
<td>32</td>
<td>Lock Manager</td>
<td>98</td>
<td>To synchronize access to shared data by means of exclusive locks.</td>
</tr>
<tr>
<td>33</td>
<td>Bi-Lock Manager</td>
<td>100</td>
<td>To synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks.</td>
</tr>
</tbody>
</table>

### 3.2 Catalog of CPN patterns

In this section, we describe 33 CPN patterns using the selected pattern format. The patterns presented have been gathered empirically. The majority of patterns originate from the knowledge of experts in the field (e.g., Kurt Jensen, Wil van der Aalst, Kees van Hee, Wolfgang Reisig). Some of the patterns have been fragmentally documented by them in literature [7,8,119,130,185], whereas others have been modeled based on the analysis of models presented in the 1999-2004 CPN Workshop papers [66].
The first pattern group consists of three patterns (Deterministic XOR-Split, Non-deterministic XOR-split and OR-split), which address issues related to data-based control-flow routing. Where specific properties of data elements supplied need to be analyzed in order to decide which of the available tasks has to be executed, the nature of the decision associated with the evaluation of the data elements is either deterministic, non-deterministic or a combination thereof. The Deterministic XOR-Split pattern describes a situation where exactly one of several tasks is selected based on the explicitly defined mutually excluding conditions. The Non-Deterministic XOR-Split pattern allows either of tasks satisfying the same set of data conditions to be selected non-deterministically. Whereas the OR-split pattern considers situations where data conditions associated with each of the available tasks may overlap. The evaluation of these conditions may potentially result in one or more tasks being selected for execution.

Pattern 1  DETERMINISTIC XOR-SPLIT  

**Intent**  To allow at most one transition out of several possible transitions to execute, based on the fulfillment of conditions which mutually exclude each other.  

**Motivation**  In many systems, there are data structures which can be accessed by several tasks concurrently. However, it is not allowed that several tasks execute at the same time, i.e. only one task out of several possible needs to be selected. In terms of safety requirements for event-driven systems, it is allowed that after a certain event only one of two possible events happen, but not both. Similar, in some situations there is the need to ensure that at most one process can be engaged in a specified activity at a time.  

**Problem Description**  Consider the model in Figure 16 with a place **Data** and two transitions **Activity1** and **Activity2**, which have access to data elements stored in this place. Depending on the value of data element **d** supplied by the **Data** place it is necessary to ensure that only one of the transitions **Activity1** and **Activity2** executes. In the net in Figure 16 both transitions are enabled, so the choice of an activity is not explicitly defined.

![Figure 16: The problem of a non-deterministic choice](image)

**Solution 1**  In order to define explicitly which transition out of several possible options is selected for execution, all transitions have to be associated with guards containing conditional expressions which mutually exclude each other. The evaluation of mutually excluding expressions results in the selection of at most one transition.  

**Implementation of Solution 1**  The list of instructions below describes how to implement Solution 1 for the Deterministic XOR-Split pattern (cf. Figure 17):

- Place **Data**, where data for the execution of activities is stored, is connected to transitions **Activity1** and **Activity2**. The flow of data element **d** occurs in the direction of the arcs connecting the place and the transitions.
Mutually excluding conditional expressions, which cover all possible values of data elements that can be stored in the Data place, are added to the transition guards. In Figure 17, the guard for the Activity2 transition evaluates to True and enables the execution of this transition if the data element d is bigger than N. The guard associated with the Activity1 transition evaluates to True in all other cases, i.e. if the value of data element d is less or equal to N. Note, that since the data element included in the expression of the guard can be of composite form, several conditional statements may be specified in each guard respectively.

**Solution 2** In order to define explicitly which transition out of several possible options is selected for execution, each transition must be associated with conditional expressions which mutually exclude each other. Evaluation of these expressions must be performed in advance and must result in the selection of at most one transition.

**Implementation of Solution 2** The list of instructions below describes how to implement Solution 2 for the Deterministic XOR-Split pattern (cf. Figure 18):

- The Data place, which provides data inputs to transitions Activity1 and Activity2, is decoupled from the transitions by introducing two intermediate places Data Act1 and Data Act2. Places Data Act1 and Data Act2 store only data elements that satisfy the enabling criteria for the Activity1 and Activity2 transitions respectively.
- The Define branch transition is introduced in order to move the responsibility for evaluation of conditional expressions and selection of an activity away from the activities Activity1 and Activity2.
- The Data place is connected to the Define branch transition in order to supply the input data element d. This transition is connected to places Data Act1 and Data Act 2. The inscriptions of arcs connecting the transition with these places include conditional expressions for selecting the corresponding branch.
- In Figure 18, conditional expressions if d>N then 1’d else empty and if d<=N
then `1'd else empty` are used\footnote{Note that in CPN Tools an expression “if b then 1’d else empty” can be replaced by a shortcut “b%d”.}. Note that considered expressions mutually exclude each other. Based on the evaluation of these expressions, data element \( d \) is placed into the Data Act2 place if the value of data \( d \) is bigger than \( N \) or to the Data Act1 place otherwise.

**Consequences** This pattern can be applied to make an explicit selection of at most one activity, event or a task, that is modeled by means of a transition, based on the characteristics of data provided as an input.

The Deterministic XOR-split pattern provides two solutions. The moment at which mutually excluding conditional expressions are being evaluated in order for exactly one out of several possible transitions to be selected (i.e. as late as possible or in advance) is the main distinction between Solution 1 and Solution 2 of the Deterministic XOR-Split pattern. In Solution 1, this decision is made at the latest possible moment, while in Solution 2 as early as possible. Solution 2 can be applied when the selection of a transition has to be performed based on conditional statements which are evaluated before the transition actually becomes enabled. This solution allows the transitions to abstract from the boolean expressions involved in the selection procedure, thus making these expressions transparent to them.

This pattern addresses the problem of data based control-flow routing, which is similar to the ones addressed in the Non-Deterministic XOR-Split pattern (cf. page 38) and the OR-Split pattern (cf. page 40).

**Examples**
- Depending on the type of a file provided by a user (e.g., a text document or a picture), either the Word editor or the Paint editor is launched.
- In financial institutions, requests from private individuals and corporate representatives follow different paths in a process.

**Related Patterns** This pattern is similar to the Non-Deterministic XOR-Split pattern (cf. page 38) and the OR-Split pattern (cf. page 40).
these activities, whilst still ensuring that only one activity may execute at a time. In this net, such a behavior cannot be obtained as there is a strict separation of the conditional expressions on the basis of mutual exclusion.

![Diagram](image)

**Figure 19:** Implementation of Solution 2 of the Deterministic XOR-split pattern

**Solution** In order to allow the non-deterministic selection of precisely one transition out of several possible ones, every transition has to be associated with overlapping conditional expressions.

**Implementation of Solution** The list of instructions below describes how to implement the *Non-Deterministic XOR-Split* pattern (cf. Figure 20):

- The *Data* place, where data elements determining the execution of activities is stored, is connected to transitions *Activity1* and *Activity2*.

- Conditional expressions in transition guards are specified as being overlapping for values where both transitions could be enabled. Figure 20(a) illustrates that data values between N1 and N2 (N2>N1) can be processed by either of the activities, while all other values outside of the specified range can be handled by only one activity. Note that the choice of activities, when the common condition is satisfied, is non-deterministic. It is not possible to predict which of the activities will handle values in this range.

- If only one of the two transitions should be selected non-deterministically for all possible data values, the transition guards can be omitted as visualized in Figure 20(b).

![Diagram](image)

**Figure 20:** Implementation of the Non-deterministic XOR-split pattern

**Consequences** This pattern can be applied to realize the non-deterministic selection of a task from a set of possible tasks, while ensuring that only one task may execute at a time. It addresses the problem of data based control-flow routing, in a similar way to the *Deterministic XOR-Split* pattern (cf. page 36) and the *OR-split* pattern (cf. page 40).

In comparison to the mutually excluding conditional expressions, the overlapping conditions in some way extend the options available for handling data. Note that in general,
overlapping and mutually excluding conditions can be combined. In contrast to the overlapping conditions, one can specify conditions that are too restrictive, i.e. where some data elements do not result in any of the transitions being enabled. For instance, if one task handles data with a value smaller than 5, and another task handles data with a value bigger than 10, then data in the range from 5 to 10 will not be handled at all. This may cause undesirable blocking or even deadlock. Conditions that are too strict can be applied if one can ensure that only data in the specified range is likely to occur. However, the usage of insufficient conditional expressions is not desirable.

Examples
- Any application suitable for processing of a textual documents (e.g., Wordpad, Notepad, or Word) can be launched for processing of a text-file supplied by a user.
- Two receptionists process requests of visitors to the city hall. When both receptionists are free, either of them may be approached by a new visitor.

Related Patterns This pattern is similar to the Deterministic XOR-Split pattern (cf. page 36) and the OR-Split pattern (cf. page 40).

Pattern 3 OR-SPLIT

Intent To allow any number of transitions to be selected for execution, based on the fulfillment of a specific conditional expression.

Motivation The usage of mutually exclusive conditional expressions associated with a group of transitions ensures that only one of these transitions is selected for execution. In some situations, there is no need to pose strict restrictions on the number of transitions which may execute simultaneously. Thus, the number of transitions that may execute concurrently may vary depending on the fulfillment of a specified condition. For instance, when distributing work items, an employee may receive a work item if she specializes in the specific type of work being distributed, however some types of work may be executed by all employees.

Problem Description Figure 21 presents two transitions Activity 1 and Activity 2, at most one of which may execute when a condition common to both of them is satisfied. Although this diagram allows either of the tasks to be selected when \( N1 < N2 \), it does not allow all of the tasks to execute concurrently.

\[
\text{Data} \quad \text{Activity1} \quad \text{Activity2}
\]

\[
\begin{align*}
\text{Data} & \quad \text{Data} \\
\text{Activity1} & \quad \text{Activity2}
\end{align*}
\]

\[ [d>N1] \quad [d<=N2] \]

Figure 21: Implementation of Solution 1 of the Non-Deterministic XOR-split pattern

Solution In order to enable the execution of all, some or only one transition out of several available options based on the fulfillment of common or mutually exclusive conditional expression, extend Solution 2 of the Deterministic XOR-Split pattern with overlapping data conditions.
Implementation of Solution  The list of instructions below describes how to implement the OR-Split pattern (cf. Figure 22):

- Solution 2 of the Deterministic XOR-Split pattern (cf. page 36) is extended by introducing overlapping conditional expressions in inscriptions of the arcs from the Define branch transition to places Data Act1 and Data Act2, which provide input data to transitions Activity1 and Activity2 respectively.

- When the Define branch transition fires, the specified data is added to the outgoing place if a condition in the expression associated with the arc to the outgoing place is satisfied. As such, if the value of data element supplied by the Data place is in the range between \( N_1 \) and \( N_2 \) \((N_1 < N_2)\), then this data element will be provided as an input to both transitions Activity1 and Activity2. However, if the data value is less than \( N_1 \), then only transition Activity1 will execute; and if the data value is bigger than \( N_2 \), then it will only be provided as an input to transition Activity2.

![Figure 22: Implementation of the OR-split pattern](image)

Consequences This pattern can be applied to allow one or more tasks to execute concurrently depending on the fulfillment of the conditional expressions associated with the each of these tasks. Similar to Solution 2 of the Deterministic XOR-Split pattern, in this pattern, tasks processing data are not aware of conditions based on which they have been enabled. This allows tasks to concentrate on the actual processing of data, without being involved in the evaluation of the related conditions.

Note that this pattern is able to support the functionality of the Asynchronous Router pattern (cf. page 60) for distributing data elements to a number of targets on the conditional basis. After having distributing data amongst the targets, the data may need to be merged later on into a single data entity. In order to ensure the correct branching and synchronization of the data elements, the Distributed Data Processing pattern may need to be applied (cf. page 65). To realize such a behavior, it may be necessary to keep track of the selected branches by means of the Boolean variables.

Example

- Specialized data files can be processed only by a dedicated program, whereas for reading of a text file any of available programs can be launched.
- Doctors working at a hospital specialize in a particular kind of treatment, however a patient may count on one or more specialists able to offer the first-aid help.

Related Patterns This pattern extends Solution 2 of the Deterministic XOR-Split pattern (cf. page 36).
The second group of CPN patterns addresses problems related to filtering of non-complying data elements being transferred from one place to another based on the analysis of data properties. Depending on whether the state of an external data structure is involved in the evaluation of the data properties and whether the non-compliant objects are being blocked in the filter's input place, we distinguish the Blocking State-Independent Filter, Blocking State-Dependent Filter, Non-Blocking State-Independent Filter, and Non-Blocking State-Dependent Filter patterns.

**Pattern 4** BLOCKING STATE-INDEPENDENT (BSI) FILTER

**Intent** To prevent data, which does not conform to a specified property, from being processed.

**Motivation** In many different application examples, data being transferred from one source to another may be scrambled, modified or flawed. Because of this, a target data recipient instead of receiving data in an expected format, can be faced with incorrect, invalid or unknown data. To avoid this problem, there is a need to filter incoming data based on data-specific qualities or characteristics.

**Problem Description** Consider a process where a number of requests (such as product enquiries, subscription offers or insurance applications) are provided to a particular resource for processing. Accepting all requests may either be not allowed or undesirable as only requests satisfying a specific set of properties may be handled. For instance, in many insurance companies processing of claims may be performed only by authorized employees (thus it is necessary to check whether the request was delivered to the right person). The handling of requests in such situations has to be performed only if all conditions specified have been fulfilled.

**Solution** In order to prevent data, non-conforming to a certain property, from passing, use a blocking state-independent (BSI) filter. The BSI filter consumes input data and compares it against the set of specified conditions. The filter passes the data through only if it satisfies the specified set of properties, otherwise the data is blocked.

**Implementation of Solution** The list of instructions below describes how to implement the BSI Filter pattern (cf. Figure 23):

- The In place, where non-filtered data is located, is connected to the Filter transition.
- The guard of the Filter transition contains the f(x) function that specifies conditions for data filtering. Note that incoming data may be of any format, e.g., lists, composite data structures, etc.
- In this implementation alternative, the Filter transition has knowledge of data bounds within which the incoming data should fit. It passes data elements, whose value is within the boundaries indicated, from the input place to the output place Out, while blocking the rest of the data in the input place.

![Figure 23](image.png)

**Figure 23:** Implementation of the Blocking State-Independent Filter pattern

Figure 24 illustrates an example of filtering out data exceeding a certain value (x>5), i.e. only data with values bigger than five may be passed to the output place Out. The Filter
transition is enabled for the binding \( x=7 \), as this is the only value satisfying the filtering condition.

\[ \text{Figure 24: Example of applying the Blocking State-Independent Filter pattern (place In holds three tokens: one with value 1, one with value 5, and one with value 7)} \]

Consequences  This pattern should be applied in situations, where data properties are used as a means for filtering data. One of the characteristics of this pattern is its “blocking” property: the filter ensures that all non-compliant data is blocked and cannot pass through the filter. The drawback of this pattern is that it does not prevent the accumulation of non-complaining objects at the input place of the filter. To address this problem an extension of this pattern, i.e. Non-Blocking State-Independent Filter (cf. page 45), can be used instead.

In some situations, the filtering properties need to be based not only on the value of incoming data but also on some context information, e.g., the state of some external data structure. To handle such situations, an extension of this pattern, i.e. Blocking State-Dependent Filter (cf. page 43), can be applied.

Examples
- Only trigger the credit-application task for incoming application objects where age is greater or equal to 18.
- For subsequent review and participation in the conference, only accept Emails whose subject contains the ‘paper submission’ field.

Related Patterns  This pattern is extended by the Blocking State-Dependent Filter pattern (cf. page 43) and the Non-Blocking State-Independent Filter pattern (cf. page 45).

Pattern 5  BLOCKING STATE-DEPENDENT (BSD) FILTER

Intent  To prevent data being passed which does not conform to a property related to the state of an external data structure.

Motivation  The Blocking State-Independent Filter pattern (cf. page 42) provides a means of filtering out data satisfying a specific data property and blocking non-compliant data from being passed. In some situations, the analysis of data properties should be based not only on the data value but also on the state of an external data structure. For instance, only passengers who are registered in the central database are allowed to board a flight, or only passengers whose names do not appear in the ‘black list’ are able to complete passport control. The state of such an external data structure may be static or may vary over time.

Problem Description  Figure 25 illustrates the solution of the BSD Filter pattern, where before processing data element \( x \) supplied by the In place first the properties of the data element are analyzed by means of the function \( f() \). Although this solution helps in filtering out data based on the data properties, it does not allow the information stored in some external place to be used during the analysis.

Solution  In order to prevent data, which does not conform to a property based on the state of an externally located data structure (i.e. list or multi-set), from being passed,
use a blocking state-dependent (BSD) filter. The BSD filter consumes data from an input place, checks the state of the external data collection, and where the filtering conditions are fulfilled passes the data to the output place.

**Implementation of Solution** The list of instructions below describes how to implement the BSD Filter pattern (cf. Figure 26).

- The implementation of the Blocking State-Independent Filter (cf. page 42) is extended by connecting the Filter transition to an external data collection stored in the External place place. The state of this external place needs to be used in the conditions for data filtering. Note that the collection is always populated, i.e. there is always one token of the collection type residing in it. Such a place may, for instance, represent a shared database.
- In Figure 26(a), a guard of the Filter transition contains the check(l,x) function which examines an input data element x and an external collection l to identify whether a specific data property is fulfilled. For instance, the filtering condition may be used to check if no duplicates have been sent.
- In Figure 26(b), the External place stored a collection of all data values passed to the Out place, and the check(l,x) function examines whether the input data element x is contained in the list l. Similar filtering conditions may test non-containment of elements in the collection or non-compliance with a certain criteria.
- Note that the state of the data collection can be static as in Figure 26(a), i.e. do not change during the whole process execution, or be dynamic as in Figure 26(b) and vary during execution.

**Consequences** This pattern can be applied to test incoming data for fulfillment of a set of properties involving the state of an external data structure. As this pattern extends the Blocking State-Independent Filter pattern (cf. page 42), it also blocks data which does not satisfy a given set of properties in the filter’s input place. However, the problem of
accumulating non-complying data in the input place of the filter is not resolved by this pattern. In order to address this issue, an extension of this pattern, i.e. the Non-Blocking State-Dependent Filter pattern (cf. page 47), needs to be used instead.

Objects stored in an external collection can be represented as stand-alone tokens or as elements aggregated into a single collection (obtained by applying the Aggregate Objects pattern (cf. page 67)).

Examples

- To avoid unintended messages, only accept Emails from the senders whose addresses are recorded in the contact list.
- Before a candidate is invited to a job interview at a financial organization, the candidate’s credentials are checked against the historical database of employees who were involved in illegal activities.

Related Patterns This pattern is an extension of the Blocking State-Independent Filter pattern (cf. page 42). It is extended by the Non-Blocking State-Dependent Filter pattern (cf. page 47). Furthermore, it can be combined with the Aggregate Objects pattern (cf. page 67).

Pattern 6 NON-BLOCKING STATE-INDEPENDENT (NBSI) FILTER

Intent To prevent data that does not conform to a specified property from being passed, while avoiding the accumulation of non-conforming data at the filter’s input place.

Motivation The Blocking State-Independent Filter pattern (cf. page 42) offers a solution for preventing the data that does not meet a certain property from being passed. However, it does not resolve the problem of non-conforming data accumulating in the filter’s input place. In some situations, instead of blocking or ignoring the non-conforming data, it is necessary to reroute this data or to store it elsewhere.

Problem Description Figure 27 illustrates the solution of the Blocking State-Independent Filter. In this solution, any data elements which do not fulfill a specific property (as specified by the guard of the Filter transition) cannot be passed through, i.e. they accumulate in the In place. Assume that non-complying data elements need to be used for processing elsewhere in the process. In this solution it is not possible due to blocking of the non-complying data in the filter’s input place.

Solution In order to prevent data, that does not conform to a certain property, from being passed, while avoiding the accumulation of non-conforming data in the filter’s input place, use a non-blocking state-independent (NBSI) filter. The NBSI filter consumes all data from the input place, analyzes it against a set of the specified conditions, and passes it to a specific output depending on whether it conforms or not with the filter property.

Implementation of Solution The list of instructions below describes how to implement the NBSI Filter (cf. Figure 28):
• The In place, where non-filtered data is located, is connected to the Filter transition which tests incoming data.

• The Filter transition is connected to two output places Passed and Not passed, where data conforming and not conforming to the filter property is placed respectively. Note that the Not passed place is optional (cf. Figure 28(b)), i.e. it is required only if data elements non complying with a certain property need to be accumulated in order to be used elsewhere.

• Arcs connecting the Filter transition with output places are associated with mutually exclusive filtering conditions: a filtering condition cond which must be fulfilled by data in order to pass through the filter and the negation of this condition which filters non-conforming data.

Figure 28: Implementation of the Non-Blocking State-Independent Filter pattern

Figure 29 illustrates an example of filtering out data of Integer type, whose value is below the specified range (i.e. \( x \leq 5 \)).

Figure 29: Example of applying the Non-Blocking State-Independent Filter pattern

Consequences This pattern can be used to analyze a stream of incoming data elements according to a set of specified rules. It is essential for this pattern to avoid blocking of non-conforming data at the filter’s input place, since all data elements need subsequently to be used somewhere else in the process.

In contrast to the Blocking State-Independent Filter pattern (cf. page 42), this pattern handles all incoming data routing data satisfying a filter property to one place, and non-conforming data to another place. This filter works deterministically, i.e. it has sufficient knowledge about all possible data values and in order to filter the data it uses mutually exclusive data conditions as suggested by Solution 2 of the Deterministic XOR-Split pattern (cf. page 36).

Example

• Patients arriving at the emergency department of a hospital are examined against a list of life-threatening symptoms. Patients that require emergency treatment are
immediately directed to the emergency care unit, while the other patients are sent to appropriate specialists.

- For security reasons, before entering the museum, visitors and their belongings need to be checked. Women and children are separated from men, and both two groups are analyzed concurrently.

**Related Patterns** This pattern is an extension of the *Blocking State-Independent Filter* pattern (cf. page 42). It uses Solution 2 of the *Deterministic XOR-Split* pattern (cf. page 36).

**Pattern 7 NON-BLOCKING STATE-DEPENDENT (NBSD) FILTER**

**Intent** To filter-out data that does not conform to a property involving the state of an external data-structure, while avoiding accumulation of non-conforming data at the filter’s input place.

**Motivation** The *Blocking State-Dependent Filter* pattern (cf. page 43) prevents data elements that do not fulfill a property involving the state of external data structure from being passed. However, it does not handle the problem of the accumulation of non-conforming data elements at the filter’s input place. In some situations, data which does not satisfy the filter property needs to be rerouted to another place instead of being blocked and ignored by the filter.

**Problem Description** Figure 30 illustrates the solution of the *Blocking State-Dependent Filter*. In this solution, any data elements which do not fulfill a specific property (potentially involving the state of an external data structure) cannot be passed, i.e. they accumulate in the \(\text{In} \) place. This solution does not allow to use non-complying data elsewhere in the process by blocking it in the filter’s input place.

![Figure 30: Problem of data accumulation associated with the BSD filter](image)

**Solution** In order to filter-out data elements that do not conform to a property that involves the state of an externally located data structure (i.e. list or multi-set), while avoiding the accumulation of non-conforming data elements in the filter’s input place, use a *non-blocking state-dependent (NBSD) filter*. The NBSD filter consumes data from the input place, checks the state of the external data collection, and passes on data elements that fulfill the filtering condition to one output and non-compliant data to another output.

**Implementation of Solution** The list below describes how to implement a *Non-Blocking State-Dependent Filter* pattern (cf. Figure 31):

- The solution of the *Blocking State-Independent Filter* pattern is extended in the following way. The Filter transition is connected to two output places: one place Passed will serve as the destination for the data elements conforming to the filter...
property and the other place Not passed will be the destination for non-conforming data elements respectively. Note that the Not passed place is optional, i.e. it is required only when non-complying data elements need to be accumulated in order to be used elsewhere.

- Arcs connecting the Filter transition with its outgoing places have to be associated with filtering conditions, i.e. a condition which must be fulfilled for data elements passing through the filter, and the negation of this condition for filtering out non-conforming data elements.

\[
\text{if cond}(l,x) \text{ then } 1^*x \text{ else empty} \\
\text{if not(cond}(l,x)) \text{ then } 1^*x \text{ else empty}
\]

\(x\) Filter Passed
\(l\) State

Figure 31: Implementation of the Non-Blocking State-Dependent Filter pattern

The net in Figure 32 illustrates that data elements supplied to the filter are passed through only if they are not contained in the external data collection, while redundant data, i.e. the data which is already present in the external store, is filtered out. Note that other filtering conditions can be also used.

\[
\text{if not(contains}(l,x)) \text{ then } 1^*x \text{ else empty} \\
\text{if (contains}(l,x)) \text{ then } 1^*x \text{ else empty}
\]

\(x\) Filter Passed
\(l\) State

Figure 32: Example of applying the Non-Blocking State-Dependent Filter pattern

Consequences This pattern can be applied in order to examine data against a specified property based on the state of an external data structure in order to filter out non-conforming data elements. This pattern ensures that all incoming data is handled, thus avoiding the accumulation of non-conforming data at the filter’s input place.

This pattern can be considered as an extension of the Blocking State-Dependent Filter pattern (cf. page 43) with Solution 2 of the Deterministic XOR-Split pattern (cf. page 36).

Examples
- Handling of insurance claims, where all incoming claims should be analyzed and either reviewed in detail or rejected.
• A family doctor, based on the records available about a patient, decides whether to prescribe a medicine or to direct the patient to hospital for further tests.

**Related Patterns** This pattern extends the *Blocking State-Dependent Filter* pattern (cf. page 43) by using Solution 2 of the *Deterministic XOR-Split* pattern (cf. page 36).

---

The third pattern group offers solutions to issues related to identity management. This group consists of two patterns: *ID Matching* and *ID Manager*. The *ID Matching* pattern addresses the problem of matching data objects of the same data type after their original value has been modified. In order to distinguish objects of the same type, this pattern proposes a solution based on associating an identifier with each data object. The *ID Manager* pattern addresses the problem of uniqueness of object identifiers by introducing an id generator. Object identifiers are often used in this pattern catalog for referencing a specific object in a database or another collection, as well as for referring to specific parties engaged in the message exchange.

---

**Pattern 8 ID MATCHING**

**Intent** To make information objects distinguishable.

**Motivation** In CPNs, a token can serve as a representation of an information object (e.g., a process instance, a case, an item, etc). A place may contain multiple objects of the same type. In some cases, it is necessary to compare the original value of an object with the value of the same object after it has been modified. As a consequence of modification, the value of the object changes, and the object may lose its identity. Consequently, it becomes impossible to distinguish which of the modified values corresponds to which of the original values of the objects.

**Problem Description** Figure 33 illustrates the problem of object matching. Initially, two objects `obj` of the same type `T` are present in the *Start* place. These objects serve as an input for two functions `f1` and `f2`, which replace the values of the processed objects with a randomly generated integer number. After applying the functions, values produced by them need to be matched by the *Match pair* transition for each specific object.

![Figure 33: Problem of losing identity by multiple identical objects](image-url)

Tokens accumulate in the *Return f1(x)* and *Return f2(x)* places after the `f1` and `f2` functions are applied respectively. At the moment of modification, objects lost their identity, i.e. it becomes impossible to distinguish which of the values present in place *Return f1(x)* correspond to which of the values in place *Return f2(x)*.
Since tokens are consumed from places in a non-deterministic order, the value produced by the function \( f_1 \) for one object can be matched with a value produced by the function \( f_2 \) for another object. Such behavior is undesirable and may lead to inconsistent results and incorrect matching operations.

**Solution** In order to solve the problem of distinguishing individual objects, couple each object with an identifier. The availability of identifiers makes it possible to distinguish objects of the same type even when they have the same values.

**Implementation of Solution** The following list of instructions describes how to implement the **ID Matching** pattern (cf. Figure 34).

- Type \( T \), associated with the type of objects to be distinguished, is replaced by a multiset type \( T_{xID} \). Type \( ID \) is an arbitrary type selected to serve as an identification for an object (for example, INT, STRING, etc.). For instance, in case of integers, identifiers can be represented as 1, 2, 3, etc. One could also use a more complex/composite data type to encode id’s.

- Each identical object is coupled with a unique identifier (the problem of the ID uniqueness is addressed in the **ID Manager** pattern on page 51). For this, inscriptions on the arcs, which contain a variable representing an object, are replaced with a corresponding pair \( \text{variable of object type, variable of ID type} \). For instance, a variable \( x \) of type \( T \) can be replaced by a pair \( (x, \text{id}) \), where \( \text{id} \) is of type \( ID \).

- To refer to an object, the object identifier \( \text{id} \) is coupled with the value of the object. In order to match values corresponding to the same object, the **Match pair** transition is introduced. This transition is enabled only for objects whose identifiers match. Different variables representing identifiers, for instance \( \text{id1} \) and \( \text{id2} \), can be matched using the guard of the transition performing the matching, i.e. \( [\text{id1} = \text{id2}] \).

![Figure 34: Implementation of the ID Matching pattern](attachment:image.png)

Figure 34 illustrates how to incorporate identifiers into the example presented in the **Problem description** section. Note that in the initial marking, place **Start** contains two identical objects \( \text{obj} \) that are coupled with integer identifiers 1 and 2. When combined with identifiers, objects form distinct pairs. Even if the value of an object now changes, it will be possible to refer to the object by means of the identifier associated with it.

**Consequences** This pattern can be applied when it is necessary to refer explicitly to a specific object from a group of objects of the same type, distinguish between identical objects, or organize objects by referring to identifiers rather than actual values of the objects.

The **ID Matching** pattern helps to solve the problem of referring to in-distinguishable objects by means of object identifiers. However, this pattern does not guarantee that
the identifiers used are unique. In order to ensure that identifiers used for referencing to specific objects are unique, the *ID Matching* pattern must be combined with the *ID Manager* pattern (cf. page 51).

**Examples**
- Every year the level of sugar in the blood of patients with high risk of diabetes is analyzed. The results obtained are compared with those earlier stored under the patient’s name in a database in order to estimate the effectiveness of the treatment.
- Prior to sending the evaluation result to the review-assessment task, it is assigned a unique identifier to ensure it is still identifiable should its original value be changed during the review activity.

**Related Patterns** This pattern can be combined with the *ID Manager* pattern (cf. page 51) to ensure uniqueness of identifiers used for distinguishing identical objects.

**Pattern 9  ID MANAGER**

**Intent** To ensure the uniqueness of identifiers used for distinguishing of identical objects.

**Motivation** The *ID Matching* pattern (cf. page 49) solves the problem of distinguishing between identical objects by assigning an identifier to each of these objects. However, it does not guarantee that the identifiers used are unique. Since CPNs allow the use of multisets, where the same object can be created multiple times, identical objects may also have the same identifier, and thus become in-distinguishable. This may lead to confusion and diminish the value of the notion of object identity.

**Problem Description** Figure 35 illustrates the problem of ID uniqueness. The **Start** place contains two objects **a** and **b** of the same type **T**, each coupled with an identifier. In terms of the correctness of the specified net, no problems can be detected. However, in the current marking both objects are associated with the same identifier. These objects cannot be distinguished due to the non-uniqueness of their identifiers.

![Figure 35: Problem of non-unique identifiers](image_url)

**Solution** In order to guarantee the uniqueness of identifiers, used for distinguishing identical objects, use an *ID manager*. The *ID manager* ensures that only unique identifiers are generated. Optionally, the ID manager may contain the functionality for verifying and controlling the consistency of allocated identifiers.

In general, it is sufficient to have the mechanism of ID generation implemented. The *ID generation* operation is responsible for generation of new id’s. The uniqueness of identifiers used for object identification in this case is guaranteed by the fact that once assigned to
an object, the identifier will not be reassigned to another object in the future under any circumstances. In some situations, the number of identifiers that can be coupled to objects may be limited. When old objects are outdated, they need to be replaced with new ones. For this, old objects have to be decoupled from their identifiers in order for these identifiers to be reassigned again. The ID deletion operation is responsible for removing unused ID’s. Finally, the ID approval operation is responsible for examining whether the returned ID or the ID supplied by the ID generator, is not in use.

Implementation of Solution  The list of instructions below describes how to implement the ID Manager pattern.

- **ID generation** Figure 36(a) illustrates a mechanism for generating ID’s. Unique identifiers, produced by the Generate ID transition, are stored in the Fresh ID place. The Last ID place keeps track of generated identifiers, by storing the last produced one. Alternatively, a list of all allocated ID’s can be maintained. The Last ID place stores a token with an arbitrary integer value, that is incremented each time the Generate ID transition fires. The incrementing operation applied to the most recently generated identifier ensures the uniqueness of identifiers used. Note that in Figure 36(a) identifiers of INT type are used, however any data type having a total order can be used instead.

![Diagram](a) ID generation

![Diagram](b) ID deletion/verification

| Figure 36: Implementation of the ID Manager pattern |

- **ID deletion** To maintain the consistency of identifiers it might be necessary to remove id’s returned, for instance, due to object destruction, from the list of allocated identifiers. After the returned identifier has been deleted, it may be reallocated again. Figure 36(b) illustrates the operation of deleting the returned identifier from the list of allocated identifiers. The Destroy ID Transition removes returned identifier id from the list of allocated identifiers lid using the del() function:

\[
\text{fun del}(x, y::z) = \text{if } x=y \text{ then } z \text{ else } y::(\text{del}(x,z)) |
\]

\[
\text{del}(x, []) = []; \\
\]

The consistency manager can be obtained by combining the operations of inserting, deleting, and verifying identifiers.

\[\text{(Note that in CPN Tools there is a built-in function rm that can be used for this purpose.}\]
• **ID verification** Figure 36(b) illustrates the mechanism of verifying the uniqueness of identifiers before they are coupled to objects. Identifiers which are not yet allocated are identified using Solution 2 of the Filter pattern and added to the Approved ID’s place. Transition Approve ID takes as input a new id supplied for checking and a list of the allocated identifiers lid stored in place Existing ID’s, and applies the elt() function to check whether the id is an element of the list of the existing identifiers.

\[
\text{fun } \text{elt}(x,y::z) = \text{if } x=y \text{ then true else elt}(x,z) | \text{elt}(x,[]) = \text{false};
\]

Unique id’s are also added to the list of allocated identifiers lid in order to keep the list up-to-date. Note that allocated identifiers are aggregated into a list, for which the Aggregate Objects pattern (cf. page 67) is used. In addition, one can apply the Queue pattern (cf. page 75) to keep identifiers in the list in a strictly specified order.

Figure 37 shows how to incorporate the ID Manager pattern into the example described in the Problem description section. Note that only functionality of the ID Generation is used, because (re-)allocation of identifiers is not required in this example.

**Consequences** This pattern can be combined with the ID Matching pattern in order to guarantee the uniqueness of identifiers used for distinguishing identical objects, to keep track of allocated identifiers and to ensure the consistency of allocated identifiers. In addition, this pattern can be used to solve the problem of the data inconsistency occurring when identifiers of the outdated objects have to be reassigned. Note that in its solution, this pattern uses Blocking State-Dependent Filter pattern (cf. page 43) in combination with the Aggregate Objects pattern (cf. page 67).

**Examples**

- The tax office handles requests from visitors. When visitors arrive at the tax office, they receive a ticket with a number which specifies their place in the queue. The ticket number must be unique in order to avoid several visitors approaching the same tax officer at the same time.
- Employees with identical names are issued different email aliases so that they can be distinguished.

**Related Patterns** This pattern uses the Blocking State-Dependent Filter pattern (cf. page 43) and the Aggregate Objects pattern (cf. page 67) in its solution. It can be combined with the Queue pattern (cf. page 75).
The fourth group consists of three CPN patterns (Asynchronous Transfer, Synchronous Transfer and Rendezvous patterns), which address problems related to data exchange between two processes. The Asynchronous Transfer pattern describes situations where during data exchange, the sending process may continue executing without waiting for a response to be received for a message sent by it. The Synchronous Transfer pattern describes the situation where the sending process becomes blocked whilst it waits for a response to message that it has sent. Finally, the situation where the exchange of data between the processes may happen synchronously in both directions is described by the Rendezvous pattern.

When considering request-reply interactions, it is important to be aware that the delivery of messages is not always guaranteed. In Chapter 5 of this thesis, we examine various aspects of interactions involving two or more parties. Don’t miss the discussion of request-reply interactions on page 233.

Pattern 10  ASYNCHRONOUS TRANSFER

Intent  To allow the transfer of data from one location to another, while avoiding blocking of the sender.

Motivation  In a distributed environment, several processes may operate independently of each other until one process needs to interrupt another process in order to transfer data to it. Although the processes interact with each other through a data channel, they must remain independent.

Problem Description  Assume that there are two processes running independently as Figure 38 illustrates. One process produces data for transfer to another process. After sending the data element with identifier id1 and value val1, a new data element is generated using the f() function. The other process analyzes the data received from the first process. Waiting for the notification of the data acceptance by the second process may cause the first process to block, thus postponing the preparation of data for subsequent data transfers.

Solution  In order to send data from one place to another, while avoiding blocking of the sender, use an asynchronous transfer. The asynchronous transfer is established through a placeholder which stores data arriving from the sender until the receiver picks it up.
Implementation of Solution  The list of instructions below describes how to implement the Asynchronous Transfer pattern (cf. Figure 39):

- To achieve the asynchronous transfer of data from one process to another, place Request is introduced between the Send and Receive transitions of the corresponding processes. The format of data in this place must incorporate data identifiers known both to the sender and to the receiver. The receiver will use the data identifier as a reference when consuming the data from the place.
- The Send transition puts data in the placeholder Request in the form of a tuple \((id, vl)\). The Receive transition uses the ID Matching pattern (cf. page 49) to extract the value of the data element from the placeholder.

Figure 40 demonstrates how to incorporate the Asynchronous Transfer pattern into the net presented in the problem description section. The first process sends data element to the second process via the Request place. After sending the data, the first process continues preparation of data for the next transfer, and the second process analyzes the data received.

Consequences  This pattern can be applied to establish a communication channel between multiple parties, when the sender of data does not require an instant acknowledgement of the data being received. This promotes the independence of communicating parties and avoids blocking by the sender.

The advantage of this pattern is that it allows the sender to function independently from the receiver. This pattern is similar to the Synchronous Transfer pattern (cf. page 56) and the Rendezvous pattern (cf. page 58), because they address similar kind of problems in the context of the data transport in distributed environments.
Examples

• An employee, who needs to send a letter, does not wait until a mail carrier arrives to pick it up, and puts the letter in a mailbox. The mail carrier will pick the letter up at a later time of their own choosing.

• Participants, who are online at different times, use Web message boards, newsgroups, or e-mail, to interact asynchronously.

Related Patterns This pattern uses the ID Matching pattern (cf. page 49). This pattern is similar to the Synchronous Transfer pattern (cf. page 56) and the Rendezvous pattern (cf. page 58).

Pattern 11 SYNCHRONOUS TRANSFER

Intent To allow the transfer of data from one location to another, ensuring that a party that posted a request is blocked until the requested information becomes available.

Motivation In a distributed environment, several processes may proceed independently of each other until one of them is interrupted during the transfer of data to another process. The data sender needs to get a rapid response on the data request sent, and may not proceed until this response is obtained. Such a request/respond communication strategy is used, for example, by people gathering at the same time for chatting or instant messaging.

Problem Description Assume that there are two processes running independently (Figure 41). Process 1 produces data for transferring to the process 2. Process 2 requires data produced by process 1 in its calculations. Process 1 will provide data only if the request for data transmission has been received from the other process.

Solution In order to transfer data from one location to another, ensuring that the sender remains blocked until the response from the receiver arrives, use a synchronous transfer. A synchronous transfer is established through two placeholders which temporarily store requests posted by one process and replies provided by another process as a response to the posted requests.

Implementation of Solution The list of instructions below describes how to implement the Synchronous Transfer pattern (cf. Figure 42):

• Two processes, one playing a role of an initiator of the synchronous transfer and another playing the role of the responder, are defined. In the initiating process, transitions Send and Receive are responsible for sending data to and receiving data from the responder process respectively.
Section 3.2 Catalog of CPN patterns

The Send transition has to be connected to the Receive transition through a waiting place Wait, in which the initiator of the data exchange will pause until the requested data becomes available.

Between the initiator and the responder processes two places Request and Response are introduced. The former place stores requests posted by the initiator process until the request is consumed by the responder process. The latter place stores data sent by the responder process to the initiator process once the response to the request is received.

The Ackn transition of the responder process is connected to the Send and Receive transitions of the initiator process via the placeholders Request and Response, indicating the direction of flow of data (requests) by direction of the arrows. The same behavior could also be obtained by fusing three transitions Send data, Send, and Receive into a single transition.

Note that since the initiating process may post multiple requests, the replies received have to be correlated with the requests sent. In order to identify which response corresponds to which of the requests sent, the requests have to be coupled to unique identifiers. These identifiers will be used by the initiator process to correlate requests stored in the Wait place and responses in the Response place using the ID Matching pattern (cf. page 49).

Figure 43 demonstrates how to incorporate the synchronous data transfer into the net presented in the problem description section. Note that in order to realize the desired behavior, Process 1 is extended with place Wait and transition Receive.

Consequences This pattern can be used to provide data produced by one process to another process, ensuring that the initiator of the data exchange is blocked until it receives a response from the other process.
This pattern is similar to the *Asynchronous Transfer* pattern (cf. page 54) and the *Rendezvous* pattern (cf. page 58), because they address similar kind of problems in the context of data transportation in a distributed environment. In comparison to the *Asynchronous Transfer* pattern (cf. page 54), the disadvantage of the *Synchronous Transfer* pattern is that the sender is blocked until the receiver finishes processing of the request received. In the same way as the *Rendezvous* pattern (cf. page 58), the *Synchronous Transfer* pattern needs to synchronize the sender and the receiver. However, in this pattern the synchronization is done sequentially, while in the *Rendezvous* pattern it is done concurrently.

**Examples**
- Any kinds of interactions, where an instant response from participants is expected.
- Subroutine calls from a program on one machine to the library routines on another machine.

**Related Patterns** This pattern is similar to the *Asynchronous Transfer* pattern (cf. page 54) and the *Rendezvous* pattern (cf. page 58). It can be combined with the *ID Matching* pattern (cf. page 49).

**Pattern 12 RENDEZVOUS**

**Intent** To allow multiple processes to synchronously exchange data in both directions.

**Motivation** In some situations, it is necessary to model a channel, which only transfers data messages but does not store them, allowing sending and receiving of the messages at the same time and in both directions.

**Problem Description** Assume that there are two processes running independently as shown in Figure 44. Process 1 produces data for transferring to process 2 based on the data it has provided. Process 2 correspondingly processes data received from process 1 and sends the results of the processing back to it. In order to avoid unnecessary waiting, both processes need to be able to exchange, i.e. send and receive, data concurrently.

**Solution** In order to allow for synchronous data exchange between multiple processes in both directions, use a rendezvous. A rendezvous enables concurrent exchange of data between multiple processes by connecting senders and receivers to a single transition, which reveals data available for sending and broadcasts it to the corresponding recipient(s).

**Implementation of Solution** The list of instructions below describes how to implement the *Rendezvous* pattern (cf. Figure 45):
- For each of the processes, participating in a concurrent data exchange, an input place and an output place are defined. The input places In1 and In2 provide data that needs to be transferred. The output places Out1 and Out2 store the data received.
The input and output places have to be connected to the **Exchange** transition, which will consume data from the input places and put data in the output places simultaneously. Note that amount of data swapped between the processes could vary.

Unlike the **Synchronous Transfer** pattern where the exchange of information can be performed as a series of requests and replies, this pattern concentrates on the *concurrent* exchange of information at a particular place and time.

Figure 46 illustrates how to implement concurrent data exchange in the example described earlier. Note that transitions **Send data** and **Process data** are fused into one transition **Exchange**, which synchronizes concurrent communication between process 1 and process 2.

**Consequences** The **Rendezvous** pattern facilitates synchronous exchange of data between two or more actors in both directions. This pattern is similar to the **Synchronous Transfer** pattern (cf. page 56) and the **Asynchronous Transfer** pattern (cf. page 54), because they address similar kinds of problems in the context of data transportation in a distributed environment. In contrast to the **Synchronous Transfer** pattern, in this pattern the data producers and consumers are tightly coupled and must execute simultaneously for data delivery to occur.

The **Rendezvous** pattern can be applied for broadcasting data to multiple recipients. The disadvantage of this pattern is in the tight coupling it requires between senders and receivers. If the tight dependency between senders and receivers is not desirable during data exchange, then the **Broadcasting** pattern (cf. page 63) can be applied instead. The drawback of using the **Broadcasting** pattern is that it is based on the **Asynchronous Transfer** pattern, which does not ensure that target recipients will obtain data simultaneously.
Example

- Two insurance companies agree to exchange data about the claims received from clients involved in the other company on a daily basis.
- A business event where various organizations meet to exchange their details for the purpose of future collaboration.

Related Patterns This pattern is similar to the Synchronous Transfer pattern (cf. page 56), the Asynchronous Transfer pattern (cf. page 54) and the Broadcasting pattern (cf. page 63).

The fifth pattern group consists of three patterns (Asynchronous Router, Broadcasting, and Distributed Data Processing), which address problems related to data exchange during one-to-many and many-to-one interactions. The Asynchronous Router pattern describes situations where a source sends data to a number of targets on an asynchronous basis. The data sent by a source is dedicated only to one target, and the source is not blocked after performing the data transfer. The Broadcasting pattern describes situations where the same data needs to be transferred to several targets simultaneously. The Distributed Data Processing pattern describes situations where after decomposing a data element into smaller parts and distributing these amongst several targets for concurrent processing, there is a need to gather the results of processing the distributed data parts and aggregate them into a single entity for use elsewhere.

Identity management plays an important role in the patterns considered. The Asynchronous Router pattern utilizes target identifiers in order to check whether a data element sent is delivered to the correct recipient. For the Broadcasting pattern target identifiers may need to be utilized to prevent the same data object being consumed by a recipient multiple times. The Distributed Data Processing pattern may utilize data identifiers in order to match up the results of processing objects which were distributed earlier for concurrent processing.

The matching of object and target identifiers in these patterns is closely related to the issue of message correlation discussed in a detail in Chapter 5 on page 256.

Pattern 13 ASYNCHRONOUS ROUTER

Intent To enable asynchronous transfer of data from a single source to a dedicated target, ensuring that old targets can be easily removed and new targets can be added without affecting the source.

Motivation In some situations, there is a need to transfer data from a source to any of several available targets asynchronously. This can be achieved by applying the Asynchronous Transfer pattern (cf. page 54), which allows a source and target to work independently, however it requires the parties involved in the interaction to know each other’s identities. To deliver data directly to a target, the source requires knowledge about all of the target recipients. Due to the tight coupling between the source and target(s), changes in any of the target(s) may directly affect the source, thus providing minimal flexibility when manipulating any of the targets.
**Problem Description**  Figure 47 presents the problem of directly addressing data messages between a data source and targets. A data element \texttt{mes} of type \texttt{Message} needs to be sent by transition \texttt{Source} to a specific target, e.g., \texttt{target1}, \texttt{target2} or \texttt{target3}. In order to specify to which target a message is being sent, the source specifies a target identifier (of type \texttt{TargetID}). However, the targets have no knowledge about how the source selects a target and what information it uses for this purpose.

![Figure 47: Problem of direct addressing](image)

Changes on the target side such as the addition of a new target, no information about the address of which is available, are not possible in this diagram. In addition, changes in any of the targets directly affect the source, and may even influence the connection between the source and the rest of the targets.

**Solution 1** In order to decouple a data source from a set of targets, which communicate asynchronously, while ensuring that data sent by the source is received by the target to which it was directed, introduce an \textit{asynchronous router}. An asynchronous router will direct all data received from the source to a place, where it will be stored until the intended target picks it up at a time of its choosing.

**Implementation of Solution 1** The list of instructions below describes how to implement Solution 1 of the \textit{Asynchronous Router} pattern (cf. Figure 48(a)):

- The source place from which data is to be distributed to the targets must be of type \texttt{MesxTargetID}, where \texttt{Mes} is a message to be sent and \texttt{TargetID} is an identifier of the target.
- The Route transition takes data from the source place and puts it in the temp place which serves as temporary storage for all data sent by the source.
- Targets represented by transitions \texttt{target 1}, \texttt{target 2} and \texttt{target 3} are connected to the temp place, ensuring that only data whose identifier corresponds to the target identifier is consumed by the target. For this, the \textit{ID Matching} pattern (cf. page 49) has to be applied (a guard comparing the target identifier with the identifier of the data stored in place temp must be added for every target-transition).

Note that data from source to temp is sent asynchronously, as the \textit{Asynchronous Transfer} pattern (cf. page 54) describes. Since targets identifiers are all different, no two targets can consume the same data (as the \textit{Deterministic XOR-Split} pattern is used (cf. page 36)) at once.

Note that data, which is routed from the source to temp, is consumed by a target transition in a non-deterministic order. In order to ensure that targets consume data...
in the order of arrival, the data in place temp can be aggregated into a collection by applying the Aggregate Objects pattern (cf. page 67), while data manipulation in some other ordering sequence can be enforced by applying the Queue pattern (cf. page 75).

Solution 2  In order to decouple a data source from a set of targets, with which it communicates asynchronously, while ensuring that data sent by the source is received by the target to which it was directed, introduce an asynchronous router. The asynchronous router will direct data directly to a specific target.

**Implementation of Solution 2**  The list of instructions below describes how to implement Solution 2 of the Asynchronous Router pattern (cf. Figure 48(b)).

- The Route transition is introduced, to which the source place is connected, providing the composite data element \((mes,t)\) which includes an information object to be sent to the target and the identifier of the target.
- For each of the targets an input place into which the router will place a dedicated data element is introduced. As such, for targets 1, 2 and 3 places \(b\), \(c\), and \(d\) are introduced respectively.
- Every outgoing arc of the Route transition is associated with a filtering condition that determines whether a data element should be routed down this arc. For this, the ID Matching pattern (cf. page 49) is used. For instance, to examine whether a data element supplied by the router is intended for the target1 transition, a filtering condition \(\text{if } t=t_1 \text{ then } 1\text{'mes} \text{ else empty}\) is added. The resulting construct incorporates Solution 2 of the Deterministic XOR-Split pattern (cf. page 36).

**Consequences**  This pattern can be applied to avoid a direct dependency between a source and targets which are communicating asynchronously. By adding a new target or removing an existing one, the source will not be affected.

This pattern allows decoupling of source and targets in two different ways. Solution 1
places all data routed from the source in temporary data storage. From the target’s point of view, if the delivery of data was organized and initiated by the source, then after applying this solution, targets should become more active and take the initiative for obtaining the data at a time of their choosing.

Solution 2 routes data from the source to an input place of a target, without involving the target in the procedure of selecting a dedicated data. The drawback of this solution is that the router is impacted each time a new target is added.

This pattern includes the Asynchronous Transfer pattern (cf. page 54) to ensure that data is sent asynchronously, and the Deterministic XOR-Split pattern (cf. page 36), which guarantees that every data element will be consumed by one-and-only-one intended target.

The major characteristic of this pattern is that a source, producing data for multiple targets, sends it asynchronously to a single dedicated target. When it is necessary to broadcast data from a source to a set of targets, so that every target receives the same data, the Broadcasting pattern (cf. page 63) should be applied.

Since this pattern is based on asynchronous communication, the data source does not know whether a target received the data element that was sent to it. If for example the connection was broken, but the target did receive the data element, the source may try to retransmit the same data element again. Thus this pattern does not guarantee that duplicated data is not transferred from the source to a target. In order to address this problem, this pattern can be combined with the BSD Filter pattern (cf. page 43).

Examples

- The secretary of a department is responsible for distribution of holiday cards. Instead of delivering a card directly to every employee of the department, the secretary puts the cards in the employee’s post-boxes. Employees pick up their cards at a time of their choosing.
- The canteen provides organized meetings with coffee and tea assuming meeting participants will serve themselves.

Related Patterns This pattern includes the Deterministic XOR-Split pattern (cf. page 36), the Asynchronous Transfer pattern (cf. page 54) and the ID Matching pattern (cf. page 49). This pattern is similar to the Broadcasting pattern (cf. page 63), and can be combined with the Blocking State-Dependent Filter pattern (cf. page 43).

Pattern 14  BROADCASTING

Intent To broadcast data from a single source to multiple targets, while avoiding creating a direct dependency between them.

Motivation In some situations, there is a need to transfer data from a single source to a set of targets, so that all targets receive the same information. This can be done using the Asynchronous Router pattern (cf. 60), however the data exchange achieved this way requires tight coupling between the source and the targets. Such data transfer is performed asynchronously, whereas the concurrent distribution would be more efficient. Moreover, direct addressing from a source to targets can become cumbersome, when the number of targets and other target-related information is not known in advance or may change over time.

Problem Description Figure 49 presents the problem of direct addressing of data messages between a source and multiple targets. In order to broadcast the same data mes to several targets, the source transition needs to be connected to each of the targets, thus
providing data directly to each of them. If the number of target recipients varies, the source will be directly affected. In terms of programming, this necessitates recompilation of the source each time a new component is added or removed. Thus, tight coupling between a source and targets minimizes flexibility when broadcasting to a series of targets.

Figure 49: The problem of direct data addressing

Solution 1 In order to loosen the connection between a source and targets, ensuring that the same data is received by each of the targets, decouple the source from the targets by introducing an intermediate placeholder. The source will provide data to the placeholder, and targets will take the data from this place at their own initiative.

Implementation of Solution 1 The list of instructions below describes how to implement Solution 1 of the Broadcasting pattern (cf. Figure 50 (a)):

- Places directly connecting the source transition with targets are merged in one place **Router**. In this place objects are stored as separate tokens, but it is possible to aggregate all data into a collection by applying the Aggregate Objects pattern (cf. page 67).
- The merged place **Router** is connected to the targets by means of bidirectional arcs. An arc from the **Router** place to a target provides data **mes** to a target, while the arc in the opposite direction returns the data back to the **Router** place in order to allow other targets to utilize this information.

Solution 2 In order to loosen the connection between a source and targets, ensuring that the same data is received by each of the targets, decouple the source from the targets by means of broadcasting. The data provided by the source for broadcasting will be distributed to all targets simultaneously.

Implementation of Solution 2 The list of instructions below describes how to implement Solution 2 of the Broadcasting pattern (cf. Figure 50(b)):

- The output place a of the source transition, where the data which needs to be broadcasted locates, and places b, c and d providing data to the targets are connected to the Broadcast transition. The Broadcast transition consumes data from the a place and distributes it simultaneously to all its outgoing places.
- In order for targets to receive the data from the source, the type of places a, b, c and d should be the same (i.e. **Message** type).

Consequences This pattern can be used to broadcast data from a single source to multiple targets, ensuring that every target receives the same data. The Broadcasting
pattern decouples a source from the targets, ensuring that every target receives the same data. The advantage of this pattern is that the source does not need to consider the delivery of data to targets, but only to the router. Targets themselves take care of consuming data from the router.

In comparison to Solution 2, Solution 1 of this pattern shifts the responsibility for obtaining the broadcasted data to targets. It is however not guaranteed that every target will consume data elements only once because the broadcasted data accumulates in the intermediate placeholder. In order to solve this problem, this pattern needs to be combined with the Region Flush and Blocking State-Dependent Filter patterns. The Region Flush (cf. page 86) will remove the content which has been broadcasted, and the BSD Filter (cf. page 43) will ensure that the content is removed only after each of the targets has consumed the data exactly once.

Examples
- Broadcasting of TV programs to all receivers connected to the network.
- After each update, a configuration manager puts a new version of the tool in the shared directory, where it can be accessed by all employees.

Related Patterns This pattern can be combined with the Aggregate Objects pattern (cf. page 67), the Region Flush pattern (cf. page 86), and the BSD Filter (cf. page 43).

Pattern 15 DISTRIBUTED DATA PROCESSING

Intent To decompose a data element in smaller parts in order for them to be processed in parallel, subsequently merging the processed data later on.

Motivation In most business processes and information systems, where continuously growing amounts of data require processing, there is a need to improve the processing efficiency by introducing parallelism, in relation to both flow and structure of information. Depending on the processing context, the nature and complexity of data, it might be...
necessary to involve several either identical or specialized entities that can process data concurrently, rather then letting a single entity doing all of the work sequentially.

**Problem Description** Consider a complex data structure such as a compound request received from a client for registering a bank account and a set of insurance policies. Processing of such a request can be done more efficiently if several employees, each responsible for a constituent part of the request, are involved. To enable efficient processing of the request, it needs to be decomposed into independent parts and passed for processing to corresponding parties respectively. Figure 51 illustrates a problem of sequential data processing. A composite data element \((a,b,c)\) is processed in several stages, thus resulting in longer waiting time than when processed concurrently.

![Figure 51: Problem of sequential data processing](image)

**Solution** In order to scale the throughout of data processing, use *distributed data processing*. The compound data unit is divided into smaller parts, which are distributed between several concurrent processing streams, either specializing or performing the same set of operations. After each of the streams completed the processing of the data elements, the results of processing are merged back into a single entity.

Depending on the data structure, the complexity of data, and the context of data processing, the data distributor either spreads the input data equally or divides it among several different locations for independent processing.

**Implementation of Solution** The list of instructions below describes how to implement the *Distributed Data Processing* pattern (cf. Figure 52):

![Figure 52: Implementation of the Distributed Data Processing pattern](image)

- A transition *Distribute* is introduced to distribute data received from the source place \(\text{In}\) between a set of the output places \(\text{p1, p2, and p3}\).
- The distribution rules, specifying what data element is to be provided to each of the output places of the *Distribute* transition, have to be defined and encoded in functions *partA()*, *partB()*, and *partC()* respectively.
- The results of processing by each of the parallel streams are provided to the \(\text{p4, p5, and p6}\) places. The data elements from each of these places are merged by means of the *Merge* transition in to a single entity. Note that the types of places \(\text{p4, p5, and p6}\) can be adjusted in order to allow multiple tokens to be consumed from each of the places. One could also consider using the *Aggregate Objects* pattern (cf. page 67) in order to address all data elements accumulated in each of these places as a single entity.
Consequences This pattern can be applied to support parallel processing of data. It can be used to decompose a single compound request into several simpler requests. Typically, after processing data, it is necessary to combine the results of processing back into a single data unit. This pattern assumes that all threads processing the data elements will also provide the results of processing for subsequent merge. Note that in order to ensure that after distributing data the corresponding results of data processing are merged correctly, each of the distributed parts may need to associated with an identifier (as the ID Matching pattern suggests). The usage of a unique identifier may help to perform matching and subsequent merging of data elements correctly.

The distribution of data has the same structure as the Asynchronous Router pattern (cf. page 60) and the Broadcasting (cf. page 63) patterns. When distributed or broadcasted data need to be merged, these patterns can be combined with the Distributed Data Processing pattern.

Examples
- To distribute the load between several processors a complex process is divided into several threads which are distributed over available processors.
- Distribution of the papers within a group of researchers in order to speed up the reviewing process.

Related Patterns This pattern is similar to the Broadcasting pattern (cf. page 63) and the Asynchronous Router pattern (cf. page 60). It can be combined with the ID Matching pattern (cf. page 49).

The sixth pattern group consists of four patterns: Aggregate Objects, Deaggregate Objects, Capacity Bounding and Containment Testing. The Aggregate Objects and Deaggregate Objects patterns allow objects to be organized into a collection or for the retrieval of a single object from a collection in order for it to be processed as a single entity respectively. As a consequence of having multiple objects available at one place, it may be necessary to limit the capacity of the object collection. This problem is addressed by the Capacity Bounding pattern. The Containment Testing pattern addresses the problem of testing for the absence or presence of particular objects in the collection.

Pattern 16 AGGREGATE OBJECTS

Intent To allow a set of information objects to be manipulated as a single entity.

Motivation In many cases, it is natural to represent an information object (e.g., an order, a car, a message) as a single entity, i.e. there is a one-to-one correspondence between objects in a “real system” and tokens in the model. Sometimes, it is necessary to aggregate objects into one token to enable a collection of objects to be referenced as a single entity.

Problem Description Figure 53 illustrates the problem addressed by this pattern. In the original model, the objects place is of type T and transitions put and get add and remove tokens from this place. Note that each token corresponds to an object.

Suppose that it is necessary to perform an operation from the following list:
- Count the number of objects in the objects place;
- Select an object from the objects place with some property relative to the other objects (e.g., the first, the last, the smallest, the largest, the cheapest, etc.);
Chapter 3 Colored Petri Nets Patterns

Figure 53: Problem of accessing multiple/specific objects

- Modify all objects in a single action (e.g., increase the price by 10 percent);
- (Re-) move all objects in one batch (e.g., remove a set of outdated files, items, etc. at once, rather than one by one).

None of these operations can be directly applied in the diagram shown above. Note that it is only possible to inspect one token at a time and this is a non-deterministic choice. Moreover, this choice can be limited by transition guards and arc inscriptions, but it is memory-less and not relative to the other tokens in the place. This makes it very difficult or even impossible to realize operations mentioned above.

Solution In order to allow a set of information objects to be manipulated as a single entity, aggregate objects into a single token of “collection type”.

Implementation of Solution The list of instructions below describes how to implement the Aggregate Objects pattern (cf. Figure 54).

Figure 54: Implementation of the Aggregate Objects pattern

- Type T of place objects, where multiple objects may reside, is replaced with the collection type LT (e.g., list, set, bag). In this example, the collection type list is chosen: color LT = list T.
- Arcs between transition put and place objects are replaced by a bi-directional arc with the following inscriptions. An inscription of the arc supplying an object to the collection contains a function \( \text{add}(x, l) \), which adds an object \( x \) of type \( T \) to the list \( l \).

By referring to the list \( l \), all objects contained in it can be addressed simultaneously.

Figure 55 illustrates application of the Aggregate Objects pattern to a model presented in the problem description section. In this implementation, the \( \text{add}(x, l) \) function is represented as \( x::l \) in order to achieve the last-in-first-out behavior (cf. also the LIFO Queue pattern on page 79).

Figure 55: Model after applying the Aggregate Objects pattern

By introducing a collection type, it becomes possible to refer to the collection of objects as a single entity and perform operations on multiple objects contained in the collection at once. In order to check the number of objects in the collection or test the availability of objects satisfying a particular property, an extension of this pattern, i.e. the Containment Testing pattern (cf. page 73) can be used. An example in Figure 56 shows how to calculate the number of objects in the collection. Note that there is always precisely one token in place objects representing all objects. The \( \text{count} \) transition takes the current list of
objects and sends the size of the list to the **Number of objects** place. Note that the `size(l)` function for determining the size of the collection is predefined in CPN Tools.

**Figure 56:** Example of defining the size of the collection

In a similar way, it is possible to modify all objects in a single action (for instance, increase the price by ten percent) and to remove all tokens (simply by returning a token with a value `[]`).

**Consequences** This pattern can be applied to organize multiple objects into a collection and/or perform an operation on a group of objects or the whole collection at once.

In principle, this pattern is not concerned with the order in which tokens are taken from the collection. The example used in the implementation section uses last-in-first-out ordering (cf. the **LIFO Queue** pattern on page 79). Nevertheless, if the problem of order is relevant, one should apply an extension of this pattern by adding the **Queue** pattern (cf. page 75), or one of its specializations.

This pattern can be combined with the **Deaggregate Object** pattern (cf. page 69), whose intent is to extract an object from the collection in order to use it elsewhere. Furthermore, the **Containment Testing** pattern (cf. page 73), which extends this pattern, can be used to test the absence/presence of tokens satisfying a certain property in a place.

**Examples**
- The salary administration of a university divides employees into different groups: students, PhD students, and professors. All PhD students get a salary increase of 10%. The salary administration does not need to adjust the salary slips for every PhD student individually, but does it in one step by increasing the salaries of the whole group.
- The documents are collected and organized in one file. This allows the whole file to be taken and sent for processing elsewhere.

**Related Patterns** This pattern is extended by the **Queue** pattern (cf. page 75) and the **Containment Testing** pattern (cf. page 73). It can be combined with the **Deaggregate Object** pattern (cf. page 69).

**Pattern 17** DEAGGREGATE OBJECT

**Intent** To allow the manipulation of an object aggregated into a collection as an independent entity.

**Motivation** The **Aggregate Objects** pattern (cf. 67) describes how to aggregate objects into a collection in order to manipulate them as a single entity. In some situations, objects stored in the collection need to be addressed separately. Such objects can be retrieved from the collection in order to be updated, destroyed or used elsewhere in the process without affecting other objects in the collection.

**Problem Description** After individual objects have been placed in a collection `objects` (cf. Figure 57), they can not be referenced as a single entity. Suppose that it is necessary to perform an operation on a single object from a given collection `l` stored in place `objects`. Such an operation can not be performed without addressing the whole collection.
**Solution** In order to allow the manipulation of an object contained in a collection of objects as a single entity, the object has to be deaggregated. After a required object has been retrieved from the collection, it can be manipulated as a single entity.

**Implementation of Solution** The list of instructions below describes how to implement the *Deaggregate Objects* pattern (cf. Figure 58).

- In order to deaggregate objects from the collection of objects stored in the `objects` place, the `unpack` transition is introduced which consumes the list of objects, extracts all objects from it and places them in the `out` place and returns an empty collection to the `objects` place.
- In order for a object specifying a specific property to be retrieved from the collection, this pattern can be combined with the *Blocking State-Dependent Filter* pattern (cf. page 43).

**Consequences** This pattern can be applied to retrieve specific objects from the collection in order to use them elsewhere in the process as a separate entity. This pattern is similar the *Containment Testing* pattern (cf. 73) where the collection of objects is examined in order to identify whether an object satisfying is (not) contained in the collection. However, the main purpose of that pattern is to test for presence in the collection rather than to deaggregate a single object from it. Finally, the *Deaggregate Object* and the *Containment Testing* pattern (cf. page 73) can be combined in order for an object with a certain property to be extracted from the collection.

The *Aggregate Objects* pattern (cf. page 67) can be used in combination with the *Deaggregate Objects* pattern in order to model a temporary storage of objects in a collection that is updated when old objects are retrieved and new objects are added. A collection of objects achieved in this way is not ordered. To enforce objects in the collection retain a strictly specified order, objects have to be sorted by applying one of the variants of the *Queue* pattern (cf. page 75).

**Examples**
- A collective holiday is organized for all employees of an organization at the same time. All reservations are made in the name of the organization. Because the duration of stay for many employees varies, the return ticket needs to be booked for each of the employees separately.
- For each member of an alumni club, once a year the statement about membership payment status needs to be sent. The secretary of the club has to send the payment details provided by the board to each of the members.
Related Patterns This pattern can be used in combination with the Aggregate Objects pattern (cf. page 67) and the Containment Testing pattern (cf. page 73).

Pattern 18  CAPACITY BOUNDING  

Intent To prevent over-accumulation of objects in a certain place.

Motivation Places in CPNs are potentially unbounded and may accumulate an unlimited number of tokens. In some situations, for instance for modeling of a network buffer, it is necessary to limit the number of tokens which a certain place is allowed to contain.

Problem Description In the net presented in Figure 59 different resources stored in the Objects place are accessed by transitions Put and Get. Assume that it is necessary to prevent over-accumulation of the resources due to the limited size of the storage (let the storage size be N). In such a net it is not possible to limit the capacity of the Objects place since by definition places in CPNs are unbounded.

Solution 1 In order to prevent over-accumulation of information objects in a certain place, use an anti-place. An anti-place is a place corresponding to the original place, which in combination with the original place and its incoming/outgoing transitions forms a feedback construct.

We talk about the feedback between two transitions Put and Get if they depend on each other in such a way that Get both consumes an output token from Put and produces an input token for Put (cf. Figure 60). The Anti-place stores N tokens (()), where the number N defines the maximal capacity of the Place. Tokens stored in the Anti-place are of type Unit, a colorset with only one element (()).

Implementation of Solution 1 The list of instructions below describes how to implement Solution 1 of the Capacity Bounding pattern (cf. Figure 61).

- A new place (Anti-place) of type Unit is added to the net. Note that a multi-set type can be used instead, which allows for multiple instances of tokens of the same type.
- The initial marking of the Anti-place is set to N' (()), where N is the capacity bound of the original place Objects.
- Outgoing and incoming transitions of the Objects place are connected to Anti-place, such that incoming transition of the Objects place corresponds to the outgoing transition of the Anti-place and vice versa.
Chapter 3 Colored Petri Nets Patterns

Solution 2  In order to prevent over-accumulation of objects in a given place, use a *counter-place*, which will count the number of objects present in the place. The *Blocking State-Dependent Filter* pattern can be used to examine the state of the counter and prevent new objects from being added if the counter has reached the maximal place capacity.

**Implementation of Solution 2** The list of instructions below describes how to implement Solution 2 of the *Capacity Bounding* pattern (cf. Figure 62).

- A new place *Counter* of the type *INT* is added to count how many objects are accumulated in the *Objects* place.
- The counter is connected to transitions *Put* and *Get*, which add and remove objects from the *Objects* place. When a new object is added to the *Objects* place, the counter is incremented. When an object is removed from the *Objects* place, the counter is decremented.
- A value *N* is declared to represent the maximal capacity of the *Objects* place.
- A guard \([n<N]\) is associated with a transition *Put* which compares the status of the counter, i.e. how many objects are contained in place *Objects*, with the maximal capacity of this place. If there is some free capacity available, then a new object can be added. Otherwise, transition *Put* remains disabled until some of the objects have been removed.

Note that this solution incorporates the *Blocking State-Dependent Filter* pattern (cf. page 43), which examines the state of the counter, and based on the value of the counter, determines whether it is possible to add a new object.

Solution 3  In order to prevent over-accumulation of information objects in a given place, aggregate objects into a collection using the *Aggregate Objects* pattern. The current size of the collection is examined using the *Blocking State-Dependent Filter* pattern (cf. page 43) to prevent new objects from being added to the collection if the collection size has reached the maximal place capacity.

**Implementation of Solution 3** The list of instructions below describes how to implement Solution 3 of the *Capacity Bounding* pattern (cf. Figure 63).
Figure 63: Implementation of Solution 3 of the Capacity-Bounding pattern

- Type T of place Objects is changed to the collection type LT=list T.
- One of the variants of the Queue pattern (cf. page 75) is applied to specify in which order objects must be retrieved from the collection. In this example, the LIFO Queue pattern (cf. page 79) is applied.
- A value N is declared to represent the maximal capacity bound of the Objects place. In this particular example, N=2 which means that at most two tokens can be present in the Objects place at once.
- Transition Put, which evaluates the size of the collection, i.e. how many objects it contains, and compares it with the maximal capacity of this place, is associated with a guard \([\text{size}(l)<N]\). If some free capacity is available, then a new object can be added. Otherwise, transition Put remains disabled until some of the objects have been removed.

Note that this solution incorporates the Blocking State-Dependent Filter pattern (cf. page 43), which examines the state of the counter and prevents from new objects being added if the capacity of the place has been reached. The Deaggregate Objects pattern (cf. page 69) is used in this solution for retrieving objects from the queue.

Consequences This pattern provides a means for bounding the capacity of places. However, it is not applicable for places where an upper bound cannot be defined or does not exist.

All three solutions presented in this pattern require knowledge about the maximum number of objects a place can hold. In contrast to solutions 1 and 2, where every object is represented by a separate token, Solution 3 aggregates all objects into one collection and provides extra flexibility by allowing operations to be performed on a group of objects at once. Although the second solution has a more complex realization than other solutions, it allows the exact value of the current place capacity to be monitored and used elsewhere in the model.

Examples
- The number of applications that can be processed concurrently in the city hall is limited by the number of officers available.
- Requests for residence permit extension are handled by a group of four immigration officers. Whilst all officers are occupied no new requests can be registered.

Related Patterns This pattern uses the Blocking State-Dependent Filter pattern (cf. page 43); Solution 3 also uses the Aggregate Objects and Deaggregate Object pattern (cf. page 67 and page 69).

Pattern 19  CONTAINMENT TESTING

Intent To allow the (non)-availability of objects with particular properties in a given location to be tested.

Motivation In CPNs it is easy to test the presence of a particular object in a place by linking this place to a transition, which fires and consumes the object satisfying a certain
property from the place. In some situations, it may be necessary to test the number of tokens or the color of tokens present in a place. Potentially, it might also be necessary to have a transition which is enabled if a place does not contain a token satisfying a certain property. For instance, a corrective action has to be taken when incorrect data is encountered in a database or when a bottleneck in a process is identified. However, in CPN it is not possible to test for the absence of tokens in a certain place.

**Problem Description** Consider the situation presented in Figure 64. Objects of type T are placed in and taken from the Object place. Now suppose that we want to check whether the Object place does not hold a token with a specific value. In such a net this is not possible, because CPNs only allows testing for the presence of a token in a place by linking it to a transition and subsequently firing it consuming a token from this place when doing so.

![Figure 64: Example illustrating the problem of non-containment property of place](image)

**Solution** In order to check whether objects satisfying a certain property are (not) contained in a place with unbounded capacity, aggregate the objects into a collection as the Aggregate Objects pattern suggests, and connect this place to an external transition will test the status of the collection via the test arc. A test arc is a bidirectional arc that consumes the contents of a place and returns the contents unmodified back to the place.

**Implementation of Solution** The list of instructions below describes how to implement the Containment Testing pattern (cf. Figure 65).

![Figure 65: Implementation of the Containment Testing pattern](image)

- Objects of type T are aggregated into a list \(LT=\text{list } T\).
- The current list of objects \(l\) has to be examined by the \(\text{elt()}\) function\(^3\) enclosed in the guard of transition \(\text{Test}\). This function identifies if \(y\) is an element of the list \(l\). Note that any other function, performing the analysis of elements stored on the collection could be used instead.

\[
\begin{align*}
\text{fun elt}(y,[]) &= \text{false} \\
\text{elt}(y,x::l) &= \text{if } x=y \text{ then true else elt}(y,l);
\end{align*}
\]

\(^3\)Note that CPN Tools has a built-in function \(\text{mem}\) that can be used for selecting an element from a list.
Note that the capacity of the `Objects` place can be bounded if needed by applying the `Capacity Bounding` pattern (cf. page 71).

**Consequences**  This pattern can be applied to test the absence of objects satisfying a certain property in a place. If objects that do not satisfy the conditions specified have been identified in a place, it might be necessary to remove them. To do this, this pattern can be combined with variants of the `NBSI Filter` or `NBSD Filter` patterns (cf. pages 45 and 47 respectively).

**Examples**
- A medical assistant makes appointments with patients over the telephone. As long as emergency-patients are on the phone, the assistant continues handling their requests. When no emergency-patients are left on the phone, the assistant switches to the patient waiting in the queue at the reception counter.
- Families with children and disabled people are selected from the passenger’s queue as the first ones to board the plane.

**Related Patterns**  This pattern uses the `Aggregate Objects` pattern (cf. page 67). It can be combined with the `NBSI Filter` or `NBSD Filter` patterns (cf. pages 45 and 47).

---

The seventh pattern group consists of six patterns. The `Queue` pattern, the `FIFO Queue` pattern, the `LIFO Queue` pattern, the `Random Queue` pattern and the `Priority Queue` pattern address the problems of coordinating the order in which data elements aggregated into a collection are utilized. The `Queue` pattern describes a generic approach to enforcing that objects be added to a collection and removed from it in a particular order. Different approaches to queue management, e.g., sorting based on the order of arrival, priority associated with objects or random selection are discussed in the `FIFO Queue`, `LIFO Queue`, `Random Queue` and `Priority Queue` patterns. The situation where the availability of a particular data element defines the order in which tasks are executed is described by the `Prioritized Execution` pattern.

---

**Pattern 20  QUEUE**

**Intent**  To allow the manipulation of queued objects in a strictly specified order.

**Motivation**  In many systems, there are buffers where a variable number of objects are required to queue in between two steps in a process. This pattern assumes an unbounded queue. The objects need to be placed in a queue and retrieved according to a specific queuing policy. By exchanging one policy for another, it is possible to obtain the desired ordering of the buffered objects.

**Problem Description**  Assume that a collection of objects in the form of a queue is given, and that it is necessary either to add an object to the collection, or to select an object and remove it from the queue. The order in which objects are being added to or removed from the collection may depend on the properties of an individual object (e.g., age, weight, etc.), the location in the queue (FIFO, LIFO), or a time-stamp (cf. Figure 53).

**Solution 1**  In order to enforce that elements of a queue move in a strictly specified order, and enable that several queued elements to be moved in one go, use the `Aggregate Objects` pattern in combination with the `Deaggregate Objects` pattern. The objects are added to a list and removed from it based on a predefined ordering algorithm.
Implementation of Solution 1 The list of instructions below describes how to implement Solution 1 of the Queue pattern (cf. Figure 66).

- Objects of type $T$ to be queued have to be aggregated in a collection of a list type $LT$, such that color $LT = list T$. The $add()$ function adds objects in a specified order. The $select()$ function retrieves a required element from the queue, and the $rest$ function puts an updated queue back in the Queue place. Objects can be added to the queue and removed from it based on a first-in-first-out, last-in-first-out or object priority basis as described in patterns FIFO Queue, LIFO Queue and Priority Queue (cf. pages 78, 79, and 81) respectively.

Solution 2 In order to enforce that elements of a queue move in a strictly specified order, such that each of the queued objects is distinguishable as a separate entity, objects have to be augmented with a number indicating their position in the queue. Furthermore, for each queue it is also necessary to keep track of the objects at the head and tail of the queue in order to facilitate the addition and removal of objects.

Implementation of Solution 2 The list of instructions below describes how to implement Solution 2 of the Queue pattern (cf. Figure 67).

- Two variables identifying at the queue bounds (i.e. head and tail) are introduced. These variables need to be updated when the first element is inserted to the queue and the last element is retrieved from it respectively. The upper bound is incremented when a new element is added to the queue. The lower bound is incremented when an element is removed from the queue. By using a lower/upper bound one can directly refer to the first/last element in the queue respectively.
- In Figure 67 every object $x$ is coupled to a number. Variables $a$ and $b$ keep track of elements in the queue, i.e. variable $a$ stores the position of the first element in the queue, while variable $b$ points to the last added element. When a new element is added to the queue, the value of $b$ is increased by 1, thus the pointer is moved to the last element. When an element needs to be removed from the queue, the element with
position \(a\) (the first element of the queue) is supplied to transition \texttt{get}. After the object is retrieved, variable \(a\) is incremented in order to point to the (new) beginning of the queue. Note that although this applies to the \textit{FIFO} queue (cf. page 78), for the current pattern any ordering can be applied.

**Solution 3** In order to enforce that elements of a queue move in a strictly specified order, such that every queued element, (which consists of an object and an object identifier), is distinguished separately, a sorted collection of object identifiers needs to be introduced. The \textit{ID Matching} pattern helps in defining the queue element to be retrieved next based on the matching of object identifier with the identifiers stored in the introduced collection.

**Implementation of Solution 3** The list of instructions below describes how to implement Solution 3 of the \textit{Queue} pattern (cf. Figure 68).

![Figure 68: Implementation of Solution 3 of the Queue pattern](image)

- A place \texttt{Identifiers queue} is created in order to store the collection of object identifiers. To maintain the ordering of objects by means of identifiers, the identifiers have to be sorted according to a desired policy (i.e. FIFO, LIFO, random, etc.).
- When adding a new element to the queue \texttt{Queued elements}, the identifier of the object is added to the list of identifiers. Note that in Figure 68, the identifiers are added to the tail of the list in order to achieve a first-in-first-out behavior.
- To retrieve an element from the queue, the \textit{ID Matching} pattern has to be applied. An element from the queue, whose identifier matches the identifier extracted from the head of the list of identifiers, will be retrieved.

**Consequences** The \textit{Queue} pattern presents three solutions, each of which can be applied for managing the order of objects as they are inserted in or removed from a certain place. Although the first solution is more flexible, it hides the behavior inside each of the functions. The second solution provides less flexibility and has a more complex realization. However, it captures the required behavioral logic in the diagram structure itself and does not encapsulate the ordering functionality in functions. The third solution offers the flexibility of the first solution and exposes the behavioral logic in the same way as the second solution. In contrast to the second solution, referencing the first and the last elements of the queued objects, the third solution provides more flexibility in defining the order of removal of elements from the queue, and similar to the first solution can be realized according to first-in-first-out, last-in-last-out, or other desired ordering policy. The selection of the solution depends on the context, within which the pattern needs to be applied, and whether it is necessary to distinguish objects as separate entities (as in the second and third solution) or perform operations on several objects simultaneously (first solution).
This pattern is a generic representation of queue management. The specializations of this pattern addressing specific ordering policies are described in the patterns FIFO Queue, LIFO Queue, and Random Queue (cf. pages 78, 79 and 80) respectively. In this pattern, the ordering algorithm is fixed either by functions or by the net structure, so that all elements of a queue are treated in a uniform way. In some situations, there is a need to make the ordering of objects more flexible and responsive to certain object properties, e.g., age, weight, time-stamp, etc. This problem is addressed by the Priority Queue pattern (cf. page 81), which is a special type of the Queue pattern.

To manage the order of elements queued in a buffer with a limited capacity, this pattern should be combined with the Capacity Bounding pattern (cf. page 71) whose main intent is to bound the capacity of unbounded places.

Examples

- The city hall handles requests from citizens. In order to minimize waiting time for each citizen, visitors that arrive first are served first.
- In order to rent a house, people register at the housing agencies. Based on the date of an application or urgency of the application an ordering is imposed on the waiting queue. For allocation of apartments in new districts, a candidate is randomly selected from the list of reactions received.

Related Patterns This pattern uses the Aggregate Objects pattern (cf. page 67) and the Deaggregate Objects pattern (cf. page 69) in Solution 1, and uses the ID Matching pattern (cf. page 49) in Solution 2. The Priority Queue, FIFO Queue, LIFO Queue, and Random Queue patterns (cf. pages 81, 78, 79 and 80) are specializations of this pattern. This pattern can be combined with the Capacity Bounding pattern (cf. page 71).

Pattern 21 FIFO QUEUE

Intent To allow manipulation of queued objects in a strictly specified order, such that an object, which arrives first, is consumed first.

Motivation The Queue pattern (cf. 75) allows a variable number of objects to queue in-between two steps in a process, providing a means for manipulating objects in a strictly specified order. As was mentioned for the Queue pattern, there are many scheduling policies, according to which manipulation of queued objects can be done. In some situations, there is the need to retrieve objects from the queue in the order of arrival.

Problem Description Assume that a collection of objects takes the form of a queue and that it is necessary either to add an object to the collection or to select an object from the collection and remove it from the queue, whilst ensuring that an object, which arrived first, is retrieved first.

Solution In order to enforce that elements of a queue move in the order of arrival, use a first-in-first-out queue. In the FIFO-queue new elements are added to the tail of the list, where the queued elements are stored, and old elements are removed from the head of the list.

Implementation of Solution The list of instructions below describes how to implement a FIFO Queue (cf. Figure 69).

- Objects of type T to be queued are aggregated into a collection of a list type LT, such that color LT = list T.
Figure 69: Implementation of the FIFO Queue pattern

- Transition put adds an object $x$ to the end of the list $1$ with help of the concatenation function $1 \^\^ [x]$. After the object $x$ is added, the updated list is returned as an input to transition put.
- Transition get removes the first element of the list $x::1$ and puts an updated list back in the queue place. In this way, objects that arrive first are retrieved first.

Note that other solutions presented for the Queue pattern (cf. page 75) are also possible, e.g., figures 67 and 68 refer to a FIFO queue.

Consequences The FIFO Queue pattern is a specialization of Solution 1 of the Queue pattern, which is applied in the situation where multiple objects are aggregated into a collection with unbounded capacity that is sorted in the order of arrival.

Examples
- A butcher shop handles requests from clients. In order to keep the waiting time fair, customers who arrived first are served first.
- In order to rent a house, people register at the housing agency. Based on the date of application, the (FIFO) order in the waiting queue is defined.

Related Patterns This pattern is a specialization of the Queue pattern (cf. page 75).

Pattern 22 LIFO QUEUE

Intent To allow manipulation of queued objects in a strictly specified order, such that the most recently added object is retrieved first.

Motivation The Queue pattern (cf. 75) allows a variable number of objects to queue in-between two steps in a process, providing a means of manipulating objects in a strictly specified order. As was mentioned in the Queue pattern, there are many scheduling policies, according to which the manipulation of queued objects can be undertaken. In some situations, after placing objects in a queue there is a need to retrieve the most recently added object first. Such queueing structure is known in the software community under term “stack”.

Problem Description Assume that a collection of objects takes the form of a queue, and that it is necessary either to add an object to the collection or to select an object from the collection and remove it from the queue, ensuring that the last object added is the first one to retrieved.

Solution In order to enforce that elements in a queue move in the order of arrival, use a last-in-first-out queue. In the LIFO-queue a new object is added to the head of the objects list, where the queued elements are stored, and an object is also removed from the head of the list.

Implementation of Solution The list of instructions below describes how to implement the LIFO Queue pattern (cf. Figure 70).

- Objects of type $T$ that are to be queued are aggregated in a collection of the list type $LT$, such that color $LT = list T$. 
Chapter 3 Colored Petri Nets Patterns

Figure 70: Implementation of the LIFO Queue pattern

- Transition \texttt{put} adds an object \(x\) to the head of the list \(1\) via function \(x::1\). After the object \(x\) is added, the updated list is returned as an input to the transition \texttt{put}.
- Transition \texttt{put} removes the first element of the list \(x::1\) and puts an updated list back into the queue place. In this way, the most recently arrived object is retrieved first.

Consequences This pattern can be applied to ensure strict ordering of objects which are inserted in or removed from from a collection, such that the most recently added object is retrieved first. In computer society, this pattern is known as stack, which represents a container of nodes and two operations, e.g., pop and push. These operations add new nodes to the top of the stack and removing the node from the top of the stack respectively. The \textit{LIFO Queue} pattern is a specialization of Solution 1 of the \textit{Queue} pattern (cf. page 75) which addresses the issue of manipulating objects aggregated into a collection with unbounded capacity.

Examples

- Inventory accounting in which the most recently acquired items are assumed to be the first sold.
- An example of a LIFO is a stack of plates in a cafeteria. In such a stack, only the top plate is accessible to the user, whereas other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, pushing the stack of plates down.

Related Patterns This pattern is a specialization of the \textit{Queue} pattern (cf. page 75).

\textbf{Pattern 23 RANDOM QUEUE}

\textbf{Intent} To allow manipulation of queued objects such that an object is added to the collection in some specified order and an arbitrary object is consumed from it.

\textbf{Motivation} The \textit{Queue} pattern (cf. 75) allows a variable number of objects to queue in-between two steps in a process, providing a means of manipulating objects in a strictly specified order. In some situations, the order in which objects are inserted into the queue is unimportant, since an arbitrary object from the queue needs to be consumed.

\textbf{Problem Description} Assume that a collection of objects takes the form of a queue, and that it is necessary either to add an object to the collection or select an arbitrary object from the collection and remove it from the queue.

\textbf{Solution} In order to enforce that elements of a queue move in an arbitrary order, objects are added either to the tail or to the head of the list in which the queued elements are stored, and randomly removed from it.

\textbf{Implementation of Solution} The list of instructions below describes how to implement the \textit{Random Queue} pattern (cf. Figure 71).

- Objects of type \(T\) to be queued are aggregated in the collection of list type \(LT\), such that color \(LT = list \ T\).
Objects can be placed in the list in any order, i.e. either at the head of the list \(x::l\) or at the tail of the list \(1^{\wedge}[x]\). In Figure 71 objects are added to the head of the list.

Transition \textit{get} picks a random element of the list using the function \textit{rand}.

\[
\text{fun \ rand(l) = List.nth(l, discrete(0, size(l)-1));}
\]

An updated list, i.e. the list without the withdrawn element, is supplied by the function \textit{rest()}.

\[
\text{fun \ rest(x, h::l)= if x=h then l else h::rest(x,l);}
\]

Note that Figure 53 is a possible solution for this pattern, provided that the non-deterministic selection of tokens is done in a “fair” way.

**Consequences** This pattern can be applied to add and remove elements from the queue in any order. The \textit{Random Queue} pattern is a specialization of Solution 1 of the \textit{Queue} pattern (cf. page 75) addressing the issue of manipulating objects aggregated into a collection with unbounded capacity.

**Examples**

- In order to rent a house, a person must subscribe to a housing agency. After subscribing, registered members may react to the available houses. A notary of the housing agency randomly selects a person who will get the house.

- The selection of a candidate who reacted on the same accommodation advertisement and who have the same waiting time is performed by a notary on the random basis.

**Related Patterns** This pattern is a specialization of the \textit{Queue} pattern (cf. page 75).

**Pattern 24 PRIORITY QUEUE**

**Intent** To allow the selection of queued objects in the order of object priority.

**Motivation** In many systems, as Solution 1 of the \textit{Queue} pattern describes (cf. 75), there are buffers where a variable number of objects, aggregated into a collection, are queued in-between two steps in a process. The order in which objects are being added to and removed from the collection may be based on the priority associated with an object.

**Problem Description** Assume that a collection of objects in form of a queue is given, and that it is necessary either to add an object to the queue or select an object and remove it from the queue. The \textit{Queue} pattern solves this problem by capturing a predefined ordering algorithm in the net structure and functions. Specializations of the \textit{Queue} pattern which queue objects in the order of their arrival treat all objects in uniform way, i.e. no matter what the value of an object is, an object is retrieved only when its turn comes (first-in-first-out, last-in-first-out, etc). However, this approach does not allow the order of objects in a queue to vary depending on certain object properties, i.e. the value of an object or the priority associated with it.
**Solution**  In order to allow for the manipulation of objects in an order whose priority is defined by object-specific properties, use a *Priority Queue*. When accessing objects in a queue, the object with the highest priority is removed first.

There are several alternatives for implementing this solution. The selection of an appropriate implementation depends on one of the following context conditions:

- Sorting of a queue based on object priority is done at the moment of object insertion (cf. corresponding implementation alternatives 1 and 2).
- Sorting of a queue based on object priority is done at the moment of object retrieval (cf. corresponding implementation alternatives 3 and 4).
- Sorting of a queue based on object priority after object insertion but before the object retrieval (cf. the implementation alternative 5).

**Implementation 1 of Solution**  A collection of queued objects has to be sorted at the object insertion in order of ascending priority (cf. Figure 72). The priority of an object is based on the object’s value. The first element of the list, i.e. the object with the highest priority, is retrieved first. The list of instructions below describes how to implement the *Priority Queue* pattern (cf. Figure 72).

![Figure 72: Implementation 1 of the Priority Queue pattern](image)

**Assumption**: initially, the collection of objects stored in place **Objects**, is either empty or sorted.

- **Objects** is of type LT. This is a list type which collects objects of type T, i.e. "color LT = list T". In this model, objects are sorted in order of descending priority, i.e. an object with the highest priority is withdrawn from the queue first. Of two objects of INT type, the object with the highest priority is the one, whose value is greatest (cf. the `higherPriority()` function).

  ```haskell
  fun higherPriority (p1, p2) = (p1>p2);
  (* p1 has higher priority than p2 if p1 is greater than p2 *)
  ```

- Objects in the **Objects** place are stored in ascending order, i.e. the first element has the largest value, while the last element has the smallest value. The **Put** transition determines the priority of the new object and inserts it in the queue, ensuring that the queue remains properly sorted (cf. the `pinsert()` function).

  ```haskell
  fun pinsert(elm, []) = [elm]
  | pinsert (elm, q::queue) =
    if higherPriority (elm,q) then elm::q::queue
    else q::(pinsert(elm, queue));
  ```

- The **Get** transition removes the object with the highest priority (and correspondingly the largest value from the queue) by taking the first element of the collection `x::l`.

**Implementation 2 of Solution**  A collection of queued objects has to be sorted at object insertion in order of ascending priority (as shown in Figure 73). In contrast to implementation 1, the object’s priority is not based on the value associated with an object, but is passed as a separate value coupled to the corresponding object. Since the collection of queued elements is composed of pairs (**obj_value**, **priority_value**), the second element
of a pair, i.e. priority, is a parameter for sorting. The first element that will be retrieved from the queue is the element with the highest priority.

\[
(x, \text{prio}) :: \text{Put} \rightarrow \text{Objects}
\]

\[
\text{Get} \rightarrow x :: \text{Out}
\]

**Figure 73:** Implementation 2 of the Priority Queue pattern

*Assumption:* initially, the collection of objects stored in place `Objects` is either empty or properly sorted.

- In Figure 73 transition `Put` takes a pair and sorts the queue using the `sort()` function:

  \[
  \text{fun sort}((x,p), []) = [(x,p)] \mid \text{sort}((x,p), ((y,q)::\text{queue})) =
  \]

  \[
  \text{if higherPriority}(p,q) \text{ then } (x,p)::(y,q)::\text{queue} \text{ else } (y,q)::(\text{sort}((x,p), \text{queue}));
  \]

- The collection of objects 1 is sorted. Transition `Get` takes the first pair from a list `(x,\text{prio})`, which is the pair with the highest priority `\text{prio}`, and returns back the updated list 1.

**Implementation 3 of Solution** Objects are inserted in a queue in any order (objects stored in the collection are not sorted). The collection needs to be sorted based on priority of objects at the moment of object retrieval (cf. Figure 74). In Figure 74, a merge-sort algorithm is used for sorting the list in descending order, however without loss of generality any other sorting algorithm may be used.

\[
\text{In} \rightarrow x :: \text{Put} \rightarrow \text{Objects}
\]

\[
\text{Get} \rightarrow x :: \text{Out}
\]

**Figure 74:** Implementation 3 of the Priority Queue pattern: another alternative

- First, objects are inserted into the collection 1 in any order, i.e. either at the beginning `x::1` or at the end of the list `1::x`.
- Then, a transition guard `[x::\text{templ}=\text{sort}(1)]` added to the `Get` transition, assigns the sorted list 1 to a new list \text{templ}.
- The first element \text{x} of this list is passed to the \text{Out} place, while the rest of the list \text{templ} is put back in the \text{Objects} place.

**Implementation 4 of Solution** Pairs of objects and priorities associated with the objects are inserted in a queue in any order. The collection needs to be sorted based on the value of object priority at the moment of object retrieval (cf. Figure 75).

- Pairs of objects and the priorities associated with them are stored in pairs in the list 1 in any order. When an object needs to be retrieved from the collection, the list 1 is sorted based on the value of the priority element. This implementation, similar to implementation 2, uses pairs of elements, where the priority of an object is specified. Similar to implementation 3, it uses sorting upon retrieval with help of the merge-sort algorithm. Note that any other sorting algorithm can be used instead.
- A sorted list \text{sl} consists of pairs, from which only the first pair, i.e. \text{hd}(\text{sl}), is passed to place \text{Out}. 
Implementation 5 of Solution  
Objects are handled in a FIFO (first-in-first-out) order. The sorting of objects is neither done at object insertion nor upon the object retrieval. In contrast to all of the other implementations considered, sorting in this implementation alternative is done externally, i.e. with help of the Sort transition (cf. Figure 76). This transition can fire when a new object is added to the collection.

This implementation alternative is “non-safe”, given that there is not a strict ordering of firing transitions Sort and Get. In an ideal case, transition Sort must fire before transition Get in order to ensure that all elements in the collection are sorted and the element with highest priority is taken first. However, this may not occur. If a new element is added to the top of the list, and its priority is lower than the preceding element, it should be placed by transition Sort in the correct location. Assume, that transition Get fires first, before the list was sorted. In this case, a newly arrived element with a lower priority will be consumed and the element with the highest priority will stay in the collection.

Consequences  
This pattern can be utilized to retrieve objects from a queue in the order defined by priorities specified in terms of object properties like price, weight, age, etc. To apply this pattern, one should determine how the object priority is defined and when the sorting should be performed, i.e. upon object insertion or upon the object retrieval from the queue respectively. The Priority Queue pattern allows for manipulation of objects in the order defined by object-specific properties. To make this possible, this pattern uses a sorting algorithm which is hidden inside one of the functions. Many sorting algorithms have been developed, which differ in the efficiency and speed of sorting. We leave the selection of the sorting algorithm out of the scope of this pattern, since this information can be found elsewhere.

Implementations 1 to 4 are safe in the sense that they ensure that the retrieval of objects in the order of specified priority. Implementation 5, in contrast to the other implementation alternatives, also performs the sorting of objects, however it is not safe since it cannot guarantee that sorting will be done at the right moment, i.e. preceding object retrieval. To solve this problem, one could apply the Prioritized Execution pattern (cf. page 85).

Example
The service desk of a company distributing coffee-machines handles complaints of clients. Every day there is a list of urgent complaints received, which must be resolved within the same day. However, if too many complaints are received on the same day, and the service desk is not able to handle them all in time, they are scheduled as tasks with the highest priority for the next day.

The human resources department schedules meetings based on the availability of employees and the urgency of the subjects to be discussed.

**Related Patterns** This pattern is a specialization of the *Queue* pattern (cf. page 75). It can be combined with the *Prioritized Execution* pattern (cf. page 85).

**Pattern 25 PRIORITIZED EXECUTION**

**Intent** To coordinate the execution order of two tasks, such that in situations when both of them are simultaneously enabled, one of them always executes before the other.

**Motivation** In some situations, the execution order of two tasks has to be regulated according to the priorities associated with these tasks. The high-priority task always needs to be executed before the low-priority task.

**Problem Description** Figure 77 presents an example of the prioritized enabling of two transitions A and B. Having both inputs in places p3 and p4 available, transition B may fire before transition A has completed. In this situation, it is necessary to ensure that the low-priority transition B fires only after transition A has completed.

![Figure 77: Example of the prioritized execution problem](image)

In such a net it is necessary to ensure that the B transition becomes enabled only after all inputs have been consumed by transition A from the p1 and p2 places. In this diagram, if the absence of tokens in the p1 and p2 has not been tested, the B transition can fire as soon as the collection of objects in the p3 and p4 places becomes non-empty. In CPN, it is possible to check the presence of tokens in a place by linking the place to a transition, which fires and consumes tokens from the input place; however the absence of tokens in the place cannot be checked.

**Solution** In order to coordinate the execution order of two tasks, where one task has priority over the other, test the absence of tokens in the input place of the high-priority task and only then execute the low-priority task. To do this, aggregate the tokens in the place that needs to be tested into a collection as described by the *Aggregate Objects* pattern, and connect an arc to this place for checking the number of objects in the collection.

**Implementation of Solution**

The list of instructions below describes how to implement the *Prioritized Execution* pattern (cf. Figure 78).
The type $T$ of the $p1$ and $p2$ places are replaced with a collection of type $LT$ according to the "Aggregate Objects" pattern (cf. page 67). In this case a list type ($\text{color } LT = \text{list } T$) is used. Note that the order of objects in the collection is unimportant.

- The initial marking of the $p1$ and $p2$ places has been set to the empty list $[]$ if originally this place contained no elements. Otherwise, the list has to be filled with the corresponding elements.
- The enabled high-priority task consumes the required number of tokens from the $in1$ place, and returns the rest of the list to the place.

**Consequences** The "Prioritized Execution" pattern can be used for enforcing conditional enablement of two tasks such that one of them always executes before the other. The enabling of a low-priority task is based on testing for absence of objects in inputs of the high-priority task. This pattern can be combined with the "Containment Testing" pattern (cf. page 73) for testing of the "non-containment" property of places to check that an object satisfying a certain property is not present in a place.

**Examples**
- A medical assistant makes appointments with patients via the telephone. As long as patients continue to call, the assistant continues handling their requests. When no patients remain to be answered, the assistant switches to serving the patient waiting in the queue at the reception counter.
- Of two patients who simultaneously arrive at the first-aid unit of a hospital, the patient with the most life-treating ailment is treated first.

**Related Patterns** This pattern uses the "Aggregate Objects" pattern (cf. page 67). It can be combined with the "Containment Testing" pattern (cf. page 73).

---

So far, we have considered the management of objects residing in a particular place. The eighth pattern group considers two situations where there is a need to manage objects residing in a region that may contain multiple places. The "Region Flush" pattern describes how the content of a particular region can be emptied immediately. The "Content Setting" pattern describes how the content of a particular region can be reset to another state.

**Pattern** 26  REGION FLUSH

**Intent** To clear the content of all places in a particular region.
Motivation In some situations, where a particular region in a process needs to be canceled, all content within this region needs to be instantly removed. When clearing up such a region, all data is discarded and the process may be initiated again.

Problem Description Figure 79 illustrates a part of the journey booking process where both a plane and a hotel need to be booked. Each of the bookings is performed separately, and may consist of several steps, thus being characterized by different states. The payment is performed only when both bookings have been successfully completed. However if a hotel or a plane booking is unsuccessful (i.e. there is a token in the nok place), the other booking needs to be canceled.

Figure 79: Problem of the region cancelation

To cancel a booking process, one needs know which places in the booking region are populated and empty the content of these places. This could be done by connecting all places to the Cancel transition via a “reset” arc, which consumes all of the contents of the place (if any), however this construct is not available in CPNs. Therefore, in order to empty each of the places in the region, one needs to add a separate transition consuming both a token from this place and the token from the nok-place. If a region contains many places, this solution results in a very large and complex model, thus a more elegant way of emptying the contents of the region is required.

Solution In order to clear the contents of all places in a particular region, aggregate objects in each of the places as the Aggregate Objects pattern suggests, and connect each of these places to the Cancel transition in order to empty the content of all places at once.

Implementation of Solution The list of instructions below describes how to implement the Region Flush pattern (cf. Figure 80).

Figure 80: Implementation of the Region Flush pattern
To clear the content of the p1, p2, ok1 and ok2 places, where all objects are accumulated in a single collection, these places are connected to the Cancel transition which consumes collections lt and lt1, and places an empty list [] back in them.

- Figure 81 illustrates how to clear the content of four places. If the region contains more places, these steps have to be repeated for every place.

**Implementation 2 of Solution** Figure 81 demonstrates an alternative implementation for emptying the contents of a region. This implementation is based on the assumption that objects cannot be aggregated in a collection and are stored as separate entities in a place. To calculate the number of objects in the place, the counter-place is introduced.

- For each of the places in the region whose contents needs to be emptied, a counter place is added whose value is incremented when an object is added to the place and decremented when an object is removed from it. This counter is necessary in order to know how many objects have to be removed from the place whose contents is being flushed. If the capacity of the place needs to be bounded, Solution 2 of the Capacity Bounding pattern (cf. the next pattern) should be applied.

- Both the counter place and the place whose contents need to be emptied are connected to the Cancel transition. The current value of the counter i indicates the number of objects to be removed from the p2 place.

Note that although in this solution all tokens have the same value, it also applies to situations where tokens are associated with different types and distinct values.

**Consequences** The Region Flush pattern can be combined with the Content Testing pattern (cf. page 73) in order to enable objects satisfying a particular property to be removed from a place (or all places in a region). In situations where after emptying a region the process needs to be reset to its original state, one could use an extension of this pattern, i.e. the Content Setting pattern (cf. next pattern for details).

In its solution, this pattern uses the Aggregate Objects pattern (cf. page 67) in order to remove all objects available in a place at once. Note that by aggregating objects into
a collection, an empty collection can be tested, which would not be possible if the place stored objects as separate entities.

**Examples**
- After patient’s death the relevant records in the patient database are destroyed.
- For security reasons, upon transfferral from one department to another an employee’s data must be removed from the employee computer.

**Related Patterns** This pattern uses the *Aggregate Objects* pattern (cf. page 67) in its implementation 1. It can be combined with Solution 2 of the *Capacity Bounding* pattern (cf. page 71), and is extended by the *Content Setting* pattern (cf. page 89).

**Pattern 27 CONTENT SETTING**

**Intent** To reset the content of a particular region to another state.

**Motivation** The *Region Flush* pattern allows the content of places in a particular region to be removed. In some situations, after the region has been emptied it may be necessary to reset it to its original or another state. This may be necessary, for instance, when a process that failed needs to be restarted or when requirements for process execution has changed and the state of the process needs to be adjusted.

**Problem Description** Figure 82 illustrates the process of trip booking where both a hotel and a plane need to be booked. In case, one of the bookings fails, the other booking process needs to be stopped and the both procedures have to be restarted. This situation is different from that described in the *Region Flush* pattern where a region is canceled and all contents are inconsequentially removed from it.

**Solution** In order to reset the content of a region, in each place within the region aggregate objects into a collection as described by the *Aggregate Objects* pattern, and connect the *Reset* transition to it such that it will remove all objects from the various collections simultaneously and add new objects into them.

**Implementation of Solution** The list of instructions below describes how to implement the *Content Setting* pattern (cf. Figure 83).
- Objects in places p1 and p2 are aggregated into a collection as described by the *Aggregate Objects* pattern (cf. page 67).
The reset transition is added, whose incoming arcs remove all the content of the places within the region by consuming lists \( \text{lt} \) and \( \text{lt1} \), and put back a new value into each of them.

**Consequences** This pattern allows all of the contents in a particular region to be reset to another state. For this, the contents of all places in the region are emptied by applying the *Region Flush* pattern (cf. page 86) and then the selected places are reset to another value.

**Examples**
- After having modified the personal settings of their mobile phone, a user may choose to delete all changes and reset them to default values.
- When during an operation a patient develops high pressure, the operation needs to be halted, and all associated resources have to be prepared for the next operation.

**Related Patterns** This pattern extends the *Region Flush* pattern (cf. page 86).

---

The last pattern group consists of seven patterns, which address problems of data organization and access. The *Shared Database* pattern describes how to provide different levels of data visibility in a model and share access to data stored in one place between multiple transitions. The *Database Management* pattern describes an interface for accessing a shared database. The *Concurrent Access* pattern addresses the problem where multiple tasks need to use data stored in a particular place simultaneously. The *Copy Manager* pattern describes how to maintain the consistency of data distributed in multiple places. The *Lock Manager* and *Bi-Lock Manager* patterns describe different variants of accessing data stored in a shared database on the basis of shared and exclusive locks.

---

**Pattern 28  SHARED DATABASE**

**Intent** To enable the centralized storage of data shared by multiple tasks with support for different levels of data visibility (i.e. local, group, or global).

**Motivation** In Petri nets, a transition is only aware of data directly supplied to its input places. It has no knowledge of data related to other transitions. In other words, the visibility of data relative to a transition is local. In some cases, it is necessary to make
the local data visible to a group of transitions or to all transitions contained in a model, providing for group and global visibility respectively.

**Problem Description** Assume that within a given chain of transitions (cf. Figure 84), it is necessary to pass some data from the start of the chain A to the end D. Although intermediate steps do not change the data at all or do so only infrequently, the data is passed through the whole sequence of transitions rather than made being available upon request, i.e. only at the moment when it is needed.

![Figure 84: Problem of data sharing](image)

Data element x with an identifier id is sequentially passed from place A to place D via transitions T1, T2 and T3. Although transition T2 does not use the data element x, it knows that it is passed. In terms of network transfers, this leads to overloading of traffic, increases the duration of the data transfer and slows down overall performance. From a security point of view, it might be desirable to limit a set of transitions to that sub-set which is authorized to access the data element, thus minimizing information outflow.

**Solution** In order to centralize the storage of data shared between multiple tasks, use a *shared database*. A shared database is a place that provides access to data for transitions connected to the database.

In order to enable *global visibility* of data in a model, all transitions in the net must be connected to the shared database. In order to limit the visibility of data to a certain group of transitions and thus establish *group visibility*, the shared database has to be connected only to those transitions which are allowed to access the data. *Local visibility* can be obtained by connecting only one transition to the database. Thus, the number of transitions connected to the shared database determines the visibility of data in a model.

**Implementation of Solution** The list of instructions below describes how to implement the *Shared Database* pattern (cf. Figure 85).

![Figure 85: Implementation of the Shared Database pattern](image)

• A place *Shared data*, where data shared between transitions will be stored, is introduced. The format of data stored in the database must allow all data stored in the database to be referenced by means of unique identifiers (as required by the *Database Management* pattern which defines the interface for accessing the shared database on page 92). Instead of passing data elements through a whole chain of transitions which have no access to the shared database, it is sufficient to pass only identifiers corresponding to the data elements. The desired level of data visibility is determined by the number of transitions which are connected to the database.
• The identified transitions are connected to the shared database by means of bidirectional arcs. An arc directed to a transition supplies the transition with the current value of the data element requested by transition. For details of accessing data in a database see the Database Management pattern.

• Organize the data stored in the shared database into a data structure of the collection type, so that the whole database corresponds to a single token, using the Aggregate Objects pattern (cf. page 67).

**Consequences** This pattern can be used to share data between a group of transitions, so that each transition can access the data in the database whenever it needs to. One of its possible uses is to restrict access to data by unauthorized transitions by defining different levels of data visibility.

The definition of the level of data visibility is one of the most important decisions the developer of a model needs to take during the early stages of the development. Carefully selecting the degree of data visibility may help in preventing unauthorized data access, information outflow or making a model cumbersome by passing irrelevant or complex data. In almost every model there is the need for at least one global database, which stores persistent information (often non-variable or less-frequently variable), and several local databases, the visibility in which is limited to a specific group of transitions.

The drawback of implementing this pattern is that it can make a model look “spaghetti-like” due to the multiple arcs connecting transitions and a shared place. Model complexity may increase dramatically if multiple databases have to be introduced, thus creating for each group an extra place with corresponding arcs. Therefore, one should make a trade-off between increased model complexity, introduced by the shared places, and the importance of (non)limited access to data.

As it was mentioned in the implementation section, for structuring the database, this pattern can be combined with the Aggregate Objects pattern (cf. page 67). In some situations, it is necessary to make data, which is available in one shared database, also available in other locations. To make this possible, this pattern should be combined with the Copy Manager pattern (cf. page 96).

**Examples**

• A supervisory board is composed of ten people, three of whom are responsible for the budget. Each of the three people has access to the bank account. The visibility of the account data in this case is limited to these three people.

• Two companies, each of which has a number of internal projects, are involved in a joined venture. The members of both companies have access to the joint venture (global visibility). However, the members of one company do not have access to the internal projects of the other company, since the visibility of those projects is limited to the particular company’s staff members.

**Related Patterns** This pattern is extended by the Database Management pattern (cf. page 92). It can be combined with the Aggregate Objects pattern (cf. page 67) and the Copy Manager pattern (cf. page 96).

**Pattern 29 DATABASE MANAGEMENT**

**Intent** To specify an interface for accessing data, stored in a shared database, for read-only and modification purposes.
M**otivation** The *Shared Database* pattern (cf. 90) provides a solution for centralizing data shared between several transitions, allowing support for different levels of data visibility. Usually, a shared database is used either as a static data provider, which contains data for read only purposes and prohibits modifications, or as dynamic storage, where data is accessed for read/write purposes. To make the distinction between these two types of databases, their access interfaces should be clearly differentiated.

**Problem Description** Assume that two independent threads need to retrieve data from the *Shared Database* for read-only and modification purposes respectively (cf. Figure 86). According to the *Shared Database* pattern, it is necessary to connect transitions *Read-only* and *Modify* to the *Shared Database* place, which makes the data stored in the database visible and shared between these transitions. However, when connecting these transitions to the database, it is not clear what interface must be used for retrieving data for read-only and modification purposes respectively.

![Figure 86: Problem of accessing the shared database](image)

**Solution** Assumption: all data stored in a shared database has a unique identifier for referencing purposes.

In order to retrieve a data element from a shared database, use identifiers associated with data elements stored in the database as a reference. For read-only purposes, the value of the data element is retrieved and put back unmodified. For modification purposes, the value of the data element is retrieved and the modified value is put back in the database.

**Implementation of Solution** The list of instructions below describes how to implement read/write interfaces for shared database (cf. Figure 87). It is the generalization of both read and modify operations, which allows a shared database of the dynamic type\(^4\) both to read and modify data stored in it. The specialization of this pattern, an interface of accessing a shared database of the static type, can be derived from the interface of accessing a dynamic database by removing features which allows for data modification.

- The *Shared Database* place is connected to the *Modify* transition, providing the value of a data element with the requested identifier in the form \((id1, value)\).
  
  Note that the data identifier serves as a key for retrieving the value of a data element from the database by using the *ID Matching* pattern (cf. page 49).
- A guard of transition *Modify* contains a function \(f()\) which defines the new (modified) value of the data element.
- After the guard has been evaluated, the new value \(val_{1,m}\) is placed back in the *Shared Database*.

---

\(^4\)We use the terms dynamic database to refer to a database whose content can be changed, and static database to refer to a database whose content is fixed and used for only for reading purposes.
Figure 87: Implementation of the read/write interface of the Shared Database

Figure 88 illustrates the interface for “read-only” access to the shared database.

Figure 88: Implementation of read-only access to the shared database

The Retrieve transition provides an identifier \( id \) for accessing the value \( val1 \) of a data element stored in the shared database for the read-only purposes (this value is subsequently returned back unmodified).

Figure 89 illustrates the access interfaces implemented in the example presented in the Problem Description section.

Figure 89: Example of access interfaces to the shared database

Consequences This pattern can be applied to realize an access interface to a static shared database for read-only purposes and a dynamic database required for both reading and modification of data stored in the database.

The Database Management pattern clarifies the interfaces required for accessing shared databases on a read-only and read/write basis. However, this pattern does not deal with problems which might appear during the simultaneous access of data for modification by several transitions. In order to synchronize concurrent access to the shared data, the Shared Database pattern should be combined with the Lock Manager pattern (cf. page 98) or with
the *Bi-Lock Manager* pattern (cf. page 100). The *Lock Manager* pattern ensures exclusive access to all data elements by means of exclusive locks. The *Bi-Lock Manager* pattern provides shared access for reading and exclusive access for writing by means of shared and exclusive locks respectively.

**Examples**
- Databases with bibliographical information relating to books, articles, and other published materials accessed by users on-line.
- Centralized storage of the patient data in a hospital, which is available in all functional departments of the hospital.

**Related Patterns** This pattern uses the *ID Matching* pattern (cf. page 49). It can be combined with the *Lock Manager* pattern (cf. page 98) or the *Bi-Lock Manager* pattern (cf. page 100).

**Pattern 30 CONCURRENT ACCESS**

**Intent** To provide concurrent access to common data elements by a series of individual tasks without implying any changes to these data elements.

**Motivation** In situations, where the same object needs to be used by different entities, the entities need to queue and wait until the object is released by preceding users before it actually can be utilized. Having only one object available, the users may access it only on an interleaved basis (as described in the *Shared Database* pattern on page 90), and cannot utilize it concurrently. This may result in long waiting times and introduce undesired dependencies between previously independent parties.

**Problem Description** Figure 90 represents a place realized according to the *Shared Database* pattern. Transitions T1, T2 and TN read the value of the object stored in the Object place. In this diagram, when one transition consumes a token from the Object place, others may not access it and thus must wait until it is available.

![Figure 90: Problem of concurrent testing](image-url)
Solution  In order to enable concurrent access to a data element stored in a particular place by a series of individual tasks, replicate the value of the data element as many times as there are tasks that require access to it.

Implementation of Solution  Figure 91 demonstrates implementation of the Concurrent Access pattern.

![Diagram of Concurrent Access pattern](image)

- Assuming that less than N transitions need to use the content of the Object place, the value of this place is replicated by adding N’e tokens to it via the Set transition. When necessary, all objects can be removed from the Object place via the Get transition. Furthermore, the content of the Object place can be updated by means of the Update transition. Having as many objects in the Object place available as there are transitions testing it allows for concurrent access by all parties.

Consequences  The Concurrent Access pattern allows the contents of a place to be used by multiple transitions concurrently. Note that the access to the data element is read only, thus no modifications resulting in data inconsistency can be made. This pattern is similar to the Shared Database pattern (cf. page 90) where objects stored in the database can be accessed for reading or writing purposes. However, in this pattern the objects stored in the shared place may be accessed concurrently, while in the Shared Database the access to the objects is performed on an interleaved basis. In case a particular property of objects stored in the shared place needs to be analyzed, this pattern can be combined with the Containment Testing pattern (cf. page 73).

Examples  
- A library keeps several copies of each publication, to allow the same information source to be accessed by several readers simultaneously.
- A department has requested several license keys for a software tool to allow multiple students to work with the tool during instructions.

Related Patterns  This pattern is similar to the Shared Database pattern (cf. page 90) and can be combined with the Containment Testing pattern (cf. page 73).

Pattern 31  COPY MANAGER

Intent  To make data, stored in a shared database, available at other locations for local use, whilst maintaining the consistency of data in all places.

Motivation  In many situations, data stored in a central database needs to be accessed concurrently in different locations, which are often independent of each other. Such locations need to be able to work with data even when the connection to the central database
Problem Description Assume that there is a central database in which data related to employees is stored. Different departments, e.g., finance department, housing service, education center, etc. need to access this data concurrently. As a consequence of an exclusive access restrictions to the database, one department can not access data until it is released by another department. Such dependencies result in long waiting times and inefficient work practices. In addition, there is the possibility that access to the data stored in the database will be limited for all departments if the network connecting these departments to the database is temporarily unavailable. Thus, there is the need to make the data stored in the central database locally available.

Solution In order to make data stored in the shared database available at other locations, use a copy manager. The copy manager replicates data from the central database to local storage. The copy manager maintains data consistency by updating data in the main database when the local copy has been modified and by synchronizing local copies with the main databases when it has been modified.

Implementation of Solution The list of instructions below describes how to implement the Copy Manager pattern (cf. Figure 92).

- In order to replicate data stored in a shared database in one go, data in the shared database is aggregated into a collection (as the Aggregate Objects pattern on page 67 describes). Copying elements one-by-one is inefficient and time-consuming, therefore it is not considered as an implementation option.
- A new place Local copy is created, where a local copy of the data available in the central database will be stored. Note that the types of places Shared Database and Local copy are the same.
- The Shared Database place, where the central database is stored, is connected to the newly created local database by the Replicate transition which copies all data from one place to the other.
- After the data stored in the local database has been modified by the Modify transition, the old value stored in the central database is updated with a new value via the Report change transition. In order to update the original database according to changes performed in the local database, only elements that were modified have to be reported. The Report Change transition takes the whole data collection from the Shared Database and replaces the value of the data element with the specified id.

```plaintext
fun update(((id,a)::queue,id2,y)) = if (id=id2) then
  (id,y)::queue else (id,a)::update(queue,id2,y);
```

Figure 92: Implementation of the Copy Manager pattern
• The Local copy place must be regularly synchronized (with the Shared Database place) by executing the Replicate transition. This ensures that data in the local copy coincides with data stored in the shared database.

Consequences This pattern can be applied to manage the consistency between data stored in one place and that stored at another place by means of data replication.

Since multiple local databases may modify data elements stored in their copies, these changes should be communicated to the central database to ensure data consistency. Because multiple local databases may want to update the same data in the shared database concurrently, there is the need to incorporate the Bi-Lock Manager pattern (cf. page 100), which solves synchronization problems resulting from concurrent database access by means of shared and exclusive locks.

Note that even if no data is changed in the local copy, it is still necessary to periodically replicate the data in order to incorporate recent changes introduced either by the central database itself or by other local databases for overall data consistency. When several parties try to modify the same information at once, some updates may be lost. This problem can be solved by enforcing the use of locks each time a data element needs to be modified. This problem is addressed by the next pattern.

Examples
• Mobile employees require a reliable software solution that allows them to access their company data locally on a mobile device, modify this data, and synchronize the changes with a database on a remote server in a timely fashion.
• Synchronization of databases between different departments.

Related Patterns This pattern can be combined with the Bi-Lock Manager pattern (cf. page 100).

Pattern 32 LOCK MANAGER

Intent To synchronize access to shared data by means of exclusive locks.

Motivation When several related processes execute concurrently, often they share some resources or data stored in a shared database. The Shared Database pattern (cf. 90) allows multiple entities to access data stored in the shared database concurrently, but it does not protect them from overwriting each other’s changes accidentally or reading inconsistent data due to changes that are in progress.

Problem Description Assume that two owners of the same (joint) bank account withdraw money from different cash-dispensers simultaneously. If both cash-dispensers access the account concurrently, it is not clear whether one or both amounts will be subtracted from the account and what the final balance would be. Thus, unsynchronized access to information stored in the shared database can lead to data inconsistency.

Figure 93 illustrates two processes that retrieve data with an identifier \( \text{id} \) from the Database place, subsequently apply function \( f() \) to modify the value of the data element retrieved and substitute the old data value with a newly calculated one. Such a net potentially exhibits problems with the consistency of data in the database, when the value of the data element utilized by one process gets updated by another process during the time that the first process is using the data element.

Solution In order to synchronize access to a shared data, use a lock manager. The lock manager allows only one user to use a shared data element at a time. In order to access
a shared data, an actor must acquire a lock. A user that owns a lock must release a lock before the shared data element can be used by another user.

**Implementation of Solution** The list of instructions below describe how to implement the Lock Manager pattern (cf. Figure 94). The lock manager consists of two parts: lock acquisition and lock release.

- The Locks place contains a list of locks for objects that have already been acquired. This place is connected both to transitions that acquire access to an element stored in the database and transitions that release elements after they have been processed. In particular, the Locks place is connected to the retrieve1, retrieve2, update1 and update2 transitions.

- When a request to retrieve an object with an identified id is supplied by the in1 or in2 places, the enabling condition associated with transitions retrieve1 and retrieve2 is checked respectively. Guards associated with these transitions, [not(mem lid id)] check if an object with identifier id is not contained in the list of acquired locks lid.

- If an identifier for the object that needs to be retrieved from the database has not been found in the list of locks lid, the permission to access the object with the given identifier in the Database place is given. After processing this object, the corresponding transition (update1 or update2) returns (potentially modified) object to the Database place. Furthermore, the lock for the given is released by means of the del(id,lid) function.

Note that Figure 94 illustrates two processes accessing elements of the database for processing. Since the mechanism of lock acquisition/release is the same for each of the parties, as emphasized in the diagram, it can be used to controlling access when other processes are added.

**Consequences** This pattern can be applied to ensure the synchronization of concurrent access to the data shared between multiple users by means of exclusive locks, i.e. allowing only one user access the data at a time. This pattern can be applied in combination with shared databases described by the Shared Database pattern (cf. page 90) in order to maintain the consistency of data accessed by multiple users simultaneously.

Note that this pattern uses the ID Matching pattern (cf. page 49) in its solution for analyzing requests for data access. In order to keep a track of locks, this pattern aggregates them into a list using the Aggregate Objects pattern (cf. page 67).

In situations where exclusive access to shared data needs to be combined with shared access, instead of this pattern its extension the Bi-Lock Manager pattern (cf. page 100) can be used.

**Examples**
Pattern 33  BI-LOCK MANAGER

Intent  To synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks.

Motivation  When several related processes execute concurrently, they often share resources or data. The Shared Database pattern (cf. 90) allows multiple users to access data stored in a shared database concurrently, however it does not protect them from overwriting each other’s changes accidentally or reading inconsistent data due to changes that are in-progress. Often in practice, there is a need to provide shared and exclusive access to data stored in a shared database so that multiple users can read data concurrently, however only one user should be able to modify the corresponding data element at a time.

Problem Description  The Lock Manager pattern (cf. 98) solves the problem of synchronizing access to a shared resource by means of exclusive locks. No matter what the purpose of the access is, i.e. reading or modification, this pattern allows only one user to access the data element at a time. In some situations, it is necessary to differentiate between two types of data access, i.e. access for reading and access for writing, so that multiple users can access an object for reading purposes but for writing purposes only one user can access it at a time.

Figure 95 illustrates a shared database that may need to be accessed by two processes concurrently. In this net, processes may both read data from and write new data to the database. Due to the absence of the access control, a data element that has been retrieved by one process for reading purposes may be modified meanwhile by the other process, without being noticed by the first process and thus resulting in data inconsistency.

Solution  In order to synchronize access to a shared data element, allowing for both shared and exclusive access, use a bi-lock manager. The bi-lock manager provides shared and exclusive locks which must be acquired by users in order to read and modify data.
respectively. Shared locks allow multiple users access the data at the time for reading purposes, while exclusive locks ensure exclusive access for data that needs to be modified.

**Implementation of Solution** The list of instructions below describes how to implement the Bi-Lock Manager pattern (cf. Figure 96).

- In order to differentiate between exclusive and shared locks, two places **Read locks** and **Write locks** are introduced. These places store locks, acquired for reading or modifying a data element in a database, in the form of a collection of data identifiers. When a request for reading/writing has been granted by **Read** or **Modify** transitions, an identifier of the data element that is being accessed is added to the list of locks.
- In order for an exclusive lock to be granted, the guard of the **Modify** transition needs to be fulfilled. This guard specifies that the data element requested for modification should not be locked for reading nor for writing:
  
  \[
  \text{not(mem \ lut id), not(mem \ lut id)}
  \]

- In order for transition **Read** to be enabled, an identifier \( id \) of the data element which has been requested for reading, should not be contained in the list \( \text{luid} \). In other words this data element should not be modified by any other process in order for is
value to be retrieved from the database.

- After reading a data element, transition \texttt{Release read lock} releases the read lock by removing the corresponding identifier from the list of the read locks \texttt{lid} via the \texttt{del()} function.
- After modifying the value of a data element, the \texttt{Release write lock} transition returns an updated value \texttt{y} to the \texttt{Database} place.

\textbf{Consequences}  This pattern can be applied to ensure a synchronized access to data shared between multiple users. The \textit{Bi-Lock Manager} is an extension of the \textit{Lock Manager} pattern (cf. page 98). Similar to the \textit{Lock Manager} pattern, this pattern should be used in combination with a shared database as described in the \textit{Shared Database} pattern (cf. page 90) in order to manage access to data stored in the database.

This pattern can combined with the \textit{Copy Manager} pattern (cf. page 96), whose intent is to replicate data stored in a central database into local copies providing for overall data consistency. To prevent overwriting the same data in the shared database during synchronization with multiple local databases, the \textit{Copy Manager} pattern requires exclusive access to the shared database for modification purposes and shared access for reading purposes.

\textbf{Examples}

- Account access by banks, credit card companies and insurance companies, which provide shared access for reading but exclusive access for writing.

\textbf{Related Patterns}  This pattern extends the \textit{Lock Manager} pattern (cf. page 98). It can be used in combination with the \textit{Shared Database} pattern (cf. page 90) and the \textit{Copy Manager} pattern (cf. page 96).

When describing the CPN patterns we indicated the type of problems they address. In order to support users in selecting a suitable pattern from the pattern catalog, in Section 3.3 we concentrate on the classification of CPN patterns. Furthermore, in order to simplify the navigation through the pattern catalog we analyze the relationships between the patterns and graphically depict them in the form of a relationship diagram.

\section{Classification of CPN patterns}

This section focuses on the classification of the CPN patterns. In Sub-section 3.3.1, we analyze relationships between patterns and visualize them in the form of a relationship diagram. In order to support users in selecting a suitable pattern from the pattern catalog, in Sub-section 3.3.2 we group the patterns into clusters and explain how such a classification can be used in practice.

\subsection{Relationships between CPN patterns}

In this section, we analyze relationships between the CPN patterns, and organize them into a relationship diagram, which allows navigation through the pattern catalog in order to identify related patterns. The main purpose of this classification is to provide a holistic view of the catalog of patterns, providing a means for a user to select a number of patterns and to determine how individual patterns can help in solving a given problem. The selected types of relationships can help to trace other patterns related to a chosen pattern, and
Figure 97: The CPN patterns relationship diagram
compare the chosen pattern with similar patterns in order to select an optimal solution for a problem in the given context.

The 33 CPN patterns that have been identified, together with the relationships between them, form a pattern language. In order to classify the CPN patterns we have examined the nature of the relationships between the patterns. We used three types of primary relationships: problem-oriented, solution-oriented, and implementation-oriented; and two types of secondary relationships: problem similarity, and combinable solutions to describe the relationships between patterns. Some of the relationship types are based on Zimmer’s classification [238]. Figure 97 shows the relationships between various patterns. The graphical representation and the text depict the type of a relationship. To understand the diagram, we first need to explain the different types of relationships.

Primary relationships We consider three types of primary relations:

- **Problem-oriented relationship:** Pattern A is a specialization of a more general pattern B. More specifically, pattern A deals with a specialization of the problem that pattern B addresses, and has a similar but more specialized solution than pattern B. Pattern A includes all the properties of pattern B, but adds further restrictions by including some specialized characteristics. Note that a specialization often adds more context to the problem thus making it less generic. An example of this relationship is the FIFO Queue pattern (cf. page 78) which is a special variant of the Queue pattern (cf. page 75).

- **Solution-oriented relationship:** Pattern A uses pattern B in its solution. When searching for a solution to a problem addressed by pattern A, one of the sub-problems is found to be similar to the problem addressed by pattern B. Thus, the solution of pattern B is a composite part of the solution of pattern A. Whenever pattern A is used, pattern B should also be considered, since it forms a part of A. All instantiations of pattern A use pattern B. Some examples of this relationship are: the Lock Manager pattern (cf. page 98) which uses the ID Matching pattern (cf. page 49), and the Asynchronous Router pattern (cf. page 60) which uses the Asynchronous Transfer pattern (cf. page 54).

- **Implementation-oriented relationship:** Pattern A syntactically extends the implementation of pattern B. Pattern A addresses a set of requirements which have additional or slightly different functionality to those in pattern B. However, it is the implementation of B, which is syntactically extended by A, rather than the problem or a solution. For example, the implementation of Content Setting pattern (cf. page 89) extends the implementation of the Region Flush pattern (cf. page 86).

Secondary relations In addition to the three kinds of primary relationships which directly link patterns with each other, we also consider two types of secondary relationships:

- **Problem similarity relationship:** Pattern A is similar to pattern B. Pattern A addresses a problem similar to the one addressed by pattern B. Patterns A and B can be considered as alternatives of each other; therefore, one can compare them and select the pattern which best fits. For example, the Deterministic XOR-Split pattern (cf. page 36) is similar to the Non-Deterministic XOR-Split pattern (cf. page 38).

5Since a pattern may have multiple solutions, the relationship may need to refer to a specific solution. For the sake of clarity we use the mnemonics “s1”, “s2” and “s3” as identifiers for referring to the first, second, and third solution and to make relations between pattern solutions explicit.
3.3 Classification of CPN patterns

- **Combinable solutions relationship**: Pattern A can be combined with pattern B. Neither of the patterns is a part of the other. Combining the solution of pattern B with the solution of pattern A can help to solve a more complex problem than a single pattern solves in isolation. This relationship can help in finding other patterns which can be used in addition to pattern A. For example, the *Shared Database* pattern (cf. page 90) can be combined with the *Copy Manager* pattern (cf. page 96); the *Prioritized Execution* pattern (cf. page 85) can be combined with the *Priority Queue* pattern (cf. page 81).

Figure 97 shows the five types of relationships which exist between CPN patterns. Note that the secondary relationships depicted only represent some typical examples. The solid arrows represent primary relationships while the dashed lines represent secondary relationships. The problem-oriented relation is labeled “is specialization of”. The solution-oriented relation is labeled “uses”. The implementation-oriented relation is labeled “extends”. Problem similarity is denoted by dashed lines (without dots) while combinable solutions are denoted by dashed lines with dots. Note that details regarding the combination of one pattern with another one, or similarities between patterns are not indicated in the relationship diagram, but can be found in the *Consequences* and *Related patterns* sections of the individual patterns.

### 3.3.2 Clustering of CPN patterns

In this section, we classify the CPN patterns into categories in order to simplify the process of selecting a pattern from the CPN pattern catalog. Although the CPN pattern relationship diagram presented in Figure 97 allows for navigation through the catalog of the CPN patterns, it is not sufficient to classify the patterns precisely and unambiguously.

Given the nature of CPNs, the CPN patterns aim to solve problems in the domain where data and control-flow perspectives interplay. In this domain, three pattern groups can be distinguished:

- patterns where the data perspective dominates, but which must be considered in the context of the control-flow (for instance, the *ID Matching* pattern (cf. page 49) refers to the property of objects rather than to the control-flow);
- patterns where the control-flow perspective dominates, but which are data-based (for instance, the data-based control-flow routing patterns such as *Deterministic XOR-Split* pattern (cf. page 36));
- patterns where both data perspectives and control-flow perspectives are equally important (for instance, the *Priority Queue* pattern (cf. page 81) where both aspects of data-based analysis and control-flow routing are important).

In the main, this classification is helpful only for the purpose of CPN patterns discovery. In this case, patterns where the control-flow perspective plays a prominent role can be selected, and based on their formal specification the relevant control-flow structures can be identified in the model being analyzed. However, when a user needs to choose a particular pattern without having clearly defined the problem according to the role of the data and control-flow perspectives, this classification appears to be less meaningful and somewhat subjective.

In order to provide a more useful means for selecting an appropriate pattern, we adopt the classification approach presented in [118] to categorize the CPN patterns. This classification is based on the *intent* of each pattern. The rationale for each pattern has been analyzed according to a hierarchical three-level structure where *common components* contain
**diagnostic elements**, and in turn diagnostic elements contain **supplementary components**. This structure will be used in describing different pattern clusters in Table 3.1.

Common components define the set of related meanings, on the basis of which different patterns can be grouped. For instance, patterns addressing the problem of creating new elements or entities, belong to the same group with the common component create. This indicates the intent of a pattern from the process (functionality) point of view. For example, patterns, whose main intent is to manage or control something, will be combined into a group with a common component control.

Diagnostic elements define the contrastive features which distinguish the patterns belonging to the same common component. For instance, patterns belonging to the same common component control, i.e. control patterns, may involve different participants or differ in terms of control parameters. Additionally, patterns, whose main purpose is to control such features as order, throughput, and quantity, belong to the same group with a common control component and can be distinguished by the diagnostic elements Order, Throughput, Quantity respectively.

Supplementary components address additional features with extended definitions of meanings. This component addresses special circumstances when applying a pattern. This feature could be applied to distinguish one pattern from other patterns belonging to the same common component with the same diagnostic elements; however, multiple patterns may have the same supplementary component.

Using the nested format described above (i.e. common components, diagnostic elements, supplementary components), we are able to classify the 33 CPN patterns as illustrated in Table 3.1.

Let’s consider how the given classification can be used in the example of one pattern. The Aggregate Objects pattern (cf. page 67) is classified under the common component “Assemble”, the diagnostic component “Data objects”, and supplementary component “by aggregating into a collection”. It is interesting to see how the Queue pattern (cf. page 75), although it extends Aggregate Objects pattern, is classified completely differently. This can be explained by the clear difference in the pattern’s intent.

Although the CPN patterns presented in this chapter originate from the practice, it is interesting to see how frequently they are actually applied during modeling. In Section 3.4, we analyze the selection of models in order to identify which CPN patterns they utilize and how often. We illustrate the process of pattern identification using as an example one of the models analyzed.

### 3.4 Analysis of CPN patterns usability in practice

In this section, we present the results of an analysis of the frequency of use of CPN patterns in practice. To draw conclusions regarding the actual use and applicability of patterns in practice we analyzed 20 models provided by the participants of CPN workshops that took place in Aarhus, Denmark in 2005-2007. The models were created using CPN Tools and were made by expert users from different organizations (cf. [155]).

To answer the following two research questions: “Which patterns are used in practice?” and “How often are patterns used in practice?” we defined several metrics. Each of the models has been analyzed with respect to the model complexity, by calculating the total number of places and transitions in the model, the number of distinct patterns used in a model, and the total number of patterns including the repeating ones used in these models. These metrics are visualized in Figure 98.
### Table 3.1: Classification of CPN patterns

<table>
<thead>
<tr>
<th>Common component</th>
<th>Diagnostic component</th>
<th>Supplementary component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Order of data objects (Queue)</td>
<td>by predefined scheduling policy:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queue (page 75), FIFO Queue (page 81), LIPO Queue (page 79), Random Queue (page 80)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by objects’ priority:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priority Queue (page 81)</td>
</tr>
<tr>
<td></td>
<td>Consistency of distributed data</td>
<td>by regular data replication:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Copy Manager (page 96)</td>
</tr>
<tr>
<td></td>
<td>Concurrent access to shared data</td>
<td>by means of exclusive locks:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock Manager (page 98)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by means of shared and exclusive locks:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bi-Lock Manager (page 100)</td>
</tr>
<tr>
<td></td>
<td>Throughput of data</td>
<td>by inspecting content:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSI Filter (page 42), NBSI Filter (page 45)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by inspecting state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSD Filter (page 43), NBSD Filter (page 47)</td>
</tr>
<tr>
<td></td>
<td>Number of objects in place</td>
<td>by bounding the place capacity:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Capacity Bounding (page 71)</td>
</tr>
<tr>
<td></td>
<td>Uniqueness of data objects</td>
<td>by generating fresh identifiers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ID manager (page 51)</td>
</tr>
<tr>
<td></td>
<td>Discern</td>
<td>by using object identifier:</td>
</tr>
<tr>
<td></td>
<td>Identity of data objects</td>
<td>ID Matching (page 49)</td>
</tr>
<tr>
<td></td>
<td>Visibility of data objects</td>
<td>by sharing access to data objects:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shared Database (page 90)</td>
</tr>
<tr>
<td></td>
<td>Choose</td>
<td>by exclusive data conditions:</td>
</tr>
<tr>
<td></td>
<td>Single branch/ task</td>
<td>Deterministic XOR-Split (page 36)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by non-exclusive conditions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Deterministic XOR-Split (page 38)</td>
</tr>
<tr>
<td></td>
<td>Several branches/tasks</td>
<td>by overlapping data conditions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR-Split (page 40)</td>
</tr>
<tr>
<td></td>
<td>Transfer</td>
<td>by blocking the sender:</td>
</tr>
<tr>
<td></td>
<td>Data from one process to another</td>
<td>Synchronous Transfer (page 56)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by non-blocking the sender:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous Transfer (page 54)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by simultaneous exchange:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rendezvous (page 58)</td>
</tr>
<tr>
<td></td>
<td>Distribute</td>
<td>by decomposing/merging data objects:</td>
</tr>
<tr>
<td></td>
<td>Data to several places</td>
<td>Distributed Data Processing (page 65)</td>
</tr>
<tr>
<td></td>
<td>for concurrent processing</td>
<td>by routing:</td>
</tr>
<tr>
<td></td>
<td>Data to dedicated target indirectly</td>
<td>Asynchronous Router (page 60)</td>
</tr>
<tr>
<td></td>
<td>Data to multiple targets indirectly</td>
<td>by decoupling source from targets:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Broadcasting (page 63)</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>by aggregating into a collection:</td>
</tr>
<tr>
<td></td>
<td>Data objects</td>
<td>Aggregate Objects (page 67)</td>
</tr>
<tr>
<td></td>
<td>Extract</td>
<td>by deaggregation:</td>
</tr>
<tr>
<td></td>
<td>Data object</td>
<td>Deaggregate Objects (page 69)</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>by read/write operations:</td>
</tr>
<tr>
<td></td>
<td>Data objects in the shared database</td>
<td>Database Management (page 92)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>by accessing individual copy:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Concurrent Access (page 95)</td>
</tr>
<tr>
<td></td>
<td>Prioritize</td>
<td>by testing comparative data availability:</td>
</tr>
<tr>
<td></td>
<td>Execution of two tasks</td>
<td>Prioritized Execution (page 85)</td>
</tr>
<tr>
<td></td>
<td>Inspect</td>
<td>by (non)containment testing:</td>
</tr>
<tr>
<td></td>
<td>Collection of data objects</td>
<td>Containment Testing (page 73)</td>
</tr>
<tr>
<td></td>
<td>Clear</td>
<td>by emptying every place:</td>
</tr>
<tr>
<td></td>
<td>Content of a region</td>
<td>Region Flush (page 86)</td>
</tr>
<tr>
<td></td>
<td>Reset</td>
<td>by content setting:</td>
</tr>
<tr>
<td></td>
<td>A region to a particular state</td>
<td>Content Setting (page 89)</td>
</tr>
</tbody>
</table>
The complexity of models analyzed varied in terms of places from 8 to 73, and in terms of transitions from 7 to 40. Note that the number of distinct patterns used in the models is not proportional to the parameters representing the model complexity, and varies between 2 and 12. Each of the models contained at least 2 patterns, and none of the models contained all CPN patterns. In the majority of models, some of the patterns were used multiple times, which is evident from the Number of distinct patterns and Total patterns used metrics presented in Figure 98. The difference between these metrics is larger in the hierarchical models containing multiple pages of similar design structure.

The frequency of pattern use for each of the patterns is a relative metric, calculated as the number of times a given pattern has been used in all considered models divided by the total number of patterns used in these models. The sum of relative frequencies for all models equals 1. This metric is graphically depicted in Figure 99. In the considered set of models, 19 out of the 33 CPN patterns were used. This is the consequence of the nature of the modeled processes, which are characterized by different constraints and requirements. Note that this analysis has been performed in order to get an idea about how often CPN patterns are used in practice, and not to prove that each of the patterns is indeed used. It is natural to expect that not all patterns would occur in the models analyzed as the CPN patterns address different kinds of problems and the scope of the models selected for investigation is restricted. Moreover, all of the CPN patterns have been discovered based on existing solutions and experts knowledge, thus they do originate from practice. Nevertheless their application is context-specific.

As Figure 99 illustrates, from 19 patterns used, most often used is the Non-Deterministic XOR-Split pattern (ID2). The BSD Filter pattern (ID5) has often been used for data-based control-flow routing (its relative frequency is 0.104). In the same frequency range, the Aggregate Objects pattern (ID16) was often applied in combination with the Shared Database pattern (ID28). Note that the Database Management pattern (ID29) was used for accessing each of the shared databases identified. Although the Shared Database pattern can be used to provide local, group and global visibility, no shared databases which would provide for global visibility have been identified in the models analyzed. Whereas
the shared databases providing for local visibility were very often used. In multiple cases, the data elements stored in a shared database were organized in the collection, however storages of data elements where each element is represented as a separate entity have also been identified.

Although the Aggregate Objects pattern was often used for organizing data stored in shared databases, in several models this pattern was used for queuing purposes. Of all Queue patterns (ID20) identified, several occurrences of FIFO ordering (ID21) have been determined. In other cases, no particular queuing strategy was identified where the content of a queue was replaced with new elements (the Content Setting pattern (ID27)) or where the content of the whole collection was completely reset (the Region Flush pattern (ID26)). The Containment Testing pattern (ID19) has been used in three of the analyzed models to check the (non) containment of a specific element in the collection.

The ID Matching pattern (ID8) and the ID Manager pattern (ID9) were often used in order to distinguish between the resources stored in a shared database. Furthermore, identifiers were used in combination with the Distributed Data Processing pattern (ID15) applied to distribute data for parallel processing and subsequent merging of the obtained results. In a number of models, where the Broadcasting pattern (ID14) was used, also the Distributed Data Processing pattern was applied for synchronization purposes. As no diagrams modeling data transfer protocols were analyzed, no patterns related to inter-process communication were identified.

To further explain why in Figure 99 such a scale is used, let’s consider how to calculate the relative frequency for one of the patterns. The Aggregate Objects pattern (ID16) has been used 26 times in all analyzed models. In total, 286 patterns were used in these models. Thus, the relative frequency of the Aggregate Objects pattern is: \( \frac{26}{286} = 0.09 \).

In contrast to the Aggregate Objects pattern (ID16) which belongs to a group of patterns where the data perspective dominates, the Distributed Data Processing pattern (ID15) belongs to a group of patterns where the control-flow perspectives dominates. Patterns Deterministic XOR-Split (ID1) and Non-Deterministic XOR-Split (ID2) belong to the same group, i.e. where the control-flow perspective dominates. These two patterns were used at
least as frequently as patterns Blocking State-Independent Filter (ID4) and Blocking State-Dependent Filter (ID5) which belong to a group of patterns where data and control-flow perspectives are equally important. Note that the pattern Blocking State-Dependent Filter (ID5) was used more often than the Blocking State-Independent Filter (ID4) since it is included in the solution/implementation of several other patterns (e.g., ID Manager (ID9) and Capacity-Bounding (ID18)).

Figure 100: Examples of CPN patterns identified in a model

Figure 100(a)-(d) illustrates patterns identified in one of sub-pages of the model supplied for analysis by the authors of [169]. The pattern identifiers are added to the model next...
to the constructs distinguishing the corresponding patterns. Place offered work items together with transitions offers and selects form the queue structure (pattern Queue), where objects are aggregated into a list (pattern Aggregate Objects) (Figure 100(a)). When a collection is retrieved from the offered work items place, it is updated and thus set to another value (pattern Content Setting). The guard associated with transition selects tests the presence of element wi in the collection wis (pattern Containment Testing).

Transitions selects and reject are associated with guards involving the state of the offered queue, which corresponds to the Blocking State-Dependent Filter pattern (cf. figures 100(b) and (c)). In Figure 100(c), place selected and the two outgoing transitions selects and reject, which are associated with two mutually-exclusive guard conditions, correspond to the Deterministic XOR-Split pattern. Testing the non-containment property of the offered work items place is realized by transition Reject, which corresponds to the Containment Testing pattern. In this figure two shared databases are used, i.e. the activity map and user map places. Transition completes merges the data supplied by places assigned work items and completed (the Data Merge pattern).

Finally in Figure 100(d) the data merge and data distribution constructs associated with the Distributed Data Processing pattern are visualized, which are associated with the selects transition. The data elements provided to this transition are combined and used for withdrawing an offer, and data element (u,wi) supplied to the transition is decomposed and distributed over the assigned work item and approved places.

Although we did not encounter all 33 patterns in the analyzed models, we discovered that more than a half of the patterns were applied in practice. Figure 100 shows that not all patterns are used equally frequently. Patterns with a low frequency can be seen as special-purpose patterns, while more frequently used ones can be seen as common patterns.

### 3.5 Related work

Nowadays, there is a broad understanding of what a pattern is, i.e. a relationship between a problem, a solution and a certain context. Patterns have been identified and documented in various fields and domains (as has been in detail discussed in Section 2.4).

In the context of Petri nets some initial attempts to capture patterns have been made. Earlier work by Jensen [130,144], van der Aalst [7], van Hee [119], and Reisig [185] identify some patterns in an implicit and/or fragmented manner. In [175], Petri nets are used to represent workflow and communication patterns in the context of web-services. In [89], Fabian et al. illustrate some elementary Petri net constructs for operation lists, however these concentrate only on the control-flow aspect of manufacturing processes. In [123], Heuser and Richter provide a set of constructs for modeling information systems with Petri nets. They concentrate mostly on the problems related to the bounded capacity of places, and identify several extensions to Petri nets by means of restoring and maintaining arcs required for realization of state-based problems. Note however that such extensions have no practical value since they are not supported by contemporary CPN development tools.

In [129,159], Naedele and Janneck address the need for systematic description of Petri-net modeling knowledge. They provide a starting point in demonstrating four patterns from their experience and underline the need to define domain specific presentation styles. To describe the patterns, a variant of the pattern format introduced by Gamma et al. [99] is used. Furthermore, the authors indicate that the design of a pattern language, at least for a restricted set of patterns, is one of the most important research areas.
One of the few papers, linking CPNs to patterns is [172]. However, here CPNs are merely used as an underlying representation of the dynamic object-oriented architecture and the real focus is on patterns found in concurrent software designs. The paper that is probably most closely related to our work is [109] by Gries et al. They define three patterns in terms of classical Petri nets using a pattern language similar to ours. For a given example they analyze the use of these patterns. Our work differs from these papers in at least two ways. First of all, we use CPNs (rather than classical nets) and focus on the interplay between control flow and data flow. Second, our set of patterns is more comprehensive as is illustrated by the number of patterns, classification, and relationships.

3.6 Summary

In this chapter, we have presented a set of 33 CPN patterns. The patterns collected focus on the interplay between control flow and data flow. Although expressed in a specific language, the patterns can be applied to other languages suitable for modeling dynamic systems dealing with data and concurrency.

The language and the patterns have been developed in an explorative manner. This means that we applied empirical methods to gather information, based on observation, content analysis, and simulation. In order to discover patterns we used application models, publications, tutorials, workshop materials, opinions from experts, and feedback from model developers. We applied the content analysis technique for extracting the patterns from the literature sources, and verified the correctness of models, which represent solutions for certain problems by simulating them in CPN Tools.

We do not claim that the CPN patterns gathered are complete, since they are the result of an explorative work and were not derived in a systematic manner. This means that new patterns are likely to be added to the pattern language. The implementations of CPN patterns have been made available to the CPN community in the form of a pattern library [158]. We encourage members of the CPN community to extend the catalog of patterns by including additional ones not covered here. Moreover, these patterns can serve as a language enhancing interactions between developers, by allowing them to communicate problems and solutions unambiguously.

Not only can the collection be extended, but it can also be used as a basis for automatic recognition of patterns in models. The knowledge provided in this chapter can be used for improvement of user interfaces in the CPN development tools. In particular, various implementations can be embedded in the tool to allow user automatically select a suitable pattern rather than designing it from scratch on the basis of the information provided in the CPN catalog. In order to assist users in selecting patterns from the pattern catalog we classified the patterns in several clusters and visualized relations between the patterns by means of a relationship diagram.

As it has been mentioned earlier, we chose CPNs as a formal foundation for specifying the semantics of patterns related to different perspectives of PAISs for its ability to explicitly model concurrent behavior, state, data, strict formal semantics and an intuitive graphical notation. Whenever applicable, in the subsequent chapters we will indicate which CPN patterns have been used when designing CPN diagrams.
Part II

Patterns for Process-Aware Information Systems
Part II of this thesis refines the conceptual foundation for PAISs. It does this by undertaking a comprehensive revision of the patterns related to the control-flow perspective of a business process, identifying requirements relevant to the interaction of a business process with the external environment, and identifying requirements for improved process flexibility that are necessary when dealing with changes imposed by the operational environment.

![Diagram](image)

**Figure 101:** Scope of research: Part II

Figure 101 illustrates the scope of work presented in this part of the thesis. In particular, we address the following perspectives:

- **Control-Flow**: the structure of the process definition describing the order of tasks and relations between them;
- **Service-Interaction**: the order of message interactions between different processes, relations between them, and the manner in which the messages exchanged are processed and correlated;
- **Process Flexibility**: the ability to support foreseen and unforeseen changes in the environment that may impact the operation of a process by adapting the structure of the associated process definition.

In order to avoid the potential for misinterpretations we formally define the semantics of the requirements identified for each of these perspectives in the form of CPNs. While doing so, we indicate which of the CPN patterns presented in Chapter 3, have been used.

This part of the thesis is organized as follows: Chapters 4, 5, and 6 systematically describe requirements for the control-flow, service-interaction, and process flexibility perspectives using the pattern-based approach. Note that although data and resource perspectives are also relevant for understanding the requirements for PAISs, they are not addressed in this thesis as they have been examined by Russell and presented in form of data patterns and resource patterns in [194].
The perspectives considered complement each other in the following way. The workflow control-flow patterns described in Chapter 4 concentrate on tasks and ordering relations between them forming a set of recurring constructs. The control-flow patterns on their own are incapable of describing the behavior of interactive business processes. The interaction of a process with an external environment, for consuming or providing a particular service, has to be specified by means of the service interaction patterns described in Chapter 5. The description of the control-flow of a business process by means of the workflow control-flow patterns is based on the assumption that all future process instances will be based on a predefined process definition. Such a definition is constructed assuming that all possible execution paths are known in advance and that no changes in an external environment will occur or impact on original process definition. This makes the process definition relatively rigid and incapable of dealing with changing requirements imposed by an external environment of a highly-volatile nature. Effective business processes must be able to accommodate changes in the environment in which they operate. The ability to encompass such changes, termed process flexibility, is presented in Chapter 6 using a set of process flexibility patterns.
Chapter 4

Workflow Control-Flow Patterns

In this chapter, we concentrate on the control-flow aspects of a business process, i.e. the tasks constituting a process and the control-flow dependencies between them. In particular, we present the fundamental constructs for describing the structure of a process model in the form of workflow control-flow patterns (Section 4.1). For each of the patterns, we provide a formal semantics using CPNs, and establish a set of evaluation criteria that provide an objective basis for assessing the support of a control-flow pattern by a particular offering. The control-flow patterns operate at a conceptual level, i.e. they specify recurring generic constructs relevant to process structure and enactment in an abstract sense, consequently they do not provide much guidance in regard to their actual realization.

Individual workflow offerings provide various sets of constructs which can be used to realize particular patterns. However, in order to describe how these constructs work, we need to define their operational semantics in a precise way. Additionally, it would be helpful if this were done in the context of a common framework. In the absence of a suitable framework, in Section 4.2 we concentrate on the core constructs used for modeling business processes that are encountered in any PAIS, and propose a means for capturing and comparing the functionality of these constructs in a language-independent way. In Section 4.3, we assess the operational support of control-flow patterns across a series of selected PAISs. Finally, we describe related work and conclusions in sections 4.4 and 4.5 respectively.

4.1 Revisiting the control-flow patterns

The work presented in this section was conducted as part of the Workflow Patterns Initiative (cf. Section 1.3.2) that was established with the aim of providing a conceptual basis for process technology. In line with this initiative, a comprehensive survey of selected workflow offerings available at that time was undertaken, which resulted in 20 generic constructs being identified and presented in the form of patterns. The patterns identified provide a means of reasoning about the implementation of certain business requirements in a particular PAIS, provide a tool for benchmarking distinct offerings, and offer a basis for language and tool development [230].

Examination of the original 20 control-flow patterns in a variety of industrial offerings
has revealed that some of the pattern definitions were subject to ambiguous interpretation and that some of the patterns were missing. In order to determine whether the original 20 control-flow patterns provide comprehensive coverage of the constructs encountered in the control-flow perspective, we have revisited the original control-flow patterns in [193]. Consequently, 23 new patterns, including new patterns and specializations of already existing ones, were identified. To remove any potential ambiguities in pattern definitions, we formalized the semantics of the patterns in the form of CPN models. The formalization is based on a set of the context assumptions explicitly listed in Section 4.1.1. The identification of the new patterns and their formalization has been done in cooperation with Russell et al. [194]. This thesis adds to this work by reclassifying the collection of patterns obtained and by defining different approaches to their operationalization.

Aiming at consistent pattern classification, we examined the existing pattern groups in order to identify how the new patterns could be positioned. The new patterns did not fit well with the original classification, i.e. not all patterns were encompassed by the existing groupings. This identified the need for a more intuitive approach to grouping patterns based on their actual purpose rather than the manner in which they were realized. With this aim in mind, we propose a new pattern classification in Section 4.1.2, and reorder the patterns respectively in the pattern catalog in Section 4.1.3.

In Section 2.2, we discussed different pattern formats that can be used for documenting patterns. In this chapter, we will describe each of the control-flow patterns in terms of the following format:

- **description**: a summary of its functionality;
- **examples**: illustrative examples of its usage;
- **motivation**: the rationale for the use of the pattern;
- **overview**: an explanation of its operation including a detailed operational definition where necessary;
- **context**: other conditions that must hold in order for the pattern to be used in a process context;
- **implementation**: how the pattern is typically realized in practice;
- **issues**: problems potentially encountered when using the pattern;
- **solutions**: how these problems can be overcome; and
- **evaluation criteria**: the conditions that an offering must satisfy in order to be considered to support the pattern.

The operation of each pattern is described in terms of a CPN model, which represents the actual semantics for the pattern rather than its exact realization. Thus the direct replication of the CPN model in a specific process language does not necessarily demonstrate support for the pattern. In order for an offering to be considered to support a pattern, specific evaluation criteria associated with the pattern have to be satisfied. Before we proceed with reordering the control-flow patterns, first we describe the relevant context assumptions.

### 4.1.1 Context assumptions

To describe the context in which the patterns can be used and operational semantics of the revised pattern definitions, as far as possible, each pattern is illustrated using the CPN formalism. This allows us to provide a precise description of each pattern that is both deterministic and executable. Whenever applicable, we identify the CPN patterns (described in Chapter 3) we used when describing the semantics of each of the control-flow
Section 4.1 Revisiting the control-flow patterns

Throughout this thesis, a description of a process at the type level is termed a process definition. A process definition is composed of a number of tasks together with constraints on their execution order. To put a process to work, its process description needs to be instantiated. An executing instance of process definition is called a process instance. Multiple process instances may run simultaneously. These process instances are independent of each other and may exhibit differing execution paths. Depending on the context conditions associated with each of the process instances, different decisions can be taken at various decision points for individual process instances. The information about the execution of different process instances, i.e. the tasks and the order in which they were executed, can be compared using execution traces. Typically, the execution traces obtained for different process instances created on the basis of a single process description differ.

The execution of a process instance begins with a start task and finishes with the end task. A task corresponds to a single unit of work. We distinguish atomic, composite and multiple-instance tasks. A task that cannot be decomposed into smaller parts and described in terms of other tasks is called an atomic task. The execution of an atomic task results in only a single task instance being initiated. A composite task is a task which has an implementation defined in terms of a sub-process. A sub-process is a process within a process that may be executed on its own right. As with any other process, a sub-process also has a start task and an end task. When a composite task is started, the point of control is passed to the start task in the corresponding sub-process, and at its completion, the thread of control is passed back to the composite task. The use of sub-processes allows for hierarchical decomposition and reuse of the process definitions. A multiple-instance task is a task, whose execution results in the initiation of one or more distinct task instances that run concurrently and independently of each other within the same process instance. The multiple-instance task is considered to be complete when a nominated number of the task instances have completed or the relevant completion condition is satisfied. We use the generic term task to refer to an atomic task, and whenever a composite or multiple-instance task needs to be used, we will refer to the task type explicitly. We use the term process fragment to refer to a collection of tasks and corresponding routing elements in a process definition.

All of the CPN models used in this chapter are based on a set of assumptions described below. For each of the models, we adopt a notation in which input places are labeled i1...in, output places are labeled o1...on, internal places are labeled p1...pn and transitions are labeled A...Z. In the case where either places or transitions serve a more significant role in the context of the pattern, they are given more meaningful names (e.g., buffer or anti-place). In general, transitions are intended to represent tasks or activities in processes, and places are the preceding and subsequent states which describe when the activity can be enabled and what the consequences of its completion are.

Unless stated otherwise, we assume that the tokens flowing through a CPN model that signify control-flow are typed piID (short for “process instance ID”) and that each executing process instance has a distinct identifier. For most patterns, the assumption is also made that the model is safe, i.e. that each place in the model can only contain at most one token (i.e. one thread of control for each case currently being executed). This provides clarity in regard to the way in which each of the CPN models describing pattern operation are intended to function.

Safe behavior is not a mandatory quality of workflow systems. Some of the systems that we examine during the course of this chapter, do implement safe process models whilst
others do not. Where a system does provide a safe execution environment, this is typically achieved in one of two ways: either (1) during execution, the state of a given case is never allowed to transition into an unsafe state. This is the approach adopted by COSA, which blocks an activity’s execution where it has a token in the place immediately after it and allowing it to execute could potentially result in an unsafe state (i.e. the following place having two tokens in it). The other alternative (2) is to detect any unsafe situations that may arise and migrate them to safe states. An example of this is the strategy employed by Staffware where any additional triggering received by an activity that is currently executing are coalesced into the same thread of control resulting in a single thread of control being delivered to outgoing branches when the activity completes.

These variations in the ways in which distinct offerings implement concurrency with a process instance lead to differences in the ranges of patterns that they are able to support and the means by which they realize them. In the following sections, we will provide a precise description of each pattern and examine the differences in the approaches to operationalization of constructs used by workflow offerings for the pattern realization.

4.1.2 Classification of control-flow patterns

In this section, we classify the control-flow patterns based on the similarity of problems that they address. In total, the control-flow patterns can be divided into nine classes: branching patterns, synchronization and merging patterns, repetition patterns, multiple instances patterns, concurrency control patterns, triggering patterns, cancelation and completion patterns, and termination patterns. An overview of the control-flow patterns classification is given in Table 4.1. We describe each of the classes identified and related patterns in detail below.

The class of branching patterns describes situations where a single thread of control diverges into one or more branches, on the basis of a preceding decision. In this context, the thread of control is either split into multiple threads of control on the same branch or into distinct threads of control distributed over one or more branches. Depending on the requirements that have to be satisfied at the moment of branching, we can distinguish five sub-classes No-Split, AND-Split, XOR-split, OR-split or Thread-split respectively. Each of these split types may exhibit different variants. The situation where a task is linked to a single outgoing branch without the thread of control being diverged into multiple threads, is characterized by the absence of branching.

The class of synchronization and merging patterns describes situations where threads of control obtained by diverging a single thread of control into one or more branches (corresponding to the branching patterns) need to be coalesced into a single thread of control. Depending on the requirements that have to be satisfied at the moment of synchronization, we distinguish five types of join construct: AND-join, Partial join, XOR-join, OR-join and thread-join.

The class of repetition patterns describes situations where there is a need for repetitive execution of a task or a process fragment. Three different approaches to the realization of repetitive execution are described by the repetition patterns: a flexible loop with multiple entry and exit-points, more restrictive forms of iteration such as while and repeat loops, and repetitive task execution which is based on self-invocation.

The class of multiple instance patterns describes situations where one or more instances of the same task need to be created and executed either sequentially or concurrently. Depending on whether there is a need to synchronize created task instances on completion,
<table>
<thead>
<tr>
<th>Pattern class/subclass</th>
<th>ID</th>
<th>Pattern name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branching</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No-Split</td>
<td>WCF-1</td>
<td>Sequence (p.123)</td>
</tr>
<tr>
<td>AND-splits</td>
<td>WCF-2</td>
<td>Parallel Split (p.124)</td>
</tr>
<tr>
<td>XOR-splits</td>
<td>WCF-3</td>
<td>Exclusive Choice (p.125)</td>
</tr>
<tr>
<td></td>
<td>WCF-4</td>
<td>Deferred Choice (p.127)</td>
</tr>
<tr>
<td>OR-splits</td>
<td>WCF-5</td>
<td>Multi-Choice (p.128)</td>
</tr>
<tr>
<td>Thread-splits</td>
<td>WCF-6</td>
<td>Thread Split (p.130)</td>
</tr>
<tr>
<td><strong>Synchronization and merging</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND-joins</td>
<td>WCF-7</td>
<td>Synchronization (p.131)</td>
</tr>
<tr>
<td></td>
<td>WCF-8</td>
<td>Generalized AND-join (p.133)</td>
</tr>
<tr>
<td>Partial joins</td>
<td>WCF-9</td>
<td>Structured Discriminator (p.135)</td>
</tr>
<tr>
<td></td>
<td>WCF-10</td>
<td>Blocking Discriminator (p.137)</td>
</tr>
<tr>
<td></td>
<td>WCF-11</td>
<td>Canceling Discriminator (p.138)</td>
</tr>
<tr>
<td></td>
<td>WCF-12</td>
<td>Structured Partial Join (p.141)</td>
</tr>
<tr>
<td></td>
<td>WCF-13</td>
<td>Blocking Partial Join (p.143)</td>
</tr>
<tr>
<td></td>
<td>WCF-14</td>
<td>Canceling Partial Join (p.144)</td>
</tr>
<tr>
<td>XOR-joins</td>
<td>WCF-15</td>
<td>Simple Merge (p.145)</td>
</tr>
<tr>
<td></td>
<td>WCF-16</td>
<td>Multi-Merge (p.146)</td>
</tr>
<tr>
<td>OR-joins</td>
<td>WCF-17</td>
<td>Structured Synchr. Merge (p.148)</td>
</tr>
<tr>
<td></td>
<td>WCF-18</td>
<td>Local Synchronizing Merge (p.150)</td>
</tr>
<tr>
<td></td>
<td>WCF-19</td>
<td>General Synchronizing Merge (p.151)</td>
</tr>
<tr>
<td>Thread-joins</td>
<td>WCF-20</td>
<td>Thread Merge (p.153)</td>
</tr>
<tr>
<td><strong>Repetition</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-21</td>
<td>Arbitrary Cycles (p.154)</td>
</tr>
<tr>
<td></td>
<td>WCF-22</td>
<td>Structured Loop (p.155)</td>
</tr>
<tr>
<td></td>
<td>WCF-23</td>
<td>Recursion (p.156)</td>
</tr>
<tr>
<td><strong>Multiple Instances (MI)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-24</td>
<td>MI without synchronization (p.158)</td>
</tr>
<tr>
<td></td>
<td>WCF-25</td>
<td>MI with a priori DTK (p.160)</td>
</tr>
<tr>
<td></td>
<td>WCF-26</td>
<td>MI with a priori RTK (p.161)</td>
</tr>
<tr>
<td></td>
<td>WCF-27</td>
<td>MI without a priori RTK (p.163)</td>
</tr>
<tr>
<td></td>
<td>WCF-28</td>
<td>Static part. join for MI (p.164)</td>
</tr>
<tr>
<td></td>
<td>WCF-29</td>
<td>Canceling part. join for MI (p.166)</td>
</tr>
<tr>
<td></td>
<td>WCF-30</td>
<td>Dynamic part. join for MI (p.167)</td>
</tr>
<tr>
<td><strong>Concurrency Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-31</td>
<td>Interleaved Routing (p.169)</td>
</tr>
<tr>
<td></td>
<td>WCF-32</td>
<td>Interleaved Parallel Routing (p.170)</td>
</tr>
<tr>
<td></td>
<td>WCF-33</td>
<td>Critical Section (p.172)</td>
</tr>
<tr>
<td></td>
<td>WCF-34</td>
<td>Milestone (p.173)</td>
</tr>
<tr>
<td><strong>Triggering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-35</td>
<td>Transient Trigger (p.174)</td>
</tr>
<tr>
<td></td>
<td>WCF-36</td>
<td>Persistent Trigger (p.176)</td>
</tr>
<tr>
<td><strong>Cancellation and completion</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-37</td>
<td>Cancel Task (p.178)</td>
</tr>
<tr>
<td></td>
<td>WCF-38</td>
<td>Cancel Region (p.179)</td>
</tr>
<tr>
<td></td>
<td>WCF-39</td>
<td>Cancel Case (p.181)</td>
</tr>
<tr>
<td></td>
<td>WCF-40</td>
<td>Cancel MI Task (p.182)</td>
</tr>
<tr>
<td></td>
<td>WCF-41</td>
<td>Complete MI Task (p.184)</td>
</tr>
<tr>
<td><strong>Termination</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WCF-42</td>
<td>Implicit Termination (p.187)</td>
</tr>
<tr>
<td></td>
<td>WCF-43</td>
<td>Explicit Termination (p.188)</td>
</tr>
</tbody>
</table>
and the moment at which it is known how many instances need to be created, several situations are distinguished. The thread of control can potentially be passed to a subsequent task immediately after multiple instances have been initiated, or after all existing instances have completed. In some situations, the flow of control needs to be passed on, prior to completion of all instances for a given task. A partial join between instances of such a task allows a subsequent task to be triggered once a threshold of concurrent task instances have completed or a specified completion condition has been met.

The class of concurrency control patterns describe situations where the restrictions are set on concurrent execution of branches, which otherwise would execute in parallel. In this context, two types of restrictions may be considered: first, where none of the branches may execute simultaneously, and second, where one branch may not proceed if the other branch is not in a certain state.

The class of triggering patterns describe situations where the moment at which a work item commences needs to be synchronized with a signal (or trigger) from external environment. In this context, triggers of both a persistent and a transient nature are considered.

The class of cancellation and completion patterns describe situations where an individual task, an arbitrary group of tasks in a process, or a complete process instance needs to be canceled during execution. Additionally, the multiple instances task is considered to be a special case, for which both the ability to cancel and to force completion during execution are recognized.

The class of termination patterns define two alternative approaches to recognizing process completion. Both implicit and explicit process completion are covered by this class.

### 4.1.3 Catalog of control-flow patterns

In this section, we present a catalog of the control-flow patterns using the classification described in the previous section as a guideline.

**Class: Branching patterns**

Branching patterns can be divided in five sub-classes. Patterns characterized by diverging a thread of control into multiple threads of control over the same branch belong to the thread-split sub-class. Patterns where a thread of control is split over all, some or only one of the outgoing branches correspond to AND-split, OR-split and XOR-split sub-classes respectively. The absence of branching (corresponding to No-split sub-class) represents the sequence of tasks, which is the basic control-flow construct that can be encountered in any workflow offering (known as the Sequence pattern). This and other branching patterns are described in detail below.

**Sub-class: No-split** The absence of split behavior is described by the Sequence pattern (note that it is not really a branching pattern). The Sequence pattern describes situations where two tasks are consecutively executed one after another in the specified order. Although sequential execution (as described by the Sequence pattern) is supported by all offerings, actual realizations of this pattern may have differing semantics. The manner in which a task reacts to triggers arriving during its execution determines whether the input provided will be inconsequentially consumed or ignored until task completion. In a similar vein, a task that provides input to a subsequent task may block until the latter completes or it may remain unblocked and able to send inputs.
In Section 4.2, we introduce a graphical notation which provides a means of visualizing the differences between various approaches to producing/consuming inputs during a task execution. On page 208, we revisit the Sequence pattern to illustrate how the proposed graphical notation can be used to depict the differences in its implementation in two offerings (Oracle BPEL PM and Staffware).

Pattern WCF-1 (Sequence)

Description A task in a process is enabled after the completion of a preceding task in the same process.

Examples

– The verify-account task executes after the credit card details have been captured.
– A receipt is printed after the train ticket is issued.

Motivation The Sequence pattern serves as the fundamental building block for processes. It is used to construct a series of consecutive tasks which execute in turn one after the other. Two tasks form part of a Sequence if there is a control-flow edge from one of them to the next which has no guards or conditions associated with it.

Overview Figure 102 illustrates the Sequence pattern using CPN diagram.

![Figure 102: Sequence pattern](image)

Context There is one context condition associated with this pattern: an instance of the Sequence pattern cannot be started again until it has completed execution of the preceding thread of control (i.e. all places such as p1 in the Sequence must be safe).

Implementation The Sequence pattern is widely supported and all of the offerings examined directly implement it.

Issues Although all of the offerings examined implement the Sequence pattern, there are however subtle variations in the manner in which it is supported. In the main, these differences center on how individual offerings deal with concurrency within a given process instance and also between distinct process instances. In essence these variations are characterized by whether the offering implements a safe process model or not. In CPN terms, this corresponds to whether each of the places in the process model such as that in Figure 102 are 1-bounded (i.e. can only contain at most one token for a process instance) or not.

Solutions This issue is handled in a variety of differing ways. BPMN, XPDL and UML 2.0 Activity Diagrams assume the use of a “token-based” approach to managing process instances and distinguishing between them, although no details are given as to how this actually occurs. Further, although individual tokens are assumed to be conserved during execution of a process instance, it is possible for a task, split or join construct to actually add or remove tokens during execution beyond what would reasonably be expected.

1A detailed review of patterns support that has been conducted in fourteen distinct offerings including workflow systems (Staffware, WebSphere MQ, COSA, iPlanet, SAP Workflow and FileNet), case handling systems (FLOWer), business process modelling languages (BPMN, UML 2.0 Activity Diagrams and EPCs) and business process execution languages and tools (BPEL4WS, WebSphere BPEL, Oracle BPEL and XPDL) can be found in [193].
Staffware simply ignores the issue and where a step receives two threads (or more) of execution at the same time, they are simply coalesced into a single firing of the step (thus resulting in race conditions). COSA adopts a prevention strategy, both by implementing a safe process model and also by disabling the task(s) preceding a currently enabled task and not allowing the preceding task(s) to fire until the subsequent task has completed.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which supports an explicit representation of dependency (e.g., directed arc) between two tasks which specifies the execution sequence.

### Sub-class: AND-split

The AND-split construct characterizes situations where a single thread of control diverges into multiple independent branches. This branching is unconditional, i.e. it results in the enablement of all subsequent branches concurrently as described by the Parallel Split pattern.

---

**Pattern WCF-2 (Parallel Split)**

**Description** The divergence of a branch into two or more parallel branches each of which execute concurrently.

**Examples**
- After completion of the capture enrolment task, run the create student profile and issue enrolment confirmation tasks simultaneously.
- Once the customer has paid for the goods, pack them and issue a receipt.

**Motivation** The Parallel Split pattern allows a single thread of execution to be split into two or more branches which can execute tasks concurrently. These branches may or may not be re-synchronized at some future time.

**Overview** Figure 103 illustrates the implementation of the Parallel Split. After task A has completed, two distinct threads of execution are initiated and tasks B and C can proceed concurrently. The CPN model of this pattern resembles the implementation of the Broadcasting CPN pattern (cf. page 63), where the main focus is on the concurrent distribution of data between multiple targets. However, the purpose of this CPN diagram is not to distribute data over multiple branches, but to split the thread of control and enable them concurrently.

![Figure 103: Parallel Split pattern](image)

**Context** There are no specific context conditions for this pattern.

**Implementation** The Parallel Split pattern is implemented by all of the offerings examined. It may be depicted either explicitly or implicitly in process models. Where it is
Section 4.1 Revisiting the control-flow patterns

represented explicitly, a specific construct exists for the Parallel Split with one incoming edge and two or more outgoing edges. Where it is represented implicitly, this can be done in one of two ways: either (1) the edge representing control-flow can split into two (or more) distinct branches or (2) the task after which the Parallel Split occurs has multiple outgoing edges which do not have any conditions associated with them or where it does these conditions always evaluate to true.

Of the offerings examined, Staffware, WebSphere MQ, FLOWer, COSA and iPlanet represent the pattern implicitly. SAP Workflow, EPCs and BPEL\(^2\) do so with explicit branching constructs. UML 2.0 ADs, BPMN and XPDL allow it to be represented in both ways.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by the provision of a construct (either implicit or explicit) that allows the thread of control at a given point in a process to be split into two or more concurrent branches.

---

**Sub-class: XOR-Split**

The XOR-split construct characterizes situations where a single thread of control is passed to exactly one branch out of several available ones. Two patterns Exclusive Choice and Deferred Choice belong to this split type. The major difference between them relates to the moment at which the selection of a branch occurs. For the Exclusive Choice pattern, the selection of a branch is conditional and the options associated with the choice are explicitly predefined, while the selection of a branch in the Deferred Choice pattern is postponed until the latest possible time when sufficient information from the operating environment becomes available and one of the alternative branches is started.

**Pattern WCF-3 (Exclusive Choice)**

**Description** The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.

**Examples**
- Depending on the volume of earth to be moved, either the dispatch-backhoe, despatch-bobcat or despatch-D9-excavator task is initiated to complete the job.
- After the review election task is complete, either the declare results or the recount votes task is undertaken.

**Motivation** The Exclusive Choice pattern allows the thread of control to be directed to a specific (subsequent) task depending on the outcome of a preceding task, the values of elements of specific data elements in the process, the results of an expression evaluation or some other form of programmatic selection mechanism.

**Overview** The behavior of the Exclusive Choice pattern is illustrated by the CPN model in Figure 104. Depending on the results of the cond expression, the thread of control is

---

\(^2\)In general, the two BPEL implementations examined – WebSphere BPEL (which is part of WebSphere Process Server) and Oracle BPEL – provide a relatively faithful implementation of the BPEL 1.1 specification hence the evaluation results are identical for all three offerings. For this reason they are not listed individually in this chapter unless there is a variation between them.
either routed to task B or C. The realization of this CPN model is similar to Solution 2 of the Deterministic XOR-Split CPN pattern (cf. page 36) with the only difference being that conditions used for branch selection are not based on the actual data passed to the A transition, but rather represent abstractions of it.

![Diagram](image)

**Figure 104:** Exclusive Choice pattern

**Context** There is one context condition associated with this pattern: the mechanism that evaluates the Exclusive Choice is able to access any required data elements or other necessary resources when determining which of the outgoing branches the thread of control should be routed to.

**Implementation** Similar to the Parallel Split pattern, the Exclusive Choice pattern can either be represented explicitly via a specific construct or implicitly via disjoint conditions on the outgoing control-flow edges of a task. Staffware, SAP Workflow, XPDL, EPCs and BPMN provide explicit XOR-split constructs. In the case of Staffware, it is a binary construct whereas other offerings support multiple outgoing arcs. BPMN and XPDL provide for multiple outgoing edges as well as a default arc. Each edge (other than the default arc) has a condition associated with it and there is also the potential for defining the evaluation sequence but only one condition can evaluate to true at runtime. There is no provision for managing the situation where no default is specified and none of the branch conditions evaluate to true nor where more than one branch condition evaluates to true (simultaneously) and no evaluation sequence is specified. SAP Workflow provides three distinct means of implementing this pattern: (1) based on the evaluation of a Boolean expression one of two possible branches is chosen, (2) one of multiple possible branches is chosen based on the value of a specific data element (each branch has a nominated set of values which allow it to be selected and each possible value is assigned to exactly one branch) and (3) based on the outcome of a preceding task, a specific branch is chosen (a unique branch is associated with each possible outcome). UML 2.0 ADs also provide a dedicated split construct although it is left to the auspices of the designer to ensure that the conditions on outgoing edges are disjoint (e.g., the same construct can be used for OR-splits as well). Likewise EPCs support the pattern in a similar fashion. The other offerings examined – WebSphere MQ, FLOWer, COSA, iPlanet and BPEL – represent the pattern implicitly, typically via conditions on the outgoing control-flow edges from a task which must be specified in such a way that they are disjoint.

**Issues** One of the difficulties associated with this pattern is ensuring that precisely one outgoing branch is triggered when the Exclusive Choice is executed.

**Solutions** The inclusion of default outgoing arcs on XOR-split constructs is an increasingly common means of ensuring that an outgoing branch is triggered (and hence the thread of

---

3 As a general comment, the notation $x'c$ on an input arc to a CPN transition means that $x$ instances of token c are required for the input arc to be enabled.
control continues in the process instance) when the XOR-split is enabled and none of the conditions on outgoing branches evaluate to true. An associated issue is ensuring that not more than one branch is triggered. There are two possible approaches to dealing with this issue where more than one of the arc conditions will potentially evaluate to true. The first of these is to randomly select one of these arcs and allow it to proceed whilst ensuring that none of the other outgoing arcs are enabled. The second option, which seems more practical, is to assign an evaluation sequence to the outgoing arcs which defines the order in which arc conditions will be evaluated. The means of determining which arc is triggered then becomes one of evaluating the arc conditions in sequential order until one evaluates to true. This arc is then triggered and the evaluation stops (i.e. no further arcs are triggered). In the event that none evaluate to true, then the default arc is triggered.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

The *Exclusive Choice* pattern describes the situation where one out of several (explicitly-predefined) options needs to be selected at the moment the choice construct is encountered. In some situations, the routing decision may not be taken immediately, because it requires some input from the operating environment. The deferral of the choice until the moment when sufficient information becomes available is described by the *Deferred Choice* pattern.

**Pattern WCF-4 (Deferred Choice)**

**Description** A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches, i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

**Examples**
- At the commencement of the *Resolve complaint* process, there is a choice between the *Initial customer contact* task and the *Escalate to manager* task. The *Initial customer contact* is initiated when it is started by a customer services team member. The *Escalate to manager* task commences 48 hours after the process instance commences. Once one of these tasks is initiated, the other is withdrawn.
- Once a customer requests an *airbag shipment*, it is either picked up by the *postman* or a *courier driver* depending on who can visit the customer site first.

**Motivation** The *Deferred Choice* pattern provides the ability to defer the moment of choice in a process, i.e. the moment as to which one of several possible courses of action should be chosen is delayed to the last possible time and is based on factors external to the process instance (e.g., incoming messages, environment data, resource availability, timeouts etc.). Up until the point at which the decision is made, any of the alternatives presented represent viable courses of future action.

**Overview** The operation of this pattern is illustrated in Figure 105. The moment of choice is signified by place $p_1$. Either task $B$ or $C$ represent valid courses of action but only one of them can be chosen. The realization of this CPN model is based on the unguarded implementation variant of the *Non-Deterministic XOR-Split* CPN pattern (see page 38). Note however that we abstract from the role of data in the selection of an appropriate branch.
Context There is one context condition associated with this pattern: only one instance of the Deferred Choice can operate at any time (i.e. place p1 is assumed to be safe).

Implementation This is a complex pattern and it is interesting to see that only those offerings that can claim a token-based underpinning (or something analogous to it\(^4\)) are able to successfully support it. COSA is based on a Petri net foundation and can implement the pattern in much the same way as it is presented in Figure 105. BPEL provides support for it via the <pick> construct, BPMN through the event-based gateway construct, XPDL using the XOR-ENTITY-split construct and UML 2.0 ADs using a ForkNode followed by a set of AcceptSignal actions, one preceding each action in the choice. In the case of the latter three offerings, the actual choice is made based on message-based event interactions. FLOWer does not directly provide a notion of state but it provides several ways of supporting this pattern through the use of user and system decisions on plan types and also by using arc guards that evaluate to NIL in conjunction with data elements to make the decision as to which branch is selected. FileNet provides partial support for the pattern as it only allows for withdrawal of timer-based branches not of all branches other than the one selected for execution.

Issue None identified.

Solution N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. If there are any restrictions on which branches can be selected or withdrawn, then the offering is rated as having partial support.

Sub-class: OR-split The OR-split construct characterizes situations where a branch diverges into multiple outgoing branches and the thread of control is passed to at least one of the outgoing branches. The selection of branches is conditional, and potentially results in subsequent enabling of one, several or all available branches as described by the Multi-Choice pattern.

Pattern WCF-5 (Multi-Choice)

Description The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

\(^4\)The use of dead path elimination when evaluating link enablement in BPEL is analogous to the use of true/false token as a means of propagating control-flow in a Petri net sense.
Example
– Depending on the nature of the emergency call, one or more of the *despatch-police*, *despatch-fire-engine* and *despatch-ambulance* tasks is immediately initiated.
– Depending on the knowledge of a candidate, either a theory exam, a vehicle-driving exam or both have to be undertaken.

Motivation The *Multi-Choice* pattern provides the ability for the thread of execution to be diverged into several concurrent threads in distinct branches on a selective basis. The decision as to whether to pass the thread of execution to a specific branch is made at runtime. It can be based on a variety of factors including the outcome of a preceding task, the values of elements of specific data elements in the process, the results of evaluating an expression associated with the outgoing branch or some other form of programmatic selection mechanism. This pattern is essentially an analogue of the *Exclusive Choice* pattern (WCF-3) in which multiple outgoing branches can be enabled.

Overview The operation of the *Multi-Choice* pattern is illustrated in Figure 106. After task *A* has been triggered, the thread of control can be passed to one or both of the following branches depending on the evaluation of the conditions associated with each of them. The realization of this CPN model is based on the *OR-Split* CPN pattern (cf. page 40) with the only difference being that the conditions used for branch selection are not based on the actual data passed to the *A* transition, but rather represent abstractions of it.

![Figure 106: Multi-Choice pattern](image)

Context There is one context condition associated with this pattern: the mechanism that evaluates the *Multi-Choice* is able to access any required data elements or necessary resources when determining which of the outgoing branches the thread of control should be routed.

Implementation As with other branching and merging constructs, the *Multi-Choice* pattern can either be represented implicitly or explicitly. WebSphere MQ captures it implicitly via (non-disjoint) conditions on outgoing arcs from a process or block construct, COSA and iPlanet do much the same via overlapping conditions on outgoing arcs from tasks and outgoing routers respectively. Both COSA and iPlanet allow for relatively complex expressions to be specified for these outgoing branches and iPlanet also allows for procedural elements to form part of these conditions. The modelling and business process execution languages examined tend to favor the use of explicit constructs for representing the pattern: BPEL via conditional links within the `<flow>` construct, UML 2.0 ADs via the ForkNode with guards conditions on the outgoing arcs and EPCs via textual notations to the OR-split construct. BPMN and XPDL provide three alternative representations including the use of an implicit split with conditions on the arcs, an OR-split or a complex gateway.

Issues Two issues have been identified with the use of this pattern. First, as with the *Exclusive Choice*, an issue that also arises with the use of this pattern is ensuring that at
least one outgoing branch is selected from the various options available. If this is not the case, then there is the potential for the process to stall. Second, where an offering does not support the Multi-Choice construct directly, the question arises as to whether there are any indirect means of achieving the same behavior.

**Solutions** With respect to the first issue, the general solution to this issue is to enforce the use of a default outgoing arc from a Multi-Choice construct which is enabled if none of the conditions on the other outgoing arcs evaluate to true at runtime. For the second issue, a work-around that can be used to support the pattern in most offerings is based on the use of an AND-split immediately followed by an (binary) XOR-split in each subsequent branch. Another is the use of an XOR-split with an outgoing branch for each possible task combination, e.g., a Multi-Choice construct with outgoing branches to tasks A and B would be modeled using an XOR-split with three outgoing branches – one to task A, another to task B and a third to an AND-split which then triggered both tasks A and B. Further details on these transformations are presented by van der Aalst et al. [12].

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. Note that the work-around based on XOR-splits and AND-splits is not considered to constitute support for this pattern as the decision process associated with evaluation of the Multi-Choice is divided across multiple split constructs.

---

**Sub-class: Thread-split** The thread-split construct characterizes situations where a single thread of control is not split into several branches, but rather into multiple threads of control in the same branch as described by the Thread Split pattern.

**Pattern WCF-6 (Thread Split)**

**Description** At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.

**Example**
- At the completion of the confirm paper receipt task, initiate three instances of the subsequent independent peer review task.
- At the birth of a child, prepare 10 post-cards for notifying close relatives.

**Motivation** This pattern provides a means of triggering multiple execution threads along a branch within a given process instance. It can be seen as a counterpart of the Thread Merge pattern (WCF-20) which merges multiple execution threads along the same branch. Unless used in conjunction with the Thread Merge pattern, the execution threads will run independently to the end of the process.

**Overview** The operation of this pattern is illustrated in Figure 107. Note that numinsts indicates the number of threads to be created.

![Figure 107: Thread Split pattern](image)

**Context** There is one context consideration for this pattern: the number of threads needing to be created (i.e. numinsts) must be known in advance.
Implementation As with the Thread Merge pattern, implementation of this pattern implies that an offering is able to support the execution of processes in a non-safe context. This rules out the majority of the offerings examined from providing any tractable forms of implementation. BPMN and XPDL provide direct support for the pattern by allowing the quantity of tokens flowing down the outgoing sequence flow from a task at its conclusion to be specified. UML 2.0 ADs allow a similar behavior to be achieved through the use of multiple outgoing edges from a task to a MergeNode which then directs the various initiated threads of control down the same branch. BPEL indirectly allows the same effect to be achieved via the <invoke> action in conjunction with suitably specified correlation sets.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. If any degree of programmatic extension is required to achieve the same behavior, then the partial support rating applies.

Class: Synchronization and Merging patterns

Synchronization and merging patterns describe situations where threads of control that have been earlier diverged over one or several branches (corresponding to the branching patterns described on page 122) need to be coalesced into a single thread of control. Patterns describing merging of several threads of control in the same branch into a single thread of control correspond to the thread-join sub-class. The AND-join, OR-join and XOR-join sub-classes describe patterns addressing synchronization of threads of control previously diverged over all, some or one of several available branches. The partial-join sub-class describes situations where a sub-set of previously enabled branches need to be merged together. Each of these synchronization types, including related patterns, are described in detail below.

Sub-class: AND-joins The AND-join construct characterizes situations where all existing threads of control associated with branches produced by a preceding AND-split pattern, have to be coalesced into a single thread of control. Two patterns Synchronization and Generalized AND-Join, correspond to this type of synchronization. The difference between these two patterns is in the operating context, i.e. the Synchronization pattern operates in a situation, where each of the incoming branches may be enabled only once, while the Generalized AND-Join pattern operates in the context where one or more branches may be triggered multiple times for the same process instance.

Pattern WCF-7 (Synchronization)

Description The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

Examples
- The despatch-goods task runs immediately after both the check-invoice and produce-invoice tasks are completed.
– *Cash-drawer reconciliation* can only occur when the store has been closed and the credit card summary has been printed.

**Motivation** *Synchronization* provides a means of reconverging the execution threads in two or more parallel branches. In general, these branches are created using the *Parallel Split* (AND-split) construct earlier in the process model. The thread of control is passed to the task immediately following the synchronizer once all of the incoming branches have completed.

**Overview** The behavior of the *Synchronization* pattern is illustrated by the CPN model in Figure 108. The pattern contains an implicit AND-join, known as the *synchronizer*, which is considered to be *activated* once it receives input on one of the incoming branches (i.e. at places p_1 or p_2). Similarly it is considered to be *reset* (and hence can be re-enabled) once input has been received on each incoming branch and the synchronizer has fired, removing these tokens. The realization of this CPN model corresponds to the synchronization part of the *Distributed Data Processing* CPN pattern (cf. page 65) with the only difference being that the focus is on merging of threads of control distributed across multiple branches, rather than on merging of data elements.

![Figure 108: Synchronization pattern](image)

**Context** This pattern has the following context condition: once the *synchronizer* has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the synchronizer (e.g., p_1 and p_2 are safe).

**Implementation** Similar to the *Parallel Split* pattern, the synchronizer can either be represented *explicitly* or *implicitly* in a process model. Staffware has an explicit AND-join construct as do SAP Workflow, EPCs, BPMN and XPDL. Other offerings – WebSphere MQ, FLOWer, COSA, iPlanet and BPEL – represent this pattern implicitly through multiple incoming (and unconditional) control edges to a task. Only when each of these arcs has received the thread of control can the task be enabled. UML 2.0 ADs allow it to be represented in both ways.

**Issues** The use of the *Synchronization* pattern can potentially give rise to a deadlock in the situation where one of the incoming branches fails to deliver a thread of control to the join construct. This could be a consequence of a design error or that one of the tasks in the branch failing to complete successfully (e.g., as a consequence of it experiencing some form of exception) or because the thread of control is passed outside of the branch.

**Solutions** None of the offerings examined provide support for resolving this issue where the problem is caused by task failure in one of the incoming branches. Where this pattern is used in a structured context, the second possible cause of deadlock generally does not arise.
**Evaluation Criteria** Full support for this pattern is demonstrated by any offering providing a construct which satisfies the description when used in a context satisfying the context assumption.

**Pattern WCF-8 (Generalized AND-Join)**

**Description** The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings. Over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct (although obviously, the timing of these triggers may vary).

**Examples**
- When all *Get Directors Signature* tasks have completed, run the *Complete Contract* task.
- Accumulate engine, chassis and body components from the various production lines. When one of each has been received, use one of each component to assemble the basic car.

**Motivation** The Generalized AND-Join corresponds to one of the generally accepted notions of an AND-join implementation (the other situation is described by the Synchronization pattern) in which several paths of execution are synchronized and merged together. Unlike the Synchronization pattern, it supports the situation where one or more incoming branches may receive multiple triggers for the same process instance (i.e. a non-safe context) before the join resets. The classical Petri net uses semantics close to this pattern. This shows that this semantics can be formulated easily. However, the intended semantics in practice tends to be unclear in situations involving non-safe behavior.

**Overview** The operation of the Generalized AND-Join is illustrated in Figure 109. Before transition C can be enabled, an input token (corresponding to the same process instances) is required in each of the incoming places (i.e. p1 and p2). When there are corresponding tokens in each place, transition C is enabled and consumes a token from each input place and once it has completed, deposits a token in output place o1. If there is more than one token at an input place, they are left intact.

The process analogy to this sequence of events is that the AND-join only fires when a trigger has been received on each incoming branch for a given process instance however additional triggers are retained for future firings. This approach to AND-join implementation relaxes the context condition associated with the Synchronization pattern that only allows it to receive one trigger on each incoming branch after activation but before firing and as a result, it is able to be used in concurrent execution environments such as process models which involve loops as well as offerings that do not assume a safe execution environment. In light of this realization the same CPN pattern has been used as for the Synchronization pattern, which indicates a specialization relations between them.

**Context** There are no specific context conditions associated with the pattern.

**Implementation** This need to provide persistence of triggerings (potentially between distinct firings of the join) means that this construct is not widely supported by the offerings examined and only FileNet provides a construct for it. Token-based process models such as BPMN and XPDL have an advantage in this regard and both modeling notations are able to support this pattern⁵. EPCs provide a degree of ambiguity in their support for this

⁵Although it is noted that these formalisms are modeling languages which do not need to implement a given construct and may leave subtle semantical issues undefined.
pattern – whilst most documentation indicates that they do not support it, in the ARIS Simulator, they exhibit the required behavior – hence they are awarded a partial support rating on account of this variance.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description for the pattern. If there is any ambiguity associated with the specification or use of the construct, an offering is considered to provide partial support for the pattern.

---

**Sub-class: partial join** The `partial join` construct characterizes situations where threads associated with a subset of branches, that were enabled by a preceding AND-split construct, have to be synchronized in order for the thread of control to be passed on. This sub-class contains six patterns which represent the revision of the original `Discriminator` pattern. The `Discriminator` requires only one incoming branch to complete in order for the thread of control to be passed to the subsequent task. The original pattern did not differentiate between distinct implementation approaches and the degree to which they were able to deal with concurrency within a process instance. When defining the semantics of the pattern, the distinction has been made between structured, canceling and blocking types of the `Discriminator`. Interesting to note that the partial (or N-out-of-M) join was considered only as a sub-case of the `Discriminator` pattern, whereas now it is considered to be a pattern in its own right with three distinct forms: structured, canceling and blocking.

The structured partial join operates in the safe context, where each branch executes precisely once before a reset takes place, and there is a single preceding AND-split. The blocking partial join can be applied in non-safe situations, i.e. where multiple triggers can be obtained on the same branch for the same process instance, but which ensure that additional execution threads within the same branch are blocked until the partial join construct has reset. The canceling partial join applies to situations where remaining branches, that are still executing after the partial join has reset, are canceled. Six patterns, i.e. the structured, blocking, and canceling discriminator and the structured, blocking, and canceling partial join are described in detail below.
Pattern WCF-9 (Structured Discriminator)

Description The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The Structured Discriminator construct resets when all incoming branches have been enabled. The Structured Discriminator occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the Structured Discriminator is associated and it must merge all of the branches emanating from the Structured Discriminator. These branches must either flow from the Parallel Split to the Structured Discriminator without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

Example
– When handling a cardiac arrest, the check_breathing and check_pulse tasks run in parallel. Once the first of these has completed, the triage task is commenced. Completion of the other task is ignored and does not result in a second instance of the triage task.

Motivation The Structured Discriminator pattern provides a means of merging two or more distinct branches in a process into a single subsequent branch such that the first of them to complete results in the subsequent branch being triggered, but completions of other incoming branches thereafter have no effect on (and do not trigger) the subsequent branch. As such, the Structured Discriminator provides a mechanism for progressing the execution of a process once the first of a series of concurrent tasks has completed.

Overview The operation of the Structured Discriminator pattern is illustrated in Figure 110. Note that this realization assumes that inputs received from branches A_1 to A_m correspond to a single process instance, i.e. multiple process instances are not considered. An untyped token (), residing in place p_2, indicates that the Discriminator is ready to be enabled. The first token received at any of the incoming places i_1 to i_m results in the Discriminator being enabled and an output token being produced in output place o_1. An untyped token is also produced in place p_3 indicating that the Structured Discriminator has fired but not yet reset. Subsequent tokens received at each of the other input places have no effect on the Structured Discriminator (and do not result in any output tokens in place o_1). Once one token has been received by each input place, the Structured Discriminator resets and can be re-enabled once again. This occurs when m-1 tokens have accumulated at place p_1 allowing the reset transition to be enabled. Once again, the combination of the Structured Discriminator and the preceding Parallel Split can also be considered as a structured component that is compositional in form and can be incorporated in other structured processes whilst retaining the overall structural form. In this realization, a variant of the Region Flush CPN pattern (cf. page 86) is used for resetting the discriminator construct by removing outputs from the remaining branches (m-1)’p_i from the p_1 place. Although the Region Flush CPN pattern suggests aggregating objects in a place in order for it to be reset, in this realization there is no need to use a collection type because the capacity of the p_1 is limited by the number of branches that can be enabled concurrently.

There are two possible variants of this pattern that can be utilized in non-structured contexts. Both of which improve the applicability of the Structured Discriminator pattern whilst retaining its overall behavior. First, the Blocking Discriminator (WCF-10) removes the requirement that each incoming branch can only be enabled once between Structured Discriminator resets. It allows each incoming branch to be triggered multiple
times although the construct only resets when one triggering has been received on each input branch. It is discussed in further detail on page 137.

The second alternative, the *Canceling Discriminator* (WCF-11), improves the efficiency of the pattern further by preventing any subsequent tasks in the remaining incoming branches to the *Canceling Discriminator* from being enabled once the first branch has completed. Instead the remaining branches are effectively put into a “bypass mode” where any remaining tasks are “skipped” hence expediting the reset of the construct. It is discussed in further detail on page 138.

**Context**

There are two context conditions associated with the use of this pattern: (1) once the *Structured Discriminator* has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the *Structured Discriminator* (i.e. i1 to im) are safe and (2) there is a corresponding *Parallel Split* and once this has been enabled none of the tasks in the branches leading to the *Structured Discriminator* can be canceled before it has been triggered, i.e. an input is expected from all m input tasks. The only exception to this is that it is possible for all of the tasks leading up to the *Structured Discriminator* to be canceled. It is interesting to note that a corollary of these context criteria is that correct behavior of the pattern is assured and relies only on local information available to the *Structured Discriminator* at runtime.

**Implementation**
The *Structured Discriminator* can be directly implemented in iPlanet by specifying a custom trigger condition for a task with multiple incoming routers which only fires when the first router is enabled. BPMN and XPDL potentially support the pattern with a COMPLEX-Join construct however it is unclear how the IncomingCondition for the join is specified. UML 2.0 ADs shares a similar problem with its JoinNode construct. SAP Workflow provides partial support for this pattern via the fork construct although any unfinished branches are canceled once the first completes.

**Issues**

One issue that can arise with the *Structured Discriminator* is that failure to receive input on each of the incoming branches may result in the process instance (and possibly other process instances) stalling.

**Solutions**
The alternate versions of this pattern provide potential solutions to the issue. The *Blocking Discriminator* allows multiple execution threads in a given process instance to be handled by a single *Blocking Discriminator* (although a subsequent thread can only trigger the construct when inputs have been received on all incoming branches and the *Blocking Discriminator* has reset). The *Canceling Discriminator* only requires the first
thread of control to be received in an incoming branch. Once this has been received, the remaining branches are effectively put into “bypass” mode and any remaining tasks in those branches that have not already been commenced are skipped (or canceled) allowing the discriminator to be reset as soon as possible.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumptions. It rates as partial support if the *Structured Discriminator* can reset without all tasks in incoming branches having run to completion.

**Pattern WCF-10 (Blocking Discriminator)**

**Description** The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The *Blocking Discriminator* construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the *Blocking Discriminator* has reset.

**Example**
- The *check credentials* task can commence once the *confirm delegation arrival* or the *security check* task has been completed. Although these two tasks can execute concurrently, in practice, the *confirm delegation arrival* task always completes before the *security check* task. Another instance of the *check credentials* task cannot be initiated if a preceding instance of the task has not yet completed. Similarly, subsequent instances of the *confirm delegation arrival* and the *security check* tasks cannot be initiated if a preceding instance of the *check credentials* task has not yet completed.

**Motivation** The *Blocking Discriminator* pattern is a variant of the *Structured Discriminator* pattern that is able to run in environments where there are potentially several concurrent execution threads within the same process instance. This quality allows it to be used in loops and other process structures where more than one execution thread may be received in a given branch in the time between the first branch being enabled and the *Blocking Discriminator* being reset.

**Overview** Figure 111 illustrates the operation of this pattern. It is more robust than the *Structured Discriminator* as it is not subject to the constraint that each incoming branch can only be triggered once prior to reset. The *Blocking Discriminator* functions by keeping track of which inputs have been triggered (via the *triggered input* place) and preventing them from being re-enabled until the construct has reset as a consequence of receiving a trigger on each incoming branch. An important feature of this pattern is that it is able to be utilized in environments that do not support a safe process model or those that may receive multiple triggerings on the same input place, e.g., where the *Blocking Discriminator* is used within a loop. In this realization, the *Non-deterministic XOR-Split* CPN pattern (cf. page 38) is used to select an arbitrary branch by the D transition from the p3 place, and the *Region Flush* CPN pattern (cf. page 86) used for withdrawing non-consumed branch outputs by the reset transition from the p3 and *triggered input* places. Additionally, the *BSD Filter* CPN pattern (cf. page 43) is used to block other process instances by means of guard conditions associated with the t1...tm transitions. Furthermore, the *Aggregate Objects* CPN pattern (cf. page 67) is used to keep track of the activated branches by recording the identifiers of the activated branches to the *triggered input* place of the collection type (i.e. list cs is used in this case).
Context There is one context condition associated with the pattern: all inputs should arrive.

Implementation In the event of concurrent process instances attempting to simultaneously initiate the same Blocking Discriminator, it is necessary to keep track of both the process instance and the input branches that have triggered the Blocking Discriminator and also the execution threads that are consequently blocked (including the number of distinct triggerings on each branch) until it completes. The Blocking Discriminator is partially supported by BPMN, XPDL and UML 2.0 ADs.

Issues None identified.

Solutions N/A.

Evaluation Criteria An offering achieves full support if it provides a construct that satisfies the description for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

Pattern WCF-11 (Canceling Discriminator)

Description The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Canceling Discriminator also cancels the execution of all of the other incoming branches and resets the construct.

Example
– After the extract-sample task has completed, parts of the sample are sent to three distinct laboratories for examination. Once the first of these laboratories completes the sample-analysis, the other two task instances are canceled and the review-drilling task commences.

Motivation This pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but it is not important that the tasks associated with each of the branches (other than the first of them) be completed.

Overview The operation of this pattern is shown in Figure 112. Inputs i1 to im to the Canceling Discriminator serve to identify the branches preceding the construct. Transitions A1 to Am signify tasks in these preceding branches. Transitions C1 to Cm indicate alternate “bypass” or “cancellation” tasks for each of these branches (these execution options are not initially available to incoming execution threads). The first control-flow token
for a given case received at any input will cause B to fire and put a token in o1. As soon as this occurs, subsequent execution threads on other branches are put into “bypass mode” and instead of executing the normal tasks (A1..Am) on their specific branch, they can execute the “cancel” transitions (C1..Cm). (Note that the bypass transitions do not require any interaction. Hence they are executed directly by the PAIS and it can be assumed that the skip transitions are executed once they are enabled and complete almost instantaneously hence expediting completion of the branch). Once all incoming branches for a given case have been completed, the Canceling Discriminator construct can then reset and be re-enabled again for the same case. In addition to the Non-deterministic XOR-Split and Region Flush CPN patterns used in the Structured Discriminator pattern, this CPN model also uses the Non-Deterministic XOR-Split CPN pattern to realize cancelation of branches that have not yet completed (i.e. the choice between executing the A transition and skipping/canceling it by executing the C transition is non-deterministic).

**Context** There is one context condition associated with the use of this pattern: once the Canceling Discriminator has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the Canceling Discriminator (i.e. i1 to im) are safe. Another assumption for this pattern is that all inputs associated with branches i1 to im will arrive.

**Implementation** In order to implement this pattern, it is necessary for the offering to support some means of denoting the extent of the incoming branches to be canceled. This can be based on the Cancel Region pattern although support is only required for a restricted form of the pattern as the region to be canceled will always be a connected subgraph of the overall process model with the Canceling Discriminator construct being the connection point for all of the incoming branches.

This pattern is supported by the fork construct in SAP Workflow with the number of branches required for completion set to one. In BPMN, it is achieved by incorporating the incoming branches and the Canceling Discriminator in a subprocess that has an error event associated with it. The error event is triggered, canceling the remaining branches in the subprocess, when the Canceling Discriminator is triggered by the first incoming
branch. This configuration is illustrated in Figure 113(a). A similar solution is available in XPDL. UML 2.0 ADs support the pattern in a similar way by enclosing all of the incoming branches in an InterruptibleActivityRegion which is canceled when the Canceling Discriminator fires.

![Figure 113: Canceling discriminator pattern in BPMN and UML 2.0 ADs](image)

**Issues** The major difficulty with this pattern is in determining how much of the process model preceding the Canceling Discriminator is to be included in the cancelation region.

**Solutions** This issue is easily addressed in structured processes as all of the branches back to the preceding split construct which corresponds to the Canceling Discriminator should be subject to cancelation. In Figure 114(a), it is easy to see that the area denoted by the dotted box should be the cancelation region. It is a more complex matter when the process is not structured, e.g., in Figure 114(b) a cancelation region can be conceived which reaches back to the first AND-split and the pattern can be implemented based on this. A formal approach to determining the scope of the cancelation region can be found elsewhere [6].

![Figure 114: Process structure considerations for canceling discriminator](image)

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the
context assumption. An offering is considered to provide partial support for the pattern if there are side-effects associated with the execution of the pattern (e.g., tasks in incoming branches which have not completed being recorded as complete).

**Pattern WCF-12 (Structured Partial Join)**

**Description** The convergence of two or more branches (say \( m \)) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when \( n \) of the incoming branches have been enabled where \( n \) is less than \( m \). Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled. The join occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the join is associated and it must merge all of the branches emanating from the Parallel Split. These branches must either flow from the Parallel Split to the join without any splits or joins or be structured in form (i.e. balanced splits and joins).

**Example**

- Once two of the preceding three Expenditure Approval tasks have completed, start the Issue Cheque task. Wait until the remaining task has completed before allowing the Issue Cheque task to fire again.

**Motivation** The Structured Partial Join pattern provides a means of merging two or more distinct branches resulting from a specific Parallel Split or AND-split construct earlier in a process into a single branch. The join construct does not require triggers on all incoming branches before it can fire. Instead a given threshold can be defined which describes the circumstances under which the join should fire – typically this is presented as the ratio of incoming branches that need to be live for firing as against the total number of incoming branches to the join, e.g., a 2-out-of-3 join signifies that the join construct should fire when two of three incoming arcs are live. Subsequent completions of other remaining incoming branches have no effect on (and do not trigger) the subsequent branch. As such, the Structured Partial Join provides a mechanism for progressing the execution of a process once a specified number of concurrent tasks have completed rather than waiting for all of them to complete.

**Overview** The Structured Partial Join pattern is one possible variant of the AND-Join construct where the number of incoming arcs that will cause the join to fire (\( n \)) is between 2 and \( m - 1 \) (i.e. the total number of incoming branches less one, i.e. \( 2 \leq n < m \)). There are a number of possible specializations of the AND-join pattern and they form a hierarchy based on the value of \( n \). Where only one incoming arc must be live for firing (i.e. \( n=1 \)), this corresponds to one of the variants of the Discriminator pattern (cf. WCF-9, WCF-10 and WCF-11).

The pattern provides a means of merging two or more branches in a process and progressing execution of the process as rapidly as possible by enabling the subsequent (merged) branch as soon as a thread of control has been received on \( n \) of the incoming branches where \( n \) is less than the total number of incoming branches. The semantics of the Structured Partial Join pattern are illustrated in Figure 115. Note that B requires \( n \) tokens in place \( p_1 \) to progress. In this realization, the same set of CPN patterns is used as for the Structured Discriminator pattern, which indicates a specialization relationship between them.

**Context** There are two context conditions associated with the use of this pattern: (1) once the Structured Partial Join has been activated and has not yet been reset, it is not
possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the *Structured Partial Join* (i.e. i1 to im) are safe and (2) once the associated *Parallel Split* has been enabled none of the tasks in the branches leading to the *Structured Partial Join* can be canceled before it has been triggered. The only exception to this is that it is possible for all of the tasks leading up to the *Structured Partial Join* to be canceled.

There are two possible variants of this pattern that arise from relaxing some of the context conditions associated with it. Both of these improve on the efficiency of the join whilst retaining its overall behavior. The first alternative, the *Blocking Partial Join* (WCF-13) removes the requirement that each incoming branch can only be enabled once between join resets. It allows each incoming branch to be triggered multiple times although the construct only resets when one triggering has been received on each input branch. It is discussed in detail on page 143. Second, the *Canceling Partial Join* (WCF-14), improves the efficiency of the pattern further by canceling the other incoming branches to the join construct once n incoming branches have completed. It is discussed in further detail on page 144.

**Implementation** One of the difficulties in implementing the *Structured Partial Join* is that it essentially requires a specific construct to represent the join if it is to be done in a tractable manner. iPlanet does so via the router construct which links preceding tasks to a target task. A router can have a custom trigger condition specified for it that causes the target task to trigger when n incoming branches are live. SAP Workflow provides partial support for this pattern via the fork construct although any unfinished branches are canceled once the first completes. None of the other offerings examined offers a dedicated construct. Staffware provides for a 1-out-of-2 join, but more complex joins must be constructed from this resulting in an over-complex process model. Similar difficulties exist for COSA. Of the business process modelling languages, both BPMN and XPDL appear to provide support for the *Structured Partial Join* via the complex gateway construct but the lack of detail on how the IncomingCondition is specified results in a partial rating. UML 2.0 ADs also suffers from a similar lack of detail on the JoinSpec configuration required to support this pattern. There is no ability to represent the construct in BPEL.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the
context assumptions. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

**Pattern WCF-13 (Blocking Partial Join)**

**Description** The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches have been enabled (where 2 ≤ n < m). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

**Example**

- When the first member of the visiting delegation arrives, the check credentials task can commence. It concludes when either the ambassador or the president arrives. Owing to staff constraints, only one instance of the check credentials task can be undertaken at any time. Should members of another delegation arrive, the checking of their credentials is delayed until the first check credentials task has completed.

**Motivation** The Blocking Partial Join is a variant of the Structured Partial Join that is able to run in environments where there are concurrent process instances, particularly process instances that have multiple concurrent execution threads.

**Overview** Figure 116 illustrates the operation of this pattern. The Blocking Partial Join functions by keeping track of which inputs have been enabled (via the triggered input place) and preventing them from being re-enabled until the construct has reset as a consequence of receiving a trigger on each incoming place. After n incoming triggers have been received for a given process instance (via tokens being received in n distinct input places from i1 to im), the join fires and a token is placed in output o1. The completion of the remaining n-m branches has no impact on the join except that it is reset when the last of them is received.

The pattern shares the same advantages over the Structured Partial Join as the Blocking Discriminator does over the Structured Discriminator, namely greater flexibility as it is able to deal with the situation where a branch is triggered more than once, e.g., where the construct exists within a loop. In this realization, the same set of CPN patterns is used as for the Blocking Discriminator pattern, which implies a specialization relationship between these patterns.

![Figure 116: Blocking Partial Join pattern](image-url)
Context There is one context conditions associated with the pattern: all inputs associated with \( m \) branches are assumed to arrive.

Implementation The approach to implementing this pattern is essentially the same as that for the *Blocking Discriminator* except that the join fires when \( n \) incoming branches have triggered rather than just the first. The *Blocking Partial Join* is partially supported by BPMN, XPDL and UML 2.0 ADs as it is unclear how the join condition is specified.

Issues None identified.

Solutions N/A.

Evaluation Criteria An offering achieves full support if it provides a construct that satisfies the description for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

Pattern WCF-14 (Canceling Partial Join)

Description The convergence of two or more branches (say \( m \)) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when \( n \) of the incoming branches have been enabled where \( n \) is less than \( m \). Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.

Example – Once the picture is received, it is sent to three art dealers for the examination. Once two of the *prepare condition report* tasks have been completed, the remaining *prepare condition report* task is canceled and the *plan restoration* task commences.

Motivation This pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but only a subset of those tasks associated with each of the branches needs to be completed.

Overview The operation of this pattern is shown in Figure 117. It operates in the same way as the *Canceling Discriminator* except that, for this pattern, the cancelation is only triggered when \( n \) distinct incoming branches have been enabled. In this realization, the same set of CPN patterns is used as for the *Canceling Discriminator* pattern, which implies that a specialization relation exists between these patterns.

Context There is one context condition associated with the use of this pattern: once the *Canceling Partial Join* has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the *Canceling Partial Join* (i.e. \( i_1 \) to \( i_m \)) are safe. Furthermore, all inputs are expected to arrive.

Implementation The approach to implementing this pattern is essentially the same as that for the *Canceling Discriminator* except that the join fires when \( n \) incoming branches have triggered rather than just the first. The *Canceling Partial Join* is supported by SAP Workflow and UML 2.0 ADs. BPMN and XPDL achieve a partial support rating as it is unclear exactly how the join condition is specified.

Issues As for the *Canceling Discriminator* pattern.

Solutions As for the *Canceling Discriminator* pattern.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. An offering is considered to provide partial support for the pattern if
Section 4.1 Revisiting the control-flow patterns

there are undesirable side-effects associated with the construct firing (e.g., tasks in incoming branches which have not completed being recorded as complete) or if the semantics associated with the join condition are unclear.

**Sub-class: XOR-joins** The XOR-join characterizes situations where one or more distinct branches need to be merged into a single branch. Two patterns Simple Merge and Multi Merge belong to this join type. While the Simple Merge pattern operates in a safe context, where at most one incoming branch may be active simultaneously, the Multi Merge pattern allows multiple active branches to be merged into a single branch without synchronizing them.

**Pattern WCF-15 (Simple Merge)**

**Description** The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

**Examples**
- At the conclusion of either the bobcat-excavation or the D9-excavation tasks, an estimate of the amount of earth moved is made for billing purposes.
- After the cash-payment or provide-credit tasks, initiate the produce-receipt task.

**Motivation** The Simple Merge pattern provides a means of merging two or more distinct branches without synchronizing them. As such, this presents the opportunity to simplify a process model by removing the need to explicitly replicate a sequence of tasks that is common to two or more branches. Instead, these branches can be joined with a simple merge construct and the common set of tasks need only to be depicted once in the process model.

**Overview** Figure 118 illustrates the behavior of this pattern. Immediately after either task A or B is completed, task C will be enabled. There is no consideration of synchronization.

**Context** There is one context condition associated with the pattern: the place at which the merge occurs (i.e. place p1 in Figure 118) is safe and can never contain more than one
token.

**Implementation** Similar to patterns WCF6–WCF9 described earlier, this pattern can either be represented explicitly or implicitly. Staffware, SAP Workflow and UML 2.0 ADs provide specific join constructs for this purpose whereas it is represented implicitly in WebSphere MQ, FLOWer, COSA and BPEL. BPMN and XPDL allow it to be represented in both ways.

**Issues** One issue that can arise with the use of this pattern occurs where it cannot be ensured that the incoming place to the merge (p1) is safe.

**Solutions** In this situation, the context conditions for the pattern are not met and it cannot be used, however there is an alternative pattern – the Multi-Merge (WCF-16) – that is able to deal with the merging of branches in potentially unsafe process instances.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

**Pattern WCF-16 (Multi-Merge)**

**Description** The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

**Example**
- The lay_foundations, order_materials and book_labourer tasks occur in parallel as separate process branches. As each of them completes the quality_review task is run before that branch of the process finishes.

**Motivation** The Multi-Merge pattern provides a means of merging distinct branches in a process into a single branch. Although several execution paths are merged, there is no synchronization of control-flow and each thread of control which is currently active in any of the preceding branches will flow unimpeded into the merged branch.

**Overview** The operation of this pattern is illustrated in Figure 119. Note that this figure is identical to Figure 118, which indicates specialization relationship between them. Any threads of control on incoming branches to p1 should be passed on to the outgoing branch. The analogy to this in CPN terms, is that each incoming token to place p1 should be preserved. The distinction between this pattern and the Simple Merge is that it is possible for more than one incoming branch to be active simultaneously and there is no necessity for place p1 to be safe.

**Context** There is one context condition associated with this pattern: the Multi-Merge pattern assumes that there is a corresponding Multi-Choice construct preceding it.
Implementation iPlanet allows the Multi-Merge pattern to be implemented by specifying a trigger condition for a task that allows it to be triggered when any of its incoming routers are triggered. BPMN and XPDL directly implement it via the XOR-join construct and UML 2.0 ADs have an analogue in the form of the MergeNode construct. EPCs also provide the XOR-join construct, however they only expect one incoming thread of control and ignore subsequent simultaneous triggers, hence they do not support the pattern. FLOWer is able to support multiple concurrent threads through dynamic subplans however its highly structured nature does not enable it to provide general support for the Multi-Merge pattern. Although COSA is based on a Petri net foundation, it only supports safe models and hence is unable to fully support the pattern. For example, both A and B in Figure 119 will block if there is a token in place p1. Staffware attempts to maintain a safe process model by coalescing subsequent triggerings of a step whilst it is active into the same thread of control hence it is also unable to support this pattern. This behavior is quite problematic as it creates a race condition in which all of the execution sequences ABC, BAC, ACBC and BCAC are possible.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumptions. Partial support is awarded to offerings that do not provide support for multiple branches to merge simultaneously or do not provide for preservation of all threads of control where this does occur.

Sub-class: OR-joins The OR-join characterizes situations where one or more branches that have been previously enabled by a corresponding OR-split, need to be synchronized into a single branch. The original Synchronizing Merge pattern did not adequately differentiate between possible context assumptions, each of which has a distinct semantics. Depending on whether there is a single corresponding OR-split construct preceding the OR-join, and whether it is possible to foresee which of the branches were enabled and when will they complete, we distinguish three variants of the Synchronizing Merge:

- **Structured Synchronizing Merge**: a join corresponding to a preceding split (i.e. Multi-Choice) which applies only to structured process models where there is a one-to-one correspondence between splits and joins;
• **Local Synchronizing Merge**: a join whose evaluation is based on directly available information about the enablement of incoming branches that is signaled, for instance, by passing true/false tokens (i.e. local semantics); and

• **General Synchronizing Merge**: a join, whose evaluation is based on the current and potential future states of a process instance (i.e. non-local semantics).

In order to differentiate between specific OR-join implementations in distinct offerings, in Section 4.2.1 we introduce a graphical notation able to capture synchronization based on both local and non-local semantics. As the analysis of non-local semantics is the research topic in its own right, we give a brief overview of approaches to handling non-local semantics on page 205.

**Pattern WCF-17 (Structured Synchronizing Merge)**

**Description** The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The **Structured Synchronizing Merge** occurs in a structured context, i.e. there must be a single **Multi-Choice** construct earlier in the process model with which the **Structured Synchronizing Merge** is associated and it must merge all of the branches emanating from the **Multi-Choice**. These branches must either flow from the **Multi-Choice** to the **Structured Synchronizing Merge** without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

**Example**
- Depending on the type of emergency, either or both of the despatch-police and despatch-ambulance tasks are initiated simultaneously. When all emergency vehicles arrive at the accident, the transfer-patient task commences.

**Motivation** The **Structured Synchronizing Merge** pattern provides a means of merging the branches resulting from a specific **Multi-Choice** (or OR-split) construct earlier in a process into a single branch. Implicit in this merging is the synchronization of all of the threads of execution resulting from the preceding **Multi-Choice**.

**Overview** It is not necessary that all of the incoming branches to the **Structured Synchronizing Merge** are active in order for the construct to be enabled, however all of the active threads of control associated with the incoming branches must have reached the **Structured Synchronizing Merge** before it can fire.

One of the difficulties associated with the use of this pattern is knowing when the **Structured Synchronizing Merge** can fire. The **Structured Synchronizing Merge** construct must be able to resolve the decision based on local information available to it during the course of execution. Critical to this decision is knowledge of how many branches emanating from the preceding **Multi-Choice** are active and require synchronization. This is crucial in order to remove any potential for the “vicious circle paradox” [140] to arise where the determination of exactly when the merge can fire is based on non-local semantics which by necessity includes a self-referencing definition and makes the firing decision inherently ambiguous.

Addressing this issue without introducing non-local semantics for the **Structured Synchronizing Merge** can be achieved by structuring of the process model following a **Multi-Choice** such that the subsequent **Structured Synchronizing Merge** will always receive pre-
Section 4.1 Revisiting the control-flow patterns

cisely one trigger on each of its incoming branches and no additional knowledge is required to make the decision as to when it should be enabled.

The implementation of this pattern is illustrated in Figure 120. The assumption associated with this alternative is that the merge construct always occurs in a structured context, i.e. it is always paired with a distinct preceding Multi-Choice. It is interesting to note that the combination of the Structured Synchronizing Merge and the preceding Multi-Choice (together with the intervening tasks) forms a structured component that is compositional in form and can be incorporated in other structured processes whilst retaining the overall structural form. This approach involves adding an alternate “bypass” path around each branch from the multi-merge to the Structured Synchronizing Merge which is enabled in the event that the normal path is not chosen. The “bypass” path is merged with the normal path for each branch prior to the Structured Synchronizing Merge construct ensuring that it always gets a trigger on all incoming branches and can hence be implemented as an AND-join construct. In this model, the BCI Filter CPN pattern (cf. page 42) is used to block inputs that do not satisfy guard conditions associated with the A transition. The selection of branches and their associated bypass paths is done according to the Deterministic XOR-split CPN pattern (cf. page 36). The split and merge of branches in this diagram is associated with the Distributed Data Processing CPN pattern (cf. page 65) with the only difference that the focus is on the splitting/merging of threads of control rather than splitting/merging data elements.

Context There are two context conditions associated with the use of this pattern: (1) once the Structured Synchronizing Merge has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch. In other words, all input places to the Structured Synchronizing Merge (i.e. p4 and p5) are safe and (2) once the Multi-Choice has been enabled none of the tasks in the branches leading to the Structured Synchronizing Merge can be canceled before the merge has been triggered. The only exception to this is that it is possible for all of the tasks leading up to the Structured Synchronizing Merge to be canceled.

Implementation The Structured Synchronizing Merge can be implemented in any process language which supports the Multi-Choice construct and can satisfy the context conditions discussed above. It is directly supported in WebSphere MQ, FLOWer, FileNet, BPMN, BPEL, XPDL and EPCs.
Issues  One consideration that arises with the implementation of the OR-join is providing a form that is able to be used in arbitrary loops and more complex process models which are not structured in form. The *Structured Synchronizing Merge* cannot be used in these contexts.

Solutions  Both the *Local Synchronizing Merge* (WCF-18) and the *General Synchronizing Merge* (WCF-19) are able to be used in unstructured process models. The latter is also able to be used in arbitrary loops. The *Local Synchronizing Merge* tends to be more attractive from an implementation perspective as it is less computationally expensive than the *General Synchronizing Merge*.

Evaluation Criteria  Full support for this pattern in an offering is evidenced by the availability of a construct which when placed in the proper context will synchronize all active threads emanating from the corresponding *Multi-Choice*.

Pattern WCF-18 (Local Synchronizing Merge)

Description  The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis of information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

Example  After advertising an Open Day at the school of modern art, visitors are expected to arrive at the particular date and time. Show-lessons start at the indicated time for all visitors that have arrived.

Motivation  The *Local Synchronizing Merge* provides a deterministic semantics for the synchronizing merge which does not rely on the process model being structured (as is required for the *Structured Synchronizing Merge*) but also does not require the use of non-local semantics in evaluating when the merge can fire.

Overview  Figure 121 illustrates one approach to implementing this pattern. It is based on the use of “true” and “false” tokens which are used to indicate whether a branch is enabled or not. After the divergence at transition A, one or both of the outgoing branches may be enabled. The determinant of whether the branch is enabled is that the token passed to the branch contains both the case id as well as a Boolean variable which is “true” if the tasks in the branch are to be executed, “false” otherwise. As the control-flow token is passed down a branch, if it is a “true” token, then each task that receives the thread of control is executed otherwise it is skipped (illustrated by the execution of the bypass task S1...Sn associated with each task). The *Local Synchronizing Merge*, which in this example is illustrated by transition E, can be evaluated when every incoming branch has delivered a token to the input places for the same case. This realization uses the *BSI Filter* CPN pattern (cf. page 42) for enabling of the A transition based on the status of its guard, whose execution results in the selection of one or more branches. The *ID Matching* CPN pattern (cf. page 49) is applied multiple times when matching branch inputs using pi as a process instance identifier. The choice between execution or skipping of tasks in each of the enabled branches is non-deterministic and is realized using the *Non-deterministic XOR-split* CPN pattern (cf. page 38).

Another possible solution is provided by Rittgen [189]. It involves the direct communication of the number of active branches from the preceding OR-Join(s) divergence to the *Local Synchronizing Merge* so that it is able to determine when to fire.
Section 4.1 Revisiting the control-flow patterns

Figure 121: Local Synchronizing Merge pattern

Context There are two context conditions associated with the use of this pattern: (1) once the Local Synchronizing Merge has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch, i.e. all input places to the Local Synchronizing Merge (places \( p_4 \) and \( p_5 \) are safe and (2) the Local Synchronizing Merge construct must be able to determine how many incoming branches require synchronization based on local knowledge available to it during execution.

Implementation WebSphere MQ, FLOWer, COSA, BPEL and EPCs provide support for this pattern. UML 2.0 ADs seems to provide support although there is some ambiguity over the actual JoinSpec configuration required.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumptions. If there is any ambiguity as to the manner in which the synchronization condition is specified, then it rates as partial support.

Pattern WCF-19 (General Synchronizing Merge)

Description The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time.

Example Figure 122 provides an example of the General Synchronizing Merge pattern. It shares a similar fundamental structure to the examples presented in Figures 120 and 121 for the other forms of OR-join however the conditional feedback path from \( p_4 \) to \( p_1 \) involving \( F \) (which effectively embeds a “loop” within the process where \( \text{cond3} \) evaluates to true) means that it is not possible to model it either in a structured way or to use local information available to \( E \) to determine when the OR-join should be enabled. In this
realization, the same CPN patterns have been used as for the Local Synchronizing Merge pattern.

![Diagram of General Synchronizing Merge pattern]

Figure 122: Problem addressed by General Synchronizing Merge pattern

**Motivation** This pattern provides a general approach to the evaluation of the General Synchronizing Merge (or OR-join) in processes. It is able to be used in non-structured and highly concurrent processes including process models that include arbitrary looping structures.

**Overview** This pattern provides general support for the OR-join construct that is widely utilized in modeling languages but is often only partially implemented or severely restricted in the form in which it can be used. The difficulty in implementing the General Synchronizing Merge stems from the fact that its evaluation relies on non-local semantics [11] in order to determine when it can fire. In fact, it is easy to see that this construct can lead to the “vicious circle paradox” [140] where two OR-joins depend on one another.

The OR-join can only be enabled when the thread of control has been received from all incoming branches and it is certain that the remaining incoming branches which have not been enabled will never be enabled at any future time. Determination of this fact requires a (computationally expensive) evaluation of possible future states for the current process instance.

**Context** There are no specific context conditions associated with this pattern.

**Implementation** FileNet is the only offering examined to support this pattern. An algorithm describing an approach to implementing the General Synchronizing Merge based on Reset-Nets is described in [232] and has been used as the basis for the OR-join construct in the YAWL reference implementation [15].

**Issues** There are three significant issues associated with this pattern: (1) when determining whether an OR-join should be enabled in a given process instance, how should composite tasks which precede the OR-join be handled, (2) how should preceding OR-joins be handled and (3) how can the performance implications of OR-join evaluation (which potentially involves a state space analysis for the case in which the OR-join appears) be addressed.

**Solutions** Solutions to all of these problems are described in [232]. It provides a deterministic means of evaluating whether an OR-join should be enabled based on an evaluation
of the current execution state of preceding tasks. It considers composite tasks to function in the same way as atomic tasks – i.e. they are either enabled or not, – and there is no further consideration of the execution specifics of the underlying subprocess. Moreover it is assumed that they will continue executing and pass the thread of control onto subsequent tasks when complete. In terms of the second issue, any preceding OR-joins are all considered to function either as XOR-joins or AND-joins when determining if the task with which they are associated can be enabled. By doing this, the “vicious circle” problem is avoided. It also offers some potential solutions to the third issue involving the use of reduction rules which limit the size of the state space evaluation required in order to establish whether the OR-join should be enabled.

Evaluation Criteria An offering achieves full support if it provides a construct that satisfies the description for the pattern.

**Sub-class: thread-joins** The thread-join characterizes situations where distinct threads of control along a single branch need to be coalesced into a single thread of control as described by the *Thread Merge* pattern. The *Thread Merge* is applied in the context, where it is always preceded by a corresponding thread-split.

**Pattern WCF-20 (Thread Merge)**

**Description** At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

**Example**

- Instances of the *register-vehicle* task run independently of each other and of other tasks in the Process Enquiry process. They are created as needed. When ten of them have completed, the *process-registration-batch* task should execute once to finalize the vehicle registration system records update.

**Motivation** This pattern provides a means of merging multiple threads within a given process instance. It is a counterpart to the *Thread Split* pattern which creates multiple execution threads along the same branch.

**Overview** The operation of this pattern is illustrated in Figure 123. Note that numinsts indicates the number of threads to be merged.

```
Pattern WCF-20 (Thread Merge)

Description At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

Example

- Instances of the register-vehicle task run independently of each other and of other tasks in the Process Enquiry process. They are created as needed. When ten of them have completed, the process-registration-batch task should execute once to finalize the vehicle registration system records update.

Motivation This pattern provides a means of merging multiple threads within a given process instance. It is a counterpart to the Thread Split pattern which creates multiple execution threads along the same branch.

Overview The operation of this pattern is illustrated in Figure 123. Note that numinsts indicates the number of threads to be merged.

```

**Context** There is one context consideration for this pattern: the number of threads needing to be merged (i.e. numinsts) must be known at design-time.

**Implementation** Implementation of this pattern implies that an offering is able to support the execution of processes in a non-safe context. This rules out the majority of the offerings examined from providing any tractable forms of implementation. BPMN and XPDL provide direct support for the pattern by including a task after the spawned task in which the StartQuantity attribute is set to the number of threads that need to be synchronized. The StartQuantity attribute specifies the number of incoming tokens required
to start a task. UML 2.0 ADs offer a similar behavior via weights on ActivityEdge objects. BPEL provides an indirect means of implementation based on the correlation facility for feedback from the <invoke> action although some programmatic housekeeping is required to determine when synchronization should occur.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. If any degree of programmatic extension is required to achieve the same behavior, then the partial support rating applies.

**Class: Repetition patterns**

Repetition patterns describe situations where the execution of a task or a process fragment needs to be repeated on a conditional basis. The Arbitrary Cycles pattern did not adequately cover more restrictive forms of iteration such as while and repeat loops, hence the class of repetition patterns has been extended with the Structured Loop pattern. Similarly, the Recursion pattern has been introduced in order to address repetitive task execution which is based on self-invocation. Three repetitions patterns (Arbitrary Cycles, Structured Loop, and Recursion), addressing different approaches to the realization of repetitive execution, are described in a detail below.

**Pattern WCF-21 (Arbitrary Cycles)**

**Description** The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.

**Example** Figure 124 provides an illustration of the pattern with two entry points: p3 and p4.

![Figure 124: Arbitrary Cycles pattern](image)

**Motivation** The Arbitrary Cycles pattern provides a means of supporting repetition in a process model in an unstructured way without the need for specific looping operators or restrictions on the overall format of the process model.

**Overview** The only further consideration for this pattern is that the process model is able to support cycles (i.e. it is not block structured).

**Context** There are no specific context conditions associated with this pattern.
Implementation Staffware, COSA, iPlanet, FileNet, BPMN, XPDL, UML 2.0 ADs and EPCs are all capable of capturing the Arbitrary Cycles pattern. Block structured offerings such as WebSphere MQ, FLOWer, SAP Workflow and BPEL are not able to represent arbitrary process structures.

Issues The unstructured occurrences of the Arbitrary Cycles pattern are difficult to capture in some types of PAIS, particularly those that implement structured process models.

Solutions In some situations, it is possible to transform process models containing Arbitrary Cycles into structured processes, thus allowing them to be captured in offerings based on structured process models. Further details on the types of process models that can be transformed and the approaches to doing so can be found elsewhere [135, 136].

Evaluation Criteria An offering achieves full support for the pattern if it is able to capture unstructured cycles that have more than one entry and/or exit point.

Pattern WCF-22 (Structured Loop)

Description The ability to execute a task or subprocess repeatedly. The loop has either a pre-test or post-test condition (or both) associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.

Examples

– While the machine still has fuel remaining, continue with the production process.
– Continue processing photographs from the film until all of them have been printed.

Motivation There are two general forms of this pattern – the while loop which equates to the classic while...do pre-test loop construct used in programming languages and the repeat loop which equates to the repeat...until post-test loop construct.

The while loop allows for the repeated sequential execution of a specified task or a subprocess zero or more times providing a nominated condition evaluates to true. The pre-test condition is evaluated before the first iteration of the loop and is re-evaluated before each subsequent iteration. Once the pre-test condition evaluates to false, the thread of control passes to the task immediately following the loop.

The repeat loop allows for the execution of a task or subprocess one or more times, continuing with execution until a nominated condition evaluates to true. The post-test condition is evaluated after the first iteration of the loop and is re-evaluated after each subsequent iteration. Once the post-test condition evaluates to true, the thread of control passes to the task immediately following the loop.

Overview As indicated above, there are two variants of this pattern: the while loop illustrated in Figure 125 and the repeat loop shown in Figure 126. In both cases, task B is executed repeatedly. The conditional choice as to whether to continue with another iteration of the loop or exit it is realized using the Deterministic XOR-Split CPN pattern (cf. page 36).

Context There is one context condition associated with this pattern: only one instance of a loop can be active at any time, i.e. places p1 and p2 (and any other places in the body of the loop) must be safe.

Implementation The main consideration in supporting the Structured Loop pattern is the availability of a construct within a modelling language to denote the repeated execution of a task or subprocess based on a specified condition. The evaluation of the condition to determine whether to continue (or cease) execution can occur either before or after the task (or subprocess) has been initiated.
WebSphere MQ provides support for post-tested loops through the use of exit conditions on block or process constructs. Similarly, FLOWer provides the sequential plan construct that allows a sequence of tasks to be repeated sequentially until a nominated condition is satisfied. iPlanet also supports post-tested loops through conditions on outgoing routers from a task that loop back to the beginning of the same task. BPEL directly supports pre-tested loops via the <while> construct. BPMN and XPDL allow both pre-tested and post-tested loops to be captured through the loop task construct. Similarly UML 2.0 ADs provide the LoopNode construct which has similar capabilities. SAP provides two loop constructs corresponding to the while loop and the repeat loop. (In fact, the SAP loop construct is more general merging both the while and repeat loop into a single construct).

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

**Pattern WCF-23 (Recursion)**

**Description** The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated.

**Example**
- An instance of the resolve-defect task is initiated for each mechanical problem that is identified in the production plant. During the execution of the resolve-defect task, if a mechanical fault is identified during investigations that is not related to the current defect, another instance of the resolve-defect is started. These subprocesses can also initiate further resolve-defect tasks should they be necessary. The parent resolve-defect
task cannot complete until all child *resolve-defect* tasks that it initiated have been satisfactorily completed.

**Motivation** For some types of task, simpler and more succinct solutions can be provided through the use of recursion rather than iteration. In order to harness recursive forms of problem solving within the context of a process, a means of describing a task execution in terms of itself (i.e. the ability for a task to invoke another instance of itself whilst executing) is required.

**Overview** Figure 127 illustrates the format of the recursion pattern in Petri net terms. Task A can be decomposed into the process model with input i1 and output o1. It is important to note that this process also contains the task A hence the task is described in terms of itself.

![Figure 127: Recursion pattern](image)

In order to implement the pattern, a process model requires the ability to denote the synchronous invocation of a task or subprocess within the same model. In order to ensure that use of recursion does not lead to infinite self-referencing decompositions, Figure 127 contains one path (illustrated by task sequence BDC) which is not self-referencing and will terminate normally. This corresponds to the *terminating condition* in mathematical descriptions of recursion and ensures that, where recursion is used in a process, the overall process will eventually complete normally when executed.

**Context** There are no specific context conditions associated with this pattern.

**Implementation** In order to implement recursion within the context of a process, some means of invoking a distinct instance of a task is required from within a given task implementation. Staffware, WebSphere MQ, COSA, iPlanet and SAP Workflow all provide the ability for a task to invoke an instance of itself whilst executing.

The actual mechanics of implementing recursion for a process such as that depicted in Figure 127 are shown in Figure 128. The execution of the recursive task A is denoted by the transitions *startA* and *endA*. When an instance of task A is initiated in a process instance pi, any further execution of the process instance is suspended and the thread of control is passed to the decomposition that describes the recursive task (in this case, task B is enabled). A new process instance id is created for the thread of control that is passed to the decomposition and a mapping function (in this example denoted by *child()* ) is used to capture the relationship between the parent case-id and the decomposition case-id, thus ensuring that once the child case has completed, the parent case can continue from the point at which it originally suspended execution and invoked the child instance of itself.

**Issues** None identified.

**Solutions** N/A.
Evaluation Criteria

An offering achieves full support if it provides a construct that satisfies the description for the pattern.

Class: Multiple Instance (MI) patterns

Multiple Instance patterns describe situations where, for a given task, several instances need to be created and executed either sequentially or concurrently. The **MI without Synchronization**, **MI with Design-Time Knowledge**, **MI with Run-Time Knowledge**, and **MI without Run-Time Knowledge** patterns identify different ways to synchronize created task instances on completion. They differ in terms of the moment at which it is known how many instances need to be created. Unlike the **MI without Synchronization** pattern, which passes the thread of control to a subsequent task immediately after task instances have been initiated, the other three patterns acknowledge the ability to pass control to a subsequent task after all existing instances have completed.

Existing multiple instances patterns assume that all task instances must complete before a subsequent task can be enabled. In recognition that the process can proceed beyond the multiple instance task prior to the completion of all instances of a given task, the notion of a partial join has been introduced for multiple instance patterns. A partial join between instances of a particular task allows a subsequent task to be triggered once a threshold of concurrent task instances have completed. Three variants of partial join for MI task are described in the **Static Partial Join for MI**, **Canceling Partial Join for MI**, and **Dynamic Partial Join for MI** patterns.

Pattern WCF-24 (MI without Synchronization)

**Description** Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data elements) and each of them must execute independently from and without reference to the task that started them.

**Example**
- A list of traffic infringements is received by the Transport Department. For each infringement on the list an **Issue-Infringement-Notice** task is created. These tasks run to
completion in parallel and do not trigger any subsequent tasks. They do not need to be synchronized at completion.

**Motivation** This pattern provides a means of creating multiple instances of a given task. It caters for situations where the number of individual tasks required is known before the spawning action commences, the tasks can execute independently of each other and no subsequent synchronization is required.

**Overview** There are two possible variants in the way in which this pattern can operate. The first is illustrated by Figure 129 in which the *create instance* task runs within a loop and the new task instances are created sequentially. Place \( p_2 \) indicates the number of instances required and is decremented as each new instance is created. New instances can only be created when the token in \( p_2 \) has a value greater than zero – the guard on the *create instance* task ensures this is the case. When all instances have been created, the next task (B) can be enabled – again the guard on task B ensures this is also the case. The selection between transitions B and *create instance*, each associated with a guard, corresponds to the *Deterministic XOR-Split* pattern (cf. page 36). The synchronization of the counter with transitions A and B corresponds to the *Distributed Data Processing CPN* pattern (cf. page 65). The conditional enabling of the B transition depends on the number of instances created, thus represents the *BSD Filter CPN* pattern (cf. page 43).

![Figure 129: Multiple Instances without Synchronization (Variant 1)](image)

In Figure 130, the task instances are all created simultaneously. In both variants, it is a requirement that the number of new instances required is known before the creation task commences. It is also assumed that task instances can be created that run independently (and in addition to the thread of control which started them) and that they do not require synchronizing as part of this construct.

![Figure 130: Multiple Instances without Synchronization (Variant 2)](image)

**Context** There is one context condition associated with this pattern: the number of task instances (i.e. `numinst`) is known at design time and is a fixed value.
Implementation Most offerings – COSA, iPlanet, BPEL, BPMN, XPDL and UML 2.0 ADs – support the sequential variant of this pattern (as illustrated in Figure 129) with the task creation occurring within a loop. SAP Workflow also does so, but with the limitation that a new process instance is started for each task instance invoked. BPMN also supports the second variant, as do Staffware and FLOWer, and they provide the ability to create the required number of task instances simultaneously.

Issues None identified.

Solutions N/A.

Evaluation Criteria An offering achieves full support if it provides a construct that satisfies the description for the pattern. Where the newly created task instances run in a distinct process instance to the task that started them or it cannot access the same data elements as the parent task, the offering achieves only partial support.

Pattern WCF-25 (MI with a priori Design-Time Knowledge)

Description Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered.

Example – The Annual Report must be signed by all six Directors before it can be issued.

Motivation This pattern provides the basis for concurrent execution of a nominated task a predefined number of times. It also ensures that all task instances are complete before subsequent tasks are initiated.

Overview Similar to WCF-24, the Multiple Instances without Synchronization pattern, there are both sequential and simultaneous variants of this pattern illustrated in Figures 131 and 132 respectively. In both figures, task C is the one that executes multiple times. The realization in Figure 131 uses the Deterministic XOR-Split CPN pattern (cf. page 36) to specify by means of transition guards which of the tasks create instance or complete MI task, i.e. B, needs has to be selected. To synchronize multiple branches, the Distributed Data Processing CPN pattern (cf. page 65) is used in association with transitions A and B.

**Figure 131:** Multiple Instances with a priori Design-Time Knowledge (Variant 1)

Context There is one context condition associated with this pattern: the number of task instances (i.e. numinst) is known at design time and is a fixed value.

Implementation In order to implement this pattern, an offering must provide a specific construct in the process model that is able to denote the actual number of concurrent
Section 4.1 Revisiting the control-flow patterns

Figure 132: Multiple Instances with a priori Design-Time Knowledge (Variant 2)

Task instances that are required. Staffware, FLOWer, SAP Workflow and UML 2.0 ADs support the simultaneous variant of the pattern through the use of dynamic subprocedure, dynamic subplan, multi-line container element and ExpansionRegion constructs respectively. BPMN and XPDL support both options via the multi-instance loop task construct with the MI_Ordering attribute supporting both sequential and parallel values depending on whether the tasks should be started one-by-one or all together. Unlike other BPEL offerings which do not support this pattern, Oracle BPEL provides a <flowN> construct that enables the creation of multiple concurrent instances of a task.

Issues Many offerings provide a work-around for this pattern by embedding some form of task invocation within a loop. These implementation approaches have two significant problems associated with them: (1) the task invocations occur at discrete time intervals and it is possible for the individual task instances to have distinct states (i.e. there is no requirement that they execute concurrently) and (2) there is no consideration of the means by which the distinct task instances will be synchronized. These issues, together with the necessity for the designer to effectively craft the pattern themselves (rather than having it provided by the offering) rule out this form of implementation from being considered as satisfying the requirements for full support.

Solutions One possibility that exists where this functionality is not provided by an offering but an analogous form of operation is required is to simply replicate the task in the process-model. Alternatively a solution based on iteration can be utilized.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. Although work-arounds are possible which achieve the same behavior through the use of various constructs within an offering such as task replication or loops, they have a number of shortcomings and are not considered to constitute support for the pattern.

Pattern WCF-26 (MI with a priori Run-Time Knowledge)

Description Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.

Examples
- When diagnosing an engine fault, multiple instances of the check-sensor task can run concurrently depending on the number of error messages received. Only when all messages have been processed, can the identify-fault task be initiated;
- In the review process for a paper submitted to a journal, the review paper task is executed several times depending on the content of the paper, the availability of referees and the credentials of the authors. The review process can only continue when all reviews have been returned;
When dispensing a prescription, the *weigh compound* task must be completed for each ingredient before the preparation can be compounded and dispensed.

**Motivation** The *Multiple Instances with a priori Run-Time Knowledge* pattern provides a means of executing multiple instances of a given task in a synchronized manner with the determination of exactly how many instances will be created being deferred to the latest possible time before the first of the tasks is started.

**Overview** As with other multiple instance patterns, there are two variants of this pattern depending on whether the instances are created sequentially or simultaneously as illustrated in Figures 133 and 134. In both cases, the number of instances of task C to be executed (indicated in these diagrams by the variable *numinst*\(^6\)) is communicated at the same time that the thread of control is passed for the process instance. In this realization, the same CPN patterns are used as for the *MI with Design-Time Knowledge* pattern.

![Figure 133: Multiple Instances with a priori Run-Time Knowledge (Variant 1)](image)

![Figure 134: Multiple Instances with a priori Run-Time Knowledge (Variant 2)](image)

**Context** There is one context condition associated with this pattern: the number of task instances (i.e. *numinst*) is known at runtime prior to the creation of instances of the task. Once determined, the number of task instances is a fixed value.

**Implementation** Staffware, FLOWer and UML 2.0 ADs support the simultaneous variant of the pattern through the use of dynamic subplan and ExpansionRegion constructs respectively. BPMN and XPDL support both options via the multi-instance loop task construct. In the case of FLOWer, BPMN and XPDL, the actual number of instances required is indicated through a variable passed to the construct at runtime. For UML 2.0 ADs, the ExpansionRegion construct supports multiple instantiations of a task based on the number of instances of a defined data element(s) passed at runtime. Oracle BPEL supports the pattern via its (unique) `<flowN>` construct.

**Issues** None identified.

\(^6\)Note that ‘numinst’ was first used as a constant and is now being used as a variable.
**Pattern WCF-27 (MI without a priori Run-Time Knowledge)**

**Description** Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.

**Example**

- The despatch of an oil rig from factory to site involves numerous transport shipment tasks. These occur concurrently and although sufficient tasks are started to cover initial estimates of the required transport volumes, it is always possible for additional tasks to be initiated if there is a shortfall in transportation requirements. Once the whole oil rig has been transported, and all transport shipment tasks are complete, the next task (assemble rig) can commence.

**Motivation** This pattern is an extension to the Multiple Instances with a priori Run-Time Knowledge pattern which defers the need to determine how many concurrent instances of the task are required until the last possible moment – either when the synchronization of the multiple instances occurs or the last of the executing instances completes. It offers more flexibility in that additional instances can be created “on-the-fly” without any necessary change to the process model or the synchronization conditions for the task.

**Overview** Similar to other multiple instance patterns, there are two variants to this pattern depending on whether the initial round of instances are started sequentially or simultaneously. These scenarios are depicted in Figures 135 and 136. It should be noted that it is possible to add additional instances of task C in both of these implementations via the add instance transition at any time up until all instances have completed and the join associated with them has fired triggering the subsequent task (B). In both realizations, the Distributed Data Processing pattern (cf. page 65) is used in association with transitions A and B to create and synchronize multiple branches respectively. In both implementation variants, the Non-Deterministic XOR-split CPN pattern (cf. page 38) is used to specify that creation of new process instance and completion of the MI task can be executed non-deterministically.

**Context** There is one context condition associated with this pattern: the initial number of task instances (i.e. numinst) is known at runtime prior to the completion of the multiple instance task (note that the final number of instances does not need to be known when initializing the MI task).

**Implementation** Only one of the offerings examined – FLOWer – provides direct support for this pattern. It does this through the dynamic subplan construct.

**Issues** None identified.

**Solutions** N/A.
Evaluation Criteria

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

Pattern WCF-28 (Static Partial Join for Multiple Instances)

Description

Within a given process instance, multiple concurrent instances of a task (say $m$) can be created. The required number of instances is known when the first task instance commences. Once $n$ of the task instances have completed (where $n$ is less than $m$), the next task in the process is triggered. Subsequent completions of the remaining $m-n$ instances are inconsequential, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled.

Example

- Examine 10 samples from the production line for defects. Continue with the next task when 7 of these examinations have been completed.

Motivation

The Static Partial Join for Multiple Instances pattern is an extension to the Multiple Instances with a priori Runtime Knowledge pattern which allows the process instance to continue once a given number of the task instances have completed rather than requiring all of them to finish before the subsequent task can be triggered.
Overview The general format of the Static Partial Join for Multiple Instances pattern is illustrated in Figure 137. This diagram focuses only on one process instance in isolation. Transition A corresponds to the multiple instance task. In terms of the operation of this pattern, once the input place i1 is triggered for a case, m instances of the multi-instance task A are initiated concurrently and an “active” status is recorded for the pattern. These instances proceed independently and once n of them have completed, the join can be triggered and a token placed in output place o1 signalling that the thread of control can be passed to subsequent tasks in the process model. Simultaneously with the join firing, the token is removed from the active place allowing the remaining n - m tasks to complete. Once all m instances of task A have finished, the status of the pattern changes to “ready” allowing it to be re-enabled. This realization uses the Distributed Data Processing CPN pattern (cf. page 65) to synchronize branches between transitions start and join. The NBSD Filter CPN pattern (cf. page 47) is used to enable the join transition when sufficient number of instances have completed, and to complete all remaining instances in one go by transition complete afterwards.

Figure 137: Static Partial Join implementation for Multiple Instances

There are two variants of this pattern which relax some of the restrictions associated with the form of the pattern described above. First, the Canceling Partial Join for Multiple Instances pattern removes the need to wait for all of the task instances to complete by canceling any remaining task instances as soon as the join fires. It is illustrated in Figure 138 and discussed further on page 166.

The second, the Dynamic Partial Join for Multiple Instances pattern allows the value of m (i.e. the number of instances) to be determined during the execution of the task instances. In particular, it allows additional task instances to be created “on the fly”. This pattern is illustrated in Figure 139 and described in further detail on page 167.

Context There are two context conditions associated with this pattern: (1) the initial number of concurrent task instances (denoted by variable m in Figure 137) is known prior to task commencement and (2) the number of tasks that need to be completed before subsequent tasks in the process model can be triggered (denoted by variable n in Figure 137) is also known prior to task commencement.

Implementation BPMN and XPDL both appear to offer support for this pattern via the

---

Note that n is a constant in this case.
Multiple Instance Loop Activity construct where the MI Flow.Condition attribute is set to complex and ComplexMI_FlowCondition is an expression that evaluates to true when exactly n instances have completed causing a single token to be passed on to the following task. However no detail is provided to explain how the ComplexMI_FlowCondition is specified hence this is considered to constitute partial support for the pattern.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description and context criteria for the pattern. It achieves partial support if there is any ambiguity associated with the specification of the join condition.

**Pattern WCF-29 (Canceling Partial Join for Multiple Instances)**

**Description** Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered and the remaining m − n instances are canceled.

**Example**
- Run 500 instances of the Protein Test task with distinct samples. Once 400 have completed, cancel the remaining instances and initiate the next task.

**Motivation** This pattern is a variant of the Multiple Instances with a priori Runtime Knowledge pattern that expedites process throughput by both allowing the process to continue to the next task once a specified number (n) of the multiple instance tasks have completed and also cancels any remaining task instances negating the need to expend any further effort executing them.

**Overview** Figure 138 illustrates the operation of this pattern. It is similar in form to that for the Static Partial Join for Multiple Instances pattern (WCF-34) but functions in a different way once the join has fired. At this point any remaining instances which have not already commenced are “bypassed” by allowing the skip task to execute in their place. The skip task executes almost instantaneously for those and the pattern is almost immediately able to reset. In addition to the CPN patterns used in the Static Partial Join for MI pattern, this realization uses the Non-Deterministic XOR-Split CPN pattern (cf. page 38) to effect cancelation of the task individual instances (i.e. either transition A or skip is executed depending on the circumstances).

![Figure 138: Canceling Partial Join implementation for Multiple Instances](image-url)
Context This pattern has the same context conditions as the Static Partial Join for Multiple Instances pattern: (1) the number of concurrent task instances (denoted by variable m in Figure 137) is known prior to task commencement and (2) the number of tasks that need to complete before subsequent tasks in the process model can be triggered (denoted by variable n in Figure 137) is also known prior to task commencement.

Implementation This pattern relies on the availability of a Cancel Task or Cancel Region capability within an offering and at least one of these patterns needs to be supported for this pattern to be facilitated. Both BPMN and XPDL appear to offer support for this pattern by associating an error type intermediate trigger with the multiple instance task. Immediately following this task is a task that issues a cancel event effectively terminating any remaining task instances once the first n-of-them have completed. However it is unclear how the ComplexMIFlowCondition should be specified to allow the cancelation to be triggered once n task instances have completed.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. An offering achieves partial support if there is any ambiguity associated with the implementation of the pattern (e.g., if it is unclear how the join condition is specified).

Pattern WCP-30 (Dynamic Partial Join for Multiple Instances)

Description Within a given process instance, multiple concurrent instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so has not been disabled. A completion condition is specified which is evaluated each time an instance of the task completes. Once the completion condition evaluates to true, the next task in the process is triggered. Subsequent completions of the remaining task instances are inconsequential and no new instances can be created.

Example

– The despatch of an oil rig from factory to site involves numerous transport shipment tasks. These occur concurrently and although sufficient tasks are started to cover initial estimates of the required transport volumes, it is always possible for additional tasks to be initiated if there is a shortfall in transportation requirements. Once 90% of the transport shipment tasks are complete, the next task (invoice transport costs) can commence. The remaining transport shipment tasks continue until the whole rig has been transported.

Motivation This pattern is a variant of the Multiple Instances without a priori Runtime Knowledge pattern that provides the ability to trigger the next task once a nominated completion condition is satisfied.

Overview Figure 139 illustrates the operation of this pattern. The multiple instance task is illustrated by transition A. At commencement, the number of instances initially required is indicated by variable m. Additional instances may be added to this at any time via the start instance transition. At commencement, the pattern is in the active state. Once enough instances of task A have completed and the join transition has fired, the next task
is enabled (illustrated via a token being placed in the output place $o_1$) and the remaining instances of task $A$ run to completion before the **complete** transition is enabled. No new instances can be created at this time. Finally when all instances of $A$ have completed, the pattern resets and can be re-enabled. An important feature of the pattern is the ability to disable further creation of task instances at any time after the first instances have been created. In this realization, the same set of CPN patterns is used as for the **Static Partial Join for MI** pattern. In addition, the **BSD Filter** CPN pattern (cf. page 43) is used to allow the dynamic creation of new process instances by the **start instance** transition once the initial tasks have been initiated up to the time when the **join** transition fires.

**Figure 139:** Dynamic Partial Join implementation for Multiple Instances

**Context** This pattern has two context conditions: (1) the number of concurrent task instances to be started initially (denoted by variable $m$ in Figure 137) is known prior to task commencement and (2) it must be possible to access any data elements or other necessary resources required to evaluate the completion condition at the conclusion of each task instance.

**Implementation** Of the offerings identified, only FLOWer provides support for the dynamic creation of multiple instance tasks (via dynamic subplans), however it requires all of them to be completed before any completion conditions associated with a dynamic subplan (e.g., partial joins) can be evaluated and subsequent tasks can be triggered. This is not considered to constitute support for this pattern.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumptions. It achieves partial support if the creation of task instances cannot be disabled once the first task instance has commenced.
Concurrency Control patterns describe situations where a number of restrictions are imposed on the execution order of several available branches, i.e. none of the branches may execute simultaneously, and where one branch may not proceed if the other branch is not in a certain state. The original Interleaved Parallel Routing pattern is extended to two new patterns: Critical Section and Interleaved Routing. These three patterns have been combined together with the Milestone pattern as each of them operate in the context where execution of one branch is dependent on the state of other branch(es).

This group of patterns consists of four patterns:

- the Interleaved Routing pattern denotes situations where a group of tasks are executed sequentially in any order, providing that none of them execute concurrently;
- the Interleaved Parallel Routing pattern imposes a partial ordering over the execution of tasks described by the Interleaved Routing pattern;
- the Critical Section pattern extends a group of tasks, whose execution must be interleaved as specified by the Interleaved Routing pattern, to a group of process fragments, thus providing the ability to prevent concurrent execution of specific parts of a process within a given process instance; and
- the Milestone pattern describes the synchronization of two distinct branches of a process instance, such that one branch cannot proceed unless the other branch is in a specified state.

Pattern WCF-31 (Interleaved Routing)

**Description** Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated.

**Example**

- The check-oil, test-feeder, examine-main-unit and review-warranty tasks all need to be undertaken as part of the machine-service process. Only one of them can be undertaken at a time, however they can be executed in any order.

**Motivation** The Interleaved Routing pattern allows a sequence of tasks to be executed in any order, but not simultaneously.

**Overview** Figure 140 illustrates the operation of this pattern. After A is completed, tasks B, C, and D can be completed in any order. The mutex place ensures that only one of them can be executed at any time. After all of them have been completed, task E can be undertaken. The realization of the mutex place corresponds to a simple variant of the Lock Manager CPN pattern (cf. page 98). Thread split and join associated with transitions A and E correspond to the Distributed Data Processing (cf. page 65), where the role of the data perspective is minimal and is used for the purpose of the process instance identification (as described by the ID Matching CPN pattern on page 49)).

**Context** There is one consideration associated with the use of this pattern: tasks must be initiated and completed on a sequential basis, in particular it is not possible to suspend one task during its execution to work on another.
Chapter 4 Workflow Control-Flow Patterns

Implementation In order to effectively implement this pattern, an offering must have an integrated notion of state that is available during execution of the control-flow perspective. COSA has this from its Petri net foundation and is able to directly support the pattern. Other offerings lack this capability and hence are not able to directly support this pattern. BPEL (although not Oracle BPEL) can achieve similar effects using serializable scopes within the context of a <pick> construct. FLOWer has a distinct foundation to that inherent in other workflow products in which all tasks in a case are always allocated to the same resource for completion hence interleaving of task execution is guaranteed, however it is also possible for a resource to suspend a task during execution to work on another hence the context conditions for this pattern are not fully satisfied. BPMN and XPDL indirectly support the pattern through the use of ad-hoc processes however it is unclear how it is possible to ensure that each task in the ad-hoc subprocess is executed precisely once.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. An offering is rated as having partial support if it has limitations on the range of tasks that can be coordinated (e.g., tasks must be in the same process block) or if it cannot enforce that tasks are executed precisely once or ensure tasks are not able to be suspended once started whilst other tasks in the interleave set are commenced.

Pattern WCF-32 (Interleaved Parallel Routing)

Description A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can be completed in any order that accords with the partial order. Moreover, any tasks in the set can be routed to resources for execution as soon as they are enabled, thus there is the provision within the partial ordering for parallel routing of tasks should more than one of them be enabled simultaneously and there is no necessity that they be routed sequentially. However, there is an additional requirement, that no two tasks can be executed at the same time (i.e. no two tasks in the set can be active for the same process instance at the same time), hence the execution of tasks is also interleaved.

Example

![Interleaved Routing pattern](image-url)
When despatching an order, the *pick goods*, *pack goods* and *prepare invoice* tasks must be completed. The *pick goods* task must be done before the *pack goods* task. The *prepare invoice* task can occur at any time. Only one of these tasks can be done at any time for a given order.

**Motivation** The *Interleaved Parallel Routing* pattern offers the possibility of relaxing the strict ordering that a process usually imposes over a set of tasks. Note that *Interleaved Parallel Routing* is related to mutual exclusion, i.e. a semaphore makes sure that tasks are not executed at the same time without enforcing a particular order.

**Overview** Figure 141 provides an example of *Interleaved Parallel Routing*. Place p3 enforces that tasks B, C and D be executed in some order. In this example, the permissible task orderings are: ABDCE, ABCDE and ACBDE. For the realization of this CPN model, the same set of the CPN patterns as for the *Interleaved Routing* pattern has been used.

**Context** There is one context condition associated with this pattern: tasks must be initiated and completed on a sequential basis and it is not possible to suspend one task during its execution to work on another.

**Implementation** In order to effectively implement this pattern, an offering must have an integrated notion of state that is available during execution of the control-flow perspective. COSA has this from its Petri net foundation and is able to directly support the pattern. Other offerings lack this capability and hence are not able to directly support this pattern. BPEL (although surprisingly not Oracle BPEL) can indirectly achieve similar effects using serializable scopes within the context of a <pick> construct although only tasks in the same block can be included within it. It also has the shortcoming that every permissible execution sequence of interleaved tasks must be explicitly modeled. FLOWer has a distinct foundation to that inherent in other workflow products in which all tasks in a case are always allocated to the same resource for completion hence interleaving of task execution is guaranteed, however it is also possible for a resource to suspend a task during execution to work on another hence the context condition for this pattern is not fully satisfied.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. It achieves a partial support rating if there are any limitations on the set of tasks that can be interleaved or if tasks can be suspended during execution.
Pattern WCF-33 (Critical Section)

**Description** Two or more connected subgraphs of a process model are identified as “critical sections”. At runtime for a given process instance, only tasks in one of these “critical sections” can be active at any given time. Once execution of the tasks in one “critical section” commences, it must complete before another “critical section” can commence.

**Example**

- Both the *take-deposit* and *insurance-payment* tasks in the holiday booking process require the exclusive use of the *credit-card-processing* machine. Consequently only one of them can execute at any given time.

**Motivation** The *Critical Section* pattern provides a means of limiting two or more sections of a process from executing concurrently. Generally this is necessary if tasks within this section require exclusive access to a common resource (either data or a physical resource) necessary for a task to be completed. However, there are also regulatory situations (e.g., as part of due diligence or quality assurance processes) which necessitate that two tasks do not occur simultaneously.

**Overview** The operation of this pattern is illustrated in Figure 142. The *mutex* place serves to ensure that within a given process instance, only the sequence BD or CE can be active at any given time. For the realization of this CPN model, the same set of the CPN patterns as for the *Interleaved Routing* pattern has been used, with the only difference being that the scope of the process fragments interleaved via the *mutex* place is larger and consists of multiple transitions.

![Figure 142: Critical Section pattern](image)

**Context** There is one consideration associated with the use of this pattern: tasks in critical sections must be initiated and completed on a sequential basis, in particular it is not possible to suspend one task during its execution to work on another.

**Implementation** Although useful, this pattern is not widely supported amongst the offerings examined. BPEL allows it to be directly implemented through its serializable scope functionality. COSA supports this pattern by including a mutex place in the process model to prevent concurrent access to critical sections. FLOWer provides indirect support through the use of data elements as semaphores.

**Issues** None identified.

**Solutions** N/A.
**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. Where an offering is able to achieve similar functionality through additional configuration or programmatic extension of its existing constructs (but does not have a specific construct for the pattern) this qualifies as partial support.

**Pattern WCF-34 (Milestone)**

**Description** A task is only enabled when the process instance (of which it is part) is in a specific state (typically in a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

**Example**
- Most budget airlines allow the routing of a booking to be changed providing the ticket has not been issued;
- The enrol student task can only execute whilst new enrolments are being accepted. This is after the open enrolment task has completed and before the close off enrolment task commences.

**Motivation** The Milestone pattern provides a mechanism for supporting the conditional execution of a task or subprocess (possibly on a repeated basis) where the process instance is in a given state. The notion of state is generally taken to mean that control-flow has reached a nominated point in the execution of the process instance (i.e. a Milestone). As such, it provides a means of synchronizing two distinct branches of a process instance, such that one branch cannot proceed unless the other branch has reached a specified state.

**Overview** The nominal form of the Milestone pattern is illustrated by Figure 143. Task A cannot be enabled when it receives the thread of control unless the other branch is in state p1 (i.e. there is a token in place p1). This situation presumes that the process instance is either in state p1 or will be at some future time. It is important to note that the repeated execution of A does not influence the top parallel branch. Note that the A transition performs synchronization of threads belonging to the same process instance (for this purpose the ID Matching CPN pattern (cf. page 49) is used).

![Figure 143: Milestone pattern](image)

Note that A can only occur if there is a token in p1. Hence a Milestone may cause a potential deadlock. There are at least two ways of avoiding this. First of all, it is possible to define an alternative task for A which takes a token from the input place(s) of A without taking a token from p1. One can think of this task as a time-out or a skip task. This way the process does not get stuck if C occurs before A. Moreover, it is possible to delay the
execution of C until the lower branch finishes. Note that in both cases A may be optional (i.e. not execute at all) or can occur multiple times because the token in p1 is only tested and not removed.

**Context** There are no specific context conditions for this pattern.

**Implementation** The necessity for an inherent notion of state within the process model means that the *Milestone* pattern is not widely supported. Of the offerings examined, only COSA is able to directly represent it. FLOWer offers indirect support for the pattern through the introduction of a data element for each situation in which a *Milestone* is required. This data element can be updated with a value when the *Milestone* is reached and the branch which must test for the *Milestone* achievement can do so using the FLOWer milestone construct. Note that this is only possible in a data-driven system like FLOWer. It is not possible to use variables this way in a classical control-flow driven system because a “busy wait” would be needed to constantly inspect the value of this variable. (Note that FLOWer only re-evaluates the state after each change with respect to data elements).

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description for the pattern. It receives a partial support rating if there is not a specific construct for the *Milestone* but it can be achieved indirectly.

**Class:** Triggering patterns

Triggering patterns describe situations where the moment at which a work item commences needs to be synchronized with a signal (or trigger) from external environment. The ability to respond to external signals within a process instance was not well covered by the original patterns other than by the *Deferred Choice* (cf. page 127) which allows a decision regarding possible execution paths to be based on environmental input. Therefore two new patterns are introduced to denote the ability of external signals to affect process execution. Depending on the nature of the trigger, we distinguish the *Transient Trigger* and *Persistent Trigger* patterns.

**Pattern WCF-35 (Transient Trigger)**

**Description** The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received.

**Examples**
- Start the *Handle Overflow* task immediately when the *dam capacity full* signal is received.
- If possible, initiate the *Check Sensor* task each time an *alarm trigger signal* is received.

**Motivation** Transient triggers are a common means of signalling that a pre-defined event has occurred and that an appropriate handling response should be undertaken – comprising either the initiation of a single task, a sequence of tasks or a new thread of execution in a process. Transient triggers are events which must be dealt with as soon as they are received. In other words, they must result in the immediate initiation of a task. The process provides
no form of memory for transient triggers. If they are not acted on immediately, they are irrevocably lost.

**Overview** There are two main variants of this pattern depending on whether the process is executing in a safe execution environment or not. Figure 144 shows the safe variant, only one instance of task B can wait on a trigger at any given time. Note that place enabled holds a token for each possible process instance. This place ensures that at most one instance of task B exists at any time. In both realization variants, the *ID Matching* CPN pattern (cf. page 49) is used for matching an external trigger with relevant process instance. Furthermore, the safe variant of the transient trigger realization uses the anti-place of the *Capacity Bounding* pattern (cf. page 71) to ensure that only one instance of task B can wait for a trigger.

**Figure 144:** Transient Trigger pattern (safe variant)

The alternative option for unsafe processes is shown in Figure 145. Multiple instances of task B can remain waiting for a trigger to be received. However only one of these can be enabled for each trigger when it is received.

**Figure 145:** Transient trigger pattern (unsafe variant)

**Context** There are no specific context conditions associated with the pattern.

**Implementation** Staffware provides support for transient triggers via the Event Step construct. Similarly COSA provides a trigger construct which can operate in both synchronous and asynchronous mode supporting transient and persistent triggers respectively.
Both of these offerings implement the safe form of the pattern (as illustrated in Figure 144). SAP Workflow provides similar support via the “wait for event” step construct. UML 2.0 ADs provide the ability for signals to be discarded where there are not immediately required through the explicit enablement feature of the AcceptEventAction construct which is responsible for handling incoming signals.

**Issues** One consideration that arises with the use of transient triggers is what happens when multiple triggers are received simultaneously or in a very short time interval. Are the latter triggers inherently lost as a trigger instance is already pending or are all instances preserved (albeit for a potentially short timeframe)?

**Solutions** In general, in the implementations examined (Staffware, COSA and SAP Workflow) it seems that all transient triggers are lost if they are not immediately consumed. There is no provision for transient triggers to be duplicated.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description for the pattern.

### Pattern WCF-36 (Persistent Trigger)

**Description** The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task.

**Examples**
- Initiate the *Staff Induction* task each time a *new staff member* event occurs.
- Start a new instance of the *Inspect Vehicle* task for each *service overdue* signal that is received.

**Motivation** Persistent triggers are inherently durable in nature, ensuring that they are not lost in transit and are buffered until they can be dealt with by the target task. This means that the signalling task can be certain that the trigger will result in the task to which they are directed being initiated either immediately (if it already has received the thread of control) or at some future time.

**Overview** There are two variants of the persistent triggers. Figure 146 illustrates the situation where a trigger is buffered until control-flow passes to the task to which the trigger is directed. Once this task has received a trigger, it can commence execution. Alternatively, the trigger can initiate a task (or the beginning of a thread of execution) that is not contingent on the completion of any preceding tasks. This scenario is illustrated by Figure 147. The realization in Figure 146 uses the *ID Matching* CPN pattern (cf. page 49) to identify the correspondence between the process instance $pi$ and an external trigger $t$ that has been received.

**Context** There are no specific context conditions associated with the pattern.

**Implementation** Of the offerings examined, COSA provide support for persistent triggers via its integrated trigger construct, SAP Workflow has the “wait for event” step construct, FLOWer and FileNet provide the ability for tasks to wait on specific data conditions that can be updated from outside the process. The business process modelling formalisms BPMN, XPDL and BPEL all provide a mechanism for this form of triggering via messages and in all cases the messages are assumed to be durable in nature and can either trigger a standalone task or can enable a blocked task waiting on receipt of a message to continue. UML 2.0 Activity Diagrams provide a similar facility using signals. Although EPCs provide support for multiple input events which can be utilized as persistent triggers, it is not
Section 4.1 Revisiting the control-flow patterns

possible to differentiate between them hence this is viewed as partial support. Note that if the pattern is not directly supported, it is often possible to implement persistent triggers indirectly by adding a dummy task which “catches” the trigger.

**Issues** None identified.

**Solutions** N/A

**Evaluation Criteria** An offering achieves full support if it has a construct that satisfies the description for the pattern. If triggers do not retain a discrete identity when received and/or stored, an offering is viewed as providing partial support.

**Class: Cancelation and Completion patterns**

Cancelation and Completion patterns describe situations where an individual task, an arbitrary group of tasks in a process, or a complete process instances need to be canceled during execution. Previous notions of cancelation only related to individual tasks and complete process instances (as specified in the Cancel Task and Cancel Case patterns respectively). The Cancel Region pattern has been introduced in order to deal with cancelation in a more general sense where arbitrary groups of tasks in a process need to be canceled during execution. Additionally, the multiple instances task is considered to be a special case, for which both the ability to cancel and to force its completion during execution are recognized (as described by the Cancel MI Task and Complete MI Task patterns respectively). Not all task instances may have completed when the decision regarding the cancelation or forced-completion of the MI task is taken. Therefore the Cancel MI Task and Complete MI Task patterns explicitly specify how the remaining instances and the flow of control are dealt with.
In Section 4.2.1 we propose a graphical notation to visually depict the cancelation set associated with a given task. Although this graphical notation is used to differentiate the operational semantics of process modeling entities encountered in any PAIS, one can extend it with multiple instance tasks in order to differentiate between cancelation and force completion.

Pattern WCF-37 (Cancel Task)

**Description** An enabled task is withdrawn prior to or during to its execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed.

**Examples**
- The *assess damage* task is undertaken by two insurance assessors. Once the first assessor has completed the task, the second is canceled;
- The purchaser can cancel their *building inspection* task at any time before it commences.

**Motivation** The *Cancel Task* pattern provides the ability to withdraw a task which has been enabled or is already executing. This ensures that it will not commence or complete execution.

**Overview** The general interpretation of the *Cancel Task* pattern is illustrated by Figure 148. The thread of control which has enabled task B is removed, preventing the task from proceeding. In this realization, we used the *Non-deterministic XOR-Split* CPN pattern (cf. page 38) to define the choice between executing task B and its cancelation on a non-deterministic basis.

![Figure 148: Cancel Task pattern (Variant 1)](image)

There is also a second variant of the pattern where the task has already commenced execution but has not yet completed. This scenario is shown in Figure 149, where a task which has been enabled or is currently executing can be canceled. It is important to note for both variants that cancelation is not guaranteed and it is possible that the task will continue executing to completion. In effect, the cancelation vs continuation decision operates as a *Deferred Choice* with a race condition being set up between the cancelation event and the much slower task of resources responding to work assignment. For all practical purposes, it is much more likely that the cancelation will be effected rather than the task being continued.

Where guaranteed cancelation is required, the implementation of tasks should take the form illustrated in Figure 150. The decision to cancel task B can only be made after it has been enabled and prior to it completing. Once this decision is made, it is not possible for the task to progress any further. For obvious reasons, it is not possible to cancel a task which has not been enabled (i.e. there is no “memory” associated with the action of
canceling a task in the way that there is for triggers) nor is it possible to cancel a task which has already completed execution.

**Context** There are no specific context conditions associated with the pattern.

**Implementation** The majority of the offerings examined provide support for this pattern within their process models. Most support the first variant as illustrated in Figure 148: Staffware does so with the withdraw construct, COSA allows tokens to be withdrawn from the places before tasks, iPlanet provides the AbortActivity method, FileNet provides the \(<\text{Terminate Branch}\)> construct and SAP Workflow provides the process control step for this purpose although it has limited usage. BPEL supports the second variant via fault compensation handlers attached to tasks, as do BPMN and XPDL using error type triggers attached to the boundary of the task to be canceled. UML 2.0 ADs provide a similar capability by placing the task to be canceled in an interruptible region triggered by a signal or another task. FLOWer does not directly support the pattern although tasks can be skipped and redone.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description for the pattern. If there are any side-effects associated with the cancelation (e.g., forced completion of other tasks, the canceled task being marked as complete), the offering is rated as having partial support.

**Pattern WCF-38 (Cancel Region)**

**Description** The ability to disable a set of tasks in a process instance. If any of the tasks
are already executing (or are currently enabled), then they are withdrawn. The tasks need not be a connected subset of the overall process model.

**Examples**
- Stop any tasks in the *Prosecution* process which access the *evidence* database from running.
- Withdraw all tasks in the *Waybill Booking* process after the *freight-lodgement* task.

**Motivation** The option of being able to cancel a series of (potentially unrelated) tasks is a useful capability, particularly for handling unexpected errors or for implementing forms of exception handling.

**Overview** The general form of this pattern is illustrated in Figure 151. It is based on the premise that every task in the required region has an alternate “bypass” task. When the cancelation of the region is required, the process instance continues execution, but the bypass tasks are executed instead of the original tasks. As a consequence, no further work occurs on the tasks in the cancelation region. However, as shown for the *Cancel Case* (WCF-39) pattern, there are several alternative mechanisms that can be used to cancel parts of a process.

**Context** There are no specific context conditions associated with the pattern.

**Implementation** The concept of cancelation regions is not widely supported. Staffware offers the opportunity to withdraw steps but only if they have not already commenced execution. FLOWer allows individual tasks to be skipped but there is no means of canceling a group of tasks. UML 2.0 Activity Diagrams are the only offering examined which provides complete support for this pattern: the InterruptibleActivityRegion construct allows a set of tasks to be canceled. BPMN and XPDL offer partial support by enclosing the tasks that will potentially be canceled in a subprocess and associating an error event with the subprocess to trigger cancelation when it is required. In both cases, the shortcoming of this approach is that the tasks in the subprocess must be a connected subgraph of the overall process model. Similarly BPEL only supports cancelation of tasks in the same scope hence it also achieves a partial rating as it is not possible to cancel an arbitrary group of tasks.
As COSA has an integrated notion of state, it is possible to implement cancelation regions in a similar way to that presented in Figure 151 however the overall process model is likely to become intractable for cancelation regions of any reasonable scale hence this is viewed as partial support.

**Issues** One issue that can arise with the implementation of the Cancel Region pattern occurs when the canceling task lies within the cancelation region. Although this task must run to completion and cause the cancelation of all of the tasks in the defined cancelation region, once this has been completed, it too must be canceled.

**Solutions** The most effective solution to this problem is to ensure that the canceling task is the last of those to be processed (i.e. the last to be terminated) of the tasks in the cancelation region. The actual cancelation occurs when the task to which the cancelation region is attached completes execution.

**Evaluation Criteria** An offering achieves full support if it provides a construct that satisfies the description for the pattern. It rates as partial support if the process model must be changed in any way (e.g., use of subprocesses, inclusion of bypass tasks) in order to accommodate cancelation regions.

**Pattern WCF-39 (Cancel Case)**

**Description** A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all subprocesses. The process instance is recorded as having completed unsuccessfully.

**Examples**
- During an insurance claim process, it is discovered that the policy has expired and, as a consequence, all tasks associated with the particular process instance are canceled;
- During a mortgage application, the purchaser decides not to continue with a house purchase and withdraws the application.

**Motivation** This pattern provides a means of halting a specified process instance and withdrawing any tasks associated with it.

**Overview** Cancelation of an entire case involves the disabling of all currently enabled tasks. Figure 152 illustrates one scheme for achieving this. It is based on the identification of all possible sets of states that the process may exhibit for a process instance. Each combination has a transition associated with it (illustrated by $C_1$, $C_2$, ... etc) that disables all enabled tasks. Where cancelation of a case is enabled, it is assumed that precisely one of the canceling transitions (i.e. $C_1$, $C_2$, ...) will fire canceling all necessary enabled tasks. To achieve this, it is necessary that none of the canceling transitions represent a state that is a superset of another possible state, otherwise tokens may be left behind after the cancelation.

An alternative scheme is presented in Figure 153, where every state has a set of cancelation transitions associated with it (illustrated by $C_1$, $C_2$, ... etc.). When the cancelation is initiated, these transitions are enabled for a very short time interval (in essence the difference between time $t$ and $t + \epsilon$ where $\epsilon$ is a time interval approaching zero), thus effecting an instantaneous cancelation for a given state that avoids the potential deadlocks that might arise with the approach shown in Figure 152. Note that time is used as a priority mechanism as CPNs do not directly support this.

A more general approach to cancelation has been illustrated in Figure 151. This may be used to cancel individual tasks, regions or even whole cases. It is premised on the creation of an alternative “bypass” task for each task in a process that may need to be canceled.
When a cancelation is initiated, the case continues processing but the “bypass” tasks are executed rather than the normal tasks, so in effect no further work is actually achieved on the case.

**Context** There is an important context condition associated with this pattern: cancelation of an executing case must be viewed as unsuccessful completion of the case. This means that even though the case was terminated in an orderly manner, perhaps even with tokens reaching its endpoint, this should not be interpreted in any way as a successful outcome. For example, where a log is kept of events occurring during process execution, the case should be recorded as incomplete or canceled.

**Implementation** There is reasonable support for this pattern amongst the offerings examined. SAP Workflow provides the process control step for this purpose, FileNet provides the `<Terminate Process>` construct, BPEL provides the `<terminate>` construct, BPMN and XPDL provide support by including the entire process in a transaction with an associated end event that allows all executing tasks in a process instance to be terminated. Similarly UML 2.0 ADs achieve the same effect using the InterruptibleActivityRegion construct. FLOWer provides partial support for the pattern through its ability to skip or redo entire cases.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. If there are any side-effects associated with the cancelation (e.g., forced completion of other tasks, the process instance being marked as complete), then the offering is rated as having partial support.

**Pattern WCF-40 (Cancel Multiple Instance Task)**

**Description** Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance task can be canceled
and any instances which have not completed are withdrawn. Task instances that have already completed are unaffected.

**Example**
- Run 500 instances of the *Protein Test* task with distinct samples. If it has not completed one hour after commencement, cancel it.

**Motivation**
This pattern provides a means of canceling a multiple instance task at any time during its execution such that any remaining instances are canceled. However any instances which have already completed are unaffected by the cancelation.

**Overview**
There are two variants of this pattern depending on whether the task instances are started sequentially or simultaneously. These scenarios are depicted in Figures 154 and 155. In both cases, transition \( C \) corresponds to the multiple instance task, which is executed \( numinst \) times. When the cancel transition is enabled, any remaining instances of task \( C \) that have not already executed are withdrawn, as is the ability to initiate any additional instances (via the create instance transition). No subsequent tasks are enabled as a consequence of the cancelation. Note that in the sequential and concurrent initiation variants of the Cancel MI Task pattern utilize the Distributed Data Processing CPN pattern (cf. page 65) to synchronize branches between transitions A and B. Additionally, we used the Deterministic XOR-Split CPN pattern (cf. page 36) to define conditions for creating new task instances, completing and canceling the MI task.

**Context**
There is one context condition associated with this pattern: it is assumed that only one instance of each multiple instance task is executing for a given case at any time.

**Implementation**
In order to implement this pattern, an offering also needs to support
Figure 154: Cancel Multiple Instance Task pattern (sequential initiation)

one of the Multiple Instance patterns that provide synchronization of the task instances at completion (i.e. WCF-29 – WCF-31). Staffware provides the ability to immediately terminate dynamic subprocedures albeit with loss of any associated data. SAP Workflow allows multiple instances created from a “multi-line container element” to be terminated when the parent task terminates. BPMN and XPDL support the pattern via a MI task which has an error type intermediate event trigger at the boundary. When the MI task is to be canceled, a cancel event is triggered to terminate any remaining MI task instances. Similarly UML 2.0 ADs provide support by including the multiple instance task in a cancellation region. Oracle BPEL is able to support the pattern by associating a fault or compensation handler with a <flowN> construct. As the <flowN> construct is specific to Oracle BPEL, there is no support for this pattern by BPEL more generally.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. If there are any limitations on the range of tasks that can appear within the cancellation region or the types of task instances that can be canceled then an offering achieves a partial rating.

**Pattern WCF-41 (Complete Multiple Instance Task)**

**Description** Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered. During the course of execution, it is possible that the task needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent tasks.

**Example**
Run 500 instances of the *Protein Test* task with distinct samples. One hour after commencement, withdraw all remaining instances and initiate the next task.

**Motivation** This pattern provides a means of finalizing a multiple instance task that has not yet completed at any time during its execution such that any remaining instances are withdrawn and the thread of control is immediately passed to subsequent tasks. Any instances which have already completed are unaffected by the cancelation.

**Overview** There are two variants of this pattern depending on whether the task instances are started sequentially or simultaneously. These scenarios are depicted in Figures 156 and 157. In both cases, transition C corresponds to the multiple instance task, which is executed `numinst` times. When the `complete` transition is enabled, any remaining instances of task C that have not already executed are withdrawn, as is the ability to add any additional instances (via the `create instance` transition). The subsequent task (illustrated by transition B) is enabled immediately. In this realization, we used the counter-place `counters` of the *Capacity-Bounding* CPN pattern (cf. page 71) in order to keep track of the total number of task instances.

The `counters` place in Figure 156 keeps track of task instances in different states: the p1 place stores one token for a given process instance that enables task instance creation, instances that have been initiated are stored in the p2 place, and instances that have completed are stored in the p3 place. Process instances that have not yet completed and which are represented as tokens in the p2 place can be forced to complete by removing tokens both from the p1 and p2 places, and producing a corresponding token in place p3.

Figure 157 represents the semantics of completing the multiple instance task, instances associated with which execute concurrently. The variable `numinst` indicates the number of instances of task C which need to be created. This number is both supplied to the p1 place and the count place. The count place keeps track of how many instances of task C
Figure 156: Complete Multiple Instance Task pattern (sequential initiation)

have been initiated (the value of the counter \( n \) is deducted with 1 each time a new task instance has been created). After execution of each of the task instances a token is placed to the \( p2 \) place. Prior to the initiation of task \( C \) there exists the possibility to force the completion of this task by firing the complete transition.

Figure 157: Complete Multiple Instance Task pattern (concurrent initiation)

Context There is one context condition associated with this pattern: only one instance of a multiple instance task can execute at any time.

Implementation In order to implement this pattern, an offering also needs to support one of the Multiple Instance patterns that provide synchronization of the task instances at completion (i.e. WCF-29 – WCF-31). FLOWer provides indirect support for this pattern via the auto-complete condition on dynamic plans which force-completes unfinished plans when the condition evaluates to true however this can only occur when all subplans have completed. Similarly, it also provides deadline support for dynamic plans which ensures that all remaining instances are forced complete once the deadline is reached, however this action also causes all subsequent tasks to be force completed as well.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. It demonstrates partial support if there are limitations on when the
completion task can be initiated or if the force completion of the remaining instances does not result in subsequent tasks in the process instance being triggered normally.

Class: Termination patterns

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Example</th>
<th>Motivation</th>
<th>Overview</th>
<th>Context</th>
<th>Implementation</th>
<th>Issues</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination patterns</td>
<td>Define two alternative approaches to recognizing process completion. An implicit termination of a process instance, which occurs when there are no remaining work items that can be executed, was previously described by the Implicit Termination pattern. The Explicit Termination pattern is introduced as an alternative approach to process instance termination which occurs when a dedicated end-node is executed and a specific nominated state is reached.</td>
<td>The research project is considered to be completed when all students have completed courses, wrote reports, and no outstanding tasks have been remained.</td>
<td>The rationale for this pattern is that it represents the most realistic approach to determining when a process instance can be designated as complete. This is when there is no remaining work to be completed as part of it and it is not possible that work items will arise at some future time.</td>
<td>Figure 158 illustrates a process model where after executing task A multiple branches can be enabled simultaneously. The given process instance does not terminate when the execution of tasks in one of the branches has completed, as there are (potentially) tasks left in other branches which still could be executed. An Implicit Termination pattern presumes that the execution of a process instance terminates when no tasks that still could be executed remain. In CPN terms this means that a process instance terminates when all tokens have reached the end of the process. Workflow offerings that do not enforce all branches in a process model to be synchronized with a single end node, but which do support implicit termination, implement the behavior which is illustrated in Figure 159. Note that this diagram represents only the semantics of the pattern and does not specify how exactly such behavior needs to be implemented in the offerings.</td>
<td>There are no specific context conditions associated with this pattern.</td>
<td>Staffware, WebSphere MQ, FLOWer, FileNet, BPEL, BPMN, XPDL, UML 2.0 ADs and EPCs support this pattern. iPlanet requires processes to have a unique end node. COSA terminates a process instance when a specific type of end node is reached.</td>
<td>Where an offering does not directly support this pattern, the question arises as to whether it can implement a process model which has been developed based on the notion of Implicit Termination.</td>
<td>For simple process models, it may be possible to indirectly achieve the same effect by replacing all of the end nodes for a process with links to an OR-join which then links to a single final node. However, it is less clear for more complex process models involving multiple instance tasks whether they are always able to be converted to a model</td>
</tr>
</tbody>
</table>
Figure 158: Example of a process with multiple branches whose execution needs to be implicitly terminated

Figure 159: Implicit Termination pattern: a process instance implicitly terminates when no tasks left which still can be execute

with a single terminating node. Potential solutions to this are discussed at length by Kiepuszewski et al. [137].

It is worthwhile noting that some languages do not offer this construct on purpose: the Implicit Termination pattern makes it difficult (or even impossible) to distinguish proper termination from deadlock. Often it is only through examination of the process log that it is possible to determine if a particular case has actually finished. Additionally, processes without explicit endpoints are more difficult to use in compositions.

Evaluation Criteria An offering achieves full support if it is possible to have multiple final nodes and the behavior of these nodes satisfies the description for the pattern.

Pattern WCF-43 (Explicit Termination)

Description A given process (or subprocess) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node
Section 4.1 Revisiting the control-flow patterns

is reached, any remaining work in the process instance is canceled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed.

**Example** The recruitment process terminates when a suitable candidate has been found for a specific position. As a consequence of this, any pending interactions with remaining candidates are stopped.

**Motivation** The rationale for this pattern is that it represents an alternative means of defining when a process instance can be designated as complete. This is when the thread of control reaches a defined state within the process model. Typically this is denoted by a designated termination node at the end of the model. Where there is a single end node in a process, its inclusion in other compositions is simplified.

**Overview** Figures 160 and 161 illustrate the semantics of *Explicit Termination* pattern in the context of a process model with a single and multiple end nodes respectively.

**Figure 160:** Explicit Termination pattern: after reaching a designated end-node, the process instance terminates

For models with a single end node, as shown in Figure 160, after executing the End transition, a token is produced to the o1 place indicating the termination of the given process instance. This assumes that any remaining work is canceled.

In Figure 161 a process model with two end nodes is shown. When any of these nodes is reached, a process instance is considered to be successfully completed. In order to illustrate that any remaining work is canceled after an end node is reached, each of the transitions End1 and End2 is associated with a cancelation region.
Chapter 4 Workflow Control-Flow Patterns

Figure 161: Explicit Termination pattern: after reaching any of the designated end-points the process instance terminates and any remaining tokens are withdrawn

**Context** There is one context condition associated with this pattern: every task in a process must be on a path from a designated start node to a designated end node.

**Implementation** COSA, iPlanet, SAP Workflow, BPMN, XPDL and UML 2.0 ADs support this pattern although other than iPlanet, none of these offerings enforce that there is a single end node.

**Issues** One consideration that does arise where a process model has multiple end nodes is whether it can be transformed to one with a single end node.

**Solutions** For simple process models, it may be possible to simply replace all of the end nodes for a process with links to an OR-join which then links to a single final node. However, it is less clear for more complex process models involving multiple instance tasks whether they are always able to be converted to a model with a single terminating node. Potential solutions to this are discussed at length elsewhere [137].

**Evaluation Criteria** An offering achieves full support for this pattern if it demonstrates that it can meet the description and context criterion for the pattern.

4.1.4 Relationships between control-flow patterns

In order to facilitate an understanding the commonalities and differences between patterns in order to enable selection of the best pattern for a specific purpose, we have analyzed which patterns can be implemented through combination of other patterns and which patterns represent a more restricted form of another pattern. Figure 162 illustrates two types of relationships between patterns: a composition relationship and a specialization relationship. A composition relationship is shown with a dashed line and specialization relationship is shown with a solid line.

An example of a composition relationship is the Multi-Choice pattern (cf. page 128) which can be realized by combining the Parallel Split pattern (cf. page 124) with the Exclusive Choice pattern (cf. page 125). Another example is the Structured Loop pattern (cf. page 155) which can be realized by combining the Exclusive Choice pattern (cf. page 125) with the Multi-Merge pattern (cf. page 146).
Section 4.2 Patterns operationalization

4.2 Patterns operationalization

An example of a specialization relationship is the Parallel Split pattern (cf. page 124) which represents a special case of the Multi-Choice pattern (cf. page 128). The Multi-Choice pattern can be configured to unconditionally enable all outgoing branches as required by the Parallel Split pattern. A pattern can represent a generic variant for several specializations. For instance, the Synchronization pattern (cf. page 131) has four specializations, whilst the Structured Discriminator (cf. page 135) has two specializations.

Having described the control-flow patterns, we now move on to an analysis of the constructs required for their implementation in PAISs.

4.2 Patterns operationalization

The control-flow patterns presented in Section 4.1.3 provide an insight into generic constructs that are commonly used in workflow offerings for describing possible behavior in
the control-flow perspective. Since the control-flow patterns abstract from the details of their operationalization, distinct offerings may implement the same pattern in different ways. To illustrate the variations in the modeling languages used by workflow systems, we briefly review a set of PAISs: YAWL, COSA, Staffware and Oracle BPEL PM. In doing so, we show that even basic constructs such as the XOR-join and AND-join (the Petri-net representations of which are given in Figure 163), which are supported by the majority of workflow systems, are not implemented in a uniform way. In order to differentiate between different implementation approaches, in Section 4.2.1 we examine the core process constructs that can be encountered in any offering and propose a formal language that offers an intuitive graphical notation to depict differences between distinct implementation approaches in a uniform and language-independent way.

In Petri-nets, states, tasks and causal dependencies between them are modeled by places, transitions, and arcs respectively. The XOR-join specifies that several distinct paths come together without synchronization (cf. a place \( p_1 \) with ingoing arcs from transitions A and B in Figure 163(a)). The AND-join specifies the synchronization of multiple paths (cf. a transition C with incoming arcs from places \( p_1 \) and \( p_2 \) in Figure 163(b)).

![Figure 163: XOR-join (a) and AND-join (b) in Petri Nets](image)

The Petri-net based workflow system YAWL offers direct support for the XOR-join and AND-join constructs (cf. Figure 164(a) and (b) respectively). Places in YAWL have unbounded capacity, and in order to enable a task, each input place must contain at least one token. YAWL also allows the modeling of an OR-join (Figure 164(c)), which is a construct that may behave like an XOR-join or an AND-join (or a mixture of the two) depending on the context in which it is used. The OR-join waits until no additional tokens can arrive and only then does it fire.

![Figure 164: Notation for (a) XOR-join, (b) AND-join, and (c) OR-join in YAWL](image)

COSA is a Petri-net based workflow management system. The main building blocks of COSA are states, activities, and transitions, which are based directly on the concepts of

---

8Note that the CPN models were only added to provide semantics and not intended as an end-user language implementation.
Petri-nets as places, transitions, and arcs respectively. Figures 165(a) and (b) depict the XOR-join and the AND-join constructs respectively. A COSA model can be considered as a safe Petri-net, i.e. at most one token can be stored in a place at any given time. Hence, activities block when the output states are not empty. For example, activity $A$ blocks when there is a token in place $s_1$. As a result, COSA behaves differently from YAWL and ordinary Petri nets.

Figure 165: Notation for XOR-join (a) and AND-join (b) in COSA: activities block if not all output states are empty

The XOR-join and AND-join constructs in Staffware are denoted by means of Step and Wait objects (cf. models (a) and (b) of Figure 166 respectively). Step $C$ behaves as an XOR-join, i.e. it is triggered when Step $A$ or Step $B$ has completed. Only one instance of $C$ can be active at a time. For instance, if Step $C$ is still active and a new trigger arrives, it will be ignored and all information associated with it will be lost. Note that this way a so-called “race condition” is created.

Figure 166: Constructs for (a) an XOR-join and (b) an AND-join in Staffware. The default semantics of a step (e.g., $C$) is an XOR-join. A wait step (sand-timer symbol) needs to be inserted to synchronize flows (AND-join)

Just like COSA, Staffware forces intermediate states to be “safe”. However, COSA enforces safeness by blocking activities, while Staffware simply removes excess triggers. The Wait object synchronizes left and top input arcs. This object may only have one left and up to 16 top arcs. However, the Wait object can only be triggered by a signal arriving at the left arc. When the object has been triggered, it starts evaluating the status of the
Chapter 4 Workflow Control-Flow Patterns

top arcs. When all input arcs provided input, the \texttt{Wait} object executes. Note that when used in a loop, the \texttt{Wait} object behaves differently. For instance, if Step B is in the loop, but Step A is not, then for the repeated enabling of Step C, it is sufficient for Step B to complete. However, if both steps A and B are in the loop, they both must complete in order to trigger Step C again.

Currently there are many systems supporting BPEL [94]. In this thesis, we selected Oracle BPEL as a representative of this class. Oracle BPEL PM implements the XOR-join and the AND-join by means of BPEL activities \texttt{<switch>} and \texttt{<flow>} as shown in Figures 167(a) and (b) respectively. In contrast to YAWL, Staffware and COSA, these constructs are applied within a structured workflow, i.e. every join is preceded by a corresponding split-construct. This way the corresponding processes are safe and the exceptional situations mentioned for COSA and Staffware cannot occur.

![Figure 167: Constructs for an XOR-join and an AND-joins in Oracle BPEL PM. Although BPEL is a textual language, a graphical interface is provided which directly reflects the BPEL code](image)

We have shown that the implementation of XOR- and AND-joins in COSA, YAWL, Staffware and Oracle BPEL PM are based on different assumptions in regard to the manner in which these constructs behave. Thus, even simple constructs such as the XOR-join and AND-join are not interpreted in a uniform way in different PAISs.

In order to describe how the control-flow patterns can be operationalized, we need to describe the semantics of the constructs used for their realization. With this aim in mind, in Section 4.2.1 we concentrate on a subset of patterns and describe the constructs required for their realization in a uniform, precise and language-independent way. To achieve this, we define a \textit{Core Process Constructs Modeling Language} that provides a graphical notation to depict various core constructs available in every PAIS. Note that in doing so, we abstract from more complex process features as multiple-instance tasks, triggers and hierarchical decomposition.

### 4.2.1 Core Process Constructs Modeling Language

In this subsection, we describe the core process modeling constructs that are typically encountered in every PAIS in the form of the \textit{Core Process Constructs Modeling Language} (CPC-ML). This language is formally defined, and offers a graphical notation to depict
individual variants of these process constructs. The goal of CPC-ML is to provide an instrument for comparing different approaches to the implementation of the control-flow patterns in distinct PAISs.

The scope of the language is limited to a single process instance, covering the details of task routing, while leaving external relationships with other process instances, processes, and the external environment out of consideration. There are two fundamental premises in regard to the semantics of CPC-ML. First, all behavior in the modeled process is associated with active tasks, i.e. tasks whose execution changes the state of the modeled system. Second, the modeled process behaviors are message-driven and discrete. Discrete means that a modeled system is characterized by a certain state at each moment in time.

Figure 168 illustrates the control-flow patterns, the core process constructs inherent to CPC-ML, and the hierarchical relationship between them.

Figure 168: Relation between control-flow patterns and core modeling constructs

In order to describe this figure, first we need to introduce a new concept. A Generic Workflow Net (GWF-net) is a language-independent representation of a workflow model, which can be created using process entities encountered in any PAIS, expressed in terms of CPC-ML constructs. Three structural constructs the GWF-net is composed of, i.e. a task, a channel, and a message are represented on the bottom layer. In terms of Petri nets, a task, a channel, and a message correspond to a transition, a place, and a token respectively. A task is an abstraction of an activity, characterized by a set of inputs and outputs, assigned to a certain resource. Note that we concentrate only on atomic tasks. A message refers to the task input/output expressed in terms of a basic or complex data structure. A message is an abstraction of the control data, used for routing purposes, and/or the production data, i.e. any information (excluding control data) that can be manipulated as a discrete entity for the purpose of executing a certain activity. Note that by using messages we abstract from the actual data contained in the messages. A channel connects tasks and is used to convey messages.

The next layer is behavioral; it addresses the details of inter-task and intra-task behaviors. The intra-task area defines the variants of the task behavior based on the task properties, while the inter-task area addresses different ways of combining the structural entities together.

At the highest level of abstraction there is a set of extensible control-flow patterns, which are obtained as a result of the interplay between the inter- and intra-task behaviors.

---

9We abstract from multiple instances tasks and composite tasks since these are special cases and are not as common as atomic tasks (note that these could be formally defined as has been done in [15]).
In this context, a control-flow pattern is defined as a three-part rule expressing a relation between a certain context (the lifecycle of a single process instance), a problem (addressing the behavioral aspects of task routing), and a solution (expressed in terms of the structural entities). When expressed using the semantics of CPC-ML, operationalization of the control-flow patterns presented in Section 4.1 can be discussed at a more detailed level.

In Figure 168, we only introduced the main concepts related to the operationalization of the control-flow patterns. Now we will present a detailed description of CPC-ML, all elements of which are graphically depicted in Figure 169.

Tasks send and receive messages to/from channels via ports, which play the role of message gates. Ports producing messages are called output ports. Ports consuming messages are called input ports. Every input port is mapped to a channel, which stores messages. We will refer to the combination of an input port and a channel to which this port is mapped as a task input, whilst we understand a task output to be the combination of an output port and a channel to which this port sends messages. We denote input and output channels as squares residing on the front and back edges of the task block respectively.

Every GWF-net may have multiple input channels and output channels, however exactly one input channel and one output channel are involved in the initiation and termination of the process instance.

Every channel is characterized by a set of parameters, such as the maximal capacity, the minimal capacity, and the enabling status. The maximal capacity parameter defines how many messages the channel may hold at once. A channel with unlimited maximal capacity is called an unbounded channel, while a channel with limited maximal capacity is called a bounded channel. We will refer to bounded channels that are able to hold at most one message at a time, as safe channels. We denote safe, bounded and unbounded channels as a single, double and triple circles respectively. The direction of the arrows represents the message flow.

The minimal capacity parameter of a channel defines at least how many messages the channel must contain in order to make a port, consuming messages from this channel,
enabled. A channel is enabled if its minimal capacity has been reached, otherwise the channel is said to be disabled.

Depending on the enabling status of a channel, an input port mapped to it can be either enabled or disabled. An input port is enabled if the channel to which this port is mapped is also enabled, otherwise the port is considered to be disabled.

Depending on the level of visibility of the transferred messages and the accessibility of them, two types of channels can be distinguished. A local channel (relative to the task input) is used for dedicated message transfer, i.e. when messages sent by a task-producer are to be received by a single dedicated task-consumer, and no other tasks may access the messages stored in this channel. The external channel (relative to the task input) is used for non-dedicated message transfer, i.e. when the message sent by a task-producer to the channel is to be consumed by one of several task-consumers which share access to the messages stored in this channel. To distinguish local and external channels graphically, we merge local channels with input ports.

Input ports which are mapped to local channels are called local input ports, while input ports mapped to external channels are called external input ports. The availability of messages in input channels is a property associated with input ports. A mandatory input port is a port, which must be enabled before the task may commence. An optional input port is a port, the enabling of which is not compulsory for task commencement. The output ports can also be mandatory or optional. An output port produces one message upon task termination. A mandatory output port always produces a message upon task termination. An optional output port produces one message upon the task termination if and only if a data-based condition associated with this port has been satisfied (we do not elaborate on the data conditions, since in the context of this work we abstract from the data perspective). We denote optional and mandatory ports as white and dark squares respectively.

Every task has a set of properties that define the input and output logic of the task and the behavior of the task in an active state. A task is in the active state after it has commenced but before it has terminated. The input sets (IS) of a task define all possible sets of input ports, which must be enabled for task commencement. The input selection mode (ism) of a task defines which set of enabled input ports is to be selected from the input sets.

The message consumption mode (mcm) of a task defines how many messages are to be consumed from the channels attached to the ports selected for consumption. In minimal message consumption mode, the minimal channel capacity is consumed from each of the channels, attached to the enabled ports selected for message consumption. Any non-consumed messages remain in the channels unless these channels are explicitly included in the task cancelation set (CS), which specifies locations from which all messages are to be removed upon task termination. In maximal message consumption mode, all messages available in the channels attached to the enabled ports selected for message consumption, are consumed at once.

The output sets (OS) of a task defines a set of output ports each of which will produce one message at the moment of task termination.

The blocking mode (blp) of output ports is a property defined for each task. In blocked mode, output ports may send messages to output channels if and only if the maximal capacity of the corresponding channels has not been reached. If the maximal capacity of the channel has been reached, the output ports are blocked and wait until the required channel capacity becomes available. In open mode, output ports may send messages to
channels whose maximal capacity has been reached, however these messages will be lost and will not modify the state of the channels.

The cancelation set (CS) defines which parts of the net should be emptied at the time of task termination. Emptying part of a GWF-net corresponds to removing messages from specified locations. Removing messages from a task corresponds to aborting execution of that task. We denote the cancelation set as a dashed-line attached to a task (cf. Figure 172).

Every task has a data-based guard, the status of which influences the enabling status of the task. Furthermore, the join logic of a given task is dependent on the jtype parameter, which specifies whether the processing of the task inputs is local, i.e. based on the messages currently available in the input channels, or future, i.e. postponed until no new messages can arrive at the task inputs.

**Definition of CPC-ML**  The following section formalizes the notions just introduced. First, we define a GWF-net.

**Definition 4.2.1. (GFW-net)**
A generic workflow net (GWF-net) $N$ is a tuple $(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, pto\text{c}, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F)$ where:

- $C$ is a set of channels.
- $LC \subseteq C$ is a set of local channels.
- $EC \subseteq C$ is a set of external channels, such that $LC$ and $EC$ partition $C$, i.e. $LC \cap EC = \emptyset$ and $LC \cup EC = C$. 
- $i \subset C$ is a set of input channels.
- $o \subset C$ is a set of output channels, such that $i \cap o = \emptyset$. 
- $T$ is a set of tasks. 
- $P$ is a set of ports.
- $IP : T \rightarrow \mathcal{P}(P)$ defines a set of input ports for each task. 
- $OP : T \rightarrow \mathcal{P}(P)$ defines a set of output ports of a task $t \in T$, such that $\forall t_1, t_2 \in T : (IP(t_1) \cup OP(t_1)) \cap (IP(t_2) \cup OP(t_2)) = \emptyset$ whenever $t_1 = t_2$ and $IP(t) \cap OP(t) = \emptyset$ for any task $t \in T$. 
- $ManP : T \rightarrow \mathcal{P}(P)$ defines a set of mandatory ports for each task, such that $\forall t \in T : ManP(t) \subseteq (IP(t) \cup OP(t))$ 
- $OptP : T \rightarrow \mathcal{P}(P)$ defines a set of optional ports for each task, such that $\forall t \in T : ManP(t) \cap OptP(t) = \emptyset$ and $(ManP(t) \cup OptP(t) = IP(t) \cup OP(t))$ 
- $ptoc : P \rightarrow C$ maps every port to a channel, such that a single input/output port is mapped to a local channel, while multiple ports can be mapped to the same external channel.

$$\forall p_1, p_2 \in P \forall c \in C : pto\text{c}(p_1) = pto\text{c}(p_2) = c \Rightarrow (p_1 = p_2 \lor c \in EC)$$

As a shorthand we use $\overline{p} = pto\text{c}(p)$ for any $p \in P$. This is generalized for sets: $\overline{X} = \{pto\text{c}(x) | x \in X\}$ for $X \subseteq P$.

- $mincap : C \rightarrow \mathbb{N}$ defines the minimal channel capacity. (Note that a channel with $mincap(c)=0$ behaves like a reset arc [79]).
- $maxcap : C \rightarrow (\mathbb{N}\backslash\{0\}) \cup \{\infty\}$ defines the maximal channel capacity.
if $\text{maxcap}(c) = \infty$, then the channel $c$ is unbounded.

if $\text{maxcap}(c) = k$, where $k \geq 2$, then the channel $c$ is bounded.

if $\text{maxcap}(c) = 1$ then the channel $c$ is safe.

- $\text{blp} : P \rightarrow \{\text{blocked}, \text{open}\}$ defines the blocking mode of all output ports. Note that $\text{dom}(\text{blp}) = \bigcup_{t \in T} \text{OP}(t)$.

- $\text{IS} : T \rightarrow \mathcal{P}(\mathcal{P}(P))$ defines input sets for each task, specifying input ports the enabling of which is sufficient for task commencement, such that $\forall t \in T \forall Q \in \text{IS}(t) : (Q \subseteq \text{IP}(t) \wedge (\text{ManP}(t) \cap \text{IP}(t) \subseteq Q))$.

- $\text{ism} : T \rightarrow \{\text{max}, \text{min}, \text{ran}\}$ defines the input selection mode of a task.

  - $\text{max}$: indicates a “maximal” set of $\text{IS}(t)$, i.e. there is not a larger set $Q$ of enabled input ports in $\text{IS}(t)$ with respect to set inclusion.

  - $\text{min}$: indicates a “minimal” set of $\text{IS}(t)$, i.e. there is not a smaller set $Q$ of enabled input ports in $\text{IS}(t)$ with respect to set inclusion.

  - $\text{ran}$: indicates any set of enabled input ports in $\text{IS}(t)$.

- $\text{OS} : T \rightarrow \mathcal{P}(\mathcal{P}(P))$ defines output sets of a task specifying what sets of output ports are to produce messages upon task termination, such that $\forall t \in T \forall Q \in \text{OS}(t) : (Q \subseteq \text{OP}(t) \wedge (\text{ManP}(t) \cap \text{OP}(t) \subseteq Q))$.

- $\text{mcm} : T \rightarrow \{\text{min}, \text{max}\}$ defines the message consumption mode, i.e. how many messages are to be consumed from the enabled inputs selected according to $\text{ism}(t)$ for the given task $t \in T$.

  - $\text{min}$: consume the number of messages specified by the minimal capacity parameter of the channel.

  - $\text{max}$: consume all messages available in the channel.

- $\text{CS} : T \rightarrow \mathcal{P}(C \cup T \setminus (i \cup o))$ specifies the task cancelation set, i.e. what additional messages are to be removed by emptying a part of the workflow.

- $\text{guard} : T \rightarrow \text{Bool}$ defines the status of the data-based task guard, which influences the enabling status of the task. (Note that the signature of this function might be misleading, since the dependency on data elements is missing due to the abstraction from the data perspective. Given a task $t$, $\text{guard}(t)$ may evaluate to true or false depending on the data values at the moment of evaluation.)

- $\text{jtype} : T \rightarrow \{\text{local}, \text{future}\}$ specifies whether the processing of task inputs is local, i.e. based on the messages currently available in the input channels, or future, i.e. postponed until no more new messages may arrive at the task inputs.

- $F = \{(c, t) \in C \times T | c \in \text{IP}(t)\} \cup \{(t, c) \in T \times C | c \in \text{OP}(t)\}$ is the flow relation.

- every node in the graph $(C \cup T, F)$ is on a directed path from some $c_1 \in i$ to some $c_2 \in o$, i.e.

  $$(\forall x \in C \cup T \exists c_1 \in i \exists c_2 \in o : (c_1, x) \in F^* \wedge (x, c_2) \in F^*)$$

  where $F^*$ is the transitive closure of $F$.

In order to illustrate how the CPC-ML graphical notation can be used in practice, let’s consider a travel agency example.
Example 4.2.2. (Travel Agency)
The typical procedure for booking a business trip consists of several steps: registering a client, booking a flight, a hotel, a car, or a combination thereof, and payment. In order to demonstrate different values of the task and channel attributes, let’s consider three situations, where: (1) payment is performed each time a booking of a car, a hotel, or a flight has been completed; (2) payment is performed only once, i.e. after all initiated bookings have been completed; and (3) payment is performed only once, after the first booking has completed.

Figure 170: Task Pay executes each time one of the three preceding tasks completes

The first CPC-ML specification for multiple payments presented in Figure 170 starts with the Register task which enables tasks for booking of Flight, Hotel and/or Car. Task Pay is executed each time one of the three tasks (Flight, Hotel or Car) completes. In this graphical notation, unbounded channels are used, i.e. channels which are able to store an unlimited number of messages at any moment of time. The minimal capacity mincap of all channels is set to 1, specifying that exactly one message is required from each channel to enable a port attached to the given channel. Although channels are able to store multiple messages, the message consumption mode mcm of all tasks is set to minimal, meaning that exactly one message (as specified by the minimal channel capacity) will be consumed by the corresponding ports, while the rest of the messages will be ignored and thus kept in the channels for subsequent task enabling. In order to show that as the result
of the execution of task `Register` a message is sent to a single task (Flight, Hotel or Car) or their combination, the output set of task `Register`, i.e. OS(\textit{Register}), lists all set of channels that may be enabled upon the termination of the considered task. The enabling of all tasks in Figure 170 is based on the messages currently available in the channels. This is reflected by the \textit{jtype} parameter, which is set to \textit{local}.

![Diagram of task specifications]

\textbf{Figure 171}: Task \textit{Pay} executes only once, i.e. after all started tasks have completed.

The second CPC-ML specification shown in Figure 171 combines individual payments into one payment. Task \textit{Pay} waits until each of the tasks enabled by task `Register` completes. Note that task \textit{Pay} does not synchronize incoming channels if and only if a flight, a hotel or a car is booked. However, if the trip contains two or three elements, task \textit{Pay} is delayed until all have completed. This mechanism is reflected by the parameter \textit{jtype} of task \textit{Pay} which is set to \textit{future}. Moreover, to indicate that the maximal set of input ports from the ones specified in the input sets IS(\textit{Pay}) is to be selected for the message consumption, the input selection mode ism(\textit{Pay}) is set to maximal. For instance, if tasks Hotel and Car were executed, i.e. the messages were placed in channels c5 and c6, then both ports i6 and i7 (attached to these channels respectively) will be enabled.

The third CPC-ML specification shown in Figure 172 enables all three tasks (Flight, Hotel and Car) but executes task \textit{Pay} after the first task has completed. After the payment all running tasks are canceled. In contrast to the two earlier specifications, this CPC-ML specification associates a non-empty cancelation set with task \textit{Pay}. The cancelation set
of task Pay contains all channels and tasks which will be emptied the moment the task completes. Graphically the cancelation set of a task is visualized as a dashed rectangle attached to the given task, the scope of which is also indicated by the attribute \( CS \).

Semantics of CPC-ML  Definition 4.2.1 on page 198 specifies the syntax of the GWF-net in mathematical terms, however it does not give any semantics. In order to do this, we need to define the state space and state transitions.

The state space of GWF-net consists of a collection of messages, which serve as wrappers for data\(^{10}\). In order to deal with identical messages which may accumulate in channels we use bags also known as multi-sets. The state of a channel is represented by a multi-set of messages. In order to define the state space, we first introduce some notations.

\(^{10}\)Note that in this specification we do not consider the data perspective.
**Notation**

We denote input ports and output ports of a task \( t \in T \) as \( \bullet t \) and \( t \bullet \), and input channels and output channels of the task as \( \mathcal{I}t \) and \( t\mathcal{O} \) respectively, such that:

\[
\bullet t = IP(t) \\
t \bullet = OP(t) \\
\mathcal{I}t = \mathcal{I} \bullet \\
t \mathcal{O} = t \mathcal{O}
\]

A bag over alphabet \( A \) is a function from \( A \) to the natural numbers \( \mathbb{N} \). For some bag \( X \) over alphabet \( A \) and \( a \in A \), \( X(a) \) denotes the number of occurrences of \( a \) in \( X \), and is referred to as the cardinality of \( a \) in \( X \). \( \emptyset \) denotes the empty bag, \( [a,a,b] \) and \( [a^2,b] \) denote the bag containing two \( a \)'s and one \( b \). Let \( \mathcal{B}(A) \) denote the set of all bags over \( A \). The sum of two bags \( X \) and \( Y \), denoted \( X \cup Y \), is defined as \( [a^n]|a \in A \land n = X(a) + Y(a) \) \). The difference of \( X \) and \( Y \), denoted as \( X - Y \), is defined as \( [a^n]|a \in A \land n = \max((X(a) - Y(a)), 0) \) \). The size of a bag is denoted as \( \text{size}(X) = \sum_{a \in A} X(a) \). The restriction of \( X \) to some domain \( D \subseteq A \), denoted as \( X \upharpoonright D \), is defined as \( [a^X(a)|a \in D] \). Restriction binds more strongly than sum and difference (note that the binding of sum and difference is left-associative).

Bag \( X \) is a sub-bag of \( Y \), denoted as \( X \subseteq Y \), iff for all \( a \in A \), \( X(a) \leq Y(a) \). \( X \subseteq Y \) iff \( X \subseteq Y \) and for some \( a \in A \), \( X(a) < Y(a) \). Note that any finite set of elements from \( A \) also denotes a unique bag over \( A \), namely the function yielding 1 for every element in the set and 0 otherwise. Therefore, finite sets can also be used as bags. If \( X \) is a bag over \( A \) and \( Y \) is a finite subset of \( A \), then \( X - Y, X \cup Y, Y - X, Y \cup X \) yield bags over \( A \). Let \( \text{set}(x) \) denote a function which transforms a bag \( x \in \mathcal{B}(A) \) into a set, such that \( \text{set}(x) = \{ a \in A | x(a) \geq 1 \} \).

Now we can formally define the state space. In this definition, we consider channels and tasks as locations and the state space is a bag over all locations.

**State space**

**Definition 4.2.3. (State space)**

Let \( N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) \) be a GWF-net. A workflow state \( s \) is a multi-set over the channels and tasks, i.e. \( s \in S \), where \( S = \mathcal{B}(C \cup T) \) is the state space of \( N \).

Whenever there is a need to refer to a set of locations (i.e. either channels or internal task states) marked in state \( s \) by messages, we will use the function \( \text{set}(s) = \{ x \in C \cup T | x \in s \} \).

Let us consider the task lifecycle visualized in Figure 173. This figure shows the internal structure of a task \( t \). Note that in a state \( s \in S \), there is a token in \( \text{Active}_t \) if and only if \( t \in s \). A task is considered to be active if its internal state is marked by a message, i.e. after the task has commenced but before it has terminated.

**Definition 4.2.4. (Task enabling)**

Let \( N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) \) be a GWF-net. The boolean function \( \text{enable}(t,s) \) evaluates to true if and only if for a task \( t \in T \) in state \( s \in S \) the following three conditions are satisfied:

- The task guard evaluates to \( \text{true} \):
Figure 173: The internal task states

\[
\text{guard}(t) = true
\]

- One of the input sets is enabled:
  \[
  (\exists Q \in \text{IS}(t) : Q \subseteq \{ p \in \cdot \text{t} \mid s(\overrightarrow{p}) \geq \text{mincap}(\overrightarrow{p}) \})
  \]
- Let \( S' \) be a set of states reachable from \( s \) (assuming some reachability relation without executing \( t \)). If \( jtype(t) = \text{future} \), then \( \not\exists s' \in S' \mid s' \downarrow (\overrightarrow{t}) \subset s \downarrow (\overrightarrow{t}) \).

Let us clarify the semantics of the message consumption/production using a task with multiple input and output ports connected to a single external input and output channel respectively. This situation is depicted in Figure 174.

Figure 174: Message consumption/production

To illustrate the semantics, we consider two cases:

- If \( \text{IS(Task1)} = \{\{i1\}, \{i2\}\} \) and \( \text{OS(Task1)} = \{\{o1\}, \{o2\}\} \), then a single message is consumed from the channel \( c1 \) and a single message is produced on channel \( c2 \).
- However, if \( \text{IS(Task1)} = \{\{i1, i2\}\} \) and \( \text{OS(Task1)} = \{\{o1, o2\}\} \), then there is an asymmetry in message consumption/production, i.e. a single message is consumed from channel \( c1 \) and two messages are produced for channel \( c2 \). Note that such an asymmetry has to do with the rules for task enabling, message consumption and message production. If the minimal capacity of the channel \( c1 \) has been reached, both input ports \( i1 \) and \( i2 \) of Task 1 become enabled. At the time of task activation, the minimal capacity of the channel \( c1 \) is consumed once by the input ports (either \( i1 \) or \( i2 \)), while at task termination both output ports produce a message to the outgoing channel \( c2 \).

Enablement of a task \( t \) where \( jtype(t) = \text{future} \) and that has multiple inputs which need to be synchronized (we will refer to such tasks as OR-joins) needs to be postponed until no more messages can arrive, thus resulting in the enabling of a larger number of input ports for the OR-join. Since enabling of the OR-join depends on the possible future states, its semantics are non-local. Non-locality of the semantics of the OR-join has been a subject
for a debate, and as a consequence several approaches to handling non-local semantics have been proposed. In [11], Kindler et al. address the problem of non-local semantics in the context of EPCs demonstrating that there is no sound formal semantics for EPCs that is fully compliant with the informal semantics of EPCs. In [138, 139], Kindler defines a non-local semantics of EPCs using techniques from fixed point theory and a pair of corresponding transition relations. The proposed technique is claimed to be applicable to the formalization of all kinds of non-local semantics. In [151], Mendling et al. present a new semantics definition for EPCs which also covers the behavior of the OR-join. In contrast to other semantical proposals for process modeling languages with OR-joins, their definition is applicable for any EPC without imposing a restriction on the syntax and yields into sound behavior. In [231], Wynn et al. propose a general and formal approach to OR-joins in workflow using Reset-nets. The authors examine the concept of the OR-join in the context of the workflow language YAWL and propose an algorithmic approach towards determining OR-join enablement. Because the issue of non-local semantics of OR-joins is a subject of intense an investigation on its own, we consider this issue to be outside of the scope of this thesis. Thus, we simply assume that a suitable approach for dealing with non-local semantics of OR-joins is known. Therefore, we assume some $S'$ in Definition 4.2.4, where $S'$ is the set of reachable states and if $jtype(t) = future$, then the enabling of $t$ depends on this set $S'$.

State transitions

Let’s formalize the transitions possible in a given state by means of binding functions $binding_{enter}$ and $binding_{exit}$ corresponding to task commencement, which brings a task from the disabled state to the active state, and task termination, which brings a task from an active state back to the disabled state, respectively.

**Definition 4.2.5. (Task commencement)**

Let $N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F)$ be a GWF-net. The Boolean function $binding_{enter}(t, cons, prod, s)$ evaluates to true if and only if the transition $enter$ can occur for a task $t \in T$ in the state $s \in S$, while consuming the bag of messages $cons$ and producing the bag of messages $prod$, such that the following conditions are satisfied:

- The task $t$ is enabled in the given state $s$:
  
  $$enable(t, s) = true$$

- Messages to be consumed are present in the state:
  
  $$cons \subseteq s$$

- There exists a set $Q \in IS(t)$ such that:
  
  $\diamond$ Messages are consumed from inputs of the task:
  
  $$set(cons) = \overline{Q}$$

  $\diamond$ If the input selection mode is set to maximal, then a maximal set of enabled input ports of $IS(t)$ is selected, i.e. there is no a bigger set with respect to set inclusion:
  
  $$if ism(t) = max, then \forall Q' \in IS(t)(Q \subseteq Q') \Rightarrow \exists p \in Q' \setminus Q \ s(p) < mincap(p)$$
If the input selection mode is set to minimal, then a minimal set of enabled task inputs of $IS(t)$ is selected for message consumption, i.e. there is not smaller set with respect to set inclusion:

$$
\text{if } ism(t) = \text{min}, \text{ then } (\forall Q' \in IS(t) : Q' \not\subseteq Q)
$$

If the input selection mode is set to random, then any set of enabled task inputs of $IS(t)$ can be selected for the message consumption.

- One message is created for the active task state:
  $$prod = [t]$$

- The task is not active yet:
  $$t \notin s$$

- The number of messages consumed from the selected task inputs is determined by the message consumption mode of the considered task. In the minimal message consumption mode, the number of messages required for enabling of the input is consumed, while the rest of the messages remain in the input channels. In the maximal message consumption mode, all messages contained in the channels of the selected input ports are consumed. For all $c \in set(cons)$:
  $$(mcm(t) = \text{min}) \Rightarrow (cons(c) = \text{mincap}(c))$$
  and
  $$(mcm(t) = \text{max}) \Rightarrow (cons(c) = s(c))$$

Next we define the function $binding_{exit}$ which does not consider cancelations.

**Definition 4.2.6. (Task termination)**
The boolean function $binding_{exit}(t, cons, prod, s)$ evaluates to true if and only if the transition $exit$ can occur for a task $t$ in state $s$, while consuming the bag of messages $cons$ and producing the bag of messages $prod$, and the following conditions are satisfied:

- Task $t$ is active in state $s$:
  $$t \in s$$

- One message is consumed from the internal task state:
  $$cons = [t]$$

- There exists a set $Q \in OS(t)$ such that potentially one message is produced for each of the selected output ports and the maximal channel capacity is respected (cf. blocking mode):
  $$prod' = [\prod | p \in Q]$$
  if $blp(t) = \text{blocked}$, then $prod = prod'$ and $\forall c \in \tau$ $(s \cup prod)(c) \leq \text{maxcap}(c)$
  if $blp(t) = \text{open}$, then $\forall c \in \tau$ $prod(c) = \text{min}(prod'(c), (\text{maxcap}(c) - s(c)))$

Using $binding_{exit}$ function, we define $binding^{CS}_{exit}$ which takes cancelation into account.

**Definition 4.2.7. (Task termination with cancelation)**
The boolean function $binding^{CS}_{exit}(t, cons, prod, s)$ yields true if and only if there exists a $cons'$ such that $binding_{exit}(t, cons', prod, s)$ yields true and for any $x \in C \cup T$: 
\[ \text{cons}(x) = s(x) \text{ if } x \in CS(t) \]

and

\[ \text{cons}(x) = \text{cons}'(x) \text{ if } x \notin \text{CS}(t) \]

Messages are removed from all input channels of task \( t \) and from its cancelation set. This implies cancelation of tasks in the cancelation set which have not yet completed.

**Definition 4.2.8. (Task binding)**

Let \( N=(C, \text{LC}, \text{EC}, i, o, T, P, \text{IP}, \text{OP}, \text{ManP}, \text{OptP}, \text{ptoc}, \text{psend}, \text{mincap}, \text{maxcap}, \text{blp}, \text{IS}, \text{ism}, \text{OS}, \text{mcm}, \text{CS}, \text{guard}, \text{jtype}, F) \) be a GWF-net. The Boolean function 
\[ \text{binding}(t, \text{cons}, \text{prod}, s) \]

yields true if and only if one of the following conditions holds:

- The enter part of a task is enabled:
\[ \text{binding}_{\text{enter}}(t, \text{cons}, \text{prod}, s) \]

- The exit part of a task is enabled:
\[ \text{binding}_{\text{CS exit}}(t, \text{cons}, \text{prod}, s) \]

**Definition 4.2.9. (State transition)**

Let \( N=(C, \text{LC}, \text{EC}, i, o, T, P, \text{IP}, \text{OP}, \text{ManP}, \text{OptP}, \text{ptoc}, \text{psend}, \text{mincap}, \text{maxcap}, \text{blp}, \text{IS}, \text{ism}, \text{OS}, \text{mcm}, \text{CS}, \text{guard}, \text{jtype}, F) \) be a GWF-net, \( S \) its state space and \( s_1 \) and \( s_2 \) two workflow states in \( S \). \( s_1 \rightarrow s_2 \) if and only if there are \( t \in T, \text{cons}, \text{prod} \in S \) such that 
\[ \text{binding}(t, \text{cons}, \text{prod}, s_1) \text{ and } s_2 = (s_1 - \text{cons}) \uplus \text{prod} \]

\( \rightarrow \) defines a transition relation on the states of the given workflow. The reflexive transitive closure of \( \rightarrow \) is denoted \( \rightarrow^* \) and 
\[ R(s) = \{ s' \in S | s \rightarrow^* s' \} \]

is the set of states reachable from state \( s \).

The state space \( S \) and transition relation \( \rightarrow \) define a transition system \((S, \rightarrow)\) for a given GWF-net.

This completes the formalization of GWF-nets. Using this formalization we can also reason about the correctness of GWF-nets. For example, we can generalize the well-known soundness property as shown in Definition 4.2.10.

**Definition 4.2.10. (Soundness)**

Let \( N=(C, \text{LC}, \text{EC}, i, o, T, P, \text{IP}, \text{OP}, \text{ManP}, \text{OptP}, \text{ptoc}, \text{psend}, \text{mincap}, \text{maxcap}, \text{blp}, \text{IS}, \text{ism}, \text{OS}, \text{mcm}, \text{CS}, \text{guard}, \text{jtype}, F) \) be a GWF-net.

- \( N \) has the option to complete iff for any state \( s \in \bigcup_{c \in i} R([c]) : \exists c \in o[c] \in R(s) \).
- \( N \) has no dead tasks iff for any \( t \in T \) there is a state \( s \in \bigcup_{c \in i} R([c]) \) such that \( t \in \text{set}(s) \).
- \( N \) has proper completion iff for any state \( s \in \bigcup_{c_1 \in i} R([c_1]) \) and any \( c_2 \in o : (s \geq [c_2]) \Rightarrow (s = [c_2]) \).

\( N \) is sound iff \( N \) has the option to complete, has no dead tasks, and has proper completion.

Note that three GWF-nets used in the travel agency example 4.2.2 on page 200 have the option to complete and have no dead tasks. In contrast to the GWF-nets depicted in Figures 171 and 172, the GWF-net in Figure 170 has no proper completion because of the multiple messages produced by task Pay directed to the end channel \( e_1 \).
CPC-ML in action Having precisely described the core process modeling constructs using CPC-ML, we now want to illustrate its use with respect to the operationalization of the control-flow patterns defined in Section 4.1.3.

As an example, we illustrate that even the simplest control-flow patterns such as Sequence (WCF-1) can be operationalized using different approaches\textsuperscript{11}. Thereafter, we show how CPC-ML can be used to assess the functionality of distinct PAISs and we compare realizations of specific behavioral constructs.

For the Sequence pattern, we describe a selected CPC-ML specification, its alternative representations, and the mapping of the implementations in Staffware and Oracle BPEL PM to the CPC-ML notation. Note that to describe different variants of the Sequence pattern, we use relevant task attributes as variation points. By setting each of the task attributes to a specific value, a configuration of a pattern variant can be specified.

Pattern WCF1 - Sequence

Description A task in a process is enabled after the completion of a preceding task in the same process.

Selected CPC-ML specification Figure 175 shows the CPC-ML notation of the Sequence pattern. In terms of CPC-ML, task $t_2$ is executed after the execution of task $t_1$. Messages are transferred via a safe external channel $c_1$. The blocking mode of an output port $o_1$ of task $t_1$ is chosen to be open to allow task $t_1$ to complete if the channel $c_1$ is non-empty. The enabling of task $t_2$ is based on the messages currently available in the channel, i.e. $jtype$ is set to local. Messages are consumed in the minimal message consumption mode. The choice of the input selection mode for this net is irrelevant, since the number of the input ports is limited to one, thus no option for the input selection is available. However, to make the specification complete, $ism$ for $t_2$ is set to minimal. For the sake of convenience, the input sets of task $t_1$ and output sets of task $t_2$, which may vary without influencing the behavior of a pattern, are omitted in this net. This is one of many possible interpretations of the pattern.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{sequence.png}
\caption{WP1 - Sequence. Selected CPC-ML specification}
\end{figure}

Alternative CPC-ML specifications The variation points, based on which alternative configurations of the Sequence pattern can be obtained, are listed below:

- **Channel characteristics.** Channel capacity: the maximal channel capacity can be increased from safe to bounded or unbounded, depending on the number of messages

\textsuperscript{11}More complex patterns can be analyzed in a similar vein as has been done in [156].
the channel is able to store at once. If the minimal channel capacity $\text{mincap}(c1) = 1$
does not change, this notation is equivalent to the main notation. Note however, that
by increasing the minimal channel capacity, the enabling of the input port connected
to this channel would be delayed until the specified minimal channel capacity is
reached.

Channel positioning: an external channel can be used (as in Figure 175) with the
additional requirement that no tasks other than those comprising the sequence are
connected to this channel. However, for direct message transfer, the local channel
type can be selected.

- **Blocking of output ports.** Output port $o1$ of task $t1$ can be set to open or blocked.
  If the safe channel $c1$ is not empty, then an open output port $o1$ for task $t1$ is allowed
to produce a message, however this message will be lost. If the output port is in the
blocked mode, the completion of task $t1$ will be postponed until the capacity of the
channel $c1$ is freed. Note that if the maximal capacity of the channel $c1$ is unbounded,
the output port $o1$ of task $t1$ is always open and is never blocked since the maximal
capacity of the channel cannot be reached.

- **Message consumption mode.** In the minimal message consumption mode, the
  number of messages required to enable the channel (equivalent to the minimal channel
  capacity) is consumed, the rest of the non-consumed messages are stored in the
  channel for the subsequent task execution. To consume all messages available in the
  channel, the message consumption mode should be set to maximal.

Note that in the initial definition of the Sequence pattern on page 123 such attributes as
the message consumption mode, the blocking of output ports, and the channel capacity
are not specified, which makes the pattern definition subject to multiple interpretations.

**Staffware implementation** The Staffware model of the Sequence and its corresponding
CPC-ML interpretation are presented in Figures 176 (a) and (b) respectively. The behavior
of this pattern can be described by means of the CPC-ML attributes as follows. Messages in
Staffware are transferred via a safe channel ($c1$ is chosen to be safe, i.e. $\text{maxcap}(c1) = 1$).
The enabling of task $B$ is based on the messages available locally ($jtype(B) = \text{local}$).
The consumption of messages is performed in the minimal message consumption mode
($\text{mcm}(B) = \text{min}$). In Staffware, messages can be sent to a channel even if the capacity
of the channel has been reached. The second message will cancel the first one. Although
Staffware has no notion of ports, the described behavior corresponds to the open mode of
the output port ($\text{blp}(o1) = \text{open}$) in the CPC-ML terms.

**Oracle BPEL PM implementation** The (sequence) construct presented in Figure 177,
which also corresponds to the BPEL code listed below, allows the definition of the collection
of tasks to be performed in the lexical order.

```xml
<sequence name="Sequence_1">
  <empty name="A"/>
  <empty name="B"/>
</sequence>
```

The fact that BPEL is structured and acyclic, implies that it does not facilitate rea-
soning about attributes such as the input selection mode and the blocking mode of the
ports. Activity $B$ is enabled as soon as it is triggered by a message, therefore $jtype$ of
the corresponding CPC-ML task is set to local. It is sufficient to have a single trigger for
enabling a task, therefore the message consumption mode of task $B$ is set to minimal, i.e.
$\text{min}$. 
Variation points expressed in terms of CPC-ML for each of the control-flow patterns serve as classification criteria for distinguishing different variants of the same pattern. As an example of this we have illustrated that even such a basic pattern as Sequence, which is supported by all workflow systems evaluated in [193], has several variants. To distinguish between different pattern variants, it is important to identify the characteristics of channels used to model the pattern, specifying explicitly the maximal channel capacity that indicates the number of messages the channel is able to store at once; the minimal channel capacity and the message consumption mode specifying how many messages are required to enable tasks in the sequence; and the blocking mode of the output ports of tasks in a net defining whether the tasks producing messages may complete when the maximal channel capacity has been reached.

Although such a classification could potentially lead to an explosion of the number of patterns, it offers the potential to increase the precision of pattern definitions from an operational perspective significantly, thus avoiding possible misinterpretations. Moreover, the properties of the basic model entities that are captured in CPC-ML are orthogonal. By explicitly identifying the various dimensions and by distinguishing all combinations of values in different dimensions we can obtain a large set of variants of the control-flow patterns.
CPC-ML is a powerful tool for the analysis of workflow systems. So far, we have used CPC-ML as a means of comparing the expressiveness of the implementations of the control-flow patterns in Staffware and Oracle BPEL PM. Note that CPC-ML can easily be used for the analysis of other workflow systems. To illustrate this, we will show the mapping of the XOR-join and the AND-join constructs implemented in COSA, GLIF and YAWL to CPC-ML. Some of these systems were also mentioned at the beginning of Section 4.2.1 on page 192. The mapping of process modeling languages to CPC-ML allows us not only to reason about features of specific languages but also to compare them using CPC-ML as an evaluation tool. Figure 178 depicts the mapping of the XOR-join construct to CPC-ML (cf. models (a) and (b) respectively).

![Figure 178: The XOR-join: mapping of COSA to CPC-ML](image)

COSA activities A, B and C and states s1 and s2 correspond to CPC-ML tasks A, B and C and channels s1 and s2 respectively. The maximal capacity of channels in COSA is bounded to one, therefore the channels are safe. In COSA, tasks A and B can not be enabled if state s1 is not empty. This corresponds to the blocked mode set for all output ports in the GWF-net.

Figures 179(a) and (b) show the mapping of the AND-join construct implemented in COSA to CPC-ML respectively. COSA activities A, B and C and states s1, s2 and s3 correspond to CPC-ML tasks A, B and C and channels s1, s2 and s3 respectively. Activity C requires both inputs from states s1 and s2, which corresponds to the synchronization of incoming threads. Such AND-join behavior is denoted in CPC-ML by means of the input sets of task C. Similar to the description of the XOR-join implementation in COSA, this model corresponds to a GWF-net with safe channels, blocked output ports, and maximal input selection mode. The enabling of tasks is based on locally-available information.

In contrast to COSA, a language for modeling clinical guidelines GLIF has no notion of states that could be directly mapped to channels of CPC-ML. Figure 180 shows the mapping of the XOR-join construct in GLIF to CPC-ML. The action steps A and B are followed by the patient state step C, which merges the outputs of A and B. Step C is executed as soon as an input from either of these branches has arrived, therefore the message consumption mode for task C is set to min. The output ports of tasks A and B
Figure 179: The AND-join: mapping of COSA to CPC-ML

Figure 180: The XOR-join: mapping of GLIF to CPC-ML

are set to the open mode, which means that they may produce messages even if previous messages have not yet been processed by task C.

Figure 181 depicts the mapping of the AND-join construct in GLIF to CPC-ML. Since the threshold of the synchronization step C can be set to a subset of incoming branches, the input consumption mode of step C is set to minimal. Note however that the message consumption mode for this step is set to maximal in order to ensure that all inputs provided are consumed. The merge of inputs is based on the locally available information, therefore jtype(C) is set to local.

Figure 182, Figure 183 and Figure 184 depict the mapping of the XOR-join, AND-join
Section 4.2 Patterns operationalization

Figure 181: The AND-join: mapping of GLIF to CPC-ML

Figure 182: The XOR-join: mapping of YAWL to CPC-ML

and OR-join implemented in YAWL to CPC-ML respectively. In contrast to COSA, YAWL allows an unlimited number of messages to be stored in places, therefore channels in the GWF-net are chosen to be unbounded. Since the maximal capacity of the channels can never be reached, all output ports produce messages in the open mode. The input selection mode for all joins is set to maximal, meaning that the maximal set of enabled input ports from the perspective of set inclusion is selected by task $C$. The join logic is incorporated into the input sets of task $C$, the enabling of which is based on local and future states for XOR-join, AND-join, and OR-join respectively. The messages are consumed from channels in minimal mode, i.e. one message is consumed from every channel selected for message consumption.
We have shown that CPC-ML can also be used for specifying the functionality of systems such as COSA and YAWL, and thus is generic enough to serve as a point of reference when assessing and comparing modeling languages employed by PAISs. The examples presented highlight many subtle differences between systems and many possible interpretations of the control-flow patterns. This concludes the discussion of requirements for PAIS from the control-flow perspective. We now move on to a comprehensive evaluation of control-flow pattern support in a selection of PAISs.

4.3 Tool evaluations

In this section, we show how the control-flow patterns can be used for assessing and comparing process modeling languages employed by a wide range of PAISs. In particular, we
concentrate on two extreme approaches utilizing process models: a BPEL-based process engine (Oracle BPEL PM) and clinical Computer-Interpretable Guideline (CIG) modeling languages. The medical community claims that CIG modeling languages are very different from workflow management systems, as they require the support of processes of more flexible and unpredictable nature, therefore it is interesting to investigate the actual differences between workflow management systems and CIG modeling languages from the control-flow perspective. In Section 4.3.1, we give a brief overview of languages selected for the evaluation. The evaluation of the BPEL-based process engine is presented in Section 4.3.2, whereas the evaluation of the CIG modeling languages is presented in Section 4.3.3.

We have selected Oracle BPEL Process Manager 10.1.3.1 as a candidate offering which supports processes based on the BPEL standard. Although this tool provides functionality for inter-process communication, in this evaluation we concentrate only on the control-flow perspective in the context of a single process instance. Oracle BPEL PM can be considered to be a system employing an imperative approach, i.e. it explicitly specifies tasks and the order in which they should be performed.

Clinical CIG modeling languages utilize an alternative approach in regard to addressing particular modeling challenges. Instead of dictating what should be done and how, these languages guide users through the process and provide recommendations for decision making. The main goal of CIG modeling languages is to assist users in making decisions by providing computer-interpretable representations of the clinical knowledge contained in clinical guidelines. To analyze CIG modeling languages from the control-flow perspective, we chose four standards, i.e. Asbru, PROforma, EON and GLIF, supported by the AsbruView, Tallis, and Protege-2000 tools respectively.

4.3.1 Background

Before we proceed with a more detailed analysis of the control-flow patterns support, we first give some background information about BPEL and CIG modeling languages.

Business Process Execution Language for Web-Services (WS-BPEL)

Oracle BPEL Process Manager 10.1.3.1 is a tool providing facilities for modeling, deploying and managing BPEL-processes [166] (v.1.1). It provides a graphical interface for visualizing BPEL process modeling activities, which are described below.

The Business Process Execution Language for Web-Services (WS-BPEL) is a business process modeling language that is executable. We chose Oracle BPEL PM as an example of a system that provides operational support for WS-BPEL. WS-BPEL is an orchestration language, which allows multiple services to be linked together. It takes the viewpoint of a single participant in the specification of such an interaction. Processes deployed as web-services can be linked together and communicate by means of messages. The messaging mechanism that WS-BPEL relies on is defined by the Web Services Description Language (v1.1) [60].

Apart from its ability to specify abstract processes, i.e. providing a high-level view on the interaction between communicating processes, WS-BPEL also provides constructs for specifying executable processes. While WS-BPEL supplies only a textual description of processes expressed in the form of an XML-schema, Oracle BPEL PM provides also a graphical interface to visualize each of the process modeling activities.
A generic BPEL-document contains definitions of partner links, variables, fault handlers, and the description of a structure of a business process. Partner links define parties that are involved in an interaction; these are declared in the <partnerLinks> section. Each partner link is of type PartnerLinkTypes which represents dependencies between services, each playing one of the two specified roles (i.e. a service provider or a service consumer). The variables section defines data variables used in a process and messages exchanged by the given process during its execution. Each message has a type whose definition is based on an XML-schema and is defined in a corresponding WSDL-document. The fault handlers section defines activities that must be performed in response to faults that may happen during interaction. Finally, the <process> definition contains a description of the process behavior. WS-BPEL offers a set of process activities that represent the actions associated with tasks which comprise a business process. We will briefly describe each of these process activities below.

The <receive>, <reply>, <invoke> and <pick> are activities used in WS-BPEL to send and receive messages between processes. The <invoke> activity allows a business process to invoke an operation provided by another process. Depending on whether such call is one-way or request-response, the requestor process may specify how the expected response needs to be processed. The <receive> activity is executed when a message whose message type matches the corresponding port type is received (ports are considered to be gates for messages for a particular type). An alternative means of receiving a message is provided by the <pick> construct. It includes multiple branches, each of which can be associated with a particular action, e.g., on message receive <onMessage> or on the basis of a particular event <onEvent>. A race condition exists between activities included in these branches, i.e. only the first one can execute, while the rest are withdrawn. The <reply> activity allows a business process to send a message in response to a message that was previously received by any of Inbound Message Activities (IMA) (i.e. <receive>, <onMessage>, or <onEvent>).

The <assign> activity is used to update variables with new data. The <validate> activity is used to validate the values of variable against the corresponding XML definition. Faults are generated inside a business process by means of the <throw> activity. Waiting for a given period of time can be accomplished via the <wait> activity. It can also be used as a timeout event in the pick construct. The <empty> activity represents an empty task, that is used primarily for synchronization purposes. The <exit> activity is used to end the execution of the business process immediately.

The <scope> activity is used in WS-BPEL to define a nested activity. If execution within the scope fails, the <compensateScope> activity can be performed. The <compensate> activity is used to compensate all inner scopes that have completed successfully. The <rethrow> activity is used to rethrow the fault which was originally caught by the enclosing fault handler.

The following set of constructs defines the execution order of tasks in a process. The <sequence> activity defines a set of tasks to be performed sequentially in a defined order (i.e. in the order that activities are listed). The <flow> activity specifies multiple branches that run concurrently. By means of links associated with tasks in these branches, a partial order can be enforced. The <forEach> activity iterates an enclosed activity a specified number of times, either sequentially or in parallel. This construct corresponds to the <flowN> construct in Oracle BPEL PM. The <if> activity is used to describe conditional choice, where one activity is selected from a set of choices. The <while> activity defines conditions for repeating the execution of a child activity. Finally, the <repeatUntil>
activity defines the repetitious execution of a child activity until a specified condition is satisfied.

**Computer-Interpretable Guideline modeling languages**

Clinical practice guidelines and protocols are being applied in diverse areas including policy development, utilization management, education, clinical decision support, conduct of clinical trials, and workflow facilitation. The main intent of clinical guidelines is to improve the quality of patient care and reduce costs. Difficulties associated with accessing information contained in conventional guidelines and applying it in practice have motivated the development of decision-support systems whose main purpose is to create computer-interpretable representations of the clinical knowledge contained in clinical guidelines.

We introduce the main concepts of four functioning Computer-Interpretable Guideline (CIG) modeling languages selected for evaluation, i.e. Asbru, EON, GLIF, and PROforma, by modeling the following patient diagnosis scenario in each of the corresponding tools. A patient is registered at a hospital, after which he is consulted by a doctor. The doctor requests the patient to give a blood test and a urine test. When the results of both tests become available, the doctor makes a diagnosis and confirms the treatment strategy.

Figure 185 presents the scenario modeled in AsbruView [34]. AsbruView is a markup tool among many others (e.g., Delt/A [75, 215], URUZ [142], and CareVis [22, 52]) that were developed to support the authoring of guidelines in Asbru [34]. A process model in Asbru [201] is represented by means of a time-oriented skeletal plan.

![Figure 185: The patient-diagnosis scenario modeled in AsbruView](image)

The root plan (marked as Plan A in Figure 185) is composed of a set of other plans, which are depicted with different colors for the purpose of differentiation. The plans are represented as 3-dimensional objects, where the width represents the time axis, the depth represents plans at the same level of the decomposition (i.e. which are performed in parallel), and the height represents the decomposition of plans into sub-plans. Parent plans are considered to be completed when all mandatory sub-plans have been completed. Enabling, completion, resumption and abortion conditions can be specified for each plan where required. As the time axis shows, plans **Register patient**, **Consult doctor**, **Test phase**
and Define treatment are executed sequentially. The Test phase plan is a parallel plan consisting of two activities Ask for urine test and Ask for blood test. In this model, we used only two types of plans: sequential (root plan) and parallel (Test phase plan). AsbruView also support the visualization of the Any-order Plan, Unordered Plan, Cyclical Plan, and If-then-else Plan, and two types of actions: Ask and Variable Assignment.

Figure 186: The patient-diagnosis scenario modeled in EON/Protege

An EON model of the patient-diagnosis scenario created in the Protege-2000 environment is illustrated in Figure 186. Protege-2000 is an ontology-editor and knowledge-base framework [162]. The main modeling entities in EON [211] are scenarios, action steps, branching, decisions, and synchronization [210,212]. A scenario is used to characterize the state of a patient. There are two types of decision steps in EON, i.e. a Case step and a Choice step, which allow exactly one path or more than one path to be selected respectively. An Action step is used to specify a set of action specifications or a sub-guideline that are to be carried out. Branch and Synchronization steps are used to specify parallel execution.

GLIF 3.5 [46] is a specification method for the structured representation of guidelines. To create a model in GLIF, an ontology schema and a graph widget have to be loaded into the Protege-2000 environment. Figure 187 illustrates the GLIF model of the patient-diagnosis scenario.

In GLIF 3.5, five main modeling entities are used for process modeling, i.e. an Action Step, a Branch Step, a Decision Step, a Patient Step, and a Synchronization Step. An Action Step is a block used to specify a set of tasks to be performed, without constraints set on the execution order. It allows the inclusion of sub-guidelines into the model. Decision steps, combining a Case Step and a Choice Step from GLIF 3.4, are used for conditional and unconditional routing of the flow to one of multiple steps. Branch and Synchronization steps are used for modeling concurrent steps and synchronization of the parallel branches respectively. A Patient Step is a guideline step used for describing a patient state and for specifying an entry point(s) to a guideline. A remarkable feature of GLIF 3.5 is its ability to specify several alternative entry-points to a process model, the execution of any of which results in the initiation of a new process instance. In Chapter 6 of this thesis, we focus
on different approaches to process flexibility and describe them in the form of patterns. The ability of GLIF3.5 to support multiple entry-points corresponds to the *Alternative Entry-Points* process flexibility pattern described on page 335. Interesting to note, that this pattern is not supported by typical workflow systems and is the typical characteristic of CIG modeling languages.
PROforma [93] is a formal knowledge representation language for authoring, publishing and executing clinical guidelines. It deliberately supports a minimal set of modeling constructs: actions, compound plans, decisions, and enquiries that can be used as tasks in a task network. In addition, a keystone may be used to denote a generic task in a task network. All tasks share attributes describing goals, control flow, preconditions, and postconditions. A model of the patient-diagnosis scenario created in Tallis is shown in Figure 188. Note that in PROforma control-flow behavior is captured by modeling constructs in combination with scheduling constraints. Scheduling constraints are visualized as arrows connecting two tasks, meaning that the task at the tail of the arrow may become enabled only after the task at the head of the arrow has completed.

Unlike other CIG modeling languages, PROforma combines two approaches to process specification: an imperative approach and a declarative approach. As a consequence of this, it is capable of expressing the majority of process flexibility patterns (cf. page 323) that can be included in the process definition at design-time.

4.3.2 Evaluation of Oracle BPEL PM

In this section, we consider the evaluation results obtained from a detailed analysis of the control-flow patterns in Oracle BPEL PM 10.1.3.1. In total, Oracle BPEL PM supports 20 of the 43 patterns directly and 3 patterns indirectly. The evaluation results are summarized in Table 4.2.

All branching patterns are well supported by Oracle BPEL PM, the only exception being that realization of the Thread Split pattern (WCF-6) is only indirectly achievable through the use of the <invoke> construct in conjunction with programmatic extensions. From the Synchronization patterns, Oracle BPEL PM supports a basic variant of the Synchronization pattern and all structured-type patterns. The native support for the Structured Discriminator pattern (WCF-9) and the Structured Synchronizing Merge patterns (WCF-17) is due to the block-structured nature of the process modeling language.

Note however that more flexible variants of the Discriminator pattern, i.e. the Blocking Discriminator and Canceling Discriminator, are not supported. Although a more generic variant of the Structured Synchronizing Merge, i.e. the General Synchronizing Merge, that requires “future”-based semantics is not supported, Oracle BPEL PM allows the Local Synchronizing Merge to be realized using links within the <flow> construct.

The Repetition pattern Structured Loop (WCF-21) is supported by Oracle BPEL PM in two ways, i.e. via the <repeatUntil> and the <while> constructs. No support for the Arbitrary Cycles pattern (WCF-23) is provided due to the block-structured nature of the modeling language. Nor is there the support for self-invocation as required by the Recursion pattern (WCF-22).

Although Oracle BPEL PM has no notion of a multiple instance task, three of the Multiple Instance patterns can be realized. The MI without Synchronization pattern (WCF-24) can be implemented using the <invoke> construct within the <while> loop. The MI with a priori Design-Time Knowledge pattern (WCF-25) and the MI with a priori Run-Time Knowledge pattern (WCF-26) are supported through the use of the <flowN> construct, which allows multiple instances of a task to be created based on the dynamically set variable N indicating the desired number of instances.

Of the four concurrency control patterns, Oracle BPEL PM directly supports only one of them: the Critical Section (WCF-33). Despite the fact that BPEL has a notion of serializable scopes, providing concurrency control in governing access to a common set of
shared variables, the realization of serializable scopes in Oracle BPEL PM does not behave as expected. In Oracle BPEL, the serializable scopes they behave as if they executed in sequence rather than concurrently. This means that the Interleaved Routing and Interleaved Parallel Routing patterns are not supported.

Since events in Oracle BPEL PM are of a durable nature, there is a direct support for the Persistent Trigger pattern (WCF-35), but no corresponding support for the Transient Trigger pattern (WCF-36).

Cancelation and Completion patterns are very well supported by Oracle BPEL PM as follows. The Cancel Task pattern (WCF-37) is achieved by associating a fault or a compensation handler with a task. The Cancel Case pattern (WCF-39) is directly supported via the \texttt{<terminate>} construct. The Cancel MI Task pattern (WCF-40) is supported in a \texttt{<flowN>} construct by associating it with a fault or compensation handler. The Cancel Region pattern (WCF-38) is not directly supported, since there is no means of canceling arbitrary groups of tasks, although tasks within the same scope can be canceled. Oracle BPEL PM does not support the Complete MI Task pattern (WCF-41) since there is no notion of a multiple instance task.

Only one means of denoting process instance termination, - Implicit Termination, - is supported by Oracle BPEL PM. The process instance terminates when there are no more tasks remaining.

4.3.3 Evaluation of CIG modeling languages

In this section, we provide a detailed examination of the CIG modeling languages. Table 4.2 summarizes the comparison. As the results of the analysis show, PROforma offers direct support for the largest number of patterns (22 out of 43) among the examined offerings. Asbru and GLIF offer support for 20 and 17 patterns respectively. Even less patterns are supported by EON (it supports only 11 patterns).

More detailed analysis of the pattern support reveals that all offerings examined directly support the majority of the Branching and Synchronization patterns, which are relatively common in business processes used in practice. Note that the Structured Synchronizing Merge pattern (WCF-17) is not supported by any of the examined offerings. While PROforma supports this pattern directly, Asbru adds a time restriction to the synchronization process to approximate the desired behavior. The semantics of the synchronization blocks in EON and GLIF are not precise enough, i.e. they do not specify what happens to the active tasks after the Synchronization task has been executed. This is also the reason why some of the variants of the Synchronizing Merge are not supported by EON and GLIF.

None of the examined modeling languages support the concept of a multiple instance activity and, therefore, patterns from the Multiple Instances pattern group are not supported. In the business process domain, multiple threads of execution that relate to the same activity are often supported (e.g., an insurance claim with a variable number of witness statements or an order containing multiple order lines). Similar situations may arise, for example, when a clinical trial is executed for groups of patients. To identify whether there is a need for CIG modeling constructs supporting multiple instances, more research needs to be done addressing the nature of the clinical guidelines requirements (note that this falls outside of the scope of work presented in this thesis).

Although EON and GLIF have the notion of patient state, they lack the notion of process state, thus providing no support for the Concurrency Control patterns. The only language that employed these concepts is PROforma. PROforma is the only system/language
Table 4.2: Support for the Control-flow Patterns in (1) Asbru, (2) EON, (3) GLIF, (4) PROforma, and (5) Oracle BPEL PM

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCF-1</td>
<td>Sequence</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-2</td>
<td>Parallel Split</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-3</td>
<td>Exclusive Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-4</td>
<td>Deferred Choice</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-5</td>
<td>Multi-Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-6</td>
<td>Thread Split</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+/-</td>
</tr>
<tr>
<td>WCF-7</td>
<td>Synchronization</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-8</td>
<td>Generalized AND-join</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-9</td>
<td>Structured Discriminator</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-10</td>
<td>Blocking Discriminator</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-11</td>
<td>Canceling Discriminator</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WCF-12</td>
<td>Structured Partial Join</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WCF-13</td>
<td>Blocking Partial Join</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-14</td>
<td>Canceling Partial Join</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WCF-15</td>
<td>Simple Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-16</td>
<td>Multi-Merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-17</td>
<td>Structured Synt. Merge</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-18</td>
<td>Local Synt. Merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-19</td>
<td>General Synt. Merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-20</td>
<td>Thread Merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+/-</td>
</tr>
<tr>
<td>WCF-21</td>
<td>Structured Loop</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-22</td>
<td>Arbitrary Cycles</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-23</td>
<td>Recursion</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-24</td>
<td>MI without synch.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>WCF-25</td>
<td>MI with a priori DTK</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+</td>
</tr>
<tr>
<td>WCF-26</td>
<td>MI with a priori RTK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>WCF-27</td>
<td>MI without a priori RTK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-28</td>
<td>Static partial join for MI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-29</td>
<td>Canceling partial join for MI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-30</td>
<td>Dynamic partial join for MI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-31</td>
<td>Interleaved Routing</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-32</td>
<td>Interleaved Parallel Routing</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-33</td>
<td>Critical Section</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-34</td>
<td>Milestone</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WCF-35</td>
<td>Transient Trigger</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WCF-36</td>
<td>Persistent Trigger</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-37</td>
<td>Cancel Task</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-38</td>
<td>Cancel Region</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
</tr>
<tr>
<td>WCF-39</td>
<td>Cancel Case</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-40</td>
<td>Cancel MI Task</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-41</td>
<td>Complete MI Task</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>WCF-42</td>
<td>Implicit Termination</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>WCF-43</td>
<td>Explicit Termination</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
supporting the Milestone pattern (WCF-34). In contrast, Asbru supports all variants of the Interleaved Routing pattern (i.e. WCF-31, WCF-32 and WCF-33) directly via its Any-order type plans.

All of the languages analyzed except EON support the Cancelation and Completion patterns relatively well. Note that none of the systems has the functionality for canceling an arbitrary group of tasks available. PROforma plays a leading role in supporting the Trigger patterns. In addition to the Transient Trigger and Persistent Trigger patterns, it also offers the notion of force triggers, i.e. triggers that force a certain task to execute such that any non-satisfied pre-conditions for its execution are ignored. In terms of process instance termination, all the languages examined provide support for implicit termination, i.e. the process instance terminates when no tasks are left to execute.

Based on the results of evaluating CIG modeling languages, we can conclude that from the control-flow perspective these languages are very similar to the process languages of workflow management systems. This is remarkable since one would have expected dedicated constructs allowing for more flexibility given the more dynamic nature of care processes.

4.4 Related work

There have been multiple attempts to define business process modeling languages, aiming to provide a single point of reference for business process specification. In [56], Casati et al propose a WorkFlow Description Language (WFDL) where a workflow model is composed of workflow tasks and routing tasks are used to denote partial and iterative joins. In [182], Reichert and Dadam present the ADEPT Workflow Model that is based on the concept of symmetrical control structures (e.g., splits, joins and loops specified as symmetrical blocks with a single start and end points). In [143], Kradolfer and Geppert describe a TRAMs workflow model, where control flow is specified through start and end conditions associated with workflow tasks.

Various formal techniques such as process algebra [42, 124], pi calculus [177], state charts [224] and Petri nets [102] have been used as a basis for developing process modeling languages. Of these, Petri nets have established themselves as the most pervasive conceptual basis for describing control-flow in a process language. In [2], van der Aalst advocates three main advantages that Petri nets offer: (1) availability of a formal semantics despite the graphical nature, (2) state-based reasoning (instead of event-based), and (3) a large number of analysis techniques. In spite of these advantages, very few PAISs directly use Petri nets as a basis for describing control-flow. Nevertheless, many of the languages are very similar to Petri Nets in many respects.

The varying interpretations of basic process modeling concepts employed in distinct PAISs have motivated the definition of a single standard providing a common vocabulary, a universal reference model, and operational semantics. One of the most widely cited attempts in this direction was the Workflow Reference Model [63] by the Workflow Management Coalition (WFMC) which aimed to identify the range of constructs that should be embodied in a workflow engine. Due to relatively limited set of control-flow constructs it described, and their incomplete and imprecise definition, it failed to provide a universal source of reference. A more considered pattern-based approach aiming to cover the insufficiencies identified has been introduced by the Workflow Patterns Initiative.

The first application of a pattern-based approach to identification of generic workflow constructs in [9] resulted in a set of patterns relevant to the control-flow perspective. A
subsequent step towards gathering control-flow patterns has resulted in a collection of twenty control-flow patterns that aimed to provide a formal basis for understanding the requirements of PAIS from the control-flow perspective. These patterns were applied for evaluation of capabilities a large set of commercial and research workflow systems and web services standards.

In [135], the expressiveness of the systems was evaluated using control-flow patterns. In [81], the control-flow patterns were utilized to examine the capabilities of UML 1.4 Activity Diagrams, with the aim of identifying their strengths, weaknesses and areas for possible improvement. Similar investigations into the expressiveness of languages have been performed for BPEL4WS [225], BML [227], UML 2.0 Activity Diagrams [228], and BPMN [226].

In recognition that the original workflow patterns were limited to the control-flow perspective, and a comprehensive description of a workflow process also requires an understanding of the data and resource perspectives [128], in the context of the Workflow Patterns Initiative additional research was conducted aiming at the identification of requirements in the data and resource perspectives. As a result, 40 data patterns [191] and 43 resource patterns [192] have been identified and documented. In [190], subsequent research has also proposed a patterns-based approach to workflow exception handling.

Some of the control-flow patterns presented in this chapter are related to control-flow patterns of Jablonski et al [128]. However, [128] aims at the development of a WFMS that can be extended with new patterns rather than characterizing existing control-flow structures and using them for evaluation. Note that the term of workflow patterns has been already used by other authors, however their focus was more on workflow architecture [152] or on data-flow related issues arising in the context of collaborative work on document processing [147].

The workflow control-flow patterns formed the starting point for the development of Yet Another Workflow Language (YAWL) [15]. This language extends Petri nets, illustrating support for almost all control-flow patterns. It also aimed to support all data, resource and exception handling patterns identified in [190–192] respectively.

In [107], Greco et al. investigate the problem of mining unconnected patterns in workflows as the authors believe that detecting sequences of tasks which frequently occur together are crucial for discovering meaningful execution patterns. For this purpose, they propose a mechanism for graphical analysis of the frequencies of process instances. Business process discovery has become an interesting topic in the research and has been addressed by many authors [98,100,233]. Following on from the topic of discovering workflow patterns in the form of workflow regions in a model, Truong et al. analyze performance metrics for use in evaluating the performance of Grid workflows.

In [48], Braghetto et al. propose an alternative business process definition language, the suitability and expressiveness of which is evaluated using the set of the workflow control-flow patterns. In [51], Cardoso proposes a metric for computing the complexity of BPEL-processes, and uses the control-flow patterns, data patterns, and resource patterns as a basis for the evaluation. Philippi and Hill [173] propose the APRIL process modeling language, where they use control-flow patterns to illustrate the animation aspects of the language; furthermore, the authors identify future plans to implement all control-flow patterns. In [49], Brambilla et al. address the topic of process modeling in web application. The authors analyze various tools and use the workflow patterns as a means of benchmarking.

In their paper [133], Joncheere et al. aim at the definition of a new generation of workflow systems, tailored for grid service composition. In order to identify requirements
for this workflow system, the authors extensively use workflow control-flow patterns. The topic of grid environments and grid optimizations has been also addressed in [58, 69, 112].

In [91], Fortino et al. propose an agent-based approach for modeling and enactment of distributed workflows. In their model, the authors use a workflow schema based on the workflow control-flow patterns. Zhao et al. also refer to workflow patterns in their work [237] related to agent-based flow control. In [197], Savarimuthu et al. apply the workflow patterns also for monitoring and controlling of a multi-agent based workflow system.

In a series of related works [104–106], Gomes et al. present an approach for extending tools for composition of application by with design patterns and operators, aiming at providing a better way to manipulate and manage the execution of available components.

### 4.5 Summary

In this chapter, we have revisited the control-flow patterns. In order to specify the operational semantics of the control-flow patterns precisely, we developed a set of CPN diagrams that define the conceptual operation of each pattern. Where applicable, we identified specific CPN patterns that were used during the CPN-modeling of control-flow patterns. An interesting observation is that many CPN diagrams designed to describe the semantics of the control-flow patterns resemble implementations of the CPN patterns, however they are not the same thing. The CPN patterns describe generic structures that might be encountered in CPN models of various kinds of systems, while the CPN models of the control-flow patterns represent the semantics of the control-flow constructs in business process modeling languages. Despite the similarity in the CPN models structure, the context in which they operate and their actual realization differ significantly.

To provide an objective basis for pattern evaluations, we have established a set of evaluation criteria for each of the control-flow patterns. Furthermore, we have classified patterns into groups based on the similarities in problems addressed by each of the patterns. The control-flow patterns have a wide range of potential application areas, each of which we will expand on subsequently. These include:

- Reuse of accumulated knowledge;
- Benchmarking of offerings;
- Enhancement of existing tools;
- Development of new tools and standards;
- Education and training.

By systematically documenting common constructs used in process modeling, the control-flow patterns facilitate the **reuse of accumulated knowledge**. Proven in practice solutions communicated in form of patterns can be reused by practitioners without having to reinvent them over and over again. The control-flow patterns also provide a basis for **benchmarking of offerings**: the degree of pattern support can be checked across distinct PAISs and standards. The results of the evaluation can be used as a trigger for **enhancement of existing tools** by redesign or extension with missing patterns. The patterns have directly influenced the development of many academic and commercial tools as well as the **development of standards** including BPMN and BPEL. In particular, FLOWer 3.0 of Pallas Athena has extended the functionality with the synchronizing merge construct, Staffware Process Suite has been extended based on the patterns for multiple instances,
Chapter 4 Workflow Control-Flow Patterns

and BizAgi of Vision Software has been completely redesigned to support more patterns. Furthermore, open source workflow systems Ivolutia Orchestration, OpenWFE, Zebra, and Alphaflow were inspired by the workflow patterns. The Workflow Patterns Initiative has also influenced the development of a Yet Another Workflow Language (YAWL) [1], which aims to provide both workflow language that is based on the workflow patterns and also an open-source reference implementation that demonstrates the manner in which these constructs can interoperate. In addition, the proved in the practice solutions can be used for education and training purposes.

Apart from describing the capabilities of PAISs to support control-flow patterns, we have illustrated that realization of the same control-flow constructs can be based on distinct assumptions that impose syntactic restrictions on the process modeling languages employed by different PAISs. In this context, we showed that operationalization aspect of the control-flow patterns can be made more precise by setting the structural and behavioral properties of the basic process modelling constructs in more detail, as described in the CPC-ML.

Expressing the control-flow patterns in terms of CPC-ML forces us to think of the behavior inherent to a pattern from the perspective of all entity attributes, thus allowing for more precise definitions of the patterns from the operationalization point of view. Depending on the value of the attributes set for the modeling entities, a single pattern may have different notations which correspond to the same behavioral pattern or be treated as different behavioral pattern variants depending on the values chosen for the variations points. The practical value of CPC-ML is multilateral: it can be used as

- a means of precise specification of the control-flow patterns, and
- a deterministic basis for capturing the operationalization semantics of process modeling constructs in distinct PAISs.

Furthermore, CPC-ML can be used by workflow system developers to prevent them from imposing restrictive constraints on workflow specifications, and identifying pattern variants a system needs to support.

In this chapter, we have concentrated on the internal aspects of a process, focusing on issues related to control-flow within a process instance. However, this knowledge is insufficient to describe interactions between processes. Interactive processes provide and/or use services to/from other processes. In order to understand requirements for such processes, interactions with the external environment have to be examined. The next chapter is devoted to the analysis of requirements from the service interaction perspective.
Chapter 5

Service Interaction Patterns

In Chapter 4, we concentrated on the internal aspects of a business process and presented the fundamental constructs for describing the structure of a process model in the form of control-flow patterns. In this chapter, we shift our focus from the internals of the process to the interaction of the process with the external environment. The insights that we provide in this chapter are crucial for understanding the requirements associated with service interaction. These are especially important for organizations which need to integrate third party applications and external services into their business processes, and for PAISs aiming to provide support for interactive business processes. A significant issue in the field of service orientation is that the requirements associated with service interactions are not very well understood. Some aspects of these interactions are ambiguous in their interpretation, and some relevant attributes are not considered at all. Furthermore, these requirements are limited to simple interactions, i.e. they do not cover interaction scenarios of a more complex nature.

In this chapter, we consolidate the results of an in-depth analysis of the requirements in service interaction and present them in the form of service interaction patterns. We start by introducing concepts relevant to service interaction in Section 5.1. Then in Section 5.2, we adapt an approach to compactly present service interaction patterns identified in the form of the configurable framework. In Section 5.3, we perform the tool evaluation using the patterns identified. We discuss the related work in Section 5.4 and draw the conclusions in Section 5.5.

5.1 Introduction

In light of current trends in the development of PAISs (as described in Section 1.2), the SOA paradigm has gained widespread acceptance as an approach to supporting the integration of software applications within and across organizational boundaries. In this context, distributed processes can be deployed as stand-alone services and interconnected using Web-based standards (cf. Appendix B). The functionality offered by a service can be linked to execution of a certain task in a process. Effectively, interaction between the process requesting a service and the process offering the service is performed via tasks dedicated to send and/or receive units of information in form of messages.

The interaction of one process with another process, can be characterized using one
of four basic interaction types presented in Figure 189. In unidirectional interactions, depicted in Figure 189(a) and (b), a dedicated task in one process either sends a message to another process, or receives a message from it. From the orchestration perspective, i.e. the point of view of a single party, process A sends a message to process B via Task A1, and receives a message from process B via Task A2. From the choreography perspective, i.e. the global view on the interaction between processes, in each of the interactions presented in Figure 189(a) and (b) both parties execute a send task and a receive task.

![Diagram of basic types of inter-process interactions](image)

**Figure 189:** Basic types of inter-process interactions

Two types of bidirectional interactions, presented in Figure 189(c) and (d), are possible: an asynchronous interaction and a synchronous interaction. An asynchronous interaction is the interaction between two parties, where one party sends a request message to another party, and expects a reply message to be received back in response to the first message. Assuming that the processing of the request may take a significant amount of time, in an asynchronous interaction the requestor party is not blocked and can continue processing. An asynchronous interaction can also be represented as a series of related interactions, where process A sends a request message to process B, which responds with a reply message at a later time. In contrast, in a synchronous interaction, a task in the requestor party is blocked until a reply message is received from the responder party.

Since services offered by different organizations may potentially be developed using
distinct technologies, they must adopt a common set of standards in order to allow such services to be easily integrated. Although contemporary technologies for web-services development and integration have reached a reasonable level of maturity in realizing simple bilateral interactions, many issues remain unresolved when it comes to managing more complex long-running interactions involving multiple (and possibly loosely-coupled) services [64].

The requirements for service interactions between multiple processes can be described using the concepts of orchestration and choreography. The difference between them is in the point of view from which an interaction is considered. Whilst orchestration defining the rules for a business process from the point of view of a single participant, choreography provides a global view on the interaction between the processes. The latest and the most significant standard, addressing issues of orchestration, is the Business Process Execution Language for Web-Services (WS-BPEL or BPEL) [164]. It provides a set of process activities for modeling of structured workflows combined with capabilities of incoming and outgoing message interactions. The W3C’s Web Services Choreography Definition Language (WS-CDL) [217] adopts a global view in order to define common ordering conditions and constraints under which messages are exchanged between participants in an interaction. While BPEL and WS-CDL aim to support a coordinated exchange between participants, the scope of the requirements they address is limited to bilateral interactions [40, 64].

To specify requirements in service interaction more extensively than had previously been done in the context of BPEL4WS and WS-CDL, thirteen service interaction patterns [37] covering bilateral, multilateral, competing, atomic and causally related interactions were identified. We have systematically reviewed the thirteen service interaction patterns presented in [37] and concluded that the scope of these patterns is limited to simple interaction scenarios, and that these patterns potentially suffer from ambiguous interpretation due to their imprecise definition. We have taken up the challenge of reconciling the requirements for service interaction and describing them using the pattern approach. Understanding these requirements in detail is crucial for improving existing standards in the field, making correct design choices when developing SOA-based PAISs, evaluating and selecting the right technology for realization and integration of business processes.

The scope of the work presented in this chapter goes beyond simple interactions: we concentrate on various aspects of long-running interactions between multiple tightly and loosely-coupled processes. Due to the large number of patterns identified, instead of listing each pattern separately, we present them in the form of a configurable framework. The framework proposed consists of five pattern families. Each pattern family combines a large set of pattern variants related to the same set of concepts. Pattern variants are generated from a pattern configuration by assigning different values to each of the configuration parameters defined for a particular pattern family. Figure 190 illustrates the main building blocks the framework consists of, and indicates how many meaningful pattern variants can be generated for each of the pattern families identified. Note that the original thirteen service interaction patterns [37] correspond to pattern variants belonging to different pattern families presented in this chapter.

In order to exclude the possibility of ambiguous interpretation, for each of the pattern configurations we provide a precise formal semantics in the form of CPNs. For every pattern family, we have developed a set of CPN models and tested them using CPN Tools [66]. Furthermore, for each pattern family we propose an intuitive notation that allows different values of configuration parameters to be graphically depicted. Thus, the graphical notation serves as a means for both visualizing and distinguishing pattern variants.
The five pattern families described in this chapter address the following problems:

- **The Multi-party Multi-message Request-Reply Conversation** pattern family considers conversations in which a single party interacts with multiple parties via multiple messages. It addresses the problems of non-guaranteed response and different aspects of message handling, e.g., sorting, consumption, and utilization of messages. This pattern family combines 1072 pattern variants.

- **The Renewable Subscription** pattern family identifies requirements related to long-running conversations (often termed subscriptions) where either of the two parties involved in the conversation can take the initiative when initializing and renewing a subscription. In total, 20 pattern variants can be derived from the pattern configuration for this family.

- **The pattern families Message Correlation, Message Mediation and Bipartite Conversation Correlation** address the problem of correlation, where the first pattern family addresses the problem at a low-level of abstraction, while the latter two do so at a higher-level.

  - In the context of interactions between two tightly-coupled parties, the Message Correlation pattern family addresses the issue of correlating incoming messages with previously exchanged messages related to the same conversation. This pattern family combines 100 pattern variants.

  - The Message Mediation pattern family concentrates on message exchange between two loosely coupled parties that interact via an intermediary. Two types of messages mediation, named Mediated Introduction and Mediated Interaction, are distinguished where the role of the intermediary is to introduce one party to another or to forward messages back-and-forth between these parties. This pattern family combines 72 pattern variants for Mediated Introduction and 320 pattern variants for Mediated Interaction.

  - The Bipartite Conversation Correlation pattern family concentrates on message correlation in the context of a long-running conversation between two parties, whose knowledge used for the message correlation may change during the course
of the conversation. This pattern family combines 18 meaningful pattern variants.

In Section 5.2, we present the conceptual foundation for the framework in the form of a meta-model, which is later augmented with the concepts specific to a particular pattern family. The five pattern families: Multi-party Multi-Message Request-Reply Conversation, Renewable Subscription, Message Correlation, Message Mediation, and Bipartite Conversation Correlation are described in sections 5.2.1, 5.2.2, 5.2.3, 5.2.4 and 5.2.5 respectively. The operational support for the pattern variants identified in selected PAISs is examined in Section 5.3.

5.2 Configurable framework for service interaction patterns

In this section, we present a framework for describing new service interaction patterns. To systematically describe the various patterns identified, pattern variants sharing the same set of concepts are combined into a single pattern family. The differences between pattern variants belonging to the same pattern family are described by means of pattern attributes. For each of the pattern attributes a range of possible values is identified. The set of all pattern attributes identified for a given pattern family forms a pattern configuration. By instantiating all pattern attributes, a pattern configuration representing a specific pattern variant can be obtained.

The pattern families introduced in this section adopt a consistent presentation format. We depict each pattern configuration using an intuitive graphical notation, consisting of a set of labels each of which corresponds to a specific pattern attribute. By configuring these attributes, a graphical notation of a specific pattern variant is obtained.

The meta-model describing the key concepts shared by all pattern family is illustrated by the UML class diagram shown in Figure 191. Later, various parts of this core model will be extended with concepts specific to each of the pattern families.

![Figure 191](UML-meta-model-of-service-interaction-concepts-shared-by-all-pattern-families)
For the purpose of this thesis, a *conversation* is defined as the communication of a set of contextually related messages between two or more parties. A *party* is an entity involved in communication with other parties by means of sending/receiving messages. A party may represent a process, a service, a business unit, etc. A *message* is a unit of information that may be composed of one or more data fields.

The association *involves* in Figure 191 shows that two or more parties may be involved in a conversation, represented as *Party* and *Conversation* classes respectively. The composition relation between *Conversation* and *Message* classes indicates that at least one message is exchanged in a conversation. A message may represent a *request* or a *reply* as visualized by specializations *Request* and *Reply* of the *Message* class. There is an association between *Request* and *Reply* message types indicating that there may be zero or more replies corresponding to a given request. In case of the one-way interaction, no reply is required, whereas in case of the two-way interaction a reply message may be either not sent at all, or one or more replies may be sent in response to the request message received.

To clarify the semantics of the pattern configuration we use a formalism based on CPNs. For each pattern family, we have designed a (set of) CPN model(s) which is executable in CPN Tools [66]. Declarations used within CPN models are consistent with the concepts described in the meta-model and pattern attributes characterizing the pattern configuration.

The pattern attributes that influence the detailed semantics of each pattern variant are described separately. Pattern attributes (also referred to as parameters) represent orthogonal dimensions used for classifying different aspects of service interaction within the context of the given meta-pattern. All possible combinations of the attribute values result in a large set of pattern variants, each of which can be easily derived from the pattern configuration and are depicted by a corresponding label. In total, for the five pattern families identified, 1602 meaningful pattern variants can be generated.

The five pattern families, from which this framework is composed of, are described using the following format:

- **Description**: service interaction scenarios to which the given pattern family applies.
- **Examples**: examples illustrating the application of the given pattern variant in practice.
- **UML meta-model**: a model describing the concepts specific to a given pattern family.
- **Visualization**: a graphical notation representing a pattern configuration and the detailed description of configuration parameters.
- **Illustrative example**: classification of one of the examples using the configuration parameters and its representation using the graphical notation defined for the given pattern family.
- **CPN semantics**: the semantics of a generic pattern illustrated in the form of CPNs presented by means of the screen-shots of the models and their corresponding description.
- **Issues**: issues that can be encountered when applying a pattern variant from the given pattern family in practice and corresponding solutions.

In the remainder of this section, we describe each of the pattern families in more detail.
Various graphical notations exist for depicting service interactions (e.g., BPMN [165], WS-CDL [217], Let’s Dance [235]). They focus on different aspects of service interaction and adopt different approaches to its visualization. Even when describing basic interactions, using these notations not all interaction parameters are taken into account. For example, there is the possibility of a deadlock situation in the following scenario where two parties, a customer and a provider, are engaged in an order/purchase process. Whilst the provider waits for the payment to arrive before delivering the ordered product, the customer may wait for the product to be delivered before paying for it. In order to prevent an endless waiting process, it is important to specify under which conditions an interaction is completed/aborted. In BPMN and WS-CDL this problem is solved by identifying control-flow dependencies between the parties. Let’s Dance adopts the interaction modeling approach, where interactions are visualized as basic building blocks in the choreography model. However, parameters such as the possibility for missing requests and replies are not included in these notations. Furthermore, the interaction attributes such as conditions for (force)completion of an interaction or rules describing consumption and utilization of messages received are also not fully addressed.

On page 234, we introduce a graphical notation for depicting the dynamic nature of request-reply interactions which can be configured according to the required interaction characteristics.

5.2.1 Pattern family: Multi-party Multi-message Request-Reply Conversation

The first pattern family described using the format discussed in the previous section is the Multi-party Multi-message Request-Reply Conversation pattern family.

**Description** A requestor posts a compound request consisting of \( N \) sub-requests to a set of \( M \) parties and expects a reply message to be received for each sub-request. There exists the possibility that some parties will not respond at all and also the possibility that a responder will not reply to some sub-requests. The requestor queues all incoming messages in a certain order. The enabling of the requestor for consumption of reply messages depends on the fulfillment of certain activation criteria. The requestor should be able to, optionally, consume a subset of the responses and even process a subset of the consumed set - hence allowing for use in cases where only the best or fastest responses are needed. The number of times the requestor may consume messages from the queue can be specified explicitly.

**Example**
- Requests to submit an abstract or to submit a paper are issued by an editor to a list of 117 people registered for participation in a workshop. Only papers and abstracts submitted before the deadline will be reviewed. If a large number of papers arrive, only the first 50 will be reviewed and only 10 best papers out of the reviewed ones will be published.

**UML meta-model** An object diagram illustrating the Multi-party Multi-message Request-Reply Conversation pattern family at a conceptual level is presented in Figure 192. A conversation consists of a set of messages (cf. the composition relation between Conversation and Message). A conversation involves an initiating process (e.g., requestor), and at least one following process (e.g., responder), depicted by the associations requestor and
A process may be or may not be involved in multiple conversations (cf. the multiplicity of the association involves). The requestor generates at least one Request message, while the responder returns one or more Reply messages or does not react at all. The relation between Request and Reply messages is depicted by the corresponds to association, and the sending of request and reply messages by a party is illustrated by the is sent by and is produced by dependency relations. Request messages issued by a requester can be composite. This means that the requestor may send several sub-requests in a single message concurrently to a single or to multiple parties.

**Figure 192:** UML meta-model of Multi-party Multi-message Request-Reply Conversation

**Visualization** The graphical notation of the configurable Multi-party Multi-message Request-Reply Conversation is shown in Figure 193. The parties are visualized as rectangles. Directed arrows represent the direction in which a party sends a message. A message containing a single request is visualized as a black token, while a compound request is represented by multiple overlapping tokens. Parameters specific to a given party are visualized as labels residing within the boundaries of a rectangle representing a party. This graphical notation is used to specify the following set of configuration parameters:

**Figure 193:** Graphical notation: Multi-party Multi-message Request-Reply Conversation
Section 5.2 Configurable framework for service interaction patterns

Figure 194: Variants of graphical notation for Multi-party Multi-message Request-Reply Conversation

- **N** - a parameter denoting a list of sub-requests sent by a requestor to a responder in a single message.
  
  **Range of values:** $\text{size}(N) \geq 1$.
  
  **Default value:** $\text{size}(N)=1$.
  
  **Visualization:** This parameter is depicted by the dots on the arc from the requestor to the responder(s). For $\text{size}(N) > 1$ and $\text{size}(N)=1$ the graphical notations depicted in Figure 194 (1a) and (1b) are used respectively.

- **M** - a parameter denoting a list of responders involved in the conversation.
  
  **Range of values:** $\text{size}(M) \geq 1$.
  
  **Default value:** $\text{size}(M)=1$.
  
  **Visualization:** For $\text{size}(M) > 1$ and $\text{size}(M)=1$ the graphical notations depicted in Figure 194 (2a) and (2b) are used respectively.

- **Possibility of non-responding parties** - a parameter specifying whether some of the responders will ignore the request issued by the requestor.
  
  **Range of values:**
  
  - No: all $M$ responders will reply at least something (for example, a request to report the level of income to the tax-office obliges all receivers to reply);
  - Yes: some responders may not reply at all (for example, only interested parties react on the invitation to participate in a social event).
  
  **Default value:** No.
  
  **Visualization:** Figure 194 depicts the graphical representation of four variations, where: in (4a) and (4b) all $M$ responders will produce at least some replies; in (4c) and (4d) some responders may not reply to all requests received. (See also the next configuration parameter).

- **Possibility of missing replies** - a parameter specifying whether the responder will not reply to some of the sub-requests (i.e. it is the choice of the responder to engage in a conversation or not, and to reply to all or only to some of the received requests).
Chapter 5 Service Interaction Patterns

Range of values:

- No: responders reply to all sub-requests (for example, the responder answers on all questions in the tax declaration);
- Yes: responders reply only to some sub-requests (for example, a client subscribes only to two out of the five journal offers received).

Default value: No.

Visualization: Figure 194 depicts the graphical representation of four variations, where: in (4a) and (4c) no replies are lost; in (4b) and (4d) some replies may not reach the requestor. Note that this notation is combined with the notation of the previous parameter.

- **Sorting of queued messages** - a parameter specifying an ordering discipline according to which response messages queued by the sender are sorted.

  Range of values:

  - FIFO: oldest message is the first one in the queue;
  - LIFO: newest message is the first one in the queue;
  - PRIO: sorting based on some criterion (for instance, on price);
  - NoQueue: messages are not queued and are consumed upon arrival if the sender is ready to process them, otherwise they are lost.

  Default value: NoQueue.

  Visualization: Figure 194 (5a)-(5d) depicts the graphical notation of different policies that can be applied for sorting messages in the queue.

- **Enabling condition** - a parameter specifying the condition that has to be fulfilled to enable the requestor to consume replies.

  Range of values:

  - a timeout (for example, requests for purchase on a discounted basis are accepted only until the expiration of the discount period);
  - a specified minimal number of messages K (0<K≤N);
  - a Boolean condition, examining the properties of the queued messages (for example, at least three low-cost offers are required in order to select the best of them).

  Default value: K=1.

  Visualization: the E label residing at the requestor’s side in Figure 193 substituted with one of the graphical notations presented in Figure 194 (3a), (3b) and (3c) which denote activation conditions based on a timeout, availability of specific number of messages and a Boolean expression respectively.

- **Consumption index** - a parameter specifying the number of reply messages to be consumed by the requestor from the queue.

  Range of values:

  - 0: none of the messages are removed from the queue (for example, messages enable the receipt process, but it may be necessary to leave them on the queue for another process to use);
  - S: S messages are removed from the queue such that 0<S<K, where K is the number of replies sufficient for activation of the requestor (as specified in the enabling condition);
  - All: all messages contained in the queue are removed.

  Default value: All.

  Visualization: the C label residing at the requestor’s side in Figure 193 substituted
with a suitable value, i.e. 0, S or All.

- **Utilization index** - a parameter specifying the number of messages from those consumed that are used by the requestor for the processing.
  
  **Range of values:**
  
  - 0: no messages are used for processing (for example, if no messages were consumed, or if none of the consumed messages are required by the receiving process);
  - 1: one message is used for processing (for instance, the best offer from the available ones is selected);
  - UN: a specific number of messages is used for processing such that $1 < UN < C$, where $C$ is the number of messages consumed;
  - All: all consumed messages are used for processing.

  **Default value:** All.

  **Visualization:** the $U$ label residing at the requestor’s side in Figure 193 substituted with its value, i.e. 0, 1, UN or All.

- **Consumption Frequency** - a parameter specifying the number of times the requestor performs the consumption of messages from the queue.
  
  **Range of values:**
  
  - 1: the requestor is activated only once, after this all remaining and arriving messages are destroyed;
  - FN: the requestor consumes messages FN number of times, $1 < FN$, after which all remaining and subsequent messages are destroyed;
  - $\infty$: the requestor consumes messages as long as they continue to arrive.

  **Default value:** 1.

  **Visualization:** the $F$ label residing at the requestor’s side in Figure 193 substituted with its value, i.e. 1, FN, or $\infty$.

The pattern variant representing a scenario in which every parameter is set to the default value is presented in Figure 195. A party A sends a single request to a party B, who sends a reply back. The party A does not queue messages, and consumes them as soon as they arrive. Only one message is necessary for party A to become enabled and start the consumption and processing of messages. All other messages that may arrive later will be discarded.

**Figure 195:** Notation for the default Multi-party Multi-message Request-Reply Conversation pattern variant
Meaningful configurations Various pattern variants can be obtained by setting configuration parameters to a specific value. The total number of possible configurations can be calculated by multiplying the number of values defined for each of the configuration parameters. One could conceive that for the Multi-party Multi-message Request-Reply Conversation pattern family in total 6912 configurations would be permissible. This number is obtained by counting all possible combinations of parameter settings. However, few of them represent meaningful pattern variants. Therefore, we try to provide a more accurate number.

Meaningful configurations are illustrated in Figure 196 that represents an alternative to commonly used tree structure. As the latter is used to determine the number of paths from the top parent node to bottom children nodes, this representation can be used to calculate the total number of meaningful pattern configurations. The configuration parameters correspond to nodes, and arcs directed downwards correspond to the leaves. Each arc is associated with a set of values the configuration parameters may take, thus multiple values associated with an arc compactly represent multiple arcs each associated with a single value.

![Figure 196: Meaningful configurations of Multi-party Multi-message Request-Reply Conversation (total 1072 configurations)](image)

In order to arrive at an arithmetic expression for calculating the total number of meaningful configurations, we define a set of rules describing how to construct such an expression. We start with an empty expression and extend it gradually by moving from the top to the bottom of the tree structure in Figure 196. Where a parameter has a single outgoing arc, the expression is extended by multiplying it by the number of values associated with the arc. For example, starting with an empty expression and processing the parameter N (i.e. number of sub-requests in a message) with 2 values on its outgoing arc (representing a single request in a message or multiple ones), will result in a total expression of “2 * ...". Processing the next node M (i.e. number of responders) will result in the same extension to the expression, i.e. “2 * 2 * ...". When dealing with a parameter with several arcs outgoing from the same point the following extension to the expression is made: “∑ \( A \) \( n_i \) * (...),
where \( A \) is the number of arcs outgoing from the same point and \( n_i \) is a number of values associated with the arc \( i \). Then each of the bracket-pairs needs to be filled in separately. Each bracket pair has to be initialized with an empty expression if the bottom of the structure has not been reached, or be treated as 1. The described procedure has to be recursively applied to every bracket.

Using the rule described, a total number of 1072 meaningful configurations can be calculated (e.g., \( 2^2 \times 2^4 \times (1^1 \times (1^1 \times 1^1 + 1^3 \times 1^1)) + 3^2 \times (1^1 \times (1^1 \times 1^1 + 1^3 \times 1^3 + 1^3 \times 1^1) + 1^1 \times (1^1 \times 1^1 + 1^3 \times 1^1) + 1^1 \times (1^1 \times 1^1 + 1^3 \times 1^1)) = 1072 \)). It is interesting to see why not all pattern configurations are considered to be meaningful. The configuration parameters specifying the number of sub-requests in a message, the number of responders involved in the conversation, the possibility of non-responding parties and missing replies are orthogonal, i.e. any combination of their values is possible. Distinct approaches for calculating the meaningful configurations start from the sorting algorithm applied for messages queued. If no queuing mechanism is available (\( \text{NoQueue} \)), then incoming messages are consumed and processed upon arrival. This means that enabling conditions related to counting of messages in a queue, analyzing their properties or postponed consumption are not applicable. In other cases, when messages are queued, they are sorted according to one of the sorting algorithms, i.e. \( \text{FIFO}, \text{LIFO} \) or \( \text{PRIO} \). To differentiate between rules for consumption and processing for queued and non-queued messages, the latter are visualized separately.

If messages are not queued, then they are consumed immediately for processing, i.e. the number of messages required for starting the message consumption \( K = 1 \). The message arrived may be either discarded, in which case the consumption index is \( C = 0 \), or it is consumed \( C = \text{All} \). If no messages are consumed, none of them can be utilized (i.e. \( U = 0 \)). If all messages were consumed, then either none, one or all of them can be utilized. Note that in this case, consuming one or all messages has the same semantics. Because there is no queue available, no messages are left for future consumption, thus the only possible value of consumption frequency is \( F = 1 \).

If messages are queued, the consumption of messages from the queue may be enabled based on a timeout, the availability of messages satisfying a certain property and the availability of a minimal number of messages. In the first two cases, if the consumption of messages is enabled, either none or all messages are consumed from the queue (\( C = 0 \) or \( C = \text{All} \)). If the enabling of message consumption is based on the availability of a minimal number of messages in the queue, then all, none or a subset of these messages (\( C = S \)) may be consumed for processing. Only in the case when not all messages available in the queue are consumed, is there the possibility for multiple consumption (\( F = \text{FN} \) or \( F = \infty \)).

Illustrative example To illustrate how the pattern configuration of the \textit{Multi-party Multi-message Request-Reply Conversation} can be applied in practice, we revisit the example presented earlier for which we describe the corresponding pattern variant by defining values for the configuration parameters.

In this example, the editor plays the role of the requestor and people registered for participation in a workshop represent multiple responders (\( \text{size}(M) = 117 \)). The editor sends a request to submit an abstract or to submit a paper, thus the request is of a composite nature and \( \text{size}(N) \geq 1 \). Since there is no guarantee that the responders will reply, there is the possibility of non-responding parties. Since not all responders may respond by sending an abstract and a paper, there is also the possibility of missing replies. Reply messages are sorted according to a FIFO policy. The enabling condition for consumption of replies for processing is set to the timeout corresponding to the nominated deadline. The consumption index is set to 50 papers, this means that all other messages will be
discarded. The utilization index is set to 10 (only the 10 best papers will be reviewed). The consumption frequency is set to 1, no messages are stored in the queue for subsequent processing. The graphical notation representing the pattern configuration for this example is shown in Figure 197.

![Figure 197: Notation for the paper submission example](image)

**CPN semantics** To avoid an ambiguous interpretation of the pattern variants related to Multi-party Multi-message Request-Reply Conversation we formalize the semantics by means of CPNs. Figure 198 depicts the top view of the CPN diagram representing the pattern. Requestor and responder are represented as substitution transitions which can be unfolded to the nets depicted in Figure 199 and Figure 200(c) respectively. In every conversation, the parties exchange requests and replies of type `Message`. Since the requestor process does not expect a simultaneous reply, the interactions between the requestor and the responder are realized using the `Asynchronous Transfer` CPN pattern, described on page 54.

The requestor (whose behavior is shown in Figure 199) can send requests and receive

![Table 5.1: Data types used in Figures 198 -201](image)

| colset Party = string; |
| colset Request = string; |
| colset Requests = list Request; |
| colset Reply = string; |
| colset Replies = list Reply; |
| colset ConvId = int; |
| colset Content = union Req:Requests + Repl:Replies; | ¹ |
| colset Message = product ConvId * Party * Party * Content; | ² |

¹ The content of a message is either a list of requests or a list of replies. The CPN union type is used to specify this.

² A message is a tuple (cid,P1,P2,c) where cid is a conversation identifier, P1 is the requestor, P2 is the responder, and c is the content. Such a message is of type `Message`.

![Figure 198: CPN diagram: The top view of Multi-party Multi-response Request-Reply Conversation](image)
response messages using substitution transitions Send request and Receive response whose decompositions are shown in Figure 200(b) and Figure 201. A requestor process may have multiple process instances, whose lifecycle is shown in Figure 200(a). Process instances available for participation in a conversation are stored in the enabled place. A process instance chosen for conversation is stored in the running place. Transitions activate, deactivate and complete control the status of a process instance during its lifecycle. When an enabled process instance is activated, it has the status active and may participate in sending and receiving of messages. Meanwhile the active process instance can become inactive through deactivation or can become completed. The lifecycle of a process instances ends when it is complete and the process instance is added to the completed place. Note that in order to realize different statuses of a process instance we use the Non-deterministic XOR-split CPN pattern, described on page 38.

Each process instance is represented by a list of requests crqs that need to be sent, a list of replies crps received from the responder parties, and the status of the process instance. In order to refer to requests and replies associated with a given process instance

Table 5.2: Data types used in Figure 199

| colset Count = int; |
| colset MTime = int; |
| colset Status = with active|inactive|enabled|completed; |
| colset ConvRequest = product Parties * Requests; |
| colset ConvRequests = list ConvRequest; |
| colset ConvReply = product Parties * Replies; |
| colset ConvReplies = list ConvReply; |
| colset Pr = product ConvRequests*ConvReplies*Status; |
| colset Proc = product ConvId*Pr; |
| colset ConvInfo = record start_time: MTime * last_act:MTime |
| * nof_unique_messages: Count * nof_parties: Count * |
| total_nof_messages: Count; |
| colset Conv = product ConvId * ConvInfo * Status; |

---

3The lifecycle of a process instance starts with the activation of an enabled instance. An active instance can become inactive through deactivation, or completed when the instance lifecycle ends.

4Process instances of type Pr contain a list of requests sent, replies received and the status of the instance. When a conversation starts, a process instance is coupled with a conversation identifier.

Figure 199: CPN diagram: The Requestor page of Multi-party Multi-message Request-Reply Conversation
(a) The Process instances sub-page

(b) The Requestor’s Send Request sub-page

Figure 200: CPN diagrams of Multi-party Multi-message Request-Reply Conversation

as a single entity, we aggregate them in a collection using the Aggregate Objects CPN pattern, described on page 67.

The requestor’s Send Request sub-page in Figure 200(b) shows that the requestor, whose identifier is stored in place Requestor ID, at the moment of sending a request message creates a new conversation. In order to distinguish conversations, for each of them a conversation identifier cid is generated. The uniqueness of identifiers is ensured by incrementing a counter whose value is stored in the conversation counter place. To realize this counter we applied the ID Manager CPN pattern, described on page 51. Note
that request messages are sent by the requestor only if the list of requests is not empty. This condition is included in the guard of the \texttt{send} transition, and acts as a filter based on the \textit{BSD Filter} CPN pattern, described on page 43.

Function \texttt{create\_messages()} takes a list of conversation requests \texttt{crqs} (which is of the \texttt{ConvRequests} type), which contains a list of parties to whom a request should be sent, and a list of sub-requests that should be sent to each party, and creates as many messages as there are parties in the list. This function directly corresponds to the configuration parameter specifying that messages with N sub-requests are sent to M parties.

When request messages are created, a new conversation is created by means of the function \texttt{create\_conversation()}. This function records information about the conversation identifier, conversation-specific parameters (the start time of the conversation, the time of the last activation, a total number of messages sent, the number of parties to whom the requests have been sent, and the number of unique messages (i.e. a number of sub-requests can be contained in the single message)), and the status of the process instance. The recorded conversation information is used later on for the purpose of correlating response messages received with the requests sent and for identifying how many times the received messages can be consumed for processing. This behavior corresponds to the \textit{Shared Database} CPN pattern, described on page 90.

The responder page shown in Figure 200(c) illustrates the behavior of responders involved in the conversation. The identities of the responders are stored in place \texttt{self}. They are used to relate incoming requests to the right party, based on the party identifier. The operation of matching the party identifiers is realized according to the \textit{ID Matching} CPN pattern, described on page 49. When a responder receives a request message, it unpacks the composite requests into separate messages each containing a distinct sub-request. The parameter \texttt{prob\_all\_lost\_for\_party} corresponds to a configuration parameter specifying the probability that the responder will ignore a received composite request or will process it. If the responder decides to reply to the request, the parameter \texttt{prob\_individual\_message\_lost} is used as a configuration parameter to define the probability that a reply will be sent for every unpacked sub-request.

The requestor’s \texttt{Receive response} sub-page presented in Figure 201 illustrates the mechanism for queueing and processing incoming responses by the requestor. The requestor processes only messages addressed to it based on the requestor identity stored in the \texttt{Requestor ID} place. For this purpose the \textit{ID Matching} CPN pattern, described on page 49, is used. The response messages received are queued according to the \texttt{QueueingDiscipline()} function, which corresponds to the configuration parameter that can be set to any of the supported queueing disciplines, i.e. LIFO, FIFO or PRIO, and realized using the \texttt{Queue} (cf. page 75), \texttt{FIFO Queue} (cf. page 78), \texttt{LIFO Queue} (cf. page 79), and \texttt{Priority Queue} (cf. page 81) CPN patterns respectively. If messages should not be sorted (\texttt{NoQueue}), they are filtered out using the \textit{BSD Filter} CPN pattern, described on page 43, and consumed immediately.

Function \texttt{Consume()} corresponds to the configuration parameter specifying how many messages from the queued ones have to be consumed. One, several, or all available messages in the queue can be consumed. The consumption of messages occurs when the enabling condition (encoded as a guard of transition \texttt{Pull}) is satisfied. The \texttt{Activated} function can be configured to enable the \texttt{pull} transition based on the availability of one or several messages in a queue, upon the satisfaction of a certain condition or upon a timeout. The guarded enabling of the \texttt{Pull} transition corresponds to the \textit{BSD Filter} CPN pattern (cf. page 43).
Variation point: a number of times the sender is activated for consumption

Variation point: immediate consumption or not

Variation point: enabling condition for activating the consumption of messages

Variation point: a number of consumed messages used from the consumed ones

**Figure 201:** CPN diagram: The Requestor’s Receive Response sub-page

From the messages consumed only the number of messages defined by the Use() function are actually used by the requestor for the processing. This configuration parameter can be set to use either one, several or all consumed messages.

The MaxAct parameter corresponds to the configuration parameter specifying how many times the requestor may consume the messages from the queue for the given conversation. If the messages have been consumed the specified number of times, the process instance receives the status completed and the messages left in the queue are removed from it by means of the filter() function. Transition destroy messages is used to retrieve messages from the queue place if the incoming response messages do not need to be sorted and have to be consumed immediately upon arrival.

When describing the behavior of the Multi-party Multi-message Request-Reply Conversation pattern family in the form of CPN, we used 12 distinct CPN patterns. Table 5.3 indicates how frequently each of the patterns has been used and how many patterns in total, i.e. including the repeating ones, were applied.

**Issues** When applying pattern variants belonging to the Multi-party Multi-message Request-Reply Conversation pattern family the issue of the message correlation may arise while matching replies received with the requests sent. This issue can be addressed by applying a suitable pattern variant from the Message Correlation pattern family (described in Section 5.2.3). If the Multi-message Multi-Party Request-Reply Conversation pattern variant has to be applied in the context of a long-running conversation, where a series of requests have to be sent one after another, the given pattern variant can be combined with a suitable pattern variant from the Bipartite Conversation Correlation pattern family (described in Section 5.2.5).
Section 5.2 Configurable framework for service interaction patterns

Table 5.3: CPN patterns used in the implementation of Multi-party Multi-message Request-Reply Conversation

<table>
<thead>
<tr>
<th>CPN pattern</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Matching</td>
<td>2</td>
</tr>
<tr>
<td>ID Manager</td>
<td>1</td>
</tr>
<tr>
<td>Aggregate Objects</td>
<td>2</td>
</tr>
<tr>
<td>Queue</td>
<td>1</td>
</tr>
<tr>
<td>FIFO Queue</td>
<td>1</td>
</tr>
<tr>
<td>LIFO Queue</td>
<td>1</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>1</td>
</tr>
<tr>
<td>Shared Database</td>
<td>1</td>
</tr>
<tr>
<td>Data Distributor</td>
<td>1</td>
</tr>
<tr>
<td>BSD Filter</td>
<td>3</td>
</tr>
<tr>
<td>Asynchronous Transfer</td>
<td>2</td>
</tr>
<tr>
<td>Non-deterministic XOR-split</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>16</strong></td>
</tr>
</tbody>
</table>

The publish-subscribe interaction is a special kind of interaction between two parties, a provider and a customer, whose main goal is to establish an agreement on delivery of a particular product according the specific subscription terms. A publish-subscribe interaction is usually performed in several phases: an initiation phase where an agreement to deliver a product is established between a customer and a provider, and a renewal phase where one of the parties takes the responsibility to renew or cancel the current subscription. In software community, the publish-subscribe pattern is often used to describe a situation where an entity maintains a list of dependents and notifies them of any state change. Interesting to note that when considering publish-subscribe scenarios the focus is usually set on the initiation of a particular conversation, whereas the renewal phase is not considered at all. For example, the ‘Publish-Subscribe Channel’ described in the Enterprise Integration Patterns by Hohpe [125] concentrates on broadcasting of an event to interested receivers by splitting one input channel into several output channels and delivering an event to each of these output channels. The main focus of the Hohpe’s pattern is to announce an event to multiple receivers; hence it does not address possible variants of reacting to events by a subscriber and omits strategies for renewing an established subscription.

In Section 5.2.2, we focus on renewable subscriptions. On page 246, we describe possible variants of subscription renewal and provide a graphical notation to depict both subscription initiation and subscription renewal phases of the conversation established between a publisher and a subscriber.

5.2.2 Pattern family: Renewable Subscription

This sub-section describes the second pattern family named **Renewable Subscription**.

**Description** Two parties, a provider and a customer, are involved in a conversation with each other. A provider offers a product under specific subscription terms and a customer consumes the product. Both the provider and the customer may initiate the subscription process by sending a request message. There may be a confirmation/rejection response to
the subscription request or no response at all. Depending on the terms of subscription, the subscription can be renewed automatically, at the initiative of the customer or the provider.

**Examples**

- To apply for travel insurance, a client contacts an insurance company. The insurance company informs the client about available types of insurance and duration terms. Once the client has taken out insurance, it is automatically renewed every year. The insurance can be canceled at the client’s request at any time.

- A short-term trial newspaper subscription can be extended at the request of a reader.

**UML meta-model**

Concepts specific to the *Renewable Subscription* pattern family are illustrated by means of the class diagram in Figure 202. By *subscription* we understand a conversation between two parties, one party offering a product (cf. the *provider* association between *Party* and *Subscription*) and another party consuming it (cf. the *customer* association between *Party* and *Subscription*), to establish an agreement for delivery of a certain *product* according to pre-agreed *subscription terms*. A party may offer zero or more products and define a set of subscription terms (cf. aggregation relations between *Party*, and *Product* and *SubscrTerms*). Subscription terms define a subscription period, a number of products to be delivered during this subscription period, a response period within which the customer has to acknowledge the acceptance or rejection of the offered subscription, and how the acknowledgment should be notified. The established subscription relates to one product and a particular set of subscription terms, although these could be the same for multiple subscriptions.

Each subscription has one initiation phase and may have multiple renewal phases or may not have a Renewal phase at all (cf. the association relations between *Subscription*, *Initiation* and *Renewal*). Initiation and renewal are conversations (cf. specialization relation between *Conversation*, *Initiation* and *Renewal*) held for the purpose of establishing and renewing of a subscription respectively. Parties involved in the conversation, play the role of *requestor* or *responder*, where requestor initiates a phase by sending a *Request* message and responder replies with zero or more *Reply* messages.

**Visualization** Figure 203 illustrates the graphical notation for the *Subscription Renewal* pattern configuration. Depending on who initiates the subscription and who takes the initiative for its renewal, six subscription renewal types can be distinguished. For each subscription renewal type listed in Table 5.4 there is a separate graphical notation as shown in Figure 203.

<table>
<thead>
<tr>
<th>Subscription type</th>
<th>Initiator</th>
<th>Renewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Customer-initiated Automatically-renewed</td>
<td>Customer</td>
<td>none</td>
</tr>
<tr>
<td>(b) Provider-initiated Automatically-renewed</td>
<td>Provider</td>
<td>none</td>
</tr>
<tr>
<td>(c) Customer-initiated Customer-renewed</td>
<td>Customer</td>
<td>Customer</td>
</tr>
<tr>
<td>(d) Provider-initiated Customer-renewed</td>
<td>Provider</td>
<td>Customer</td>
</tr>
<tr>
<td>(e) Customer-initiated Provider-renewed</td>
<td>Customer</td>
<td>Provider</td>
</tr>
<tr>
<td>(f) Provider-initiated Provider-renewed</td>
<td>Provider</td>
<td>Provider</td>
</tr>
</tbody>
</table>

The parties are visualized as rectangles with a vertical line in the center of a rectangle representing internal message flow. Directed arrows between rectangles represent the direction in which a party sends a message. A dashed arrow indicates that no reply may
be sent back, i.e. the reply is optional. Every message is represented as a black token. The message properties are embedded in the rectangles attached to the message. The time sequence of message exchange corresponds to the time axis. Request and reply messages are denoted as REQ and RPL respectively. Message indexes c and p denote that message is sent by the customer or the provider respectively. Message indexes init and renew denote that the message is related to the initialization or renewal of a subscription, while index cnlrenew identifies the message is related to the cancelation of an automatically renewed subscription.

Besides selecting the subscription renewal type, there is a set of the following configuration parameters\(^5\) that have to be set to a specific value in order to differentiate pattern variants:

- **Expected initiation confirmation**: confirmation expected by the provider to the subscription initiation offer sent to the customer. This parameter applies only for the provider-initiated subscription type.
  
  **Range of values**:  
  - Yes/No: the provider requests the customer to reply with “Yes” or “No” to accept or reject an offer for initiation of a subscription. If no confirmation is received, no subscription is established.

\(^5\)We omit the default values of the configuration parameters, since these may not apply to all subscription renewal types.
Yes: the provider requests the customer to reply with “Yes” in order to initiate a subscription. If the expected response is not received, no subscription is established.

No: the provider requests the customer to reply “No” in order to terminate the initiation of a subscription which is implicitly considered to be established. If expected response is not received, the subscription is considered to be accepted.

**Visualization:** the \( Q_i \) label in the message properties of an initiation request sent by the provider to the customer, substituted with a suitable value (Figure 203)(d),(e) and (f)). An example illustrating the provider’s request with an expected confirmation “Yes” is presented in Figure 204.

- **Expected renewal confirmation:** confirmation expected by the provider to the subscription renewal offer sent to the customer. This parameter applies only to the provider-renewed subscription type.

  **Range of values:**

  - Yes/No: the provider requests the customer to reply with “Yes” or “No” to accept or reject an offer for the renewal of a subscription. If no confirmation is received, no subscription is established.
  - Yes: the provider requests the customer to reply with “Yes” in order to renew a subscription. If the expected response is not received, no subscription is established.
  - No: the provider requests the customer to reply with “No” in order to terminate the renewal of a subscription which is implicitly considered to be established. If expected response is not received, the subscription is considered to be accepted.

  **Visualization:** the \( Q_r \) label in the message properties of a renewal request sent by the provider to the customer, substituted with a suitable value (Figure 203(c) and (f)).

Besides the configuration parameters, the graphical notation in Figure 203 also contains a set of dynamic attributes. Values of dynamic attributes may vary for different examples that are characterized by the same pattern variant. The dynamic attributes describe characteristics of a subscription such as the period of subscription (SP), the specific product (Prod), the number of products to be delivered (Nr) within the subscription period, and the response period (RP) during which a subscription offer has to be accepted. Furthermore, the customer’s response to the subscription initiation offer and to the subscription renewal offer (denoted \( R_i \) and \( R_r \) respectively), and the provider’s response to the subscription initiation request or the subscription renewal request received from the customer (denoted \( P_{Ri} \) and \( P_{Rr} \) respectively) belong to the behavioral variables that may have different values in different conversations. In particular, the customer may reply to the subscription offer received from the provider with *Yes* or *No*, or not reply at all. When the customer sends a request to the provider to initiate or renew a subscription, the provider may accept or reject the request, or may not reply at all. The behavioral variables are visualized by substituting labels Pro, SP, RP, NR, Ri, Rr, PHi, and PRr, residing in the properties of messages in the corresponding subscription renewal type in Figure 203, with a suitable value.
Section 5.2 Configurable framework for service interaction patterns

![Graphical notation: Renewable Subscriptions](image)

**Figure 203:** Graphical notation: Renewable Subscriptions
The example shown in Figure 204 presents a subscription offer for 4 issues of a journal “Cosmo” that will be delivered within 30 days. The customer is expected to reply on this offer with “Yes” to accept the offer within the response period of 14 days. The values of behavioral variables $R_i$ and $PR_r$ are not specified, because these are set dynamically and thus may take different values for each conversation. In a particular instance of the subscription conversation, the customer could accept the offer from the provider and acknowledge the acceptance by sending “Yes”. When the subscription period is about to finish, the customer can request the renewal of the subscription by sending a request specifying the name of the magazine. The provider may acknowledge the acceptance of the renewal request by issuing the “Accept” response, confirming the subscription period and the number of issues to be delivered.

![Diagram of subscription process]

Figure 204: An example of the Provider-initiated Customer-renewed pattern variant

**Meaningful configurations** Figure 205 depicts the structure for calculating meaningful pattern configurations of Renewable Subscriptions. Applying the rules described on page 238 for calculating the total number of possible configurations depicted in Figure 205, results in 20 pattern variants. In this structure, the expected initiation confirmation and expected renewal confirmation configuration parameters are applied only in case the provider plays the role of the subscription initiator and subscription renewer respectively.

**Illustrative example** To describe the pattern variants used in one of the examples listed earlier, let’s define the values of the configuration parameters. In the travel insurance example, where a client contacts an insurance company to apply for travel insurance, the insurance is renewed automatically every year. The client and the insurance company map to the roles of the customer and the provider respectively. This corresponds to the Customer-initiated Automatically-renewed subscription type. The graphical notation for the pattern configuration for the given example is depicted in Figure 206. In a request to the insurance provider, the client specifies the product requested “trvl insurance”. The provider indicates to the client in the reply message the number of products to be delivered, i.e. 1, the subscription period of 1 year, and its acceptance response $PR_i$, which can be either “Accept” or “Reject”. Since the acceptance response is determined dynamically, it’s value is not specified. Within the subscription period, the client may send a cancelation request to the insurance provider to cancel the insurance.

**CPN semantics** In this section, we only describe in detail the semantics of Provider-
**Figure 205:** Meaningful configurations of Renewable Subscription (total 20 configurations)

**Figure 206:** Notation for the travel insurance example

initiated Customer-renewed subscription type. The CPN models for the remaining subscription renewal types are listed in [157].

Figure 207 illustrates the top view of the Provider-initiated Customer-renewed subscription scenario. Customer and provider are represented by substitution transitions with corresponding names that unfold to the sub-pages presented in Figure 208 and Figure 209 respectively. The definition of data types is based on the concepts and notation introduced earlier. The subscription scenario is based on the assumption, that there is a one-to-one relation between customer and provider. This means that from the customer perspective, conversations are performed with a single provider, and the same holds for the provider. This implies that customer and provider do not need to specify their identities in the messages exchanged (this could easily be added if desired). However, since multiple subscriptions can be established between customer and provider for the same product, the subscriptions have to be differentiated. For this purpose, a subscription identifier of type SID is introduced in the messages exchanged. Since interactions between the customer and the provider are of the asynchronous type, we realized them using the *Asynchronous Transfer* CPN pattern described on page 54.

The customer, whose behavior is presented in Figure 208, receives an initialization request reqInit and puts it in the Subscription offered place by means of the Receive

![Diagram of subscription scenario](image-url)
init request transition. If the customer decides to reply, the createcinitreply() function generates an initialization reply message of type RPLcinit. A variable r of type R indicates whether the request is accepted, rejected or ignored. Its value is non-deterministically determined. If the customer accepts the subscription, the subscription details are recorded in the Subscription established place by means of the recordsubscr() function. To create an initiation reply and record accepted subscriptions, we used the OR-split CPN pattern (cf. page 40) which allows both or either of these operations to be performed.

Products sent by the provider are received by the customer via transition Receive product, which examines whether the product delivered is the product expected by the customer by means of the productforme() function. For this, matching of the party

Table 5.5: Data types used in Fig. 207-210

<table>
<thead>
<tr>
<th>colset Prod = string;</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>colset RP = int;</td>
<td>7</td>
</tr>
<tr>
<td>colset SP = int;</td>
<td>8</td>
</tr>
<tr>
<td>colset Nr = int;</td>
<td>9</td>
</tr>
<tr>
<td>colset Q = with YesNo</td>
<td>Yes</td>
</tr>
<tr>
<td>colset R = with YesNo</td>
<td>none</td>
</tr>
<tr>
<td>colset PR = with Accept</td>
<td>Reject</td>
</tr>
<tr>
<td>colset SID = int;</td>
<td>13</td>
</tr>
<tr>
<td>colset REQpinit = product SID * Prod * Nr * RP * SP * Q;</td>
<td>14</td>
</tr>
<tr>
<td>colset REQprenew = product SID * Prod * Nr * RP * SP * Q;</td>
<td>15</td>
</tr>
<tr>
<td>colset REQcinit = Prod;</td>
<td>16</td>
</tr>
<tr>
<td>colset REQcrenew = product SID * Prod;</td>
<td>17</td>
</tr>
<tr>
<td>colset RPLpinit = product SID * Prod * Nr * SP * PR;</td>
<td>18</td>
</tr>
<tr>
<td>colset RPLprenew = product SID * Prod * Nr * SP * PR;</td>
<td>19</td>
</tr>
<tr>
<td>colset RPLcinit = product SID * Prod * R;</td>
<td>20</td>
</tr>
<tr>
<td>colset RPLcrenew = product SID * Prod * R;</td>
<td>21</td>
</tr>
</tbody>
</table>

---

6 Product name
7 Response period
8 Subscription period
9 Number of products to be delivered
10 Confirmation expected by provider
11 Confirmation sent by customer
12 Confirmation sent by provider
13 An identifier for distinguishing identical subscriptions
14 Init. request of provider
15 Renewal request of provider
16 Init. request of customer
17 Renewal request of customer
18 Init. reply of provider
19 Renewal reply of provider
20 Init. reply of customer
21 Renewal reply of customer

Figure 207: CPN diagram: The top view of Provider-initiated Customer-Renewal Subscription
identifiers is performed according to the ID Matching CPN pattern described on page 49. In order to block unintended messages, the BSD Filter CPN pattern (cf. page 43) is applied. When the last product has been received by the customer, the customer generates a renewal request via the createcrenewreq() function and sends it to the provider by the Send renew request transition. In order to match future replies from the provider with the request sent, the customer stores the subscription request in the Subscription requested place. When a renewal reply rplprenew is received from the provider, the replyforcustomer() function locates a corresponding request. If the provider accepted the renewal request, the function updatesubscr() records the details of the subscription renewed in the place Subscription established. From this moment on, the customer may continue receiving products and may perform subscription renewal requests again. Note that in order to filter out replies that match with requests sent both the ID Matching CPN pattern (cf. page 49) and the BSD Filter CPN pattern (cf. page 43) are used.

The provider, whose behavior is presented in Figure 209, initiates the conversation with the potential customer by sending an initialization request for subscription reqpinit of type REQpinit. Decomposition of a substitution transition Send init request is presented in Figure 210. The createpinitrequest() function generates an initialization request based on the product offers available at the provider in Product offers place and a variable q of type Q that represents the confirmation expected by the provider on the given request. This is a configuration parameter that is set dynamically to one of the values of the small color-set Q. Subscription offers sent are stored in place Offered subscription. In order to distinguish subscriptions between each other, they are assigned a conversation identifier generated by incrementing the value of a counter stored at the Conv Counter place. The use of the ID Manager CPN pattern (cf. page 51) ensures the uniqueness of the generated identifiers.

When a customer replies on the initialization request, the provider at first examines the reply message (rplcinit) received. The replyforprovider() function checks whether the reply received corresponds to any of the requests sent. The matching replies are passed through, while the other replies are blocked. To accomplish this behavior, the ID Matching (cf. page 49) and BSD Filter (cf. page 43) CPN patterns are used. Then, for correlated requests the subscription details are recorded by the createsubscr() function in the Established subscription place if the customer confirms the acceptance of the subscription. For established subscriptions, the provider delivers a specified number of products nr during the agreed subscription period sp.

Products are sent to the customer via transition Send product. When the provider receives a renewal request reqcrenew from the customer, it examines by the reqforprovider() function whether there is a corresponding subscription that could be renewed. A subscription which can be renewed is stored in place Requested subscription. If the provider decides not to neglect the reply, a reply message is created by the createrenewreply() function, and if the provider accepts the request received, the subscription details are recorded by the renewsubscr() function in the Established subscription place.

When describing the behavior of the Provider-initiated Customer-renewed Subscription in the form of a CPN, we used 7 distinct CPN patterns. The Table 5.7 indicates how frequent each of the patterns has been used and how many patterns were used in total, i.e. including the repeating ones, were applied.

**Issues** The pattern variants belonging to the family of Renewable Subscription consider one-to-one relation between a customer and a provider. However, in many real life scenarios
### Table 5.7: CPN patterns used in the implementation of Provider-initiated Customer-renewed Subscription

<table>
<thead>
<tr>
<th>CPN pattern</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Matching</td>
<td>3</td>
</tr>
<tr>
<td>ID Manager</td>
<td>1</td>
</tr>
<tr>
<td>Shared Database</td>
<td>4</td>
</tr>
<tr>
<td>Data Distributor</td>
<td>1</td>
</tr>
<tr>
<td>BSD Filter</td>
<td>4</td>
</tr>
<tr>
<td>Asynchronous Transfer</td>
<td>5</td>
</tr>
<tr>
<td>OR-Split</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>

### Figure 208: The Customer page of Provider-initiated Customer-Renewed Subscription

### Table 5.6: Data types used in Fig. 208-210

```plaintext
colset Offer = product Prod * Nr * RP * SP * Ren;
colset Subscr = product SID * Prod * Nr * SP * Ren;
colset OfferedSubscr = product SID * Prod * Nr * RP * SP * Q * Ren;
colset RequestedSubscr = product SID * Prod;
```
Section 5.2 Configurable framework for service interaction patterns

Figure 209: CPN diagram: The Provider page of Provider-initiated Customer-Renewed Subscription

Figure 210: CPN diagram: The Send init request sub-page of Provider-initiated Customer-Renewed Subscription

a customer may have multiple subscriptions with the same or different providers and a provider may have multiple subscriptions with the same or different customers. Such context conditions obviously require a deep insight into the message correlation issue. The issues of correlation can be addressed by combining a Renewable Subscription pattern variant with a suitable pattern variant from the families of Message Correlation or Bipartite Conversation Correlation. The involvement of multiple providers and customers as well as requests for multiple subscriptions can be expressed by combining a renewable subscription pattern variant with the pattern variants of the Multi-party Multi-message Request-Reply Conversation pattern family.
The term *message correlation* is often used for matching a reply message that has been received with an earlier request message that was sent, especially when multiple replies may be possible for a given request. In order to perform correlation, an entity needs to keep track of events describing executions, and each such log must include identifiers that associate a message with a specific context. Different sources of information are used as a key during correlation: a message id, a party id, a conversation id or a combination thereof. In some situations, this information is neglected, and the use of an artificial correlation identifier is enforced (the use of such correlation identifiers has been observed in Oracle BPEL PM (cf. Section 5.3.1)). In reality, information provided for correlation is rarely complete, therefore correlation is performed based on the analysis of the message content. When correlation fails (i.e. no corresponding earlier messages can be found), a new conversation may be initiated or the received message is discarded. The main goal of message correlation is to find a conversation to which a message received relates, and because a conversation may involve multiple parties, also to identify the corresponding party.

In Section 5.2.3, we focus on the fundamentals of message correlation and describe possible variants of successful correlation. The situation where a response to a message received needs to be sent to a third party is known as mediator-based interaction. We classify such scenarios on page 264. It is interesting to note that during a conversation the information used for correlation purposes may change. On page 281, we provide a graphical notation that is able to capture the dynamic character of message correlation in a bilateral context.

### 5.2.3 Pattern family: Message Correlation

The third pattern family, named *Message Correlation*, addresses issues of correlation at a lower-level of abstraction. This family is described in this sub-section.

**Description** A party communicating with other parties has to handle incoming messages in accordance with a history of established conversations. Message Correlation is the act of identifying the relevant conversation for a message received by the party.

**Example**
- An insurance company handles claims for refund of lost baggage, medical costs, etc. When a claim is received, an insurance advisor determines whether the client has a valid insurance policy and whether there are any records related to the claim received. If the client has no valid insurance, the advisor may provide the client with a new policy or may refuse to handle the claim.

**UML meta-model** Concepts specific to the *Message Correlation* pattern family are illustrated by means of the class diagram in Figure 211. A party may participate in multiple conversations with other parties (illustrated by the association relation between *Party* and *Conversation*). The party may send messages to and receive messages from other parties (cf. relations *send* and *receive* between *Party* and *Message*). It is an assumption of this pattern that messages exchanged between parties contain information about the sender, the receiver as well as additional content in the format (From, To, Content). The party sending out a message determines what information relating to the sender’s and receiver’s identity it wants to reveal in the message.

**Visualization** Figure 212 illustrates the graphical notation for the *Message Correlation*
Section 5.2 Configurable framework for service interaction patterns

Pattern configuration. A party is visualized as a rectangle. The direction of arrows linked to the party node indicates the direction of message flow. Information about the message sender contained in the message is enclosed in the From-field. The information about the message receiver contained in the message is enclosed in the To-field. Information the receiving party has about its own credentials before correlating a message received is enclosed in the Me field. The information a party has about the credentials of the other party involved in a conversation is enclosed in the You field. Information a party has about its own identity and the identity of the other party from whom a message has been received after message correlation is enclosed in fields Me and You respectively. We assume that information associated with the Content field of a message is irrelevant for message correlation, therefore we omit this field in the graphical notation.

This graphical notation contains a set of static attributes and configuration parameters. Both static attributes and configuration parameters have to be configured for each of the pattern variants. For all pattern variants, the value of static attributes is fixed and does not change, while configuration parameters can be configured in accordance with values in the specified range.

Values of the static attribute representing the knowledge of the receiving party about own identity before and after message correlation are denoted by the Me label. Me is a pair \((P_r, C_r)\) where \(P_r\) denotes the id of the party-receiver and \(C_r\) denotes the id of the conversation used by the receiving party to correlate the message received. To illustrate a pattern variant, the Me label is substituted with the value \((P_r, C_r)\). Note that Me is not a configuration parameter.

The graphical notation shown in Figure 212 illustrates that this pattern family has the following set of configuration parameters:

- **Message Sender field**: the extent of the information revealed by the sender of a message regarding its identity.

  Range of values: From is a tuple comprised of potential sender identifier and conversation identifier. The sender identifier is denoted \(P_s\) and the conversation identifier used by the sender for correlation purposes is denoted \(C_s\). Either the id of the sender, the id of the conversation or both can be missing in the From-field. Missing information can be either intentionally or accidentally underspecified by the message sender.
Figure 212: Graphical notation: Message Correlation

(for instance, when a party wants to hide its id or when it forgets to include some information). Missing information is denoted as \( \perp \). So, possible values of the From-field are \((P_s, C_s), (P_s, \perp), (\perp, C_s)\) and \((\perp, \perp)\).

- **Message Receiver field**: the extent of information specified by the sender of a message regarding the receiver’s identity.
  - **Range of values**: To is a tuple comprised of the intended receiver identifier and its conversation identifier. The receiver identifier is denoted \( P_r \) and the conversation identifier used by the receiver for correlation purposes is denoted \( C_r \). Either the id of the receiver, the id of the conversation or both can be omitted in the To-field. So, possible values of the To-field are \((P_r, C_r), (P_r, \perp), (\perp, C_r)\) and \((\perp, \perp)\).
  - **Default value**: \((P_r, \perp)\) (i.e. only the id of the intended party is specified).
  - **Visualization**: the To label shown in Figure 212 substituted with a suitable value.
  - An example specifying the default value for the message receiver field is shown in Figure 213.

- **Credentials of the message sender before message correlation**: receiver’s knowledge in regard to the credentials of the sending party involved in the conversation with the receiving party.
  - **Range of values**: You is a tuple including the potential message sender identifier and its conversation identifier. The same notation as introduced earlier is used to denote the id of the party-sender and its conversation identifier. Since the receiver’s information about the message sender may be incomplete, possible values of the You-field are \((P_s, C_s), (P_s, \perp), (\perp, C_s)\) and \((\perp, \perp)\).
  - **Default value**: \((P_s, \perp)\) (i.e. the receiving party has knowledge only about the identity of the sending party).
Visualization: the You label shown in Figure 212 substituted with a suitable value. An example specifying the default value of the message receiver field is shown in Figure 213.

- **Credentials of the message sender after message correlation:** information about the credentials of the sending party involved in the conversation with the given receiving party, updated after message correlation.

  *Range of values:* You’ is a pair comprised of the possible sender identifier and its conversation identifier. Since information the party has about the sender id and its conversation id available before the message correlation might be incomplete, some of the missing knowledge can be gained by the receiving party from the information provided in the message sender field. For instance, if You = (⊥, ⊥) and From = (Ps, Cs), then You’ = (Ps, Cs) if all missing information is recorded. Note that some of the missing information may be forgotten, and the resulting value of You’ may be (Ps, ⊥), (⊥, Cs) or may even remain unchanged (⊥, ⊥). Table 5.8 illustrates possible values of the You’ field calculated based on the information available in the You field, the information provided in the From field and the possibility of not recording the provided information.

  *Default value:* (Pr, Cr).

  Visualization: the You’ label shown in Figure 212 substituted with a suitable value. An example specifying the default value for the message receiver field is shown in Figure 213.

Figure 213 illustrates the graphical notation used for the Message Correlation pattern variant where all configuration parameters are set to the default values.

Figure 213: Default notation: Message Correlation

**Meaningful configurations** Figure 214 illustrates the structure describing meaningful pattern configurations of Message Correlation. Applying the rules for traversing the structure, described on page 238, a total number of 100 configurations can be obtained. Figure 214 is constructed from the information presented in Table 5.8 describing 25 scenarios regarding possible information gained after message arrival. As entries in Table 5.8 do not consider the values which the message receiver field (i.e. To) might take. If we do consider these values (there are 4 of them), then the total number of meaningful combinations is achieved by multiplying 25 by 4.
Table 5.8: Enumeration of all scenarios regarding possible information gained

<table>
<thead>
<tr>
<th>From</th>
<th>You</th>
<th>You’</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ps, Cs)</td>
<td>(Ps, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(Ps, ⊥)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td>(Ps, ⊥)</td>
<td>(Ps, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(Ps, ⊥)</td>
<td>(Ps, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥)</td>
<td>(Ps, ⊥)</td>
</tr>
<tr>
<td>(⊥, Cs)</td>
<td>(Ps, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(Ps, ⊥)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, Cs)</td>
<td>(⊥, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥)</td>
<td>(⊥, ⊥)</td>
</tr>
<tr>
<td>(⊥, ⊥)</td>
<td>(Ps, Cs)</td>
<td>(Ps, Cs)</td>
</tr>
<tr>
<td></td>
<td>(Ps, ⊥)</td>
<td>(Ps, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, Cs)</td>
<td>(⊥, Cs)</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥)</td>
<td>(⊥, ⊥)</td>
</tr>
</tbody>
</table>

Figure 214: Meaningful configurations of Message Correlation (total 100 configurations)

**Illustrative example** To illustrate the example presented earlier, we identify values for the configuration parameters of the *Message Correlation* pattern configuration, and show the graphical notation for the given pattern configuration (cf. Figure 215). In the insurance claims handling example, when a claim is received, an insurance advisor determines whether
the client has a valid insurance policy and whether there are any records related to the claim received. Let $Ps$ be an identity of the client specified by the client, $Cs$ be the insurance number specified by the client for handling the claim, $Pr$ be the identity of the insurance advisor, and $Cr$ be a client number associated with the client used within the insurance company. It is assumed in this example that the client has a valid insurance policy. The client specifies both information about their identity, the insurance number in their claim, and the identity of the insurance company. Since no correlation-related information is gained by the insurance advisor from the incoming message, the knowledge about the client after handling the claim does not change.

![Figure 215: Notation for the claim handling example](image)

**CPN semantics** In this section, we formalize the semantics of Message Correlation by means of CPN. The CPN diagram presented in Figure 216 illustrates the act of correlating an incoming message of type `Message` with the history of existing conversations `Conv` via transition `Correlate`. Messages that have to be processed by the party are supplied in the form `(from,to,cont)`, where `from` (of type `From`) represents the identity of the message sender; `to` (of type `To`) represents the identity of the party to which the message is intended, and `cont` (of type `Cont`) represents the content of the message. The knowledge relating to the history of existing conversations available to the party before a received message has been correlated has the form `(me,you,cont_old)`. The element `me` (of type `Me`) represents the knowledge of the receiving party itself. The element `you` (of type `You`) represents the knowledge about the message sender from previous interactions. The element `cont_old` (of type `Content`) represents the content of all messages previously exchanged in the given conversation.

Values `NoPartyId` and `NoConvId` (which correspond to the symbol ⊥ introduced in the pattern visualization section above) denote the absence of information about the party identifier and conversation identifier respectively. Note that a union type is used to incorporate such “missing values”.

The `abletocorrelate()` function matches information supplied in the incoming message with the history of previous and current conversations. In particular, matches are performed between variables `to` and `me`, and `from` and `you`. In this example, the `abletocorrelate()` function performs correlation by matching the identities of the message sender and message receiver supplied in the message with the corresponding identities available to the party-receiver in the history of conversations.

Note that instead of correlating based on matching the information contained in the incoming message with local knowledge of the party, also some analysis of the message content can be performed (this issue is discussed in more detail in the paragraph related to issues). If correlation is successful, i.e. the `abletocorrelate()` function has identified
a conversation to which the message received can be uniquely correlated, then the history of existing conversations is updated. In particular, the \texttt{merge()} function adds the content \texttt{cont} of the message received to the old content \texttt{cont\_old} of the identified conversation. The \texttt{addmissing()} function identifies if there is any missing information regarding the credentials of the message sender, and where such information is identified, adds it to the existing information. Note that since not all information provided necessarily has to be recorded, the \texttt{some()} function defines how much of the missing information will be actually recorded.

In the net presented in Figure 216, messages which can not be correlated are blocked in the \texttt{Incoming messages} place. This net could be extended with mechanisms for handling correlation failure. In particular, constructs could be added for creating a new conversation or for discarding an incoming message for which no corresponding conversation has been found in the history of existing conversations.

In this implementation, we applied the \textit{ID Matching} CPN pattern (cf. page 49) and \textit{BSD Filter} CPN patterns (cf. page 43) in order to perform the message matching and filter out unintended messages respectively. The Table 5.10 indicates how frequently each of the patterns has been used and how many patterns in total, i.e. including the repeating ones, were applied.

\textbf{Issues} The \textit{Message Correlation} pattern family identifies the issues experienced by a party receiving a message that must be correlated. The given pattern family considers correlation

\begin{table}[h]
\centering
\caption{Data types used in Fig. 216}
\begin{tabular}{l}
\hline
\texttt{colset PartyId = union smallstr + NoPartyId};  \\
\texttt{colset ConvId = union smallint + NoConvId};  \\
\texttt{colset PxC = product PartyId*ConvId;}
\texttt{colset Content = string;}
\texttt{colset From = PxC;}
\texttt{colset To = PxC;}
\texttt{colset Me = PxC;}
\texttt{colset You = PxC;}
\texttt{colset Conv = product PxC*PxC*Content;}
\texttt{colset Message = product From*To*Content;}
\hline
\end{tabular}
\end{table}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure216.png}
\caption{CPN diagram: Message Correlation}
\end{figure}

\footnote{The id of a party is a small color-set of a certain type (STRING in this case) or is denoted by \texttt{NoPartyId} if not specified.}

\footnote{The id of a conversation is a small color-set of a certain type (INT in this case) or is denoted by \texttt{NoPartyId} if not specified.}
between two tightly-coupled parties. When fewer dependencies are desired between the parties, the tight binding between them may be relaxed. The issue of message mediation among two loosely-coupled parties is addressed in the *Message Mediation* pattern family (cf. page 264). Furthermore, the *Message Correlation* pattern family concentrates on a single interaction between two parties in the context of already existing conversation. Thus, it does not provide any insight into how information used for correlation changes during the course of the conversation. The issue of varying correlation information in the context of a bipartite conversation is addressed in the *Bipartite Conversation Correlation* pattern family (cf. page 281).

From a functional point of view, message correlation is performed by a party using either a *key-matching* or a *property-analysis* approach. A party may use key-matching to uniquely identify a conversation related to the message received. For this, message fields `From` and `To` are used as keys for correlation. The mapping of these keys to the values of `You` and `Me` should be unique, i.e. no two conversations can be identified as a result of applying key-matching. In situations, where key-matching results in a non unique mapping, some additional analysis of the message content might be needed. The property-analysis approach is used to identify a conversation related to the message received based on examination of message content. For analysis purposes, other information available to the party may also be used. Based on the content of the message (for instance, message id, conversation id, time-stamp, etc.) together with other information available to the party it may be possible to determine the conversation to which the message received relates. Note that in this section we only considered key matching.

The perfect scenario leading to successful correlation would contain the maximal possible information for all configuration parameters. However, in real-life situations not all information is guaranteed to be available. For instance, information specified by the message sender in the message may be incomplete or information available to the receiver about the sender may be missing. If no identifier for the receiving party is specified in the message, no guarantee can be given that the message will be delivered to the right party. If the message sender does not disclose its id in the message, there is no guarantee that the follow-up response will be correctly delivered. If no conversation id, used by the receiver for correlation purposes, is contained in the message, then there is the possibility of it being correlated with the wrong conversation.

An exceptional situation may occur if, as a result of correlation, no conversation related to the message received can be identified (i.e. in case of the correlation failure). To deal with such an exception, a party may *create* a new conversation or *discard* the message received.

The majority of existing products rely on the key-matching approach, i.e. they perform correlation based on matching of information supplied in the message with information available to the party. These products discard messages which do not provide complete information about the credentials of the message receiver, while the identity of the party or the conversation identifier could potentially be inferred based on analysis of available message content.

**Table 5.10:** CPN patterns used in the implementation of Message Correlation

<table>
<thead>
<tr>
<th>CPN Pattern</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Matching</td>
<td>1</td>
</tr>
<tr>
<td>BSD Filter</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>
There are a number of considerations when deciding if an interaction between two parties needs to be performed directly or can be facilitated via another intermediary party. Message mediation allows messages to be passed from one party to another without requiring these parties to engage in interactions with each other directly. Mediation is often used in situations where one party needs to be introduced to another party in order to establish the basis for direct follow-up interaction or when one party wants to utilize a service offered by another party without having their credentials revealed. Although in a tripartite conversation each of the parties performs message correlation as described on page 256, the message mediator is also responsible for correlating messages exchanged amongst the parties involved. In Section 5.2.4, we analyze the particularities of message correlation in tripartite conversations.

One could argue that message mediation can be used for determining whether a message can be delivered based on some conditional logic. In order to facilitate such behavior, one needs to apply the control-flow patterns (cf. Chapter 4) to describe the internal behavior of the mediator and to specify the various interactions with the parties involved using the graphical notation of tripartite interaction presented on page 266.

### 5.2.4 Pattern family: Message Mediation

In this sub-section, we describe the fourth pattern family called *Message Mediation*. This pattern family concentrates on the mediation of messages related to a conversation between two loosely-coupled parties.

**Description** A party requesting a service may not know the credentials of the party providing this service or may know the party credentials but not be willing to engage in a message exchange with them directly. To establish a conversation between two loosely-coupled parties, the customer and the provider, a third intermediary party named “the mediator” is required. The customer posts a request to the mediator and expects a response to be received back. The mediator forwards requests from the service customer to the service provider and replies from the provider back to the customer. Alternatively, the mediator may provide a party with details of the credentials of the other party involved in the conversation to allow for future interactions to occur directly between them.

**Examples**

- To order office equipment an employee contacts a secretary, who forwards the employee’s request to a supplier. After processing the request, the supplier delivers the equipment ordered directly to the employee.
- Air-miles card owners may exchange accrued air-miles for reservation of flights provided by one of the airline partners. The miles-to-flight exchange is performed via the Air-miles web-site, which mediates data exchange between the client and the airline operator. The itinerary details provided to the client contains contact details for the airline operator for any future enquiries regarding the booking.

**UML meta-model** Concepts specific to the *Message Mediation* pattern family are illustrated by means of the class diagram in Figure 217. A conversation between a customer and a provider involves a third party, i.e. a mediator (the three parties involved in a conversation are illustrated by the association relations between *Party* and *Conversation*). Each party involved in the conversation may send and receive messages (cf. the association relations between *Party* and *Message*). Messages exchanged can be either of type *Request*
or Reply, where each reply corresponds to one request, and one request may have multiple replies.

Visualization In tripartite conversations, the mediator may play two different roles: (1) forward both a request from the customer to the provider and the provider’s reply back to the customer; (2) route a request from a customer to a provider and supply the provider with a reference to the customer’s credentials (so that the provider can reply to the customer directly). The role of the mediator is one of the configuration parameters that must be set in order to distinguish between two types of message mediation: Mediated Interaction and Mediated Introduction, which are denoted using the notation presented in figures 218 and 221 respectively.

This notation is based on the graphical notation used for the Message Correlation pattern family. It is assumed that messages exchanged between parties involved in the tripartite conversation are of the format (From, To, Them, Expose, Content), where From represents the credentials of the message sender, To represents the credentials of the message receiver, Them identifies the third party involved in the given conversation, Expose indicates permission to reveal the credentials of the message sender to the third party in the conversation, and Content represents additional content of the message (i.e. payload). As the Content field carries no information related for message routing, we omit it in figures 218 and 221. Note that compared to the basic correlation scenario described in the previous sub-section, there are two additional fields: Them and Expose.

Both graphical notations contain a set of configuration parameters (i.e. parameters that have to be set to a specific value in order to configure a pattern variant), a set of static attributes (i.e. pattern attributes whose value is fixed for all pattern variants derived from the pattern configuration), and a set of dynamic attributes (i.e. pattern attributes whose value is derived from other pattern attributes once all configuration parameters characterizing a specific pattern variant have been set to a specific value).
Let \( P_1, P_2, \) and \( P_3 \) denote the party identifiers of the customer, the mediator, and the provider respectively. Let \( C_1, C_2, \) and \( C_3 \) denote the conversation identifiers used by the customer, the mediator and the provider for correlation purposes respectively. Let \( \perp \) denote the absence of either party or conversation identifier in the message.

Figure 218: Graphical notation: Mediated Interaction

For the sake of clarity, first we will describe the pattern configuration of the Mediated Introduction, followed by the description of the Mediated Interaction pattern configuration. The graphical notation of the Mediated Interaction pattern configuration is shown in Figure 218. This notation contains the following configuration parameters:

- **Customer request for specific provider**: denotes information revealed by the customer about the identity of provider, which the mediator has to forward at the customer’s request.
  
  *Range of values*: \( \text{Them1} \) is a pair comprising a party identifier and a conversation identifier, potentially representing the credentials of the provider supplied by the customer to the mediator. The customer may know the identity of provider or may not know who the potential provider will be. If the customer knows the identity of the provider, it may indicate it in the \( \text{Them1} \) field in the form of \((P_3, \perp)\). The mediator will forward the request to the specified party. If the customer does not know the provider’s identity or does not want to specify it, the value \((\perp, \perp)\) is assigned to the \( \text{Them1} \) field.

  *Default value*: \((\perp, \perp)\).

  *Visualization*: the \( \text{Them1} \) label substituted with a suitable value in Figure 218. An example specifying a default value is given in Figure 219.

- **Customer’s permission to expose its credentials**: denotes permission granted to the mediator by the customer to disclose its credentials to the provider.

  *Range of values*: \( \text{Expose1} \) is a Boolean variable, whose value can be \text{true} or \text{false} indicating that the customer allows the mediator to expose its identity to the provider or prohibits it to do so in order to remain anonymous respectively.
Default value: false (i.e. the customer does not give permission for its credentials to be revealed).

Visualization: the Expose1 label substituted with a suitable value in Figure 218. An example specifying a default value is given in Figure 219.

• Visibility of the provider’s credentials: denotes whether permission is granted to the mediator by the provider to disclose its credentials to the customer.
  Range of values: Expose3 is a Boolean variable, indicating whether the provider allows the mediator to expose its identity to the customer or prohibits it to do so in order to remain anonymous. The provider may expose its identity in order to allow the customer to interact directly with them in the future independently of the mediator.
  Default value: false (i.e. the permission to expose the credentials is not granted).
  Visualization: the Expose3 label substituted with a suitable value in Figure 218. An example specifying a default value is given in Figure 219.

• Provider’s knowledge about credentials of customer: denotes information revealed by the provider in the response message to the mediator about the credentials of the customer. This information may be used by the mediator to correlate response messages received from the provider.
  Range of values: Them3 is a pair comprising the customer identifier and the conversation identifier. The provider may specify none, some or all information received from the mediator about the identity of the customer retrieved from the Them2 field.
  So, possible values of Them3 are \(( P1, C1 )\), \(( P1, ⊥ )\), \(( ⊥ , C1 )\) or \(( ⊥ , ⊥ )\).
  Default value: \(( ⊥ , ⊥ )\).
  Visualization: the Them3 label substituted with a suitable value in Figure 218. An example specifying a default value is given in Figure 219.

• Information about the message sender in the response from provider to mediator: denotes credentials of the provider revealed by the provider in the response message sent to the mediator.
  Range of values: From3 is a pair comprising the provider identity and the conversation identifier. The provider may underspecify some information about its identity in the message sent to the mediator. If an underspecified identity is passed by the mediator to the customer, the customer may fail to start a direct interaction with the provider in the future. Thus, possible values of the From3 field are \(( P3, C3 )\), \(( P3, ⊥ )\), \(( ⊥ , P3 )\) or \(( ⊥ , ⊥ )\).
  Default value: \(( P3, C3 )\).
  Visualization: the From3 label substituted with a suitable value in Figure 218. An example specifying a default value is given in Figure 219.

• Information about the message sender in the response from mediator to customer: denotes the identity of mediator revealed in the response message sent by the mediator to the customer.
  Range of values: From4 is a pair comprising a party identifier and a conversation identifier. The mediator may underspecify some information about its credentials, therefore possible values of the From4 field are \(( P2, C2 )\), \(( P2, ⊥ )\), \(( ⊥ , C2 )\) and \(( ⊥ , ⊥ )\).
  Default value: \(( P2, C2 )\).
  Visualization: the From4 label substituted with a suitable value in Figure 218. An example of specifying a default value is given in Figure 219.

The static attributes of the Mediated Interaction pattern configuration are listed below:
- **From1**: information specified by the customer about its credentials in the request message to the mediator. It is assumed that the customer reveals all information about its credentials, therefore \( \text{From1} = (P1, C1) \).

- **To1**: information about the mediator’s credentials specified by the customer in the message receiver field of the request message sent to the mediator. Initially, the customer has no knowledge about the conversation identifier used by the mediator for correlation, therefore \( \text{To1} = (P2, \bot) \).

- **Me**: knowledge of the mediator about its party identifier and the associated conversation identifier: \( \text{Me} = (P2, C2) \).

- **From2**: information specified by the mediator about its credentials in the request message to the provider. It is assumed that the mediator reveals all information about its credentials in order for the response message to be delivered to the correct address, therefore \( \text{From2} = (P2, C2) \).

The dynamic attributes of the Mediated Interaction pattern configuration are listed below. These can be derived given the relevant configuration parameters.

- **Cust**: information supplied by the customer about its credentials in the request to the mediator. This information is retained knowledge held by the mediator for correlation purposes. The knowledge about the identity of the customer is gained by the mediator from the *From1* field: \( \text{Cust} = \text{From1} \).

- **Prov**: information supplied by the customer in the request sent to the mediator about the identity of the provider. This information is retained knowledge held by the mediator in order to forward the request received from the customer to the party with the indicated identity (if it has been provided). This knowledge is gained from the *Them1* field: \( \text{Prov} = \text{Them1} \).

- **To2**: information about the provider’s credentials specified by the mediator in the message receiver field of the request message forwarded to the provider. The identity of the provider is set to \( \text{Prov} \) if it has been provided by the customer, or is defined by the mediator based on its implicit knowledge and set to \( (P3, \bot) \).

- **Prov’**: credentials of the provider to whom the mediator has sent the request, recorded for the purpose of the future correlations. This information is derived from the *To2* field: \( \text{Prov’} = \text{To2} \).

- **Prov’’**: knowledge about the credentials of the provider available to the mediator after receiving the response message from the provider. The mediator extends information about the provider’s identity stored in the *Prov’* field with missing knowledge gained from the *From3* field of the response message received from the provider. Since the provider could underspecify some knowledge about its identity, possible values of \( \text{Prov’’} \) are \( (P3, C3) \) or \( (P3, \bot) \).

- **ExpC**: knowledge of the mediator about permission granted by the customer to disclose its credentials to the provider. It is assumed that the mediator hides the identities of parties involved in the conversation and discloses them only if the corresponding party has given permission to do so, i.e. \( \text{ExpC} = \text{Expose1} \).

- **Them2**: information specified by the mediator in the request message to the provider about the credentials of the customer. The knowledge retained in the *Cust* field is used to set the value of the *Them2* field only if the customer has granted permission to disclose its credentials, i.e. \( \text{Them2} = \text{Cust} = (P1, C1) \) if and only if \( \text{ExpC} = \text{true} \), otherwise \( \text{Them2} = (\bot, \bot) \).
- **ExpP**: knowledge of the mediator about permission granted by the provider to disclose its credentials to the customer. This knowledge is gained from the *Expose3* field: \( \text{ExpP} = \text{Expose3} \).

- **Them4**: information specified by the mediator in the response message to the customer about the credentials of the provider. The knowledge retained in the *Prov* field is used to set the value of the *Them4* field only if the provider has granted the permission to disclose its credentials, i.e. \( \text{Them4} = \text{Prov} \) if and only if \( \text{ExpP} = \text{true} \), otherwise \( \text{Them4} = (\perp, \perp) \).

- **To4**: information about the customer’s credentials specified by the mediator in the message receiver field of the reply message sent to the customer. The mediator retrieves the credentials of the customer from the *Cust* field: \( \text{To4} = \text{Cust} \).

Figure 219 illustrates the graphical notation of the Message Interaction pattern variant where all pattern attributes are set to their default value.

![Figure 219](image_url)

**Figure 219**: Default notation: Message Interaction

**Meaningful configurations** Figure 220 illustrates the structure for calculating the total number of meaningful configurations of Mediated Interaction. Applying the rules described on page 238, 320 distinct pattern configurations can be deduced from the structure presented in Figure 220.

Depending on the customer’s permission to expose its credentials, different values of the customer’s credentials can be exposed by the provider to the mediator. If no permission has been granted by the customer to expose its credentials, the provider has no knowledge about the customer’s credentials and thus may not reveal them in the response to the mediator. Otherwise, the provider may reveal complete, partial or no information about
the customer’s credentials. The rest of the configuration parameters are independent from each other, and their values may be combined without restriction.

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Configuration Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer request for specific provider</td>
<td>(P3,⊥), (⊥,⊥)</td>
</tr>
<tr>
<td>Customer’s permission to expose its credentials</td>
<td>(false)</td>
</tr>
<tr>
<td>Visibility of the provider’s credentials</td>
<td>(true, false)</td>
</tr>
<tr>
<td>Provider’s knowledge about credentials of customer</td>
<td>(true, false)</td>
</tr>
<tr>
<td>Information about the message sender in the response from provider to mediator</td>
<td>(P1,C1), (P1,⊥), (⊥,C1), (⊥,⊥)</td>
</tr>
<tr>
<td>Information about the message sender in the response from mediator to customer</td>
<td>(P3,C3), (P3,⊥), (⊥,C3), (⊥,⊥)</td>
</tr>
<tr>
<td>Information about the message sender in the response from mediator to customer</td>
<td>(P2,C2), (P2,⊥), (⊥,C2), (⊥,⊥)</td>
</tr>
</tbody>
</table>

Figure 220: Meaningful configurations of Mediated Interaction (total 320 configurations)

Now we move to the second type of message mediation, i.e. Message Introduction. The configuration parameters of the Mediated Introduction pattern configuration are described below:

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Configuration Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credentials of the message sender</td>
<td>Me Cust</td>
</tr>
<tr>
<td>Credentials of the message receiver</td>
<td>Prov</td>
</tr>
<tr>
<td>Credentials of the third party involved in the conversation</td>
<td>From1 To1 Them1</td>
</tr>
<tr>
<td></td>
<td>From2 To2 Them2</td>
</tr>
<tr>
<td></td>
<td>From5 To5 Them5</td>
</tr>
</tbody>
</table>

Figure 221: Graphical notation: Mediated Introduction

- **Customer request for specific provider**: denotes information revealed by the customer about the identity of the provider, which the mediator has to forward at the customer’s request.

  **Range of values**: Them1 is a pair comprising a party identifier and a conversation identifier, representing the credentials of the provider supplied by the customer to the mediator. The customer may know the identity of provider or may not know who the potential provider will be. If the customer knows the identity of the provider, it may indicate it in the Them1 field in the form (P3, ⊥). The mediator will forward the request to the specified party. If the customer does not know the provider’s identity or does not want to specify it, the value (⊥, ⊥) is assigned to the Them1 field.
It is assumed that the customer has no knowledge about the conversation identifier used by the provider, since there was no direct interaction between them in the context of the previous conversation. Therefore, the values \((P3, C3)\) and \((\bot, C3)\) are excluded.

**Default value:** \((\bot, \bot)\).

**Visualization:** theThem1 label substituted with a suitable value in Figures 221. An example specifying a default value is given in Figure 222.

- **Information about the message sender in the request from mediator to provider:** denotes information revealed by the mediator about its credentials in the request message forwarded to the provider.

  **Range of values:** information provided by the mediator about its credentials in the request to the provider may be underspecified. The mediator may reveal all, some or none of its identity to the provider, providing that the mediator passes a complete reference to the credentials of the customer (the latter is required for sending a response message directly back to the customer).

  So, possible values for the From2 field are \((P2, C2)\), \((P2, \bot)\), \((\bot, C2)\) and \((\bot, \bot)\).

  **Default value:** \((P2, C2)\).

  **Visualization:** the From2 label substituted with its value in Figure 221. An example specifying a default value is given in Figure 222.

- **Information about the message sender in the response from provider to customer:** denotes the credentials of the provider revealed by the provider in the response message sent to the customer.

  **Range of values:** From5 is a pair comprising the provider identity and the conversation identifier. The provider may underspecify some information about its identity in the message sent to the mediator. However, if an underspecified identity is passed to the customer, the customer may be unable to start a direct interaction with the provider in the future. Thus, possible values of the From5 field are \((P3, C3)\), \((P3, \bot)\), \((\bot, P3)\) or \((\bot, \bot)\).

  **Default value:** \((P3, C3)\).

  **Visualization:** the From5 label substituted with a suitable value in Figure 221. An example specifying a default value is given in Figure 222.

- **Information about mediator exposed by provider to customer:** denotes the credentials of the mediator revealed by the provider in the response message sent to the customer.

  **Range of values:** Them5 is a pair comprising the party identifier and the conversation identifier, whose value is based on the From2 field. Table 5.11 illustrates possible values of the Them5 field, containing all or part of the information from the From2 field.

  **Default value:** \((\bot, \bot)\).

  **Visualization:** the Them5 label substituted with a suitable value in Figure 221. An example specifying a default value is given in Figure 222.

The static attributes of the Mediated Introduction pattern configuration are listed below:

- **From1:** information specified by the customer about its credentials in the request message to the mediator. It is assumed that the customer reveals all information about its credentials, therefore From1 = \((P1, C1)\).

- **To1:** information about the mediator’s credentials specified by the customer in the message receiver field of the request message sent to the mediator. Initially, the
Table 5.11: Enumeration of all scenarios regarding revealed credentials of the mediator

<table>
<thead>
<tr>
<th>From2</th>
<th>Them5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (P_s, C_s) )</td>
<td>( (P_s, C_s) )</td>
</tr>
<tr>
<td>( (P_s, \bot) )</td>
<td>( (P_s, \bot) )</td>
</tr>
<tr>
<td>( (\bot, C_s) )</td>
<td>( (\bot, C_s) )</td>
</tr>
<tr>
<td>( (\bot, \bot) )</td>
<td>( (\bot, \bot) )</td>
</tr>
</tbody>
</table>

The dynamic attributes of the Mediated Introduction pattern configuration are listed below. Their values are derived from other pattern attributes.

- **Cust**: information supplied by the customer about its credentials in the request to the mediator. This information is retained knowledge held by the mediator for correlation purposes. The knowledge about the identity of the customer is gained by the mediator from the \( \text{From1} \) field: \( \text{Cust} = \text{From1} \).

- **Prov**: information supplied by the customer in the request sent to the mediator about the identity of the provider. This information is retained knowledge held by the mediator in order to forward the request received from the customer to the party with the indicated identity (if it has been provided). This knowledge is gained from the \( \text{Them1} \) field: \( \text{Prov} = \text{Them1} \).

- **To2**: information about the provider’s credentials specified by the mediator in the message receiver field of the request message forwarded to the provider. The identity of the provider is set to \( \text{Prov} \) if it has been provided by the customer, or is defined by the mediator based on its implicit knowledge and set to \( (P_3, \bot) \).

- **Them2**: information specified by the mediator in the request message to the provider about the credentials of the customer. This information will be used by the provider when sending the response to the customer. The knowledge retained in the \( \text{Cust} \) field is used to set the value of the \( \text{Them2} \) field, i.e. \( \text{Them2} = \text{Cust} \).

- **To5**: information about the customer’s credentials specified by the provider in the message receiver field of the reply message sent to the customer. The provider retrieves the credentials of the customer from the \( \text{Them2} \) field: \( \text{To5} = \text{Them2} \).

Figure 222 illustrates the graphical notation of the Message Introduction pattern variant where all pattern attributes are set to their default value.

**Meaningful configurations**

Figure 223 illustrates the structure for calculating the total number of meaningful configurations of Mediated Introduction. Applying the rules described on page 238, 72 distinct pattern configurations can be deduced from the structure presented in Figure 223.
Customer does not provide the credentials of the Provider

Mediator reveals its identity: the party identifier P2 and the conversation id C2

Provider reveals the credentials of mediator including both the party id P2 and the conversation id C2

**Figure 222:** Default notation: Message Introduction

Depending on the information exposed by the mediator to the provider about the mediator’s credentials, different values of the mediator’s credentials can be exposed by the provider to the customer. If the mediator specified both the party identifier and the conversation identifier, the provider may reveal both, some or none of them. Otherwise, the provider reveals no information about the mediator’s credentials at all. The rest of the configuration parameters are independent of each other, and their values may be combined without restriction.

**Figure 223:** Meaningful configurations of Mediated Introduction (total 72 configurations)

**Illustrative example** To show how the pattern configuration can be used in practice, we analyze one of the examples presented earlier and define the corresponding pattern variant. In the air-miles exchange example, the air-miles card owner sends a request to the Air-miles web-site to exchange accrued air-miles for the reservation of a specific flight. The air-miles web-site serves as a mediator between the client and the airline operator (cf. Figure 224). When forwarding the request from the client to the selected airline operator, the mediator reveals the client’s credentials (these are needed to book a flight). The airline operator sends the details of the reservation back to the mediator, who in its turn forwards them to the client. Together with the details of the reservation made, the mediator specifies the contact address of the airline operator in order to allow the customer to contact the airline company directly in the future for any outstanding issues. In this example, the
Mediated Interaction is performed by the Air-miles web-site. The label \texttt{Expose1} is set to \texttt{true}, since the client's details have to be communicated to the airline operator. The \texttt{Them1} label contains the identity of the airline operator selected by the client. The \texttt{Expose3} label is set to \texttt{true}, since the airline operator discloses its credentials to the client in order to allow for future interactions. The \texttt{From3} label contains complete information about the credentials of the airline operator and the \texttt{From4} label also contains complete information about the Air-miles organization.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{air-miles-diagram.png}
\caption{Notation for the air-miles exchange example}
\end{figure}

\textbf{CPN semantics} In this section, we describe the semantics of both Message Mediation types in the form of CPN models. The Mediated Interaction scenario is represented in Figure 225. The three parties involved in a conversation, i.e. customer, mediator, and provider, are represented as substitution transitions that are decomposed in Figures 226, 227 and 228 respectively. The interactions between these parties are realized using the \textit{Asynchronous Transfer} CPN pattern (described on page 54).

The behavior of the customer is shown in Figure 226. The customer communicates directly with the mediator by sending a request message of type \texttt{CustRequest} to and receiving a response message of type \texttt{MedReply} from the mediator via transitions \texttt{Send request} and \texttt{Receive response} respectively. The place \texttt{Mediator identity} stores information about the identity of mediators to one of which a request message will be sent by the customer. Initially, the customer has no knowledge of the conversation identifier used by the mediator for correlation purposes. The customer may know the identity of the provider with whom it wishes to communicate directly. The customer may either provide the mediator with the identity of the provider or may leave it out. The knowledge about the identity of the provider is enclosed in the \texttt{Them1} field of the request message. The value of this configuration parameter is set by the non-deterministic \texttt{defthem1()} function that either specifies the credentials of the provider or leaves this field empty. In the request message the customer explicitly specifies whether it allows the mediator to disclose its credentials to the provider or not by setting the \texttt{Expose1} variable of Boolean type to \texttt{true} or \texttt{false} respectively.
Section 5.2 Configurable framework for service interaction patterns

Figure 225: CPN diagram: The top view of Mediated Interaction

Table 5.12: Data types used in the Mediated Interaction diagrams

<table>
<thead>
<tr>
<th>Colset Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colset PartyId = union smallstr + NoPartyId;</td>
</tr>
<tr>
<td>colset ConvId = union smallint + NoConvId;</td>
</tr>
<tr>
<td>colset PxC = product PartyId * ConvId;</td>
</tr>
<tr>
<td>colset Content = STRING;</td>
</tr>
<tr>
<td>colset From = PxC;</td>
</tr>
<tr>
<td>colset To = PxC;</td>
</tr>
<tr>
<td>colset Them = PxC;</td>
</tr>
<tr>
<td>colset Expose = BOOL;</td>
</tr>
<tr>
<td>colset ExpC = Expose;</td>
</tr>
<tr>
<td>colset ExpP = Expose;</td>
</tr>
<tr>
<td>colset Me = PxC;</td>
</tr>
<tr>
<td>colset You = PxC;</td>
</tr>
<tr>
<td>colset Cust = PxC;</td>
</tr>
<tr>
<td>colset Prov = PxC;</td>
</tr>
<tr>
<td>colset Conv = product Me * PxC * PxC * Content;</td>
</tr>
<tr>
<td>colset ConvM = product Me * Cust * Prov * ExpC * ExpP * Content; 24</td>
</tr>
<tr>
<td>colset CustRequest = product From * To * Them * Expose * Content;</td>
</tr>
<tr>
<td>colset MedRequest = product From * To * Them * Content;</td>
</tr>
<tr>
<td>colset MedReply = product From * To * Them * Content;</td>
</tr>
<tr>
<td>colset ProvReply = product From * To* Them * Expose * Content;</td>
</tr>
</tbody>
</table>

24 ConvM denotes information recorded by mediator about conversations with customer and provider. While customer and provider communicate directly only with one party and store conversations of type Conv, the mediator has to keep track of interaction with both parties at once.
The **History** place keeps track of all messages sent by the customer to the mediator, including the credentials of all parties involved in the conversation. In order to distinguish conversations, each of them is assigned an identifier, generated by incrementing the value of a counter stored in the **New Conv** place. To realize the id generator we used the **ID Manager** CPN pattern describe on page 51. Response messages sent by the mediator to response are received by the customer via the **Receive response** transition. A function `abletocorrelate()` in the transition guard checks whether the response message received can be correlated with any of the requests sent. In this example, correlation is performed based on matching the message receiver field with the identity of the customer. If the result of the correlation is successful, the missing information gained from the response message is recorded in the **Updated history** place by means of the `addmissing()` function, otherwise the message is ignored. The matching operation corresponds to the **ID Matching** CPN pattern (cf. page 49), and blocking of unintended messages corresponds to the **BSD Filter** CPN patterns (cf. page 43). Information contained in the **them4** field of the response message can be used by the customer to send a follow-up request directly to provider.

The behavior of the mediator is shown in Figure 227. Requests sent by the customer to the **Init request** place are received by the mediator via the **Receive request** transition. For each of the requests received, the mediator creates a new conversation, whose identifier is provided by the **New Conv** counter place (realized using the **ID Manager** CPN pattern described on page 51). After receiving the request, the mediator adds a record about the request received to the **History of received requests** place. This record contains information about the credentials of the customer **from1**, information provided by the customer about the identity of the provider **them1**, and information about the visibility of the customer’s credentials to the provider **expose1**. When forwarding a request received from the customer, the mediator specifies the address of the message receiver, its own credentials and shows the credentials of the customer if the customer has granted the permission to disclose its identity. The latter is done by means of the `defthem()` function. The `defto2()` function determines the identity of the provider. If the customer has specified the identity
Section 5.2 Configurable framework for service interaction patterns

Variation point:
the Customer has either specified
the identity of the Provider or not

Variation point:
the Customer has allowed its
credentials to be revealed to
the Provider or not

Variation point:
the Mediator has either
specified all information about
its credentials or only part of it

Variation point:
the Mediator has either
allowed its credentials
to be exposed or not

Variation point:
the Data Distributor
CPN pattern (cf. page 65) to replicate the message sent to the provider by
transition Forward request. This information is recorded in the History of forwarded
requests place and used for correlating future replies.

When response messages of type ProvReply, stored in place Forward response, are
received from the provider, the mediator tries to correlate them with previous requests
using the abletocorrelate() function. If the related conversation has been identified,
the mediator records the missing information gained from the response message in the
History of responses place using the addmissing() function. The mediator forwards
the response message to the customer via the Send Reply transition. In this message, the
mediator may underspecify its credentials in the \texttt{from4} field using the function \texttt{some()}. Function \texttt{defthem()} is used to define the value of the \texttt{them4} field. In particular, if the provider has granted permission for its credentials to be revealed to the customer (i.e. variable \texttt{expP} has been set to \texttt{true}), the credentials of the provider stored in variable \texttt{prov} are assigned to the \texttt{them4} field.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cpn_diagram.png}
\caption{CPN diagram: The Provider page of Mediated Interaction}
\end{figure}

The behavior of the provider is shown in Figure 228. For each of the requests forwarded by the mediator, the provider creates a new conversation and records information related to the conversation in the \texttt{History} place. The provider sends a response message to the address specified by the mediator in the \texttt{from2} field. When sending a response message, the provider may decide to reveal all, some or none of its identity to the mediator. This information is defined by the function \texttt{some} and enclosed in the \texttt{from3} field of the response message. Moreover, the provider may decide to expose its credentials to the customer, and for this, the Boolean variable \texttt{expose3} must be set to \texttt{true}.

The main page of the Mediated Introduction pattern configuration is shown in Figure 229. The behavior of the customer is identical to the one presented in Figure 226. The behavior of both the mediator and provider differs slightly from the nets described in the Mediated Interaction scenario and is presented in figures 230 and 231 respectively.

In the Mediated Introduction scenario, messages of different types are used (cf. Table 5.13). None of the messages exchanged between parties in this scenario contains the \texttt{Expose} field, because it is assumed that the credentials of the parties can be freely revealed to other parties.

The substantial difference between the behavior of the mediator presented in Figure 230 and that of the mediator in the Mediated Interaction scenario, is that credentials of the customer recorded by the mediator in the \texttt{cust} variable are assigned to the \texttt{them2} field in the response message sent to the provider without examining the permission of the customer to expose its credentials. As has been mentioned already, this is done based on
Table 5.13: Data types used in Mediated Introduction diagrams

| colset CustRequest = product From * To * Them * Content; |
| colset MedRequest = product From * To * Them * Content; |
| colset MedReply = product From * To * Them * Content; |
| colset ProvReply = product From * To * Them * Content; |

Figure 229: CPN diagram: The top view of Mediated Introduction

the assumption that no permission is required for the mediator to expose the credentials of the customer to the provider.

Figure 230: CPN diagram: The Mediator page of Mediated Introduction

The behavior of the provider is shown in Figure 231. After receiving a response message from the customer, the provider sends the response message directly to the customer. As
there was no interaction between the customer and the provider in the past, and the customer does not know the provider, the provider uses the reference to the customer’s credentials provided by the mediator in the them2 field to specify the address of the message receiver to5 in its response message.

\[
\begin{align*}
\text{1.} \quad & \text{New Conv} \\
\text{2.} \quad & \text{Receive request} \\
\text{3.} \quad & \text{Send response} \\
\text{4.} \quad & \text{History} \\
\end{align*}
\]

**Figure 231**: CPN diagram: The Provider page of Mediated Introduction

When describing the behavior of the Mediated Introduction and Mediated Interaction in the form of CPN, we used 5 distinct CPN patterns. Table 5.14 indicates how frequently each of the patterns has been used and how many patterns in total, i.e. including the repeating ones, were applied in both models.

<table>
<thead>
<tr>
<th>CPN pattern</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID Matching</td>
<td>3</td>
</tr>
<tr>
<td>ID Manager</td>
<td>6</td>
</tr>
<tr>
<td>Data Distributor</td>
<td>4</td>
</tr>
<tr>
<td>BSD Filter</td>
<td>3</td>
</tr>
<tr>
<td>Asynchronous Transfer</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>23</strong></td>
</tr>
</tbody>
</table>

**Issues** Pattern variants belonging to the Message Mediation family address scenarios where a single message is exchanged between involved parties. If multiple sub-requests have to be sent to a set of mediators at once, or if a customer has to send requests to a set of mediators, who in turn might want to forward the request to a set of providers, then the given pattern variant can be combined with another suitable pattern variant belonging the Multi-Party Multi-message Request-Reply Conversation family. One of the roles of the mediator in the tripartite conversation is to pass a reference to the identity of one party to another in order to enable their direct interaction in the future. In the course of long-
running bipartite conversations, information used for correlation may change. This issue is addressed by the *Bipartite Conversation Correlation* pattern family.

In Section 5.2.3, we focused on the fundamentals of message correlation and described possible variants of successful correlation. When considering a series of related interactions, it is interesting to note that during a conversation the information used for correlation purposes may change. On page 283, we provide a graphical notation that is able to capture the dynamic character of message correlation in the bilateral context.

### 5.2.5 Pattern family: Bipartite Conversation Correlation

In this sub-section, we describe the fifth and final pattern family, named *Bipartite Conversation Correlation*, which addresses the issue of varying correlation information in the context of a long-running conversation between two parties, i.e. the pattern family is concerned about how earlier interactions can be exploited to set up a correlation agreement.

**Description** Each of the two parties involved in a long-running conversation may indicate during the initiation of the conversation what correlation token it expects to receive in follow-up messages in order to unambiguously relate them to the existing conversation. Independently from the correlation information provided by one party during conversation initiation, the other party may forget or choose to ignore it in the follow-up interactions.

**Example**

- A client requests a telephone subscription from the telephone company. The company registers the customer and confirms the registration by sending a letter with a telephone number and the client’s name to the address specified by the client. The company expects a client to indicate the telephone number in any future enquiries. However, it is likely that when making future enquiries, the client will forget to do so or will send incomplete information.

**UML meta-model** Concepts specific to the *Bilateral Conversation Correlation* pattern configuration are illustrated by means of the class diagram in Figure 232. Two parties, a requestor and a responder, are involved in a bipartite conversation (cf. the responder and requestor association relations between Party and Conversation). Each of the parties involved in a conversation may send and receive messages (cf. associations sends and receives between Party and Message). Messages can either be of type Request or Reply, where every reply corresponds to precisely one request and multiple replies can be sent for the same request. A conversation consists of two phases: initiation and follow-up interactions (represented as specializations Initiation and Follow-up). The requestor initiates a conversation by sending a message to the other party. The requestor specifies its correlation credentials (i.e. a party identifier and a conversation identifier) in the initial message sent to the other party and expects the responding party to use the same credentials in any response messages, however the responder may choose to ignore this information in the follow-up interactions.

**Visualization** Figure 233 illustrates the graphical notation used for illustrating Bipartite Conversation Correlation. Parties are visualized as rectangles. Arrows between rectangles indicate interactions between parties, where direction of arrows corresponds to the message flow. Interactions marked as initial request and initial response correspond to the initiation of conversation, where parties notify each other about the credentials they want to use as tokens for correlation in the follow-up interactions. It is an assumption
of this pattern, that messages exchanged have the form \((\text{From}, \text{To}, \text{Content})\), specifying information about the message sender, message receiver, and the content of the message. Labels, marked as \text{From} and \text{To}, denote information about the message sender and the message receiver respectively.

Let \(P_1\) and \(P_2\) denote the party identifiers of the requestor and responder respectively. Let \(C_1\) and \(C_2\) denote conversation identifiers used by the requestor and the responder for correlation purposes respectively. Let \(\perp\) denote the absence of either party or conversation identifier in a message.

This graphical notation in Figure 233 contains a set of static attributes, dynamic attributes and configuration parameters. The static attributes refer to information that is fixed in every pattern variant. The dynamic attributes derive their value from other pattern attributes. The configuration parameters refers to information that varies in all pattern variants, i.e. the configuration parameters have to be set to a value from the defined range in order for a specific pattern variant to be configured. The list of the static attributes is shown below:

- **Requestor credentials in the initial request**: information specified by the requestor about its identity in the initial conversation request. The requestor expects the responder to use the specified credentials when sending a response message back. \(\text{From}_1\) is a pair comprising the requestor identifier and the conversation identifier used by the requestor for correlation purposes. It is an assumption, that the requestor always specifies the maximal possible information about its identity when initiating a conversation with another party: \(\text{From}_1=(P_1,C_1)\).

- **Responder credentials in the initial request**: information specified by the requestor about the credentials of the responder in the initial conversation request. \(\text{To}_1\) is a pair comprising the responder identifier and the conversation identifier used by the responder for correlation purposes. It is assumed, that initially the requestor does
not have any knowledge about the conversation identifier the responder will use for correlation: \( \text{To1} = (P_2, \perp) \).

The graphical notation in Figure 233 contains only one dynamic pattern attribute described below:

- **Requestor credentials in the initial response**: information specified by the responder about the identity of the requestor in the response to the initiation request. \( \text{To2} \) is a pair comprising the requestor identifier and the conversation identifier used by the requestor for correlation purposes, such that \( \text{To2} = \text{From1} \).
  
  **Fixed value**: \( (P_1, C_1) \).
  
  **Visualization**: label \( \text{To2} \) in Figure 233 substituted with its value.

The list of configuration parameters is given below.

- **Responder credentials in the initial response**: information specified by the responder about its identity in the response to the initial request received from the requestor.
  
  **Range of values**: \( \text{From2} \) is a pair containing the responder identifier and the conversation identifier used by the responder for correlation purposes. The requestor identifier is denoted as \( P_2 \) and its conversation identifier is denoted as \( C_2 \). It is assumed, that the requestor may forget to specify or choose not to disclose some of the information about its credentials. The absence of this information is denoted by \( \perp \). So, possible values of the \( \text{From2} \) field are \( (P_2, C_2) \), \( (P_2, \perp) \), \( (\perp, C_2) \) and \( (\perp, \perp) \).
  
  **Default value**: \( (P_2, C_2) \).
  
  **Visualization**: the \( \text{From2} \) label in Figure 233 substituted with a suitable value. An example specifying a default value is shown in Figure 234.

- **Responder credentials in the follow-up request**: information specified by the requestor about the identity of the responder in follow-up requests.
  
  **Range of values**: \( \text{To3} \) is a pair containing the possible responder identifier and the conversation identifier used by the responder for correlation purposes. Although the requestor has received information about the responder credentials in the initiation response, it may decide to use all of the information provided, some of it or no information at all. Where the requestor decides to use the credentials specified by
the responder in the FROM2 field, To3 = From2. If the requestor decides to specify only part of the information retrieved from the FROM2 field, the responder’s party identifier, the conversation identifier or both may be omitted. Thus, if From2=(P2,C2), then possible values of the To3 field are (P2,C2), (P2,⊥), (⊥,C2) and (⊥,⊥).

**Default value:** From2.

**Visualization:** the To3 label in Figure 233 substituted with a suitable value. An example specifying a default value is shown in Figure 234.

- **Requestor credentials in the follow-up request:** information specified by the requestor about its identity in the follow-up requests sent to the requestor.

  **Range of values:** From3 is a pair containing the responder identifier and the conversation identifier used by the responder for correlation purposes. Depending on what information the requestor specifies about the identity of the responder in the follow-up request, i.e. whether it used all credentials specified by the responder or only part of them, the requestor may specify either complete information about its identity as indicated in the From1 field, or only part of it. So, possible values of the From3-field are: (P1,C1), (P1,⊥), (⊥,C1) and (⊥,⊥). Note that the latter three values have the meaning only if the complete responder’s credentials have been specified, otherwise correlation performed by the responder may lead to ambiguous results.

  **Default value:** (P1,C1).

  **Visualization:** the From3 label in Figure 233 substituted with a suitable value. An example specifying a default value is shown in Figure 234.

Figure 234 presents the graphical notation for one of the bipartite conversation pattern variants where all configuration parameters are set to their default values. The requestor and the responder retrieve all information from the messages received, and specify complete information about the identity of the message sender and the message receiver.

**Figure 234:** Default notation: Bipartite Conversation Correlation

**Meaningful configurations** Figure 235 illustrates the structure for calculating the total number of meaningful configurations of Bipartite Conversation Correlation. Applying the rules described on page 238, 18 distinct pattern configurations can be deduced from the structure presented in Figure 235.
Depending on what information the responder has provided in the initiation response to the requestor, the requestor may specify the same information about the responder’s credentials in the follow-up request to realize the scenario where the responder defines the correlation information for future interactions. Alternatively, the requestor may underspecify part of the responders credentials and reveal as much information about own identity as desired. The only requirement for specifying information in the follow-up request is that the responder’s identity has to be explicitly specified. If in addition to the responder’s identifier also its conversation identifier is also available to the requestor, and these are explicitly specified in the follow-up request, the requestor may omit or underspecify the information about its identity.

![Image](image_url)

**Figure 235:** Meaningful configurations of Bipartite Conversation Correlation (total 18 configurations)

**Illustrative example** For the telephone subscription example described earlier we define the pattern configuration depicted in Figure 236. In this example, a client requests a telephone subscription from the telephone company. The company registers the customer and confirms the registration by sending a letter with a telephone number and the client’s name to the address specified by the client. The company expects a client to indicate the telephone number in any future enquiries. However, when making the next enquiry, the client forgets to specify their telephone number. In the graphical notation, the client and the telephone company map directly to the roles of the responder and the provider. The requestor omits the conversation id specified by the responder (i.e. the telephone number) in the follow-up request. Note that the requestor assumes the responder to have used C1 as an identifier.

![Image](image_url)

**Figure 236:** Notation for the telephone subscription example

**CPN semantics** A CPN diagram illustrating the semantics of the Bipartite Conversation Correlation is shown in Figure 237. Two parties, a requestor and a responder, represented as substitution transitions *Requestor* and *Responder*, are involved in a conversation. Interactions between the parties are realized using the *Asynchronous Transfer* CPN pattern.
The responder initiates a conversation by sending an initiation request `Init request` of type `Message`. In this initiation request, the requestor specifies the credentials it wants the responder to use when responding to this request. The responder sends an initiation response `Init response` of type `Message`, where it specifies its own credentials that must be used by the requestor in the follow-up requests. Messages exchanged between the parties take the form `(From,To,Content)`, where `From` is credentials of the message sender, `To` is credentials of the message receiver and `Content` is the content of the message.

The decomposition of transitions `Requestor` and `Responder` is shown in Figure 238 and Figure 239 respectively. The CPN models for bipartite conversation correlation are based on the correlation mechanisms and data types introduced for the Message Correlation pattern family (cf. Section 5.2.3).

The behavior of the requestor is depicted in Figure 238. Transitions `Send init request`, `Receive init request` and `Send follow-up request` correspond to the send conversation initiation request, receive initiation reply, and send follow-up request respectively. To initiate a new conversation, a new unique conversation id is created by the `new()` function, which increments the identifier of the last conversation. When sending out an initialization request, the requestor specifies its identity `(P1,C1)`, where P1 is a party identifier of type `PartyId` and C1 is a conversation identifier of type `ConvId`. In the credentials of the message receiver in the initiation request, the requestor only specifies the id of the responder P2. The absence of the conversation identifier is denoted by means of the value `NoConvId`. After the message has been sent, it is recorded in the `History`.

<table>
<thead>
<tr>
<th>Table 5.15: Data types used in Figs. 237-239</th>
</tr>
</thead>
<tbody>
<tr>
<td>colset PartyId = union smallstring + NoPartyId; 25</td>
</tr>
<tr>
<td>colset ConvId = union smallint + NoConvId; 26</td>
</tr>
<tr>
<td>colset PxC = product PartyId * ConvId;</td>
</tr>
<tr>
<td>colset Content = string;</td>
</tr>
<tr>
<td>colset From = PxC;</td>
</tr>
<tr>
<td>colset To = PxC;</td>
</tr>
<tr>
<td>colset Me = PxC;</td>
</tr>
<tr>
<td>colset You = PxC;</td>
</tr>
<tr>
<td>colset Conv = product Me<em>You</em>Content;</td>
</tr>
<tr>
<td>colset Message = product From<em>To</em>Content;</td>
</tr>
</tbody>
</table>

25The id of a party is a small color-set of type STRING or is denoted by NoPartyId if not specified.
26The id of a conversation is a small color-set of type INT or is denoted by NoPartyId if not specified.
function of type $\text{Conv}$. This place stores the history of all current conversations for use in correlating response messages from the responder.

![CPN diagram: The Requestor sub-page of Bipartite Conversation Correlation](image)

**Figure 238:** CPN diagram: The Requestor sub-page of Bipartite Conversation Correlation

When an initialization response arrives, the Receive init response transition checks by means of the $\text{abletocorrelate()}$ function whether the message received can be correlated with one of the conversations recorded in the place History. The correlation mechanism is based on the model presented for the Message Correlation pattern family. If the message received can be correlated, the history of conversations is updated. The $\text{addmissing()}$ function records information supplied by the responder about its identity which was not known to the requestor. Since the requestor may forget to retrieve some information, the $\text{some()}$ function defines how much information will be included in each case. By means of this function, it is possible to specify that either the party identifier, the conversation identifier or both are forgotten.

In the follow-up request, the requestor may use all or just part of the information stored in the Updated history place to specify the credentials of the message sender and the message receiver. The $\text{some}()$ function determines how much information will be underspecified in the From and To fields of the follow-up response.

The behavior of the responder is depicted in Figure 239. A conversation initiation request sent by the requestor in place Init request is consumed by transition Receive init request, which creates a new conversation identifier via the function $\text{new()}$. The
created conversation identifier will be used by the responder in the future interactions with the requestor for correlation purposes. Information contained in the initialization request received is associated with the newly created conversation identifier, and all of this information is recorded in the History place. When sending the initialization response, the responder may underspecify its identity in the from2-field by applying the some() function. After the response message has been sent, the conversation history is updated and recorded in the Updated History place. The responder expects the requestor to use the information previously provided by the responder about its identity in follow-up requests. Follow-up requests sent by the requestor to place Follow-up request are checked by transition Receive follow-up request and consumed only if the function abletocorrelate() has identified a conversation with which the message received can be correlated. In this net, all messages for which no matching conversation can be found are ignored. Note that a mechanism for handling correlation failure could be added to this model, in order to specify whether such messages have to be discarded, or whether a new conversation has to be created to process them.

Note that in this implementation we used the ID Manager CPN pattern (cf. page 51) to ensure the uniqueness of identifiers associated with conversations; the ID Matching CPN pattern (cf. page 49) and the BSD Filter CPN pattern (cf. page 43) for message matching during message correlation, and the Data Distributor CPN pattern (cf. page 65) to record the information about messages sent that is required for future correlations. Table 5.16 indicates how frequently each of the patterns has been used and how many patterns in total, i.e. including the repeating ones, were applied.

**Issues** In long-running conversations, the information used for correlation purposes may
change. If the parties involved in the conversation underspecify information about the identity of a requestor or a responder, the correlation might fail. In this pattern family, the parties involved in the conversation are assumed to already know each other. However, in some situations the identity of a responding party is not known to the requestor directly. The conversation in such a setting involves an intermediary, who either forwards requests and replies between the involved parties and hides the identity of the parties involved from each other, or brings the parties in contact by providing a reference to their identity. These scenarios are described in the Message Mediation pattern family.

### 5.2.6 Pattern-based service interaction design method

In this section, we present a pattern-based design method which describes how the configurable framework presented in Section 5.2 can be used for classifying service-interaction scenarios. This method identifies which pattern families need to be applied, the order in which they should be applied, and what the rationale is for using a specific pattern family. We illustrate this design method in action using a travel agency example.

In order to classify a particular service interaction scenario, one has to perform the steps listed below in the order specified:

1. Use the **Multi-party Multi-message Request-Reply Conversation** pattern family, as detailed on page 233, to describe the overall situation, i.e. model all parties involved and their high-level interactions. The parameters defined for this pattern family have to be configured in the order they are listed, and for each of them a specific value from the defined range has to be determined.

2. Decide whether information related to individual conversations needs to be correlated. This is especially important if interacting parties expect to receive messages related to different conversations, or originating from different parties. Conversations can be classified as short-running or long-running, depending on whether they consist of a single request-response interaction or multiple interactions respectively, and as bilateral or mediated, depending on whether the interaction between the parties involved is performed directly or via a third party-mediator respectively. The following types of interactions are possible:

   - **Short-running bilateral conversation**: to support this style of interaction use the Message Correlation pattern family, as detailed on page 256, to specify what information in regard to both party’s credentials is revealed in a message, and

---

27 Note that this only applies to conversations involving correlation, i.e. in the same application scenario multiple types may be identified.
how the message is correlated by (each of) the message receivers. The correlation information obtained helps to differentiate between messages belonging to different conversations, (and potentially involving multiple parties,) thus helping to relate replies received with previously sent requests.

- **Long-running bilateral conversation:** to support this style of interaction use the Bilateral Conversation Correlation pattern family, as detailed on page 281, to specify what information must be specified in messages and used by the parties to ensure that each message is delivered to the right destination. This gives further insight into how information used for correlation purposes may change during the course of the conversation.

Parties may be engaged in multiple conversations, therefore, it is important to specify how messages are being correlated for each conversation. In order to describe precisely how each party performs message correlation, and what information they gain from the messages received, use the Message Correlation pattern family, as detailed on page 256.

- **Short-running mediated conversation:** to support this style of interaction use the Message Mediation pattern family, as detailed on page 264, to describe what information about each party’s credentials is revealed in the messages exchanged. The role of the mediator in the conversation is determined as follows:
  
  a) **introduction:** the message mediator brings several parties in touch by revealing the credentials of one party to another and thereafter lets the parties interact directly: in this situation use the Message Introduction type of the Message Mediation pattern family to describe the interaction between these parties.

  b) **interaction:** the message mediator forwards both requests and replies between two parties, without revealing their identities: in this situation, use the Message Interaction type of the Message Mediation pattern family to describe the interaction between these parties.

- **Long-running mediated conversation:** this conversation represents either a series of short-running mediated conversations, or the combination of a short-running mediated conversation with a short-running or long-running bilateral conversation. To describe the long-running mediated conversation, each of its composite parts have to be described separately as indicated earlier.

3 For each of the conversations, determine whether the conversation is associated with the distribution of goods or services by one party to another under specific subscription terms. If the concept of subscription is relevant, use the Renewable Subscription pattern family, as detailed on page 245, to describe who is the initiator of the subscription, what the terms of subscription are, and how the subscription will be renewed.

4 The complete description of an interactive process is obtained when both the internal behavior of a party and the interactions this party is involved in are fully defined. For modeling of the control-flow in each party use the control-flow patterns described in Chapter 4.

To illustrate how this design method can be used in practice, let’s analyze a travel-agency example. Imagine that one travel agency has bought another travel agency, and it
needs to integrate the systems of both agencies in an expeditious manner. To make such an integration possible, it is necessary to identify requirements the systems to be integrated have to support and what they are actually supporting. By applying the pattern-based service-interaction design method, the agency would be able to define what aspects need to be modeled and which pattern family needs to be selected for describing the characteristics identified.

As an example, let’s consider a scenario where a client sends a request to the travel agency. The request triggers the booking of a car, a hotel, and an airplane journey. The client expects at least two offers, of which only the best and most affordable one will be selected. The travel agent seeks to provide offers within a limited period of time. To achieve this, it sends requests for booking a hotel, a car, and an airline journey to multiple service partners. Responses are expected from two car-lease agencies, three hotels and five airline operators, who are official partners the travel agency is constantly dealing with.

In order to classify this scenario, we start with applying the *Multi-party Multi-message Request-reply Conversation* pattern family (Step 1). In this step, we define the high-level interaction between the parties involved. In doing so, we use the graphical notation presented in Figure 240 to describe the conversation between the client and the travel agency from the perspective of the client, and the conversations between the travel agency and car lease, hotel rental and airline partners from the perspective of the travel agency.

![Figure 240: Travel agency: multi-party view (Step 1)](image)

We define the number of parties involved in the conversation, i.e. more than two. Since the client places multiple sub-requests to the travel agency, these are represented with multiple circles (N=3). There is no possibility of non-responding parties and missing replies since the conversation takes place at the travel agency directly. The client accepts offers
in order of their arrival, however only starts processing them when two offers have been received. This condition is represented as a Boolean expression \( B \) that must be satisfied to enable the client to start processing the information. When the enabling condition is satisfied, the client processes all offers supplied (the consumption index \( C \) is set to \( \text{All} \)), and only one of the offers is selected (the utilization index \( U \) is set to 1). The offers are processed directly and only once, therefore the consumption frequency \( F \) is set to 1.

In order to delineate conversations between the travel agency and the car lease, hotel booking and airline partners, these are depicted separately. Although the order in which conversations with each of these partner groups may be started may be different, in this example we assume it to be fixed as represented in Figure 240. A single request for booking a car, a hotel, and an airplane is sent to a group of car lease partners, a group of hotel partners, and a group of airline operators respectively. The number of partners in each of the groups is represented via parameter \( M \). This is the only parameter that distinguishes these conversations. For the rest, all configuration parameters are set to the same value. In particular, there is the possibility that some partners will not reply and that some replies will be missing. Replies from the partners are accepted by the travel agency in order of arrival. Since the waiting period is limited, the enabling condition is represented by the timeout. Upon expiration all replies received by the travel agency are consumed \((C = \text{All})\). The number of offers utilized by the travel agency, visualized as \( UN \), represents a subset of offers received. After offers from all partner groups have been received, these are communicated to the client.

Having described the high-level interactions between the parties identified, we define the details relating to the correlation of messages that are exchanged during each of the conversations (Step 2). The interaction we consider consists of two parts: first, mediated interaction between the client and the booking partners via the travel agency, and second, direct interaction between the client and a booking partner. Thus, the conversation we are considering can be classified as a long-running mediated conversation that consists of a short-running mediated interaction followed by a short-running bilateral interaction.

Let’s describe the interaction step-by-step, starting with the short-running mediated interaction. The travel agency plays the role of the message mediator through which the message interaction between the client and the booking partners is performed. We depict a specific interaction between the client, the travel agency and a booking partner, and define the configuration of the Message Mediation pattern family as depicted in Figure 241.

The client provides the travel agency with its credentials, which are openly communicated to the booking partner (i.e. parameter \( \text{Expose1} \) is set to true). The booking partner is selected by the travel agency, since the client did not request a particular partner (i.e. the \( \text{Them1} \) field is empty). The travel agency forwards the credentials of the client to the booking partner, who in response sends a reply to the travel agency. The booking partner specifies its credentials in the \( \text{From3} \) field, which are revealed to the client by the travel agency.

Having described the short-running mediated interaction, we proceed with bilateral interaction between the client and a booking partner. Since in the follow-up interaction the credentials provided by the booking partners during mediated interaction are used, the conversation is classified as long-running, therefore we apply the Bipartite Conversation Correlation pattern family. We depict this information by augmenting Figure 241 appropriately.

The reference to the credentials of the booking partner are used by the client in the follow-up conversation, where the client directly contacts the booking partner without re-
Figure 241: Travel agency: message mediation and message correlation in the context of a long-running conversation (Step 2)

quiring mediation by the travel agency. In order to facilitate correlation, the client specifies the credentials of the booking partner in the To5 field, and also includes information about their own identity in the From5 field.

To obtain the complete picture regarding correlation, for each of the parties it is important to specify exactly how incoming messages are correlated with outgoing ones, i.e. either based on a certain property of the message (known as property-based analysis) or based on matching of party and conversation identifiers (known as key-matching). In this example, the key-matching approach is used for correlating replies with previously sent requests. Furthermore, the mechanism for handling exceptional situations arising when an incoming message cannot be correlated has to be defined, i.e. either the message has to be discarded or a new conversation needs to be created. In this example, all messages that arrive from booking partners after the timeout expiration or any unintended messages that are received are ignored.

Step 3 does not apply, since the conversation considered is not related to subscriptions for a particular product.

The classification obtained provides insights into the external interactions of the travel agency with clients and booking partners. In order to complete the description, one has to describe the control-flow of each of the parties (Step 4). For this, the Workflow Control Flow patterns described in Chapter 4 can be used. Note that in Section 4.2.2 we already used the Travel Agency example and illustrated its operational semantics using the CPC modeling language.

Having illustrated how the design method presented can assist in selecting the pattern configurations for classifying complex service-interaction scenarios, we now move on to a detailed analysis of a selected PAIS from the perspective of service interaction.
5.3 Tool evaluations

This section presents the evaluation results obtained from a detailed analysis of the service-interaction patterns across selected PAISs. In particular, we consider Oracle BPEL PM as an example of a system which next to basic capabilities for modeling of business processes, also provides BPEL-based support for inter-process communication. Note that we do not consider CIG modeling languages, since their main goal is to automate support for decision-making based on medical guidelines, rather than to describe high-level interactions between business processes.

5.3.1 Evaluation of Oracle BPEL PM

Oracle BPEL PM is based on BPEL (whose concepts have been introduced in Section 4.3.1 on page 215), therefore it enables interactions across multiple organizations whose processes are deployed as Web services. BPEL is based on XML Schema [50], Simple Object Access Protocol (SOAP) [45], and Web Services Description Language (WSDL) [60]. In order to be accessible by other processes, a process that has been defined in the Oracle Process Designer, must be deployed to Oracle BPEL Server. Once deployed, a BPEL process is published as a Web service, and can be accessed through a client that uses WSDL interface definition of the given process and SOAP as a communication protocol. The role of the client may be performed by a user initiating the deployed process via the BPEL console, or by another process.

In order to send a message to a process, the client needs to know the custom data types defined in the XML schema of the target process, the message types and the port types declared in the WSDL definition of the process. Types of messages, sent and received by a process, are defined in terms of the data types declared in the associated XML schema. The portType element includes a supported set of operations, each describing the input and output messages for each operation. Thus, in order to send a message to a process, in fact an operation on the specific port type has to be called, and the type of the message sent must coincide with the type of the input message defined for the given port type.

BPEL defines the concept of a partner link, which represents a dependency between two services. A partner link specifies roles played by the services, and the port types supported by each of the roles. In order to represent an interaction of a process with another service, a valid partner link needs to be defined. In Oracle BPEL PM, in order to define the partner link, one has to look up the required service in a Universal Description, Discovery, and Integration (UDDI) browser. UDDI is a specification for maintaining a standardized Web-based distributed directory containing information about Web services, i.e. their capabilities, location and requirements, in a universally recognized format [163].

In Oracle BPEL PM, a web service can be invoked as a synchronous or asynchronous operation. A synchronous service provides an immediate response, and blocks the BPEL process for the duration of the operation. An asynchronous service does not block and continues with the BPEL process, and is used when a service may take a long time to process a client request.

Having introduced the main technologies and concepts, knowledge of which is necessary for understanding the details of implementing services in Oracle BPEL PM, we proceed with the pattern evaluations. For each of the pattern families, first, we illustrate how a selected pattern variant can be implemented in Oracle BPEL PM. For this, we describe the mapping between the configuration parameters and corresponding settings in the im-
Implementation presented. Thereafter, we discuss the support for other pattern variants by analyzing each of the configuration parameters in detail.

**Multi-party Multi-message Request-Reply Conversation**

Figure 242 shows the notation for the pattern variant where all configuration parameters are set to the default values. In this pattern variant, a Requestor sends one message to a Responder, who responds with a reply message. Messages received by the Requestor are not queued and are consumed immediately (one message is required for the requestor to start the consumption and utilization of the messages received). The Requestor consumes a message from the queue only once, and the rest of the messages are discarded.

![Notation for the default pattern variant of the Multi-party Multi-message Request-Reply Conversation pattern family](image)

We implement this example as an asynchronous interaction, because there is the possibility that the responder service will not reply. Each of the processes we consider, i.e. the requestor and the responder, needs to define the other process as a partnerLink. Since the requestor process needs to send a message to one responder process, only one partner link needs to be defined. In order to send a message, the requestor process needs an `<invoke>` activity, and in order to receive a message, one of the Inbound Message Activities (IMA), i.e. `<receive>`, `<pick>`, and `<onEvent>` is required. The responder process needs a `<receive>` activity to accept the incoming request, and a `<reply>` activity to return either the requested information or an error message (i.e. a fault).

The requestor and responder processes, implemented in Oracle BPEL PM, are illustrated in Figures 243(a) and (b) respectively. Figure 243(a) shows an asynchronous process which, upon an initiation by a client, performs an invocation of a synchronous service ResponseProcess presented in Figure 243(b) using the `SendRequestToResponder` invoke activity. The request message sent by the requestor process is specified in the RequestorInputVariable input variable of the invoke activity as shown in the code fragment in Table 5.17. Note that the type of the message sent by one process to another has to be the same both in the input variable of the invoke activity and in the output variable of the reply activity. The message types are defined in the WSDL definition for each of the interacting processes. In this case, the input and output variables are of the String type (the source code of the XML, WSDL and BPEL files can be found in [157]).

In order to specify the content of the message, prior to the invoke activity the value of its input variable has to be set. For this, the `<assign>` activity AssignInputData in Figure 243(a) is used.

The ResponseProcess process is initiated by a message received from the requestor process. The message received via the `<receive>` activity is processed by an `<assign>` activity ProcessRequest in Figure 243(b), and a response is sent back to the requestor process.
Figure 243: Implementation in Oracle BPEL PM: the default pattern variant of the Multi-party Multi-message Request-Reply Conversation

Table 5.17: Code fragment of the <invoke> activity

```xml
<invoke name="SendRequestToResponder" partnerLink="ResponseProcess"
   portType="ns2:ResponseProcess" operation="process"
   inputVariable="RequestorInputVariable"
   outputVariable="ObtainedOutputVariable"/>
```

using a replyOutput reply activity. The response message sent by the responder process is assigned to an output variable ObtainedOutputVariable of the SendRequestToResponder invoke activity. Note that the <invoke> activity has no attribute for message queueing, therefore response messages are not queued and are consumed and processed as soon as they arrive. This maps to consumption and utilization indexes whose value is set to one, and a sorting of messages configuration parameter whose value is set to NoQueue. Only messages of a specific type can be consumed by the <invoke> activity, other messages are ignored. In this implementation, as soon as the corresponding message is received by the <receive> activity of the requestor process, the flow of control proceeds. This maps to a consumption frequency configuration parameter whose value is set to one.

We have shown how the default pattern variant can be realized in Oracle BPEL PM. In order to see which other pattern variants can be realized in Oracle BPEL PM, we analyze the configuration parameters in detail below.

- **Number of sub-requests in a message**: to include multiple sub-requests in a message, a RequestorInputVariable has to be a complex data type (whose definition needs to be performed in a xsd-file (XML Schema definition) and has to be included in both the Requestor and Responder process definitions). Since the message provided
by the requestor may be of composite type, the responder has to send a separate reply corresponding to each of the elements contained in the request message.

- **Number of Responders involved in a conversation:** an invocation activity may call operations only on a single service represented as a partner link. In order to send the same message to a set of responding parties, each party must be defined as a separate **PartnerLink**, and a distinct **<invoke>** activity targeting each partner link has to be added on a separate branch of the **<flow>** construct.

- **Possibility of non-responding parties:** when a request-response invocation is performed by the Requester process, the **<invoke>** activity stays open until the response is received. This however does not guarantee that the invoked service will respond. The time interval within which a response should be received by the **<invoke>** activity is fixed in Oracle BPEL PM. If within this time period a response has not been received, a fault is thrown. To handle the fault, the **<invoke>** activity can be included in a scope, and a corresponding event handler can be associated with this scope.

- **Possibility of missing replies:** the Responder process after receiving a message from the Requestor process via an **<invoke>,<pick>** or **<onMessage>** construct may send a reply to all sub-requests in a single message, or to each of them separately. The selection of sub-requests for which a reply message will be generated can be included in the process logic (for instance, a **<switch>** construct could be used).

- **Sorting of queued messages:** receiving activities in Oracle BPEL PM need at most one message to proceed. A message is processed as soon as it has been received by a matching target activity. This corresponds to the pattern variant, where **NoQueue** is specified. In order to support scenarios where multiple messages are required in order to make a decision, the receiving activity has to be included in a loop. During each iteration, an array can be used to aggregate data from the messages received. When a sufficient number of messages have been received, the array data can be examined and the required data can be extracted from it. For realization of FIFO, LIFO and PRIO ordering of messages, custom functions need to be written for the use in the copy rules of the **<assign>** activity. Still, even with such a workaround, not all scenarios can be realized. For instance, scenarios where non-consumed messages have to be kept in the queue for future use are not possible.

- **Enabling condition:** receiving activities in Oracle BPEL PM are triggered immediately when a matching message is received by a process instance (i.e. a message of a specific type). This corresponds to an enabling condition where the number of messages required is set to one, i.e. K=1. If queuing of the messages is realized by a receiving activity inserted in a loop, the enabling condition examining properties of the messages received can be specified as part of the loop termination condition. To realize the enablement of message consumption based on a timeout, a **<wait>** construct can be used.

- **Consumption index:** receiving activities are triggered by a matching message, and this message is used for processing.

- **Utilization index:** since receiving activities can only consume one message at a time, the message consumed is also the one used for processing, i.e. U=1.

- **Consumption Frequency:** since all messages received are consumed by receiving activities, eventually no messages are left in the queue. For this reason, no other pattern variants except the one where the consumption frequency is set to 1 can be realized.

To conclude, the majority of pattern variants belonging to the *Multi-party Multi-message*
Request-Reply Conversation pattern family can be realized in Oracle BPEL PM through the BPEL process activities and some programmatic extensions. However, since native support for queues on inbound message activities is missing, pattern variants based on queueing and consumption of multiple messages are difficult or even impossible to realize.

Renewable Subscription

Figure 244 shows the graphical notation for the Provider-initiated Customer-renewed subscription. We will use this subscription type as an example for realizing the Renewable Subscription pattern family in Oracle BPEL PM.

**Figure 244:** Implementation in Oracle BPEL PM: Provider-initiated Customer-renewed subscription

In this example, two communicating parties, i.e. a customer and a provider, are involved in a conversation that consists of two phases: subscription initiation and subscription renewal. The customer and the provider have to be implemented as two processes, each declaring the other process as a PartnerLink. Figures 245 and 246 show the implementation of the provider process, and figures 247 and 248 show the implementation of the customer process respectively.

The processes presented describe the logic of the subscription renewal, where the provider sends to the customer an offer for a subscription initiation via an <invoke> activity. This offer specifies the product being distributed, the subscription period, the period within which the customer has to respond, and the expected initiation confirmation. The messages exchanged in this example are based on composite data types, whose definition must be shared between both the provider and customer processes. The code fragment in Table 5.18 declares four message types that map directly to the graphical notation in Figure 244. These are Request, Response, RenewRequest and RenewResponse defined for the RespondingCustomerProcess customer process to represent an initiation request issued by the provider, an initiation response issued by the customer, a renewal request issued by the customer, and a renewal response issued by the provider respectively. These message types are used in invocation and inbound message activities of both processes.

In general, in a renewable subscription process, both the customer and the provider of a service can play the role of the subscription initiator. For this, an initiating party has to execute an <invoke> activity, while the invoked party has to react to the message...
Table 5.18: Data types used in Provider-initiated Customer-renewed subscription

```xml
<schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    targetNamespace="http://xmlns.oracle.com/RespondingCustomer"
    xmlns="http://www.w3.org/2001/XMLSchema">

<element name="REQpinit">
    <complexType>
        <sequence>
            <element name="Prod">
                <simpleType>
                    <restriction base="string"/>
                </simpleType>
            </element>
            <element name="Nr" type="int"/>
            <element name="SP" type="int"/>
            <element name="RP" type="int"/>
            <element name="Qi">
                <simpleType>
                    <restriction base="string">
                        <enumeration value="Yes"/>
                        <enumeration value="No"/>
                        <enumeration value="YesNo"/>
                    </restriction>
                </simpleType>
            </element>
        </sequence>
    </complexType>
</element>

<element name="RPLcinit">
    <complexType>
        <sequence>
            <element name="Prod" type="string"/>
            <element name="R">
                <simpleType>
                    <restriction base="string">
                        <enumeration value="Yes"/>
                        <enumeration value="No"/>
                    </restriction>
                </simpleType>
            </element>
        </sequence>
    </complexType>
</element>

<element name="REQcrenew">
    <complexType>
        <receive>
            or
            <pick>
                activity, where the createInstance attribute is set to “yes” (the latter is needed to create a new process instance which can be correlated with a given subscription). In this example, the provider process plays the role of the subscription initiator.

        The top view of the provider process is shown in Figure 245. After a process has been
Table 5.19: Data types in Provider-initiated Customer-renewed subscription (Cont.)

```xml
<sequence>
  <element name="Prod" type="string"/>
</sequence>
</complexType>
</element>
<element name="RPLprenew">
  <complexType>
    <sequence>
      <element name="Prod" type="string"/>
      <element name="Nr" type="int"/>
      <element name="SP" type="int"/>
      <element name="PR">
        <simpleType>
          <restriction base="string">
            <enumeration value="Accept"/>
            <enumeration value="Reject"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>
</element>
</complexType>
</schema>
```

initiated, the AssignVariationPoints activity is executed setting each of the configuration parameters in the initiation request to an allowable value. The product identifier, subscription period, response period, and expected response period are the values specified by the InvokeInitCustomerInputVariable input variable for the InvokeInitCustomer invocation activity. This activity is of the synchronous type, which means that the flow of control in the provider process is blocked until a response from the customer process arrives.

Figure 246 shows the details of processing a renewal request sent by the customer when the accepted subscription is about to expire. The <switch> construct is used to model that either no response is sent by the provider or a response approving or rejecting the subscription renewal request is sent to the customer. Figure 247 illustrates the top view of the customer process, that is initialized upon arrival of a subscription offer from the provider process via the <receiveInput> activity. The assignment activity is used to set static attributes and configuration parameters of the customer response, i.e. whether the subscription offer will be accepted, rejected or ignored.

The details of processing the offer received by the customer are shown in Figure 248. Several switch cases are used to define the content of the initiation response message depending on the type of the request posted by the Provider. As such, for instance, for the subscription offer whose confirmation attribute Qi="No", no response is required to accept the subscription, while for Qi="Yes/No" an explicit response needs to be sent.

Figure 247(b) shows the logic for renewing a subscription at the instigation of the customer. The decision of the customer process to renew or not to renew the subscription is implemented via the <switch> construct with two branches. If the subscription does
not need to be renewed, an `<empty>` activity is executed, after which the customer process is terminated, otherwise the invocation activity is executed to send a message of the `RenewRequest` type to the provider process.

In this example, only one configuration parameter (Qi) needs to be set in order for the specific pattern variant to be realized. The expected initiation confirmation parameter is set in the `<assign>` activity before the subscription initiation offer is sent to the customer process.

We have shown how different pattern variants for the Provider-initiated Customer-renewed subscription type can be realized in Oracle BPEL PM. To illustrate what other pattern configurations can be realized, we discuss each of the configuration parameters below.

- **Subscription renewal type**: the other five subscription renewal types can be realized in the same manner as has been done for the Provider-initiated Customer-renewed subscription earlier. In these subscription renewal types, one of the processes, either
a customer or a provider, has to initiate the subscription. For this, an initiating party has to execute an `<invoke>` activity, while the invoked party has to react on the message received via a `<receive>` or `<pick>` activity, where the `createInstance` attribute is set to “yes” (the latter is needed to create a new process instance which can be correlated with a given subscription).

The subscription renewal can be done at the initiative of the customer, the provider or automatically. For automatically renewed subscriptions, in the customer process the renewal phase must be realized using a `<while>` construct that contains an activity for receiving the product subscriptions from the provider process. The termination condition for this `<while>` construct determines whether the subscription should be canceled or not. If a decision to terminate the subscription is taken, a message of the cancelation type has to be sent to the provider process.

Different message types, i.e. initiation, renewal, and cancelation requests and replies issued by the customer and the provider, must be defined for each of the subscription renewal types. As has been discussed, the XML-based data type declarations must be included in both the provider and the customer processes. The data type has to consist of the configuration parameters and static attributes, which when instantiated represent the pattern configuration for the specific pattern variant.

- **Expected initiation confirmation**: this configuration parameter constitutes a part of the initiation request issued by the provider process. It is defined as an element with three possible values: **Yes**, **No**, **YesNo**. For a specific pattern variant, this parameter needs to be set to one of these values depending on what confirmation approach the
provider process requires. In order to accept a subscription offer whose Qi field is set to Yes or YesNo the customer process has to send Yes in the response message, or not to send any response if the Qi field is set to No.

- **Expected renewal confirmation**: this confirmation parameter (Qr) constitutes a part of the subscription renewal request issued by the provider process, thus it applies only to provider-renewed subscriptions. The value of this configuration parameter can be set to Yes, No, or YesNo in the provider-renewal request. A message of the provider renewal request type can be defined in the same way as described for the expected initiation confirmation parameter.

The decision of the customer to accept, reject or ignore the offer can be encoded using a `<switch>` construct with three branches. Depending on the type of the confirmation expected by the provider, either a message accepting or rejecting the offer is returned or no message is sent at all.

Oracle BPEL PM provides basic primitives in the form of BPEL activities to describe the behavior of a business process based on interactions with other processes through web
service interfaces. Since BPEL concentrates on message exchange between processes, it
does not define mechanisms for conversation-oriented scenarios like renewable subscriptions
because they are considered to occur at a higher level of abstraction and they are typically
managed at application level. Nevertheless, in Oracle BPEL PM renewable subscriptions
can be realized by specifying custom message types and defining supporting business logic.

Message Correlation

An example of the Message Correlation pattern variant where all configuration parameters
are set to the default value is presented in Figure 249. In this example, a customer sends
an order request to the provider, where it specifies its address (Pr), the order id (Cs) and
the address of the customer Pr. The provider receives the order request and records the id
of the order request received for use in future correlations. We will use this example for a
sample implementation in Oracle BPEL PM.

Figure 249: Default notation: Message Correlation
There are two approaches for realizing message correlation in Oracle BPEL PM. Either the facilities of WS-Addressing [216], BPEL correlation sets or their combination can be used for this purpose. Oracle BPEL PM uses WS-Addressing to automatically set location and correlation information associated with a client role. WS-Addressing defines two concepts: endpoint references and message information headers. Endpoint references convey information providing addresses for individual messages sent to and from Web services. The endpoint reference format shown in Table 5.20 includes the **<Address>** for specifying the address of the message sender or the message receiver. Furthermore, it includes **<ReferenceProperties>** or **<ReferenceParameters>** fields suitable for including the conversation identifier used by the party for correlation.

The message information headers convey message attributes including the addresses of source and destination endpoints, and the message identity. The message information header format shown in Table 5.3.1 specifies the source address via the **From** field and the destination address via the **To** field. The address of the message destination must be explicitly specified, while the source address may be omitted (by doing so the message sender may stay anonymous). These requirements directly map to the **From** and **To** configuration parameters of the **Message Correlation** pattern family.

Endpoint references must contain an address identifying an endpoint (i.e. this information may not be left out). Reference properties and reference parameters specify individual properties and parameters required to effectively interact with the endpoint. This information is optional and may be omitted. When address and reference properties supplied in a message are compared with the actual address and reference properties of the endpoint, they are checked for equivalence. Not only the addresses of endpoints must be equivalent, but also the endpoints must contain the same number of individual properties, and for each reference property there must exist an equivalent reference property in the other endpoint. If the sender address appears to be anonymous, other information specified in the message information header about the message id or about the relation of this message to another
Table 5.22: Order type

```xml
<element name="order" type="tns:orderType"/>
<complexType name="orderType">
  <sequence>
    <element name="orderId" type="string"/>
  </sequence>
</complexType>
```

Table 5.23: Correlation set "Order"

```xml
<correlationSets>
  <correlationSet name="Order" properties="tns:orderId"/>
</correlationSets>
```

message can be used to identify the destination where reply messages can be sent to. This combines both key-matching and property-based analysis correlation methods.

In addition to the use of WS-Addressing, Oracle BPEL PM allows the specification of correlation sets, i.e. a set of properties, that are used amongst web-services to uniquely identify a conversation. To use correlation sets, custom data types must be defined in the XML Schema definition files of the communicating processes. Table 5.22 shows an example declaration of the order type that will be used to uniquely identify orders by their orderID identifiers.

The actual definition of correlation sets is done in the BPEL-source of both sender and receiver processes as shown in Table 5.23:

A receiving activity of the receiver processes a message, whose property orderID matches with the property specified in the correlation set (as shown in Figure 250).

![Correlation settings of the <Receive> activity](image)

(a) General settings
(b) Correlation settings

**Figure 250:** Correlation settings of the `<Receive>` activity

When a message with a particular correlation set is sent to a process for the first time, the correlation set has to be initialized. For this, its `initiate` attribute is set to `yes` as shown in Table 5.24.
Table 5.24: Usage of correlation sets in the <receive> activity

```xml
<receive name="receiveInput"
   partnerLink="client"
   portType="tns:Customer"
   operation="initiate"
   variable="input"
   createInstance="yes">
   <correlations>
     <correlation set="Order" initiate="yes"/>
   </correlations>
</receive>
```

We have shown how a Message Correlation pattern variant can be implemented in Oracle BPEL PM. In this implementation, both the facilities of WS-Addressing and BPEL correlation have been used. For each of the configuration parameters listed below we describe how other pattern variants can be realized in Oracle BPEL PM.

- **Message Sender field**: if BPEL correlation sets are used for correlation, then instead of the party identifier and the conversation identifier representing the credentials of the message sender, a set of properties shared by all messages in the correlated group of operations within a process instance, known as a correlation set, has to be specified.

  ```xml
  <correlationSets>?
    <correlationSet name="NCName" properties="QName-list">+
  </correlationSets>
  
  Uniquely named correlation sets together with a partner link represent the credentials of the message sender.
  
  If WS-Addressing is used for correlation purposes, then information about the message sender is specified in the From field of the endpoint reference.

- **Message Receiver field**: in Oracle BPEL PM, the credentials of the message receiver are not included in the message explicitly, i.e. they are transparently added via WS-Addressing to the To field of the endpoint reference. To specify the message sender based on BPEL-capabilities, a PartnerLink representing the party is defined. Thus the destination address can be specified in the process definition or it may be set dynamically, however it is not possible to send a message to an arbitrary party. In order to specify the conversation identifier used by the message receiver, a correlation set initiated by the message receiver must have been included in the previous interaction.

- **Credentials of the message sender before message correlation**: in BPEL terms, parties with whom the message receiver has been or will be involved in a conversation, have to be defined as partner links. Since a single dynamic partner link can be used by different parties, messages sent by these parties have to be accompanied by uniquely named correlation sets in order for these parties to be distinguished. Note that it is possible that initially the message receiver does not have any knowledge about the correlation set used by the message sender.

- **Credentials of the message sender after message correlation**: From the BPEL-perspective, if a message received by a party contains a correlation set that has not
been initialized yet, the message receiver has to initiate it. After this, the initiated correlation set can be used in the follow-up interactions. If at the moment of message receival, the correlation set has been already initiated, no information is gained.

From the WS-Addressing perspective, the information about the message sender may be retrieved from the Address, ReferenceProperties and ReferenceParameters field of the From endpoint reference.

The combination of WS-Addressing and correlation sets allows for specification of the majority of message correlation pattern variants except the ones where the identity of the target process is underspecified by the sender process.

Message Mediation

In this subsection, we illustrate how a Mediated Introduction pattern variant can be implemented in Oracle BPEL PM. We omit the implementation of the Mediated Interaction pattern variant, since the realization of the default pattern variant is very similar to the example of the Bipartite Conversation Correlation presented in the next section with the only difference being that instead of two, three parties are involved in the conversation, and for correlation of messages two kinds of correlation sets are used: one shared between the customer, mediator, and provider, and the other shared only between the mediator and the provider. The concept of correlation sets has been described in the previous section.

Figure 251 illustrates the graphical notation of the Message Introduction pattern variant where all pattern attributes are set to their default values. In this pattern variant, the customer sends a request to the mediator which in turn has to delegate it to the provider. The customer does not specify credentials for the required provider (the Them1 field is empty), therefore the mediator is responsible for its selection. The customer expects a response to be returned. In order for the provider to contact the customer directly, the mediator passes a reference to the credentials of the customer in the Them2 field. When sending the response message to the customer, the provider specifies all information about its credentials in the From5 field, and also a reference to the mediator in the Them5 field.

![Diagram of Message Introduction Pattern](image)

**Figure 251:** Default notation: Message Introduction

In the default Mediated Introduction pattern variant, a customer process named FlowA sends a request to the mediator process FlowB. The mediator process forwards the request to the provider process FlowC, who based on the information provided by the mediator dynamically, binds with the customer process and sends the response message directly to it (note that the customer process does not know the identity of the provider process
initially). The customer, mediator and provider processes are shown in figures 252, 253 and 254 respectively.

To correlate message exchange between these three processes, message invocation and receipt activities have to be associated with a shared correlation set. In this example, the order identifier is used as a property for correlation set Order described in tables 5.25 and 5.26.

Table 5.25: Order identifier in a correlation set

```xml
<correlationSets>
  <correlationSet name="Order" properties="tns:orderId"/>
</correlationSets>
```

After the customer process has sent a request, it awaits an asynchronous callback. In this example, the callback location and correlation id is transparently handled using WS-Addressing. In particular, the WSDL-file of the deployed FlowA process contains a field specifying that the ReplyTo field of the message information header is used to identify the address to which the response message should be sent. This corresponds to
the configuration parameter Them2. Furthermore, the address location is set dynamically by the caller (cf. Table 5.27).

To support all Message Mediation pattern variants, it should be possible to specify in a message the credentials of a third-party process, the message receiver should be able to dispatch information about the credentials of the third party and dynamically bind to it. These requirements are met by WS-Addressing via the concepts of endpoints and message
Table 5.26: Definition of order identifier

```xml
<complexType name="orderType">
  <sequence>
    <element name="orderId" type="string"/>
  </sequence>
</complexType>
```

```xml
<bpws:property name="orderId" type="xsd:string"/>
```

```xml
<bpws:propertyAlias propertyName="tns:orderId"
  messageType="tns:ABCARoutingFlowRequestMessage" part="payload"
  query="/tns:order/tns:orderId"/>
```

Table 5.27: Dynamic address definition via WS-Addressing

```xml
<message name = "WSAReplyToHeader">
  <part name = "ReplyTo" element = "wsa:ReplyTo"/>
</message>
```

```xml
<service name = "FlowACallbackService">
  <port name = "FlowACallbackPort" binding="tns:FlowACallbackBinding">
    <soap:address location = "http://set.by.caller" />
  </port>
</service>
```

information headers, whose format is described in tables 5.20 and 5.27 respectively.

Message information headers convey information about the source and destination endpoints of a given message in the From and To fields, and a ReplyTo address that can be used as a destination address for follow-up responses. The only mandatory field in the message information header is the address of the message destination To. This means that the message sender can stay anonymous by omitting detailed specification of the From field, while at the same time it can provide additional information in the ReplyTo field, which will be used as an address for sending response messages. The RelatesTo field may contain information specifying how the given message relates to other messages. Besides that, the MessageID field may contain a unique message identifier that can be used for correlation purposes. In the example presented, the ReplyTo field of the request sent by the mediator to the provider contains the credentials of the customer process (the Them2 field in Figure 251). The provider uses this information to specify in the To5 field the address of the party to whom the response message will be sent, i.e. the address of the customer.

We have shown how the Mediated Introduction pattern variant can be implemented in Oracle BPEL PM. Since this implementation uses both the facilities of WS-Addressing and BPEL, it is interesting to see what other pattern variants can be implemented in Oracle BPEL PM. For this, we analyze each of the configuration parameters below:

- **Customer request for specific provider**: in order to implement pattern variants, where the customer specifies the credentials of the provider, the partner link of the provider has to be set dynamically.

  In order to send a request to the provider after the mediator has been triggered by
a request received from the customer, the mediator should be able to dynamically bind to the reference provided. In WS-BPEL, this can be done via assignment of end-point references using the <Assign> activity. The reference provided is copied from the <from> field to the <to> field:

```xml
<from partnerLink="NCName" endpointReference="myRole|partnerRole"/>
<to partnerLink="NCName"/>
```

Note that the type for values specified in the from/to-assignment clause is: `<sref:service-ref>`.

One could underspecify the PartnerLink by setting it to a generic service, and assign its value during run-time using the endpoint reference of WS-Addressing. For this, a variable of the EndpointReference type needs to be declared:

```xml
<variable name = "partyReference" element="wsa:EndpointReference">
```

The value of this variable has to be assigned to the dynamically-configured partner link.

- **Information about the message sender in the request from mediator to provider**: to receive a response message back, the mediator has to specify a correlation set that is shared with the provider, otherwise the correlation set initiated by the customer service has to be passed.

- **Information about the message sender in the response from provider to customer**: the identity of the provider may be enclosed in the new correlation set (note that correlation sets have to be shared) or via end-point references as has been described earlier. An endpoint reference represents the data required to describe a partner service endpoint as a service reference container `<sref:service-ref>`. This reference can be used to dynamically determine a partner service that is not known at the moment of process instantiation.

- **Information about mediator exposed by provider to customer**: in contrast to WS-Addressing where the RelatesTo field is used to pass the reference to the related process, in WS-BPEL this can only be done in the content of the message. The message in this case has to be based on the data type defined in the XML-schema of the corresponding process.

To realize all Message Introduction and Message Interaction pattern variants, processes involved in a bilateral interaction should be able to provide references to other parties as well as to use the references provided for dynamic binding. Through the use of WS-Addressing and BPEL, Oracle BPEL PM is able to realize the majority of pattern variants (note that the pattern variants where some of the information related to the message receiver credentials is underspecified cannot be realized due to the limitations discussed in the Section 5.3.1).

**Bipartite Conversation Correlation**

Figure 255 presents the graphical notation of Bipartite Conversation Correlation pattern variant where all configuration parameters are set to their default values. In this pattern variant, the requestor and the responder retrieve all information from the message received, and specify complete information about the identity of the message sender and the message receiver in messages exchanged. In the follow-up interaction, the requestor uses information
provided by the responder as a token for correlation. We use this example as a basis for illustrating its implementation in Oracle BPEL PM.

![Diagram](image)

**Figure 255**: Default notation: Bipartite Conversation Correlation

Two processes, **Buyer** and **Seller**, one requesting a service and another offering it, are directly related and thus must be declared as partner links. To illustrate correlation information changing in the course of a conversation between these processes let us consider their implementations shown in Figures 256 and 257 respectively.

The **Buyer** sends a request to the Seller and specifies the correlation set that has to be used in the response message. Note that the correlation set has been initialized before the invocation of the Seller process, therefore its *initiate* attribute is set to *no*. Furthermore, the *pattern* attribute is set to *out* indicating that the correlation set applies to the outbound message as shown in Table 5.28.

**Table 5.28**: Correlation set specified by the Buyer process in the initial request

```xml
<invoke partnerLink="Seller" portType="seller:Seller"
      inputVariable="input"
      name="SendRequest" operation="AsyncPurchase">
  <correlations>
    <correlation set="PurchaseOrder" initiate="no" pattern="out"/>
  </correlations>
</invoke>
```

In order to be used in the process, this correlation set must be declared in the BPEL-specification for the process. The correlation sets shown in tables 5.28-5.30 are the ones used by the Buyer and Seller processes during the conversation. Each correlation set contains two properties, representing the identity of the party and the order identifier.

**Table 5.29**: Correlation set specified by the Buyer process in the initial request

```xml
<correlationSets>
  <correlationSet name="PurchaseOrder"
                  properties="cor:customerID cor:orderNumber"/>
  <correlationSet name="Invoice" properties="cor:vendorID cor:invoiceNumber"/>
</correlationSets>
```
The Seller process, receives an initiation request from the Buyer process, and uses the same correlation set PurchaseOrder for correlation. Note that this message not only creates a new process instance, but also initiates the correlation set within this process instance as shown in Table 5.30:

After processing an initiation request, the Seller process invokes the Buyer process and, in addition to the PurchaseOrder correlation set required by the Buyer, also specifies its own correlation set Invoice that must be used in the follow-up responses (cf. the code fragment in Table 5.31).

The Buyer process receives the initiation response from the Seller, and initializes the Invoice correlation set. Since the Invoice correlation set has not been used by the Buyer process in the previous interaction, it must be initiated as shown in Table 5.32.

The follow-up request sent by the Buyer process to the Seller process specifies both
Figure 257: Responder process: Seller

Table 5.30: Correlation set used by the Seller process for processing the initiation request

```xml
<receive partnerLink="Buyer" portType="seller:Seller"
operation="AsyncPurchase" variable="input" createInstance="yes"
name="ReceiveInitRequest">
  <correlations>
    <correlation set="PurchaseOrder" initiate="yes"/>
  </correlations>
</receive>
```
Realizing pattern variants, where only part of the correlation information required by the message receiver is specified by the message sender is possible by excluding the corresponding correlation set from the correlation sets included for the invocation activity. To deal with such a situation, the receiver must have a separate inbound message activity whose correlation sets matches with the one specified by the message sender. In general, such messages would not be processed, thus the corresponding conversation would not be identified.

Using the support for message correlation via WS-Addressing, where message information headers contain information both about the message sender in the From field and the message receiver in the To field, it should be possible to realize pattern variants where the message sender underspecifies its credentials. If in general, the absence of the message sender credentials makes it impossible to deliver the response message to the right destination, in WS-Addressing message information headers may contain additional information about a message id or relationship of this message to another message. By analyzing this
information, the corresponding process instance to which the response message has to be sent can be identified.

In order to define which other pattern variants can be realized in Oracle BPEL PM, we analyze each of the configuration parameters in more detail below:

- **Responder credentials in the initial response**: All correlation sets specified by the requestor process in a message initiating a conversation with the responder process must be initiated, i.e. the `initiate` attribute of every correlation set must be set to “yes”. Once initiated by the requestor, this correlation set should be used by the responder process in the follow-up responses. However, in addition to the correlation set provided by the requestor, the responder may initiate another correlation set that should be used in follow-up messages issued by the requestor. This new correlation set represents the provider credentials. However in order to correlate a message in which this correlation set is specified, the receiver process must have a corresponding inbound message activity sharing the same correlation set.

- **Responder credentials in the follow-up request**: Depending on the context of the conversation, the requestor may choose to use its own correlation set or the one provided by the responder. In order for the follow-up request issued by the requestor to be successfully processed, the responder process must have a pending activity whose correlation sets match, otherwise such a message will raise the `bpel:correlationViolation` fault. In general, the correlation set specified by the responder may be combined with a new or already existing correlation set initiated by the requestor.

- **Requestor credentials in the follow-up request**: The correlation set specified by the responder process in the initiation response must be used by the requestor in the follow-up request. It may be omitted only if the responder has a pending activity whose correlation sets match, otherwise such a message will raise the `bpel:correlationViolation` fault.

The combination of WS-Addressing and correlation sets of BPEL allow all pattern variants of the Bipartite Conversation Correlation pattern family to be implemented in Oracle BPEL PM. However, if one is restricted to native BPEL support, then the pattern variants where a party uses only a subset of credentials provided by another party as correlation information specified in the follow-up messages or where a party specifies additional information in addition to that requested by the other party would not be possible to realize in Oracle BPEL PM due to the limitations imposed by correlation sets (as described in Section 5.3.1).

### 5.4 Related work

The service interaction patterns documented by Barros et al. [37,38] describe a collection of scenarios, where a number of parties, each with its own internal processes, need to interact with one another according to a set of pre-agreed rules. These scenarios are consolidated into 13 patterns and classified based on the maximal number of parties involved in an exchange, the maximum number of exchanges between two parties involved in an interaction and whether the receiver of a response was necessarily the same as the sender of a request. Based on this classification, four groups were identified: (1) single transmission bilateral interactions (i.e. one-way and round-trip bilateral interactions where a party sends and/or receives a message to/from another party); (2) single transmission multilateral non-routed interactions (i.e. a party sends/receives multiple messages to/from different parties); (3)
multi transmission bilateral interaction (i.e. a party sends/receives more than one message to/from the same party); and (4) routed interactions.

Since the service interaction patterns [37] lacked formal semantics, their formalization by means of the π-calculus has been proposed in [73]. The majority of these patterns can be interpreted in various ways. Decker and Puhlmann formalize the patterns based on the pattern descriptions and make some additional assumptions about the value of various variable pattern attributes, however they do not specify the whole range of values the selected pattern attributes may take. Thus, they show the possibility of formalizing certain aspects of service interaction, but in fact do not make the definition of patterns less ambiguous. For example, the Racing Incoming Messages pattern specifies: A party expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes. This pattern does not specify what happens if the party receives multiple messages at once, i.e. it is not clear how many of the received messages will be consumed and whether the rest of the messages will be discarded.

In [235], Zaha et al. formulated requirements for a service interaction modeling language, in addition to the requirements covered by Barros et al. in [38]. The authors used these requirements for modeling behavioral dependencies between service interactions.

In [36], Barros et al. introduced five correlation mechanisms, five conversation patterns, and eight process instance to conversation relationships that were subsequently used for evaluation of the WS-addressing and BPEL standards. However, the framework presented by the authors does not cover relationships between different process instances. In this chapter, we addressed this issue by analyzing correlation at both a higher and a lower level of abstraction.

Aldred et al. [24] performed a detailed analysis of the notion of coupling/decoupling in communication middleware using three dimensions of decoupling, i.e. synchronization, time and space, and documented coupling integration patterns.

In [64], Cooney et al. proposed a programming language for service interaction, which has been used to describe the implementations of two service interaction patterns, i.e. One-to-Many Send-Receive and Contingent Requests [37].

This work is also related to contracting workflows and the protocol patterns of van Dijk [76], who proposed a number of protocol patterns for the negotiation phase of a transaction.

Barros et al. [39] have proposed a compositional framework for service interaction patterns and interaction flows. They provided high-level models for eight service interaction scenarios using ASM, illustrating the need to distinguish between different interpretations of the patterns.

The work of Hohpe and Woolf on Enterprise Application Integration [125] covers various messaging aspects that may be encountered during application integration. Some of the enterprise application integration patterns relate to the problems addressed in this chapter, however they do not address all aspects of these problems explicitly and lack a formal semantics. Based on implicit assumptions, the related enterprise integration patterns can be mapped to pattern variants from one or more pattern families we have presented. An example of entreprise integration patterns, Scatter-Gather, concerning the overall message flow when a message must be sent to multiple recipients, each of whom may send a reply, is presented in Figure 258(a).

This notation specifies, that a message (e.g., a quote request) is broadcast to multiple
Figure 258: Mapping of Scatter-Gather pattern to the notation of the Multi-party Multi-message Request-Reply Conversation pattern family

recipients and the responses are aggregated into a single message. Although many issues related to in-time consumption of messages, selection of the “best quote” from of them, etc. are raised in the description of the Scatter-Gather pattern, not all of them are explicitly addressed. This pattern with a set of implicit assumptions, can be mapped to one of the pattern variants of the Multi-party Multi-message Request-Reply Conversation pattern family, whose configuration is presented in Figure 258(b). This configuration reflects the broadcast of a single message to a list of three parties, responses from whom are not guaranteed. Furthermore, it specifies how messages received are to be sorted (based on order of their arrival), what is the enabling condition (in this case, the timeout is used to avoid deadlock if no responses are received) for consumption, how many messages are consumed (i.e. $C = \text{All}$), and how many of them are utilized ($U = 1$).

5.5 Summary

This chapter has identified a series of the requirements encountered in service interaction and described them in the form of service interaction patterns. It provides more comprehensive coverage of various aspects of interactions of bilateral and multilateral nature than previous work [37, 38] where 13 basic imprecisely-specified patterns were identified.

Due to the large number of patterns identified (i.e. in total 1602 new patterns), an approach to describing the patterns in the form of a configurable framework has been adopted. For this, two new concepts have been introduced: a pattern variant and a pattern family. Although these concepts have not been explicitly employed in Chapters 3 and 4, we can define the following correspondence. Patterns which represent specializations of a more generic pattern in terms of the manner in which they address a certain problem, can be considered as pattern variants. For instance, the FIFO Queue (cf. page 78), LIFO Queue (cf. page 79) and Priority Queue (cf. page 81) CPN patterns can be considered as pattern variants of a more generic pattern Queue (cf. page 75). The Blocking Discriminator WCF-pattern and the Canceling Discriminator WCF-pattern can be considered as variants of the more generic discriminator type. In both catalogs of CPN patterns and control-flow patterns, patterns addressing (structurally) similar problems and solutions are combined into groups which can be associated with pattern families. Although the pattern groups consist of similar patterns, they do not have a single pattern core from which the different pattern configurations can be derived.
Chapter 5 Service Interaction Patterns

The focus of this chapter is on service interaction. To remove any potential ambiguity and possibility for misinterpretation, a precise definition of pattern operation using the formalism of CPNs has been provided. To visualize different pattern variants, an intuitive graphical notation corresponding to a pattern configuration has been introduced. A pattern configuration is a set of pattern attributes that are set to specific values in order for a specific pattern variant to be obtained. Since the number of configuration parameters is quite large and varies for each pattern family, the graphical notation provides a convenient way to depict and distinguish distinct pattern variants.

For each of the pattern families we analyzed the meaningful pattern configurations, and calculated the total number of pattern variants per family. As such, 1072 pattern variants were identified as belonging to the Multi-party Multi-message Request-Reply Conversation pattern family, 20 pattern variants belong to the Renewable Subscription pattern family, 100 pattern variants belong to the Message Correlation pattern family, 392 pattern variants belong to the Message Mediation pattern family, and 18 pattern variants belong to the Bipartite Conversation Correlation pattern family.

The large number of pattern variants identified demonstrates the heterogenous nature of requirements in service interaction. The service interaction patterns identified are beneficial for:

- Reusing accumulated knowledge;
- Precise, light-weight description of SOA scenarios;
- Benchmarking SOA offerings; and
- Common SOA vocabulary.

We describe each of these facets in detail below.

Given the popularity of SOA and lack of understanding requirements associated with service interaction, the service interaction patterns facilitate the understanding of these requirements by explicitly delineating problems that need to be solved, and describing possible solutions to them. As a consequence of investigating various approaches in the service-oriented domain, we identified a comprehensive set of patterns, and systematically documented them in the form of a configurable framework in order to facilitate the reuse of accumulated knowledge in the field.

Different aspects of complex service-interaction scenarios can be characterized by means of applying corresponding pattern families. Given a set of pattern families, it is unclear how to perform classification of a service-interaction under consideration, which pattern families are relevant and in which order they should be applied. Therefore, in order to assist users in classifying service-interaction scenarios, we have defined a pattern-based service interaction design method. Depending on the goal of the classification and the degree of details required, this method defines what pattern families have to be used for classification, how they can be configured, and which pattern families can be combined. As a consequence, a precise, light-weight description of SOA scenarios is achieved.

In addition, benchmarking of SOA offerings can be performed in order to identify the support for different aspects of service interaction. For this, configuration parameters can be mapped to features offered by the systems, and support for realizing different values for each of the configuration parameters can be identified. Note that original patterns have been successfully used in pattern-based evaluations [38] and influenced the development of languages and standards such as Let’s Dance [235], [236], [74] and GPSL [64].

Considering that standardization in the web-services domain is a central topic at the moment, the insight into the requirements of services interaction provided in this chapter
Section 5.5 Summary

321

can be used as a source of suggestions for improving already existing standards in the field. For instance, the BPEL standard does not have concepts able to capture the meaning of all configuration parameters or these concepts are not explicitly defined. By definition, in BPEL all inbound message activities are executed as soon as a suitable message arrives. Selection of such a behavior results in quite limited capabilities to support different variants of message handling. Since BPEL intentionally does not specify a mechanism for handling of the race conditions, systems supporting BPEL-processes may employ different implementations and thus support distinct pattern variants. In this case, the configuration parameters can be used as an instrument for selection of an appropriate system. When selecting an appropriate middleware technology, in addition to identifying the required service interaction scenarios, an analysis of the (de-)coupling requirements can also be performed based on the framework identified in [23]. This framework describes three dimensions of decoupling, i.e. space, time, and synchronization, which characterize interaction by defining whether the sender uses a direct address to send the message to, whether the endpoint is concurrently operational, and whether the sender’s thread of control is blocked after a message has been sent respectively.

The pattern language also serves as a source of reference for design and development of SOA-based tools. The pattern configurations of each pattern family can be included in the user interface of a particular tool to simplify the specification of all relevant attributes of service interactions. Besides, the concepts in the configurable framework precisely define commonly used SOA terms and approaches, thus providing the basis for a common SOA vocabulary. Usage of this vocabulary helps to improve communication of problems and solutions in the domain. This becomes evident from earlier experiences in the control-flow domain, where pattern names are increasingly being used for referring to common process modeling constructs.

The ability to interact with external services is only one of many important aspects contemporary PAISs are required to address. Organizations seeking for success require their processes to be flexible, i.e. it should be possible to include foreseen and unforeseen behavior in the process definition, thus allowing the process to adapt to requirements imposed by quickly changing environment. The next chapter is devoted to investigation of requirements of PAISs from the process flexibility perspective.
Chapter 6

Process Flexibility Patterns

In Chapter 4, we described fundamental constructs for describing the structure of a process model in the form of control-flow patterns. The manner in which desired and/or possible behavior is captured in the process definition, greatly influences the flexibility given to a user in selecting a suitable execution path during process execution. As a consequence of changes in the operating environment, the execution of a process may need to be steered in the right direction in order to achieve the desired execution order of tasks. This becomes possible when multiple alternative paths have been foreseen during design-time. In situations, where the required execution path cannot be found, the need to deviate from the prescribed execution path may arise, necessitating changes to be made to process definition on the spot. In this chapter, we focus on the aspect of process flexibility. We distinguish different types of process flexibility and present them in the form of a taxonomy. For each of the flexibility types identified, we systematically analyze the requirements for process flexibility and describe them in a language-independent and precise manner using a pattern-based approach. We then utilize the process flexibility patterns identified in order to evaluate the functionality of selected PAISs.

This chapter is organized as follows. Section 6.1 presents a taxonomy for process flexibility. In Section 6.2, we describe 34 process flexibility patterns that have been identified. In Section 6.3, we use these patterns to evaluate the support for process flexibility in a selection of contemporary PAISs. In Section 6.4, we discuss related work. Finally, Section 6.5 concludes this chapter.

6.1 Taxonomy of process flexibility

In this section, we present a comprehensive description of five distinct approaches that can be taken to facilitate flexibility within a process. This is a variant of our earlier taxonomy presented in [200]. Figure 259 shows five types of flexibility: flexibility by design, flexibility by deviation, flexibility by underspecification, and two types of flexibility by change, i.e. flexibility by momentary change and flexibility by permanent change. The flexibility types identified represent orthogonal dimensions and are intended to operate independently of each other. Note however that it mainly concentrates on the control-flow perspective, and does not address issues related to other perspectives such as data handling and resource assignment.
The taxonomy is applicable to both classical (imperative) and constraint-based (declarative) process specifications\(^1\). An *imperative approach* focuses on the precise definition of how a given set of tasks has to be performed (i.e., the task order is explicitly defined). In imperative languages, constraints on the execution order are typically described either via links (or connectors) between tasks and/or data conditions associated with them. Thus for imperative languages, the process description generally takes the form of a directed graph. Although there are notable exceptions such as BPEL [164]. A *declarative approach* focuses on what should be done instead of how it should be done. It uses constraints to restrict possible task execution options. By default, all execution paths are allowed except executions that violate the constraints. In declarative languages, constraints are defined as relationships between tasks. Mandatory constraints have to be strictly enforced, while optional constraints can be violated, if needed. Thus for declarative languages, the process description consists of a set of tasks and the set of relationships between them.

![Figure 259: Flexibility types](image)

Each of the five flexibility types aim at improving the ability of business processes to respond to changes in their operating environment without necessitating a complete redesign of the underlying process definition, however they differ in the timing and manner in which they are applied. Figure 260 shows how the flexibility types identified can be classified in terms of the moment at which the need for a specific flexibility type is recognized, the moment at which steps for facilitating the anticipated flexibility needs are taken, and their relationship to actual process execution. The scope of impact associated with each of the flexibility types is defined as follows. When a decision related to the realization of a specific flexibility type affects only a particular process instance, we map it at the process instance level; when all existing process instances are affected, we map the flexibility type at the type level.

![Figure 260: Recognition and realization of flexibility types: moment and scope](image)

\(^1\)We concentrated on workflow offerings employing the imperative approach when evaluating the support for WCF and service interaction patterns in chapters 4 and 5. In this chapter, we also concentrate on systems employing the declarative approach as they provide alternative means for process flexibility.
As Figure 260 shows, the need for flexibility by design is recognized at the type level at design-time. All decisions related to the incorporation of this type of flexibility into a process definition are both taken and realized before process initiation. Thus there is a direct relationship between the flexibility incorporated into a process definition at design-time and the way it influences process execution. Flexibility by underspecification is recognized at design-time when a process definition is being created, however its complete realization is postponed until run-time. For this flexibility type, the desired behavior is set for a specific process instance, during the course of its execution. Note that different instances of the same process may have distinct realizations. Flexibility by deviation is recognized at run-time at the moment of (or shortly after) process initiation (for a specific process instance). The same classification applies to the flexibility by momentary change. The main difference between these two flexibility types is that flexibility by momentary change results in the modification of the process definition associated with a given process instance during execution, whilst realization of flexibility by deviation leaves the process definition unaffected. Finally, flexibility by permanent change can be seen as redesign of the process definition at the type level at run-time. This flexibility type is facilitated by performing changes to the process definition, potentially impacting all existing process instances.

Each of the individual flexibility types introduced above are now described in detail using a standard format including: a motivation, definition, scope and realization options. We start with flexibility by design.

6.1.1 Flexibility by design

Motivation When a process operates in a changing operational environment it is desirable to incorporate support for the various execution alternatives that may arise during execution within the process definition. At runtime, the most appropriate execution path can then be selected from those encoded in the design time process definition.

Definition Flexibility by Design is the ability to incorporate alternative execution paths within a process definition at design time allowing for selection of the most appropriate execution path to be made at runtime for each process instance depending on the circumstances encountered.

Scope Flexibility by design applies to any process which has multiple predefined execution trace and the choice of these traces can be influenced at run-time.

Realization options The most common options for realization of flexibility by design, such as parallelism, choice, iteration, etc. are thoroughly described by the control-flow patterns in Chapter 4 and have been widely observed in a variety of imperative languages. We argue that these concepts are equally applicable in a declarative setting which has a much broader repertoire of constraints that allow for flexibility by design. Note that both approaches really differ with respect to flexibility. To increase flexibility in an imperative process, more execution paths have to be modeled explicitly, whereas increasing flexibility in declarative processes is accomplished by reducing the number of constraints, or weakening existing constraints. A declarative model is most flexible when it does not have any constraints [171]. In this case, all of its constituent tasks can be executed in any order, any number of times.

Describing all possible execution paths in a process definition completely at design-time may be either undesirable from the standpoint of model complexity or impossible due to an
unknown or unlimited number of possible execution paths. The following flexibility types provide alternative mechanisms for process flexibility.

6.1.2 Flexibility by deviation

Motivation Some process instances need to temporarily deviate from the execution sequence prescribed by their associated process definition in order to accommodate changes in the operating environment encountered at runtime. The overall process definition and its constituent tasks remain unchanged.

Definition Flexibility by Deviation is the ability for a process instance to deviate at runtime from the execution path prescribed by the original process without altering its associated process definition. The deviation can only encompass changes to the execution sequence of tasks in the process for a specific process instance, it does not allow for changes in the process definition or the tasks that it comprises.

Scope The concept of deviation is particularly suited to the specification of process definitions which are intended to guide possible sequences of execution rather than restrict the options that are available (i.e. they are descriptive rather than prescriptive). These specifications contain the preferred execution of the process, but other scenarios are also possible.

Realization options The manner in which deviation is achieved depends on the specification approach utilized. Deviation can be seen as varying the actual tasks that will be executed next, from those that are implied by the current set of enabled tasks in the process instance. In imperative languages, this can be achieved by applying deviation operations such as redo, task skip, etc. For declarative approaches, deviation occurs through the violation of (optional) constraints.

6.1.3 Flexibility by underspecification

Motivation When specifying a process definition it might be foreseen that during run-time execution more execution paths are needed which must be incorporated within an existing process definition. Furthermore, often only during the execution of a process instance does it become clear what needs to be done at a specific point in a process. When all execution paths cannot easily be defined in advance in the standard way, it is useful to be able to execute an incomplete process definition and dynamically add process fragments expressing missing scenarios to it.

Definition Flexibility by Underspecification is the ability to execute an incomplete process definition at run-time, i.e. one which does not contain sufficient information to allow it to be executed to completion. Note that this type of flexibility does not require the definition to be changed at run-time, instead the definition is completed by providing a concrete realization for the undefined parts as they are encountered at run-time.

Scope The concept of underspecification is mostly suited to processes where it is clearly known in advance that the process definition will have to be adjusted or has high variability at specific points, although the exact content at these points is not yet known (and may not be known until the time that an instance of the process is executed). This approach to process design and enactment is particularly useful where distinct parts of an overall process are designed and controlled by different work groups, but the overall structure of the process is fixed. In this situation, it allows each of the individual groups to retain some
degree of autonomy in regard to the tasks that are actually executed at runtime in their respective parts of the process, whilst still complying with the overall process definition.

**Realization options** An underspecified process definition is deemed to be one which is well-formed but does not have a detailed definition of the ultimate realization of every task. An incomplete process definition contains one or more so-called *placeholders*. Placeholders are nodes which are marked as *underspecified* (i.e. their content is unknown or is specified outside of the process model) and whose content is specified when they are encountered at run-time. We distinguish two types of *placeholder enactment*:

- **Late binding**: where the realization of a placeholder is selected from a set of available process fragments. Note that to realize a placeholder one process fragment has to be selected from an existing set of predefined process fragments or defined via ripple-down rules (RDR)\(^2\). This approach is limited to selection, and does not allow a new process fragment to be constructed.

- **Late modeling**: where a new process fragment is constructed in order to realize a given placeholder. Not only can a process fragment be constructed from a set of currently available process fragments, but also a new process fragment can be developed from scratch. Therefore late binding is complemented by late modeling. Some approaches [196] limit the construction of new definitions by (declarative) constraints.

For both approaches, the realization of a placeholder can occur at a number of distinct times during process execution. Here, two distinct *moments of realization* are recognized:

- **before placeholder execution**: the placeholder is realized at the commencement of a process instance or during execution before the placeholder has been executed for the first time.

- **at placeholder execution**: the placeholder is realized when it is executed.

Placeholders can be either realized once, or for every execution of the placeholder. We distinguish two distinct *realization types*:

- **static realization**, where the process fragment chosen to realize the placeholder during the first execution is used to realize the placeholder for every subsequent execution.

- **dynamic realization**, where the realization of a placeholder can be chosen for each execution of the placeholder.

The following two types of flexibility facilitate the incorporation of execution behaviors at run-time that have not been anticipated at design-time. Depending on the extent to which the realization of flexibility by change applies, we can distinguish flexibility by momentary change and flexibility by permanent change. These two types of flexibility by change are described in detail below.

### 6.1.4 Flexibility by momentary change

**Motivation** In some situations, when executing a given process instance, the need to achieve specific behavior that has not been foreseen at design-time may arise. Sometimes

\(^2\)If late binding is used based on rules (cf. RDR in Worklets [19]), then one could argue that this is just an XOR-split. In principle, this is correct, however we still consider this flexibility by underspecification.
such behavior cannot be facilitated by temporary deviations from the existing process definition, but requires the process definition for a particular process instance to be modified.

**Definition** *Flexibility by Momentary Change* is the ability to modify a process definition at run-time such that the process definition associated with a given process instance is amended in order to realize previously not foreseen behavior.

**Scope** Flexibility by change allows processes to adapt to transient changes that are identified in the operating environment. Changes are introduced at the level of the process instance and do not affect other process instances.

**Realization options**

*Moment of allowed change* specifies the moment at which changes can be introduced in a given process instance.

- **Entry time**: changes can be performed at only the moment the process instance is created. After the process instance has been created, no further changes can be introduced to the given process instance. Momentary changes performed at entry time affect only a single process instance.

- **On-the-fly**: changes can be performed at any time during process execution. Momentary changes performed on-the-fly correspond to customization of a given process instance during its execution.

### 6.1.5 Flexibility by permanent change

**Motivation** In some cases, events may occur during process execution that were not foreseen during process design. Sometimes these events cannot be addressed by temporary deviations from the existing process definition, but require the addition or removal of tasks or links in the process definition on a permanent basis. This necessitates changing the process definition for all currently executing instances.

**Definition** *Flexibility by Permanent Change* is the ability to modify a process definition at run-time such that all currently executing process instances are migrated to a new process definition and all new process instances utilize the new process definition. The process definition constructed at design time is modified and all executing process instances need to be transferred from the old to the new process definition.

**Scope** Flexibility by change allows processes to adapt to changes that are identified in the operating environment. Changes are introduced at type level resulting in the modification of the process definition.

**Realization options**

*Moment of allowed change* specifies the moment at which changes can be introduced in a given process instance or a process definition.

- **Entry time**: changes can be performed only at the moment the process instance is created. After the process instance has been created, no further changes can be made to the given process instance. The result of permanent change performed at entry time is that all new process instances have to be started after the change to the process definition has been performed, and no existing process instances are affected (they continue execution according to the process definition with which they are associated).

- **On-the-fly**: changes can be performed at any time during process execution. Permanent changes performed on-the-fly impact both existing and new process instances.
The new process instances are created according to the new process definition, while the existing process instances may need to be migrated from the existing process definition to the new process definition.

*Migration strategy* defines what to do with running process instances that are impacted by a permanent change.

- **Forward recovery**: affected process instances are aborted.
- **Backward recovery**: affected process instances are aborted (compensated if necessary) and restarted.
- **Proceed**: changes introduced are ignored by the existing process instances. Existing process instances are handled the old way, and new process instances are handled the new way.
- **Transfer**: the existing process instances are transferred to a corresponding state in the new process definition.

We have described the five flexibility types by illustrating the motivation for each of them, giving their definition and defining the scope that each of them impact in a process. However, when it comes to the realization of these flexibility types in practice, it is unclear what the typical constructs required for realizing particular types of flexibility are, what the semantics of these constructs are and in which situations and domains they are applicable. In the next section, we address this issue by defining a set of process flexibility patterns.

### 6.2 Catalog of process flexibility patterns

In this section, we present the set of flexibility patterns that have been identified. The patterns are divided into eight groups (cf. Table 6.1). Each group addresses a common problem and is intended to describe a specific aspect of process flexibility such as *flexible initiation, flexible termination, flexible reordering, flexible selection, flexible elimination, flexible extension, flexible concurrency* and *flexible repetition*. For instance, flexible elimination patterns aim to facilitate flexibility by avoiding the execution of a particular task. This can be done already during design-time by incorporating a bypass path, at run-time by skipping the execution of a currently enabled task, or by removing this task from the process definition.

Each of the pattern groups consist of at most five patterns, and each pattern corresponds to one of the five flexibility types: flexibility by design, flexibility by deviation, flexibility by underspecification, flexibility by momentary change or flexibility by permanent change. This provides an overall structure of $8 \times 5$ possible patterns. However, not all desired aspects of flexibility can be mapped to the notions associated with the five flexibility types, therefore some pattern groups contain less than five patterns. When describing each of the pattern groups, we will indicate which configurations are meaningful and which are not. Note that the goal of the process flexibility patterns is not to fill in the process flexibility matrix, but rather to use it as a guide to meaningful types of flexible execution mechanisms.
Overview of process flexibility groups

<table>
<thead>
<tr>
<th>Group name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>The ability to initiate a process instance from a point other than the nominated entry-point to the process definition</td>
<td>page 334</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>The ability to terminate a business process on a premature basis</td>
<td>page 348</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>The ability to select an execution path appropriate to the operational circumstances</td>
<td>page 358</td>
</tr>
<tr>
<td>Flexible reordering</td>
<td>The ability to establish alternative execution ordering for the tasks in a process at run-time</td>
<td>page 369</td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>The ability to avoid the execution of a particular task</td>
<td>page 377</td>
</tr>
<tr>
<td>Flexible extension</td>
<td>The ability to enable an execution path alternative to the one prescribed by the process definition by incorporating a task that has not previously been foreseen</td>
<td>page 383</td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td>The ability to avoid unnecessary dependencies between several independent tasks by executing them concurrently</td>
<td>page 392</td>
</tr>
<tr>
<td>Flexible repetition</td>
<td>The ability to repeat the execution of a particular task variable number of times</td>
<td>page 399</td>
</tr>
</tbody>
</table>

**Table 6.1:** The Process Flexibility Matrix used to position all patterns in this chapter
6.2.1 Context assumptions

A process definition consists of a set of tasks and a set of constraints specifying the order in which these tasks have to be executed. Figure 261 shows the graphical notation we will use to visualize a process definition. Typically, a process definition contains a single start task, i.e. a task from which process execution begins, and a single end task, i.e. a task whose completion results in the termination of the process instance. Process elements that are left underspecified at design-time are marked as placeholders. We will use a square with a solid and dashed line to denote placeholders (cf. Figure 261). For some patterns we also need to differentiate individual execution instances. We will use a symbol in the form of a shaded triangle, circle or square to denote (partial) states associated with different process instances (i.e. tokens referring to particular instances). Such a symbol preceding a task in the process definition indicates that this task is enabled for execution.

To show the effect of applying a flexibility pattern to a particular process instance at run-time, both the process definition before and after applying this pattern are visualized. We depict process definitions associated with different process instances separately to show the effect of momentary changes, i.e. changes resulting in modification of the process definition associated with a particular process instance. To denote that changes of a permanent nature, performed at the type level, impact all existing process instances, we show a single process definition common to all process instances before and after the permanent change has been performed. To illustrate different execution paths within a process definition, each task may have associated split and join connectors of XOR or AND type or may have no split/join behavior at all.

![Figure 261: Process entities](image)

To describe the context in which the flexibility patterns can be used and to explain the operational semantics of our basic notations, we present a basic engine for executing process instances based on the process definitions using the CPN formalism that has been extensively described in Chapter 3. Figure 262(a) illustrates the top-level view of the basic engine able to handle both concrete and placeholder tasks. Two substitution transitions Engine and Placeholder definition interface correspond to the nets depicted in Figure 262(c) and (b) respectively.

Process definitions for which process instances will be created are stored in the Process definition place. In order to differentiate between distinct models, they are associated...
with identifiers of type ModelID to refer to a particular process model. The process definitions are of type ProcModel, and contain information about the model id (of type ModelID), the id of the start task, the id of the end task, a list of the tasks in the process definitions and a list of arcs between the tasks. The execution of a process starts with the start task specified in the process definition, and completes when no tasks are left which can still be executed.

Each task is defined as a tuple of the Task type, specifying the id of the task, the type of the task (i.e. whether it is concrete or underspecified), the type of the join connector and the type of the split connector. The join connector and the split connector are of type ConnectorType, which specify for a given task whether it needs to synchronize multiple incoming branches and produce output to multiple outgoing branches. Where both connectors have value none, the task can be characterized as sequential. The connector values XOR and AND represent XOR/AND-splits and XOR/AND-joins explained earlier on page 122 and page 131 respectively. The definition of the main data types used in Figure 262 is visualized in Table 6.2.

To create a process instance, a user places a token of type ProcInst in the Begin place, specifying the id for the process definition that needs to be populated (i.e. mid of the ModelID type), and the id that has to be assigned to the process instance being created.

![Diagram](image-url)

(a) Top view

(b) Placeholder engine

(c) Execution engine

Figure 262: Basic engine
Section 6.2 Catalog of process flexibility patterns

(i.e. pid of the piID type).

A process instance is created by the engine based on the input (mid,pid) provided by
the user and the matching model m. The Create process instance transition matches
mid, (the id of the model provided by the user), with the id associated with the process
definition m available in the Process definition place using the match(mid,m) function.
Where these match, the createinst(pid,m) function creates a process instance of type
ProcInstState. The process instance contains the id for the specific process instance pid,
a replica of the process definition m, and the current state st indicating which task is
currently enabled. Note that this does not imply that we assume an implementation where
the process model is replicated for every instance, i.e. the model could also be passed by
reference.

A task in an activated process instance, information about which is stored in the
Running instance place, can be executed only if the task is concrete (i.e. the content
of the task has explicitly been defined at design-time). If the task has been left underspec-
ified (i.e. it has the placeholder type), first the definition of this task has to be completed
by the placeholder engine illustrated in Figure 262(b). The placeholder may be completed
by creating a new process fragment, by selecting an element from the repository of pre-
defined process fragments or by composing a new process fragment. By executing one of
the transitions Create, Select or Compose, the task that previously had the placeholder
type is replaced with the desired content and its task type is set to concrete. From this
moment on, the Execute task transition in the basic engine (cf. Figure 262(c)) may fire
when the Boolean function existsEnabledTask(pid,m,st) evaluates to True, indicating
that there exist enabled tasks which can be executed.

The task to be executed next is selected non-deterministically from the set of enabled
tasks (note that we abstract from work distribution, etc. and hence can consider this
choice to be non-deterministic). The set of enabled tasks is formed by tasks whose enabling
conditions are satisfied. A task without a join connector is enabled if the task preceding it
has been executed and the arc connecting these two tasks is present in the state associated

<table>
<thead>
<tr>
<th>Table 6.2: Data types used in Figure 262</th>
</tr>
</thead>
<tbody>
<tr>
<td>colset ModelID = string;</td>
</tr>
<tr>
<td>colset piID = int;</td>
</tr>
<tr>
<td>colset ProcInst = product ModelID * piID;</td>
</tr>
<tr>
<td>colset TaskID = smallstr;</td>
</tr>
<tr>
<td>colset TaskType = with concrete</td>
</tr>
<tr>
<td>colset ConnectorType = with XOR</td>
</tr>
<tr>
<td>colset Task = product TaskID * TaskType * ConnectorType^3 * ConnectorType^4;</td>
</tr>
<tr>
<td>colset Tasks = list Task;</td>
</tr>
<tr>
<td>colset Link = product TaskID * TaskID;</td>
</tr>
<tr>
<td>colset Arcs = list Link;</td>
</tr>
<tr>
<td>colset ProcModel = product ModelID*TaskID^5 *TaskID^6 <em>Tasks</em>Arcs;</td>
</tr>
<tr>
<td>colset State = Arcs;</td>
</tr>
<tr>
<td>colset ProcInstState = product piID * ProcModel * State;</td>
</tr>
</tbody>
</table>

^3 Task split  
^4 Task join  
^5 Start task  
^6 End task
with the given process instance (as indicated by attribute st of type State). If a task has a join connector of XOR type, the process instance state must contain at least one enabled arc; otherwise, if the task has a join connector of AND type, all incoming arcs must be enabled and thus be present in the state st. After the task has been executed, the ns(pid,m,st) function determines the new state for the process instance. If the split connector associated with the executed task is of type XOR, one of its output arcs is selected non-deterministically and added to the state. If the split connector is of type AND, then all outgoing arcs are added to the state. The execution of the process instance completes when no enabled tasks are left, i.e. the existsEnabledTask(pid,m,st) function evaluates to False.

We will use the graphical notation described in Figure 261 to depict the flexibility options that are characterized by the flexibility patterns. To describe the operational semantics of the patterns, we illustrate the basic engine depicted in Figure 262 that is enhanced with flexibility extensions specific to the pattern being considered, thus focusing only on the differences between the basic and the enhanced engines in each case.

We will describe each of the process flexibility patterns using the same pattern format that was used in Chapter 4 for describing the control-flow patterns (cf. page 118):
- description: a summary of its functionality;
- examples: illustrative examples of its usage;
- motivation: the rationale for the use of the pattern;
- overview: a graphical notation illustrating the pattern and an explanation of its operation;
- context: a detailed operational definition of the pattern in terms of CPNs, illustrated as an enhancement of the basic process engine;
- implementation: how the pattern is typically realized in practice 7;
- issues: problems potentially encountered when using the pattern;
- solutions: how these problems can be overcome; and
- evaluation criteria: the conditions that an offering must satisfy in order to be considered to support the pattern.

We start with a description of patterns related to flexible process initiation.

6.2.2 Flexible initiation

This group of patterns aims to describe flexibility options related to the initiation of a business process, i.e. flexibility considerations that influence the manner in which process instances are created. Figure 263 illustrates the scope of patterns presented in this subsection and their relationship to the different types of flexibility outlined in Section 6.1. We will traverse cells in the highlighted row from left to right and analyze the mapping of the concept of flexible process initiation to each of these flexibility types.

Usually a process definition has a single entry point which triggers the initiation of a new process instance. An example of such a process is shown in Figure 264. The execution of this process starts with task A each time an instance of this process is initiated. Such an approach to process definition forces the execution of all associated process instances to start with the same task. In some situations, the execution of the process may need to start from a task other than the nominated start task prescribed by the process definition.

---

7We describe the implementation options for Oracle BPEL PM, CIG modeling languages and four systems illustrating different kinds of support for process flexibility (ADEPT1, FLOWer, YAWL, and Declare.)
Section 6.2 Catalog of process flexibility patterns

<table>
<thead>
<tr>
<th>Flexible initiation</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible termination</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible elimination</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible extension</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 263:** Process Flexibility Matrix: flexible initiation

For rigid process definitions with a single start task this is impossible, thus the start task must be executed even if it is not required.

**Figure 264:** Example of a process definition

Depending on the moment at which the need for an alternative entry-point to a process definition is recognized and the manner in which it is facilitated, we distinguish the following five patterns: *Alternative Entry-Points, Entrance Skip, Undefined Entry, Momentary Entry Change* and *Permanent Entry Change*.

Although the goal of all of these five patterns is to allow a user to start execution of a process instance from a task different to the nominated start task, these patterns differ in the moment at which the need for flexibility is anticipated. The *Alternative Entry-Points* pattern is associated with a process at design-time (i.e. all decisions allowing for flexible process initiation are defined before a process instance has been created), while the other four patterns are associated with its run-time execution. In the *Entrance Skip* pattern, the need to deviate from the nominated start task is only anticipated at the moment a process instance is created. In the *Undefined Entry* pattern, the need for flexible process initiation is anticipated at design-time, however its realization is accomplished at run-time. In the *Momentary Entry Change* and *Permanent Entry Change* patterns, the need for flexible process initiation is anticipated and realized at run-time by changing the process definition at the instance or type level.

**Pattern PF-1 ALTERNATIVE ENTRY-POINTS**

**Description** A process definition contains more than one start task, each of which represents an alternative entry point for the process definition. Any of the start tasks can be selected by a knowledgable user at run-time when creating a new process instance causing the process instance to commence from that task. This pattern characterizes a need for flexible process initiation that is anticipated at design-time, thus it corresponds to the *flexibility by design* type.
Examples
– The medical processes for handling patients in a hospital are defined in such a way that a patient may commence at any stage of the treatment process depending on their needs and their current state of health.
– In order to be admitted to the driving exam, candidates first have to pass a verbal theory exam. However, a candidate who already possesses a certificate for the theoretical examination may advance to the practical examination directly.

Motivation
The majority of structured-workflow systems prescribe a single start task as the entry point to a process definition. Despite differences in the context in which distinct process instances have to operate, the process definition created in such workflow systems enforces that the same start task has to be executed by each of the process instances. In some situations, it is necessary to skip the beginning of the process and commence execution from another task contained in the process definition. The **Alternative Entry-Points** pattern applies to situations when different execution paths, each having a distinct start task, are foreseen during the design of a process definition.

Overview
Figure 265 illustrates two process notations: the top one corresponds to a rigid process with a single entry point, and the bottom one corresponds to a process with two alternative entry points, either of which can be selected to initiate the process from the associated start task.

Context
Figure 266 illustrates the process engine expressed using the CPN formalism extended to support the **Alternative Entry-Points** pattern.

Figure 265: Alternative Entry-Points pattern

Figure 266: Engine enhanced with the Alternative Entry-Points pattern

To allow for process initiation from different start tasks, multiple start tasks have to
be included in the process definition and are stored in the Process definition place. For this, the ProcModel type is modified to ProcModel' as shown below (note that this is different from the original process definition which contains only a single start task).

\[
\text{colset TaskIDs = list TaskID};
\]
\[
\text{colset ProcModel'} = \text{product ModelID * TaskIDs * TaskID * Tasks * Arcs};
\]

In order to create a process instance, a knowledgable user supplies information in the form \((\text{mid}, \text{pid}, \text{tid})\), where \(\text{mid}\) identifies the process definition based on which a process instance needs to be created, \(\text{pid}\) identifies the process instance to be created, and \(\text{tid}\) identifies the desired start task from which the given process needs to be initiated. For this, the ProcInst type is modified to ProcInst' in order to include the identifier of the start task:

\[
\text{colset ProcInst'} = \text{product ModelID * piID * TaskID};
\]

The information provided by the user is used in the guard for the Create process instance transition, which, based on the input provided, identifies the process definition with the corresponding identifier and creates a process instance with the indicated start task \(\text{tid}\). The \(\text{isStartTask}(\text{tid}, \text{m})\) function is used to check whether the task identifier \(\text{tid}\) provided by the user corresponds to one of the start tasks defined for the indicated process definition. If the guard conditions are satisfied, the \(\text{createinst}'(\text{m}, \text{pid}, \text{tid})\) function creates a process instance which can be executed until the nominated completion conditions are met. Note that the \(\text{createinst}'(\text{m}, \text{pid}, \text{tid})\) function is different from the original \(\text{createinst}(\text{m}, \text{pid})\) function in terms of the parameters it requires, and in order to create a process instance, a start task must be provided by the user.

**Implementation** Imperative systems intended for modeling structured workflows typically do not support this pattern directly, since they require a single entry point to the process definition that must be executed for all populated process instances. This applies to YAWL, FLOWer and ADEPT1. To realize multiple entry points in such systems, one has to introduce a dummy choice construct that allows one of the tasks associated with the choice branches to be selected.

In contrast, declarative systems such as Declare allow any task, independent of the completion status of other tasks in the process definition, to be selected as a start task. CIG modeling languages such as EON and GLIF also allow multiple tasks to be marked as start tasks and depending on the patient’s state an appropriate task can be selected from which a new process instance can commence. PROforma combines both imperative and declarative approaches and allows any unconstrained task to be selected as a start task. In Oracle BPEL PM, different \(<\text{receive}>\) activities can initiate a process instance by setting the createInstance attribute to yes.

**Issues** If, during run-time, a required start task cannot be found because the corresponding execution path has not been foreseen at design-time, a user may need to choose the most suitable of the foreseen execution paths, and execute a set of tasks, which normally would not need to be executed, in order to reach the required starting point in the process. By “jumping” into the process semantic problems, e.g., deadlocks and missing data, may occur.

**Solutions** One could solve this issue by applying the Entrance Skip pattern (cf. page 338) in order to deviate from the executing tasks prescribed by the selected execution path. Alternatively, the Undefined Entry pattern (cf. page 340) could be applied in order to determine the beginning of the process at run-time. Finally, the Momentary Entry-Point
Change pattern (cf. page 343) or the Permanent Entry-Point Change pattern (cf. page 345) could be applied to change the entry-point prescribed by the process definition.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows several alternative tasks to be nominated for a process definition each of which may play the role of the start task when a process instance is initiated.

**Pattern** PF-2 ENTRANCE SKIP

**Description** At process initiation, there is the possibility of deviating from the execution path prescribed by the process definition by ignoring the nominated start task. The execution of a process instance may start from any subsequent task specified in the process definition. The pattern assumes that the act of skipping the beginning of the process is a deviation that has no effect on the process definition. This pattern characterizes a need for flexible process initiation that is anticipated at the moment of process instance creation, thus it corresponds to the *flexibility by deviation* type.

**Examples**
- Treatment of patients at a hospital starts with the registration of a patient. Patients who have been registered or are already being treated may immediately proceed to the required health unit.
- A housing agency identifies tenants to whom available accommodation will be allocated based on the length of time that they have waited. The normal selection process is skipped if a new client has requested accommodation under urgent conditions.

**Motivation** In the *Alternative Entry-Points* pattern (described on page 335), flexibility in process initiation is achieved by specifying alternative entry-points for a process definition at design-time. In some situations, where no alternative entry-points have been nominated at design-time or where no suitable entry-point can be found at run-time, the desired start task can only be reached by actually executing all tasks preceding it. This necessitates the execution of a number of tasks, which normally would be omitted. The *Entrance Skip* pattern allows the execution of a given process instance to start with the commencement of a desired task, by skipping all tasks on the path leading to it.

**Overview** Figure 267 illustrates the graphical notation for the *Entrance Skip* pattern. The top view illustrates an initiated process instance before applying the pattern. In this process instance, the thread of control, which is visualized by the filled triangle, indicates the task that is currently enabled (in this case, the start task A is enabled).

The bottom view, illustrates the process instance after the *Entrance Skip* pattern has been applied. The process definition associated with the created process instance remains the same, however instead of enabling the start task A, the thread of control is moved to a subsequent task C. Thus the tasks A and B are skipped in order to commence the process instance at task C.

![Figure 267: Entrance Skip pattern](image-url)
**Context**  Figure 268 illustrates the process engine expressed using the CPN formalism and shows how the *Entrance Skip* pattern should be realized.

In order to allow for process initiation from a task different to the nominated start task, a user needs to supply the identifier of the desired start task $tid$. For this, the information provided by a user at process initiation (which is of type $ProcInst$) is modified to $ProcInst'$ type as follows:

$$colset ProcInst' = product ModelID * piID * TaskID;$$

When the identifier for the desired start task $tid$ together with the corresponding model identifier and the identifier for the process instance to be created are made available by a user in the *Begin* place, this data is used by the *Create process instance* transition to create a process instance with the nominated start task or by the *Entrance Skip* transition to create a process instance without a nominated start task. As described for the *Alternative Entry-Point* pattern, in this engine the $isStartTask(tid,m)$ function is introduced to check whether the identifier $tid$ provided by the user corresponds to a nominated start task in the model $m$. Furthermore, the $createinst(m,pid,tid)$ function is used to create a process instance, where the start task identifier is $tid$.

---

**Implementation**  Of the wide range of contemporary offerings investigated, only FLOWer and Declare provide support for the *Entrance Skip* pattern. In Declare, a set of tasks including the start task can be skipped by ignoring optional constraints. Although in FLOWer it is not possible to advance to a desired start task directly, the same effect can be achieved by executing a series of skip operations. In ADEPT1 and YAWL, it is not possible to skip start tasks at process initiation, neither is this possible in Oracle BPEL PM. However, in Declare if none of the tasks has been marked as a start task, the process execution may start from any of the tasks defined for a process model. The decision-support systems used for enactment of clinical guidelines such as PROforma provide recommendations in the form of decisions where zero, one or more options can be selected. These can be used to encode decisions related to the execution of a certain task in the process, thus they are also applicable for describing optional start tasks. In Asbru, EON and GLIF, configuration with respect to a desired start task cannot be done at process initiation.

**Issues**  When the execution of a specific task needs to be skipped, and a subsequent task needs to be enabled, problems related to data dependencies between these tasks may arise.
Often in order to be enabled, a task requires input data which is provided by the previous task. When a task is skipped, the required data may not be available to the subsequent task.

**Solutions** In order to avoid the problem of required data elements from preceding tasks not being available at task enablement, it is possible to define a default value for data elements and use this instead.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip one or more tasks, including the start task, at process initiation.

**Pattern** PF-3 **UNDEFINED ENTRY**

**Description** A process definition contains a placeholder, associated with the beginning of the process, that is intentionally left *underspecified*. During process initiation there is the possibility of completing the specification of this placeholder with an appropriate start task. This pattern characterizes the need for flexible process initiation recognized at design-time, but whose actual realization is performed at run-time, thus it corresponds to the *flexibility by underspecification* type.

**Examples**
- The accounting program developed to calculate financial metrics which track the performance of a business can be supplied to and used in any organization. Depending on the context in which the program is to be used, the initial information which needs to be provided when initiating the accounting program may vary, thus allowing it to be tailored to a specific customer.
- An IDEAL-application, providing on-request access to bank accounts via Internet, can be used at various corporate web-sites in order to complete the process of purchasing a product ordered by clients via Internet. An interface for launching this application needs to be configured by organizations depending on their needs.

**Motivation** Often in situations where the need for flexible process initiation is recognized at design-time, a set of possible start tasks is defined, each corresponding to an alternative execution path. The *Alternative Entry-Points* pattern (described on page 335) allows different entry-points to a process to be specified by explicitly defining distinct start tasks at design-time. In some situations, it is not always clear at design-time which of the tasks may need to play the role of the start task or it may be impractical to specify all possible start tasks explicitly. The *Undefined Entry-Point* pattern allows the specification of the beginning of the process, including the start task, to be postponed until process initiation when more information related to the operational context becomes available.

**Overview** Figure 269 illustrates the graphical notation for the *Undefined Entry* pattern. The top view illustrates a process instance that has been populated from the process definition where the beginning of the process is left underspecified and marked as a placeholder. The static part of the process which consists of tasks C and D has been predefined at design time, and thus cannot be executed until the beginning of the process is defined. The bottom view visualizes the completion of the placeholder performed at runtime after applying the pattern. After process instance creation, the placeholder becomes enabled as illustrated by the execution thread preceding it. Once the placeholder is enabled, it needs to be completed. Once the content of the placeholder has been defined, the thread of control is moved to the specified start task. In this case, task B is chosen as a start task,
however a process fragment consisting of tasks A and B could have been used instead. Note that the mechanism for completing the placeholder associated with the beginning of the process is the same as that for any other placeholder in the process definition.

**Figure 269:** Undefined Entry pattern

**Context** In this section, we illustrate how flexibility in process initiation as promoted by the Undefined Entry pattern can be incorporated in the process engine. In order for the definition of a start task to be completed at run-time, the start task needs to be left underspecified in the design-time process definition and the type of this start task has to be set to placeholder.

**Figure 270:** Placeholder engine: top view

After a process has been initiated by the basic process engine presented in Figure 262(c) no tasks may be executed until the definition of the underspecified start task is completed by the placeholder engine whose main view is presented in Figure 270. In the placeholder engine, a placeholder for a given process instance can be completed either by creating a new task (e.g., the Create substitution transition), selecting a process fragment from a set of process fragments defined at design-time (e.g., the Select substitution transition), or by composing a new process fragment from existing and/or new tasks (e.g., the Compose substitution transition).

In order for a new task to be created, the substitution transition Create, whose functionality is presented in Figure 271(a), needs to be executed. The new task created by the createnewtask() function is added to the repository of process fragments by the addtasktorepos() function. The content of the selected placeholder selp is replaced by
the newly created task. In particular, the \texttt{update\_create()} function modifies both the process definition, and also updates the state of the process instance so that the created task can be executed.

When the placeholder needs to be completed with one of the previously defined process fragments, the \textbf{Select} substitution transition needs to be executed. The behavior corresponding to this transition is presented in Figure 271(b). A process element \(tp\) can be selected from the repository of process fragments \(lrep\) available in the \textbf{Repository} place on the assumption that the repository is non-empty. A process fragment, that has been randomly selected from the repository, is used to substitute the given placeholder in the process definition and in the process instance state. The update of the process definition and process instance state is performed via the \texttt{update\_select()} function.

Finally, the content of a placeholder can be defined by composing a process fragment from new tasks and process fragments available in the \textbf{Repository} place. In order to compose a process fragment, the \textbf{Compose} transition, whose behavior is shown in Figure 271(c), needs to be executed. A process fragment that will be used for completion of the selected placeholder is formed by the \textbf{Select} and \textbf{Create} transitions, and stored in the \textbf{process fragment} place. The functionality of these transitions corresponds to the ones described earlier for task creation and process fragment selection. The created and selected elements are coupled to each other in sequential order. The content of the placeholder is then replaced by the process fragment obtained and the process state is updated by the
update_compose() function in such a way that the first task in this fragment becomes enabled.

**Implementation** From the set of workflow offerings analyzed, only YAWL supports this pattern through its worklets extension [19]. A worklet is a reusable process fragment consisting of one or more tasks that can be included in the process at run-time. None of the other systems investigated, i.e. ADEPT1, Declare, FLOWer or Oracle BPEL PM, provide any means of realizing of the Underfined Entry pattern. Among CIG modeling languages, only PROforma provides a similar concept, where at design-time a keystone construct representing a generic task can be used. However, before deploying the process definition the keystone has to be specified explicitly. Thus the moment of specification is postponed not until run-time but until the latest possible moment at design-time.

**Issues** The repository of fragments used to determine the start task for a given process instance at run-time should not be empty, i.e. it must contain at least one process fragment if the placeholder is to be completed by selecting a fragment from the repository rather than by creating a new process fragment. An empty repository may potentially lead to process instances blocking.

**Solutions** When the definition of a placeholder at runtime cannot be completed because the repository of process fragments is empty, it should be possible to select a process fragment with empty content. The execution of a placeholder with empty content can be treated as if this placeholder would not exist or its completion would be skipped.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows an incomplete process definition to be enacted. An underspecified process-entry fragment must be explicitly marked as a placeholder at design-time, and it should be possible to complete its definition at run-time either before or when the placeholder is enabled.

**Pattern** PF-4  **MOMENTARY ENTRY-POINT CHANGE**

**Description** At process initiation, i.e. before the start task prescribed by the process definition has commenced, there is the possibility to change the entry-point for the process by modifying the process definition associated with the particular process instance being created. This pattern characterizes a change to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

**Examples**
- A patient transported from one hospital to another is accompanied by his/her X-ray photos. For this patient the X-ray examination should not be performed, and he/she should immediately proceed to surgery.
- A typical boarding procedure for clients at an airport is modified for premium clients according to their personal requirements (e.g., the issuing of the boarding pass and visa arrangement can be omitted to allow the client to proceed to the gate immediately).

**Motivation** In situations, where at process initiation no suitable start task can be found in the set of foreseen start tasks and it is not possible to deviate from the prescribed execution path by ignoring the nominated start task, the prescribed start task and any subsequent tasks have to be executed until the desired task is reached. The Momentary Entry-Point Change pattern allows the process definition associated with the given process
instance to be modified in such a way that its execution may start with any task in the prescribed execution path.

**Overview** Figure 272 illustrates the graphical notation for the *Momentary Entry-Point Change* pattern. The top view depicts two distinct process instances populated from the same process definition. Each instance has a process definition associated with it that is used to determine the next task to be executed. The thread of control in each of the process definitions is illustrated by the shaded triangle and circle respectively. The pattern is to be applied to a process instance whose execution thread is depicted by the shaded triangle.

![Figure 272: Momentary Entry-Point Change pattern](image)

The momentary entry-point change that is performed for one of the process instances, does not affect the process definition associated with other process instances. The bottom view illustrates that after task A and task B have been removed from one process instance, the thread of control associated with this instance is moved to the subsequent task, whilst other process instances are unaffected.

**Context** Figure 273 illustrates how the process engine expressed using the CPN formalism needs to be extended in order for flexibility for process initiation as demonstrated by the *Momentary Entry-Point Change*.

In order to incorporate the functionality for supporting flexibility by momentary change, we will extend the basic process engine as shown in the Figure 273. For context purposes, we will modify the name of transition performing the modification of the particular process instance, however the names of functions used will remain unchanged. Note that the structure of the engine allowing for momentary changes is uniform for all patterns enhancing the flexibility by momentary change. The main difference between the realization of these patterns is the functionality associated with the `change_possible()` and `update_pi()` functions. In order for the process definition associated with the given process instance to be changed, a set of conditions incorporated into the `change_possible()` function have to be satisfied. The `update_pi()` function makes the actual modifications in the process definition, and whenever necessary also updates the state of the process instance.

In the *Momentary Entry-Point Change* pattern, the process definition associated with the process instance stored in the *Running instance* place which has not commenced yet can be modified in order to start execution from a task other than the nominated start task. The `change_possible()` function checks whether there are enabled tasks that can be executed and whether the task enabled is the nominated start task. This function would evaluate to `False` if the execution of the process instance has commenced and the start task has already been executed.

When the guard associated with the *Momentary entry change* transition is satisfied, it can be executed. The `update_pi()` function updates the process instance in the following
way. First, an arbitrary task from the set of tasks associated with the given process definition \( m \) is selected. This task will be set as the new start task in the original process definition. All tasks preceding the newly selected start task will not be executed any more, therefore these tasks and all branches associated with them are removed from the process model. Then the process instance state is updated, i.e. the newly selected start task becomes enabled.

Note that this engine is realized based on the assumption that momentary changes are performed on-the-fly, i.e. after the process instance has been created.

**Implementation** ADEPT1 and Declare allow the Momentary Entry-Point Change pattern to be applied after a process instance have been created and before the start task prescribed by the process definition has been enacted. In ADEPT1, a desired task cannot be initiated directly and requires a number of delete operations to be performed. In Declare, a start task can be deleted and execution may start from another task. None of FLOWer, Oracle BPEL PM, YAWL or the clinical guideline languages allow for changes to be performed at run-time at the process instance level.

**Issues** When the beginning of a process is being removed from the process definition in order for process execution to commence from another task, the problem may arise that data elements are missing that were previously provided by the tasks that have been removed.

**Solutions** In order to allow the beginning of a process to be inconsequentially removed, each of the tasks in a process definition that may play the role of a start task must be associated with a default value which could be used for enabling the execution of the given task if no other input has been or can be provided.

**Evaluation Criteria** Full support for this pattern is demonstrated by an offering that allows any task (including the start task) to be deleted from the process model associated with a given process instance at run-time. It should be possible to mark a task other than the dedicated start task as the entry point to the process definition.

**Pattern** PF-5 PERMANENT ENTRY-POINT CHANGE

**Description** At run-time, the possibility exists to permanently change the entry point for a process by modifying the process definition. This pattern characterizes a change
that affects all future process instances directly and requires existing process instances to migrate from the old to the new process definition, thus it corresponds to the \textit{flexibility by permanent change} type.

**Examples**

- Visitors traveling to Eastern Europe are expected to acquire a visa before making ticket reservations. For countries that joined the European Union, the visa requirement is abolished. Visitors already possessing a visa continue traveling as usual. For new visitors however, the travel organization may start directly with ticket reservation.

- The masters program for international students used to start with a half-year qualification period. Due to financial reasons, the reorganization of a school is performed requiring canceling the half-year qualification program for new generations of students. This change aims at increasing the level of accepted applicants, and does not affect the students who already follow the homologation program.

**Motivation**

When the start task identifying the entry point to a process definition needs to be removed for all future process executions, there may arise the need to permanently modify the process definition by removing the nominated start task. Such an adjustment may be more efficient than applying the \textit{Momentary Entry-Point Change} pattern (described on page 343) to each newly created process instance. The \textit{Permanent Entry-Point Change} pattern allows the process definition to be modified at a type level and the execution of process instances to start from a task subsequent to the nominated start task.

**Overview**

Figure 274 illustrates the graphical notation for the \textit{Permanent Entry-Point Change} pattern. The top view depicts the execution state of several process instances (based on the same process definition) before applying the pattern.

The bottom view illustrates that for all process instances the process definition has been modified by removing task A and task B. The thread of control associated with the process instances where these tasks have not been executed yet is moved to the subsequent task C. The process instance visualized by the square that has already passed this point is not affected by the change.

**Context**

Figure 275 illustrates how the process engine expressed using the CPN formalism needs to be extended in order to cater for the \textit{Permanent Entry-Point Change} pattern.

In order to incorporate the functionality for supporting flexibility by permanent change, we will extend the basic process engine as shown in the Figure 275. For context purposes, we will modify the name of transition performing the modification of the particular process instance, however the names of the functions used will remain unchanged. Note that the structure of the engine allowing for changes at the type level is uniform for all patterns.
enhancing the flexibility by permanent change. The main difference between the realization of these patterns is the functionality associated with the `change_possible()`, `modify_m()`, `transfer_possible()` and `migrate()` functions. In order for the process definition to be changed on the type level, a set of conditions incorporated into the `change_possible()` function have to be satisfied. The `modify_m()` function makes the actual modifications in the process definition. After the process definition has been modified, the migration of existing process instances may need to be performed. The `transfer_possible` function checks whether the migration can be performed, and if it can, then the `migrate()` function updates the process definition and process state associated with a given process instance.

In the Permanent Entry-Point Change pattern, the possibility to modify the start task associated with the process definition at run-time. The Permanent entry change transition associated with this functionality can be executed only if there exist a task that can be selected as a new start task. This realization assumes that the process definition must contain at least two tasks, including the start task and the end task. The `modify_m()` function changes the original start task to the new start task that has been selected on a random basis from the set of available tasks. The original start task, all tasks preceding the newly selected start task as well as any related branches are removed from the process definition.

The migration of existing process instances in the Running instance place to the new process definition can be performed only if there are enabled tasks in the process instance. For the process instances that have been initiated already, this change is inconsequential, however for process instances that have not yet commenced the change performed may impact the process execution. The actual migration strategy defining the mapping of existing process instances to the new process definition is encoded in the `migrate()` function. Process instances whose original process definition `m` differs from the new process definition `newm` may either proceed with execution, be aborted, restarted or transferred to the new definition.

Figure 275: Engine enhanced with the Permanent Entry-Point Change pattern

Implementation ADEPT1 and Declare allow the Permanent Entry-Point Change
pattern to be applied at run-time in order to adjust a process description. Such a change implies the need for process instance migration, which applies both to existing process instances that have not yet commenced and to all future process instances. Existing process instances ignore the change introduced and continue to execute. In ADEPT1, it is not possible to change the entry point for a process in one step, but tasks must be removed one-by-one until the desired start task is reached. Definition of a new process in YAWL via worklets means that this worklet is available to all existing and future process instances via a shared repository. FLOWer, Oracle BPEL PM, and the CIG modeling languages offer no support for permanent changes to process during execution.

**Issues** When existing process instances need to be migrated to a new process definition, a problem known as the ‘dynamic change bug’ may occur [6]. The dynamic change bug is characterized by errors that may occur when transferring an old process definition to a new one (e.g., tasks may be duplicated, omitted, or even deadlock situations may arise).

**Solutions** In order to address the dynamic change bug problem, an approach proposed in [6, 10, 86, 187] can be used. For a given process definition the change region, i.e. the part of the process definition that is affected by the change, is calculated. Each process instance is analyzed in regard to the identified change region. If the thread of control in a process instance is outside of this region, the transfer to the new process definition can be safely performed, otherwise the transfer is postponed until the thread of control in the process instance leaves this region.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task (including the start task) to be deleted from the process definition at run-time, and provides migration facilities for mapping existing process instances from the old to the new process definition. It should be possible to mark a task other than the nominated start task as the new entry point to the process definition.

### 6.2.3 Flexible termination

This group of patterns aims to describe flexibility options when terminating a business process on a premature basis. Figure 276 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility. We will traverse cells in the emphasized row from left to right and analyze the mapping of the concept of flexible process termination for each of the flexibility types.

<table>
<thead>
<tr>
<th>Flexibility by</th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>Alternative entry points</td>
<td>Entrance skip</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative exit points</td>
<td>Termination skip</td>
<td>Undefined exit</td>
<td>Momentary exit change</td>
<td>Permanent exit change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Flexible reordering</td>
<td>Flexible elimination</td>
<td>Flexible extension</td>
<td>Flexible concurrency</td>
<td>Flexible repetition</td>
</tr>
</tbody>
</table>

*Figure 276: Process Flexibility Matrix: flexible termination*
Usually a process definition has an explicit end task, which needs to be executed for a process instance to complete. Such single-exit processes force the execution of all process instances to follow all of the steps prescribed by the process definition until the end task is reached. In some situations, the execution of the end task may no longer be needed at run-time or the execution of the process may need to complete before the specified end task has been reached. Depending on the moment at which the need for an alternative process termination point is recognized and the manner in which it is achieved, we distinguish the following five patterns: Alternative Exit Points, Termination Skip, Undefined Exit, Momentary Exit Change and Permanent Exit Change.

Although these patterns all describe means of achieving premature process termination, they differ in terms of the moment at which the need for such flexibility is anticipated. Of these five patterns, the Alternative Exit Points pattern is relevant at design-time. In this pattern, the need for premature process termination is recognized during process design and all possible options for its realization are incorporated in the process definition. The other four patterns are characterized by the fact that the actual decision in regard to the selection of the process termination point is taken at run-time. In the Termination Skip pattern, the need to deviate from the nominated end-task is identified after process initiation. In the Undefined Exit pattern, the need for flexible process termination is recognized at design-time, however the actual realization of this decision is postponed until the latest possible moment at run-time. In the Momentary Exit Change and Permanent Exit Change patterns, the need for flexible process termination is anticipated and realized at run-time by modifying the process definition at the instance and type levels respectively.

**Pattern** PF-6 ALTERNATIVE EXIT-POINTS

**Description** A process definition specified contains more than one end task, execution of any of which at run-time results in the termination of a given process instance. The availability of alternative exit points in a process definition gives flexibility in terminating the process execution on the premature basis. This pattern characterizes a need for flexible process termination that is anticipated at design-time, thus it corresponds to the flexibility by design type.

**Examples**
- The 6-year education program can be terminated after four, five or six years of study corresponding to attainment of the bachelor, engineer and masters levels respectively.
- The treatment of the cancer consists of several steps: where a patient has to undergo a set of chemotherapies, followed by ablation. A patient may undergo as many chemotherapies as he/she chooses and may stop the treatment at any point if the side-effects become unbearable. For patients who have recovered from the disease after several chemotherapies the treatment process stops immediately.

**Motivation** Typical process definitions contain a single end task. Once initiated such processes may complete only after the nominated end task has been completed. Under conditions which are different from the normal flow of tasks, it might be necessary to prematurely terminate process execution. The Alternative Exit-Points pattern allows several end-tasks to be incorporated in the process definition at design-time, thus providing alternative exit points for process instances. The completion of an end task from the specified set of nominated end-tasks results in the process instance terminating.

**Overview** Figure 277 illustrates the graphical notation for the Alternative Exit-Points pattern. The top view corresponds to a rigid process with a single exit point, before
applying the pattern. The bottom view illustrates the process definition after applying the pattern. Several alternative exit points are now defined, any of which can be selected to terminate the process via the related end task. The process definition in the bottom view contains a static part (i.e. task A), which is fixed and must be executed for any process instance, while a dynamic part consisting of tasks B and C, D defines an alternative way of terminating the process instance.

**Figure 277:** Alternative Exit-Points pattern

**Context** Figure 278 illustrates the process engine expressed using the CPN formalism and how the Alternative Exit-Points pattern can be realized in order to provide for flexibility in process termination. To incorporate alternative end tasks into the process definition, the original **ProcModel** type needs to be changed to the **ProcModel’** type in order to include a set of end-tasks rather than a single end-task, as shown below:

```
colset ProcModel' = product ModelID * TaskID * TaskIDs * Tasks * Arcs;
```

In the basic process engine, a process instance stored at the Running instance place can be terminated when no enabled tasks remain. When one of the alternative end tasks has been selected for execution, no tasks can be executed afterwards, and the execution of the process instance can be completed by executing the Complete process instance transition.

**Figure 278:** Engine enhanced with the Alternative Exit-Points pattern

**Implementation** None of the systems analyzed except for Declare allow multiple tasks in a model to be marked as end tasks. YAWL explicitly forces all end tasks to be synchronized into a single exit point. Oracle BPEL PM, ADEPT1 and FLOWer operate with structured models, which are characterized by a single start task and a single end task, however in Oracle BPEL PM it is also possible to introduce a `<terminate>` activity
in order to terminate process execution earlier. In Declare, different tasks can be associated with a process termination condition. After executing a task whose process termination condition is satisfied, no further tasks in the process definition can be executed. Amongst the considered CIG modeling languages, only PROforma allows process termination conditions to be associated with several tasks in a process. After executing a task whose process termination condition has been satisfied, the process terminates.

**Issues** When a decision to terminate a process instance by selecting one of the nominated end tasks is taken, there may exist tasks that are currently executing or which are enabled and may execute later. Early termination of tasks that have commenced but not yet completed may result in loss of data.

**Solutions** Depending on whether the data is produced by tasks executed at the moment of process instance termination, the decision may be taken either to allow these tasks to complete or to abort their execution and loose the data.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows several tasks to play the role of an end task, the execution of any of which causes a corresponding process instance to terminate.

**Pattern** PF-7  TERMINATION SKIP

**Description** During execution, i.e. after a process instance has been created, there is the possibility of deviating from the execution path prescribed by the process definition by ignoring all subsequent tasks. The act of skipping the execution of the currently enabled and in future to-be-enabled tasks results in the premature termination of a process instance. This has no effect on the process definition, thus this pattern corresponds to the flexibility by deviation type.

**Examples**
- In the middle of the investigation, the patient’s complaint disappeared, therefore the patient decided to stop the treatment process. All prescribed tests have been skipped.
- The recruitment process for a candidate, who did not pass the capability test, is terminated without affecting the established procedure for recruitment.

**Motivation** A typical process definition contains a set of tasks that have to be executed until the end of the process is reached. In some situations, the process needs to be terminated before the end task prescribed by the process definition has been reached. This situation may even occur if at design-time the possibility for premature process termination has been foreseen and multiple alternative exit tasks have been defined (as described in the Alternative Exit Points pattern on page 349), however the desired end-task has not yet been reached. The Termination Skip pattern allows the execution of a given process instance to end at a particular task by skipping all currently enabled and in future to be executed tasks.

**Overview** Figure 279 illustrates the graphical notation for the Termination Skip pattern. The top view shows the process instance before applying the pattern. The execution thread in this process instances indicates that task C is enabled. The bottom view shows that after applying the pattern the thread of control has been moved beyond the task D, which corresponds to process termination. Note that the process definition remains unchanged.
Chapter 6 Process Flexibility Patterns

Context Figure 280 illustrates the process engine expressed using the CPN formalism and shows how the Termination Skip pattern needs to be realized in order to support flexible process termination.

In order to realize the functionality associated with flexibility by deviation, the model of the process engine is extended with a transition performing the deviation operation and the Log place which illustrates which tasks have been executed (thus providing a means for tracking the deviations from the prescribed execution order). When describing the behavior of patterns facilitating flexibility by deviation, we will use a set of functions with uniform names, and will adjust the content of these functions in order to incorporate the desired behavior. As such, the deviation_possible() function incorporates a set of conditions that have to be satisfied in order to allow for deviation operations. The update_dev() function updates the process instance state but has no impact on the process definition associated with the given process instance.

Due to the introduction of the Log place, the structure of the functions associated with enabling and execution of the Execute task transition have been slightly modified. A new function exec_task() has been introduced, which non-deterministically selects a task from the set of currently enabled tasks for the execution. The update_exec() function updates the process instance state after the selected enabled task has been executed.

Unlike other extensions to the process engine to support momentary and permanent changes, the extension of the process model presented in Figure 280 represents a mix of the engine functionality and the environmental/user choice causing the deviation from the normal execution sequence. Depending on the engine realization, these extensions can be
incorporated as explicit functionality options or supported implicitly.

In the Termination Skip pattern, for running process instances whose process definitions and current states are stored in the Running instance place, there is the possibility either to execute a currently enabled task or to skip all current and future tasks by executing the Termination Skip transition. The execution of this transition is equivalent to completion of the process instance on a premature basis. The guard for the Termination Skip transition specifies that it can only be enabled if there are tasks enabled in the current state. By executing this transition, it is possible to jump from any execution state to the process instance termination state in which no further tasks can be executed. The termination state is calculated using the update_dev() function. In particular, the state of a given process instance is set to empty (i.e. no tasks can be executed any more), therefore transition Complete process instance becomes enabled.

Implementation Support for the Termination Skip pattern is provided by Oracle BPEL PM, FLOWer, and Declare. In FLOWer, a process instance can be terminated by executing a skip operation at the level of the root plan. In Oracle BPEL PM, an event handler can be defined at the outer scope of the process to perform the Terminate activity once a particular event occurs. In Declare, the process can be terminated after all mandatory constraints have been satisfied. The CIG modeling languages offer no support for this pattern.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip several tasks at once, resulting in the thread of control being moved to the end of the process.

Pattern PF-8 UNDEFINED EXIT

Description A process definition contains a placeholder, associated with the end of the process, that is intentionally left underspecified. During process initiation there is the possibility to complete the specification of this placeholder with an appropriate end task. This pattern characterizes the need for flexible process termination recognized at design-time, but whose actual realization occurs at run-time, thus it corresponds to the flexibility by underspecification type.

Examples

– Students starting a high-school education are obliged to follow a basic set of courses. They may select additional elective subjects when their individual academic goals become clear.

– For patients arriving at the emergency center only the admittance procedure is defined: the patient’s insurance is checked and a questionnaire is filled in. The subsequent investigative or treatment steps depend on the state of the patient and are defined next.

Motivation In many situations, it is not practical to explicitly specify how the process execution must end. This may either be because the actual end of the process is unknown or because it may vary for individual process instances. The Undefined Exit pattern allows the specification of the end of a process to be postponed until run-time, when more information related to the operational context becomes available.

Overview Figure 281 illustrates the graphical notation for the Undefined Exit pattern. The top view shows a process instance where the end of the process represented by a
placeholder is enabled. In order to complete the process, the placeholder needs to be defined. The bottom view shows that after applying the pattern, task C defining the content of the placeholder becomes enabled. Note that tasks A and B represent the static part of the process, which are to be executed for each process instance. The content of the placeholder may however vary for different process instances, e.g., instead of task C a process fragment consisting of tasks C and D could be used.

![Figure 281: Undefined Exit pattern](image)

**Context** The semantics of completing the placeholder representing the end of the process is the same as for any other placeholder. Once the placeholder is enabled, its definition needs to be completed by the placeholder engine depicted on Figure 262(b) in a similar manner to how it has been described in the **Undefined Entry** pattern (cf. page 340). In order to allow the process end to be defined at run-time, the end-task associated with the process definition has to be underspecified at design-time (its task type must be set to the **placeholder** value).

**Implementation** Of the set of workflow offerings analyzed only YAWL supports this pattern by means of worklets [19]. None of other systems investigated, i.e. ADEPT1, Declare, FLOWer or Oracle BPEL PM provide means for realization of the **Undefined Exit** pattern. Among CIG modeling languages, only PROforma provides a similar concept, where at design-time a keystone construct representing a generic task can be used. However, before deploying the process definition the keystone has to be specified explicitly. Thus the moment of specification is postponed not until run-time but until the latest possible moment at design-time.

**Issues** The same issues as identified for the **Undefined Entry** pattern (cf. page 340) apply here also.

**Solutions** See solutions identified for the **Undefined Entry** pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows incomplete process definitions to be enacted. An underspecified process-end fragment must be explicitly marked by a placeholder at design-time, and it should be possible to complete its definition at run-time either before or at the time that the thread of control reaches once the placeholder.

**Pattern** PF-9 **MOMENTARY EXIT-POINT CHANGE**

**Description** During execution, i.e. after process initiation, there is the possibility of changing the exit-point for a process by **temporarily** modifying the process definition associated with the given process instance. In this pattern, the change applies only to a specific process instance and has no effect on other (existing and future) process instances, thus it corresponds to the **flexibility by momentary change** type.

**Examples**
For a patient who developed signs of high blood-pressure an appointment made for an operation has to be canceled and the medication previously prescribed by the doctor has to be terminated immediately.

The employment of a PhD student who has been ill for more than two years is terminated. As a consequence of this all subscriptions for new courses has to be stopped and involvement of all ongoing courses for the given student is terminated.

Motivation A typical process definition contains a single exit task, which needs to be executed in order for the process to terminate. Although it is possible to include several alternative exit-points in a process definition at design-time (as the Alternative Exit-Points pattern on page 349 describes), there exists the possibility that the desired exit-point will not be found in the set of exit tasks defined. When no suitable exit task for a terminating process instance can be found at run-time, and it is not possible to deviate from the execution path prescribed by applying the Termination Skip pattern (cf. page 351), it may be necessary to temporarily modify the process definition in order to allow the given process instance to terminate prematurely. The Momentary Exit-Point Change pattern allows the execution of a process instance to be instantly completed at any point during process execution by modifying the nominated end task.

Overview Figure 272 illustrates the graphical notation for the Momentary Entry-Point Change pattern. The top view depicts two distinct process instances populated from the same process definition before applying the pattern. The pattern is applied to the process instance whose execution thread is depicted by the shaded triangle. The bottom view shows that for the given process instance tasks C and D have been removed from the process definition, and the thread of control has been moved to the end of the process. This change does not affect the other process instance.

Context Figure 283 expresses the process engine using the CPN formalism and shows how the Momentary Exit-Point Change pattern is realized. In order to incorporate the flexibility facilitated by the Momentary Exit-Point Change pattern, the basic engine is extended as has been described earlier for the Momentary Entry-Point Change pattern on page 343. The Momentary exit change transition can be executed for an initiated process instance that is stored in the Running instance place and which has not yet reached the end task yet.

Any currently enabled task that is different to the nominated end task can be selected as a new end-task. By executing the Momentary exit change transition, the original process definition associated with the given process instance is modified. The update_pi() function sets the currently enabled task to be a new end-task, ensuring that all follow-up tasks and
Any task that has not yet been executed and which precedes the nominated exit task can be marked as the new end task.

**Implementation**  ADEPT1 and Declare allow the *Momentary Exit-Point Change* pattern to be applied after process initiation. In ADEPT1, all tasks subsequent to a particular point in the process definition have to be removed one-by-one. The removal in ADEPT1 corresponds to disabling of tasks, i.e. after deleting a task a user still sees where the deleted task resided but cannot execute it. In Declare, either all unnecessary tasks have to be removed or a condition associated with the desired end task needs to be modified such that it becomes the process instance termination condition. In YAWL, when a desired point in the process has been reached, an exlet [21] can be called which will result in the termination of the given process instance. In FLOWer, Oracle BPEL PM and CIG modeling languages, no changes to the process definition can be performed at run-time on a temporary basis.

**Issues**  None identified.

**Solutions**  N/A.

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering that allows any group of tasks (including the end task) to be deleted from the process definition associated with a given process instance at run-time in a single step. It should be possible to mark any task other than the nominated end task as an exit point from the process definition.

**Pattern**  PF-10  PERMANENT EXIT-POINT CHANGE

**Description**  At run-time, there is the possibility to *permanently* change the exit point for a process by modifying the process definition. In this pattern, the change performed directly affects all future process instances, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

**Examples**
- The procedure of obtaining a visa has been modified for all applicants. Since the introduction of electronic applications, all documents are handled by an external organization, and no interviews at the embassy are required. All appointments made for interviews are canceled and no new appointments will be made from this point.
In order to save costs, a company producing products on demand decided to eliminate the number of final tests of the product functionality. This change affects all ongoing and future product production lines.

**Motivation** Due to changes in the operational environment, there may arise the need to modify the end of a process when completing the execution of current and future process instances earlier than originally defined. Although the *Momentary Exit-Point Change* pattern can be used for modifying an exit-point for a specific process instance, it requires the changes to be made for each process instance separately. The *Permanent Exit-Point Change* pattern offers a more efficient way of modifying the exit-point associated with the process definition by changing it at the type level.

**Overview** Figure 284 illustrates the graphical notation for the *Permanent Exit-Point Change* pattern. The top view depicts the execution state of several process instances populated based on the same process definition before applying the pattern. The permanent entry change performed at the type level (eliminating tasks C and D) affects process definition associated with all existing process instances, as shown on the bottom view.

**Context** Figure 285 illustrates an engine extended with the *Permanent Exit Change* pattern, which enhances flexibility in process termination, using the CPN formalism. The process engine has been extended using the structure described earlier for the *Permanent Entry-Point Change* pattern on page 345.

The content of the `change_possible()` function defines that the *Permanent exit change* transition is enabled only when the process model contains more than two tasks (including the start and end tasks). The `modify_m()` function picks an arbitrary task from the set of the tasks associated with the process definition being modified, sets it as the new end task, and removes all succeeding tasks and related branches.

The process definition `m` associated with a process instance stored in the `Running instance` place can be replaced by the `migrate()` function with the new process definition `newm` if the change performed does not course enabling problems afterwards. If a task that has not been yet executed has been selected as the new start task, then the migration can be performed directly. However, if this task has been executed already, the change either will be inconsequential or will require the process instance to be restarted. This highly depends on the migration strategy chosen for the realization of the `migrate()` function.

**Implementation** ADEPT1 and Declare are the only systems of those analyzed which allow the *Permanent Exit-Point Change* pattern to be applied at run-time in order to modify the end task in the process description. This change implies the need for process instance migration. Process instances which have not yet commenced or which have not reached the new exit task are migrated directly to the new process definition, while the process instances that have passed this execution point are terminated. In YAWL, an exlet
Figure 285: Engine enhanced with the Permanent Exit Change pattern

can be called that allows all existing process instances to be terminated. In FLOWer, Oracle BPEL PM and the CIG modeling languages there is no possibility of changing the end task in a process definition on a permanent basis at run-time.

**Issues** Identical to the issues identified for the *Permanent Entry-Point Change* pattern (cf. page 345).

**Solutions** See the solutions identified for the *Permanent Entry-Point Change* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows several tasks in a process (including the nominated end task) to be deleted from the process definition in a single step, and which provides support for process instance migration.

### 6.2.4 Flexible selection

This group of patterns aims to provide flexibility when selecting an execution path appropriate to the operational circumstances. Figure 286 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

In sequential processes, all tasks have to be executed in a predefined order. Such processes are very rigid and offer no possibility for deviating from the default execution path and selecting an appropriate alternative to it. In some situations, different execution paths may need to be incorporated in the process definition in order to allow a knowledgable user to select a suitable execution alternative during process execution. The flexible selection patterns address different ways of realizing choices between several alternative tasks, each of which corresponds to an alternative execution path. The main purpose of these patterns is to promote the availability of multiple alternatives and the ability to make a choice between them, rather than to specify the exact semantics of such a choice. In order to specify the type of choice explicitly, one has to consider different variants of branching control-flow patterns described in Chapter 4.

Depending on the moment at which the need for realizing the choice between alternative tasks is recognized and the manner in which it is achieved, we distinguish the following
five patterns: Choice, Task Substitution, Late Selection, Momentary Choice Insertion, and Permanent Choice Insertion. Of these five patterns, the Choice pattern corresponds to the design-time choice. In this pattern, the need for several alternative execution paths is recognized and incorporated into the process definition at design-time. The Task Substitution pattern relates to situations when the need for an alternative execution path is recognized at run-time, i.e. after the process instance creation, and is realized by deviating from the prescribed execution path rather than by modifying the corresponding process definition. In the Late Selection pattern, the need for alternative execution paths is recognized at design-time, however the actual realization of the chosen execution alternative is postponed until the latest possible moment at run-time. The Momentary Choice Insertion and Permanent Choice Insertion patterns correspond to the realization of future decisions related to the selection of an appropriate execution path by introducing a choice construct into the process definition at run-time at the instance and type levels respectively.

**Pattern PF-11 CHOICE**

**Description** A process definition specified at design-time contains a choice construct whose execution at run-time results in the selection of one out of several possible tasks. Such a decision may depend, for example, on the evaluation of a particular data expression or the availability of an external trigger. This pattern characterizes the need for flexible selection of an execution path which is anticipated at design-time, thus it corresponds to the flexibility by design type.

**Examples**
- When enrolling in a driving course a student has to decide what course of education he/she wants to follow: in-class lectures or self-study.
- A patient visiting a doctor may choose their preferred method of treatment: conventional medical treatment or homeopathic.

**Motivation** In order to allow for the flexible selection of an execution path from a set of possible alternatives, the decision points associated with the selection of the task to be executed next can be incorporated in the process definition at design-time. By selecting electing a task from the set of available options at run-time, a corresponding execution path becomes enabled. Such choices are common in practice as has already been
illustrated earlier by the branching class of the control-flow patterns (cf. page 122) and the 
(Deterministic XOR-Split and Non-deterministic XOR-split CPN patterns (cf. page 35). 
The decision-making may be either non-deterministic or based on the status of a data 
condition associated with a particular branch. Although multiple options may potentially 
be selected by a user, in this pattern we assume that a user makes an exclusive choice (cf. the 
Exclusive Choice WCF-pattern on page 125). The Choice pattern facilitates flexibility 
in selecting an appropriate execution path by defining a decision point in the process, 
associating a corresponding XOR-split construct with it and defining a set of alternative 
tasks which may be selected from.

Overview  Figure 287 illustrates the graphical notation for the Choice pattern. The top 
view illustrates the original process definition. Before applying the pattern, in this process 
definition after task B has completed task C always needs to be executed. The bottom 
view shows the process definition after applying the pattern. In this process definition, an 
alternative to task C, i.e. task E, is incorporated. After completing task B, a user has to 
declare which task to execute next, thus selecting either task C or task E.

\[
\begin{align*}
\text{A} & \rightarrow \text{B} \rightarrow \text{C} \rightarrow \text{D} \\
\text{A} & \rightarrow \text{B} \rightarrow \text{C} \rightarrow \hat{\text{D}} \rightarrow \text{E} \\
\end{align*}
\]

Figure 287: Choice pattern

Context  Figure 288 illustrates the basic process engine which incorporates the function-
ality for supporting the Choice pattern. In order realize this pattern, the process definition 
used as the basis for process initiation has to include a task after whose execution of which 
a choice of one out several alternative branches needs to be performed. For instance, in a 
process where after task B either task C or task E needs to be executed, the process model 
needs to be defined as follows: the type of the split-connector associated with task B has to 
be set to “XOR” (i.e. the task definition should be defined as (‘B’,generic,none,XOR)), 
and two arcs representing alternative branches have to be added to the list of arcs asso-
ciated with the process definition (i.e.((B,C),(B,E))). The Execute task transition after 
executing the currently enabled task (B) calculates the new process instance state by means 
of the ns() function. For this, the type of the split-connector of the currently executed 
task is evaluated, and if it corresponds to the XOR-split, one of the subsequent tasks is 
chosen. Since we abstract from the data perspective, in this realization a task is selected 
on a random basis.

Implementation  All of the systems analyzed allow the Choice pattern to be realized. 
In YAWL, the type of the split is associated with a task, and evaluated after it has com-
pleted. For an exclusive choice it has to be set to ‘XOR’ in order for an exclusive choice to 
be realized. In ADEPT1, in order for such choice to be defined, a dedicated construct needs 
to be inserted and in each of the branches a task from a predefined set of templates needs 
to be selected. The number of alternatives can be increased by adding extra branches. In 
Oracle BPEL PM, a task cannot be associated with a connector type, therefore in order to 
represent the XOR-split, a task must be followed by the <switch> construct with multiple 
cases, each representing an alternative branch (cf. Section 4.3.2). In FLOWer, a choice can
be specified either by means of a plan of the user-decision type or the system-decision type. In Declare, both deterministic and non-deterministic choices can be realized by means of constraints. Having multiple unconstrained tasks enables a non-deterministic choice, while by defining constraints for each of the tasks, a choice between them becomes enabled. In CIG modeling languages (cf. Section 4.3.3) GLIF and EON, there are dedicated constructs that can be used to denote an exclusive choice in the process definition. Asbru achieves such behavior by means of the If-then-else plan. In PROforma, there is a dedicated decision construct that allows one or more options to be selected at run-time.

Issues Depending on the number of options available and the conditions for defining the number of options which will execute at the same time, different types of choice constructs are possible. Having only one, several or all options selected, may require the completion of the selected options to be synchronized.

Solutions In order to know when to synchronize execution options offered by the choice construct, one has to keep track of the branches selected by means of Boolean variables as described in the Structured Synchronizing Merge and Local Synchronizing Merge control-flow patterns (cf. page 148 and page 150 respectively).

Evaluation Criteria Full support for this pattern is demonstrated by any offering that provides a decision construct which allows the selection of one of several alternative tasks to be included in the process definition at design-time.

Pattern PF-12 TASK SUBSTITUTION
Description During execution, i.e. after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by substituting the currently enabled task with another task contained in the process definition. Consequently, the enabled task is ignored and another task is executed instead. This has no effect on the process definition and other process instances, thus this pattern corresponds to the flexibility by deviation type.

Examples
- A student following a course at the driving school is unable to attend a lecture. Instead, he/she studies the lecture material at home using the material provided for self-education or for e-distance learning.
- For a patient who develops an allergy to penicillin, an alternative antibiotic treatment can be made available.
**Motivation** The *Choice* pattern allows execution alternatives that have been foreseen to be included in the process definition at design-time. However, it is possible that not all execution options have been foreseen. Therefore, at runtime there may arise the need to execute a task that is not currently enabled but which is included in the process definition. The *Task Substitution* pattern promotes flexibility by deviation by allowing a currently enabled task to be substituted with another task in the process definition. Note that such a deviation only affects the execution order of tasks and does not result in the process definition being modified.

**Overview** Figure 289 illustrates the graphical notation for the *Task Substitution* pattern. The top view shows that a process instance, in which task C is enabled, before applying the pattern. The bottom view shows that instead of executing task C, a decision to execute another task E has been taken by moving the thread of control from task C to task E. Task E is a generic representation of any task contained in the process definition. After executing this task the thread of control progresses to the task D.

![Figure 289: Task Substitution pattern](image)

**Context** Figure 290 illustrates the basic engine, which is enhanced with the *Task Substitution* pattern, using the CPN formalism. In this diagram, the *Substitute task* transition is added in order to execute a different task to the currently enabled task. The tid variable represents the task to be executed, which is selected non-deterministically by means of the dev_action() function. After executing this task, the new state of the process instance is calculated by means of the update_dev() function. In this state, task(s) subsequent to the originally enabled task are enabled. Note that since task substitution does not affect the process definition, the only way to track the effect of its operation is to record tasks that have been executed. For this purpose, the Log place is added. Whenever a task has been completed, a record containing the id of the task is added to the log.

**Implementation** Of all of the systems analyzed only FLOWer and Oracle BPEL PM offer some support for this pattern. In FLOWer, this pattern is not supported directly, but a similar result can be achieved by invoking a required task and performing a skip operation for moving the thread of control to a subsequent task. In Oracle BPEL PM, when a task becomes enabled, an exception needs to be raised, which can be handled by executing an alternative task.

**Issues** When substituting the currently enabled task with another task contained in the process definition, a problem may arise where the substitute task blocks as a consequence of specific input data elements that it requires which should have been provided by preceding tasks but being as a consequence of the fact that these tasks have not yet run.

**Solutions** In order to avoid blocking of a task caused by the absence of required input data, a default data value needs to be defined allowing this task to be enabled and executed when necessary.
Evaluation Criteria  Full support for this pattern is demonstrated by any offering that offers a means of substituting a currently enabled task with another task contained in the process definition, such that the process definition remains unchanged, and after the completion of the invoked task the thread of control can move forward as if no substitution has taken place.

Pattern PF-13  LATE SELECTION

Description  A process definition created at design-time contains a placeholder which may be completed at run-time in order to allow an alternative execution path that has not been foreseen at design-time to be taken. The completion of the placeholder is optional and is performed only if an execution path different from the default one needs to be taken at run-time. The need for flexible execution path selection in this pattern is recognized at design-time, although its actual realization is performed at run-time. It corresponds to the flexibility by underspecification type.

Examples

- The procedure of acquiring travel insurance via the Internet assumes that by default only one product is requested by a client. However, the client may decide to take this insurance as one part of a substitute part for a bigger package of products offered by the insurance company.
- When requesting a new credit card the client may choose a standard design, or create their own design before submitting their request.

Motivation  When different execution paths are foreseen at design-time, they can be included explicitly into the process definition by applying the Choice pattern (cf. page 359). However, in some situations not all execution options can be foreseen or it may be impractical to specify them all explicitly in the process definition. The Late Selection pattern offers the possibility of selecting between a particular task in the process definition or an alternative for it that has intentionally been left underspecified at design-time.

Overview  Figure 291 illustrates the graphical notation for the Late Selection pattern. The top view shows the process instance where after executing task B a decision to take
an alternative path has been taken. This is visualized by the enabled placeholder. The
bottom view illustrates that at run-time the content of the placeholder has been completed
with task E, to which the thread of control has been moved.

Figure 291: Late Selection pattern

**Context** Figure 292 illustrates the basic process engine enhanced with the flexibility fa-
cilitated by the **Late Selection** pattern. The realization of this pattern is similar to that
described for the **Choice** pattern with the only difference being that when including an al-
ternative task E in the model, the type of this alternative task has to be set to **placeholder**.
Because in the basic process engine no tasks can be executed when an underspecified task is
encountered in the set of enabled tasks, in the extended process engine the enabling condi-
tion associated with the **Execute task** transition has to be weakened. For this purpose, the
**isConcrete(pid,m,st)** function is replaced with the **existsConcrete(pid,m,st)** function which evaluates to **True** if a task of the concrete type is available in the set of enabled
tasks. Thus, when both a concrete and an underspecified task are enabled, a user may
decide whether to execute the concrete task or take an alternative path and define the
content of the underspecified task.

Figure 292: Engine enhanced with the Late Selection pattern (note the guard of the **Execute task** transition)

**Implementation** To realize this pattern, a system must support both the **Choice**
pattern and also allow a task to be left underspecified at design-time. Of all of the systems
examined, only YAWL supports this pattern via the worklet service and its ability to
associate the **XOR**-type (i.e. exclusive choice) behavior with a task split construct.
Section 6.2 Catalog of process flexibility patterns

Issues  The issues discussed for the Undefined Entry pattern (cf. page 340) and the Choice pattern (cf. page 359) apply here also.

Solutions  See the corresponding solutions identified for the Undefined Entry and the Choice pattern.

Evaluation Criteria  Full support for this pattern is demonstrated by any offering that allows a decision in regard to selecting a task or an underspecified alternative to be included in the process definition at design-time, such that the content of the task which has been left underspecified, can be completed when required at run-time.

Pattern  PF-14  MOMENTARY CHOICE INSERTION

Description  During execution, the process definition associated with a particular process instance can be modified to include a choice between a task predefined in the design-time process definition and an alternative task that previously has not been foreseen. The change applies only to the given process instance and has no effect on other (current or future) process instances. It corresponds to the flexibility by momentary change type.

Examples
– A standard education trajectory for students assumes attendance at lectures for all courses. For a handicapped student a modification to the education program can be made. Attendance at lectures can be replaced with study of the lecture material at home.
– For a passenger who did not confirm their departure date, no place on the flight could be found. The passenger has been offered the option of waiting until all passengers complete the check-in in order to find out whether any free places remain or to be booked on the next connecting flight.

Motivation  In situations, where no alternative execution paths have been foreseen and/or incorporated in to the process definition at design-time, there may arise the need to introduce an additional execution path for a given process instance. The Momentary Choice Insertion pattern anticipates the need to allow for the selection of an appropriate execution path at a later point in time. The need for flexible execution path selection in this pattern is facilitated by identifying a task for which an alternative task needs to be inserted and associating a choice construct (as described in the Choice pattern on page 359) with it.

Overview  Figure 293 illustrates the graphical notation for the Momentary Choice Insertion pattern. The top view illustrates two distinct process instances populated from the same process definition. The execution threads indicate that task B is enabled in each of the process instances. For one of the process instances at this point of time an optional path has been foreseen where instead of task C another task E may be selected.

The bottom view illustrates the process instances after the pattern has been applied to the given process instance. As the result of change, the modified process instance includes a choice construct with two alternative tasks C and E, while the other process instance remains unaffected.

Context  Figure 294 illustrates the basic engine enhanced with the Momentary Choice Insertion pattern using the CPN formalism. The engine extension is of the same form as for all of the patterns facilitating flexibility by momentary change, the first of which was described earlier on page 343. The change_possible() function specified that a choice construct can be inserted into the given process instance only if it is associated
with a sequential task with no split and joins which has not yet been executed. If there are several tasks that meet this condition, then one task is randomly selected. The \texttt{update\_pi()} function creates a bypass path for the selected task, modifies the type of the split-connector of the task at which the choice needs to made to an \texttt{XOR}-split, modifies the join-connector of a subsequent synchronization task to an \texttt{XOR}-join, and inserts a new task into the created bypass path. When during subsequent execution of the process instance the task associated with the choice is executed, either the default path or the newly created alternative path is taken.

**Implementation** Of all of the systems analyzed, only Declare allows a choice to be inserted in a process definition at run-time. By inserting a task and associating a particular constraint with it, an alternative execution path can be inserted. In ADEPT1, although it is possible to add and remove tasks at run-time, there is no functionality for inserting a choice (in one of the versions of ADEPT1, it is possible to associate a code region with outgoing arcs in order to obtain the data-based routing). It is also not possible to add an alternative branch and to modify the type of a task split construct.

**Issues** This pattern is similar to the \textit{Foreseen Bypass Path} pattern (cf. page 378), where the ability to omit the execution of a particular task by taking a bypass is foreseen at
design-time. To implement an optional bypass path for a given task, either no task or an empty task need to be inserted into the alternative path created by the Momentary Choice Insertion pattern.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows an alternative execution path, containing a task that has not previously been foreseen, to be inserted in the process definition associated with a given process instance at run-time.

**Pattern** PF-15 PERMANENT CHOICE INSERTION

**Description** During execution, there is the possibility to permanently add a choice construct between a task defined in the design-time process definition and an alternative task that previously has not been foreseen. This adds flexibility when selecting an appropriate execution path. The change performed affects all future process instances, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

**Examples**
- The product offerings of a software house are extended with a new product. All ongoing product development is unaffected, however for future orders a client can be offered a broader selection of product offerings to address their requirements.
- Due to an increased number of patients, not all patients can undergo blood tests on the same day. To distribute the load, the medical center has signed an agreement with a laboratory, to which the tests can be delivered for the required analysis. Since this contract has been signed, patients are free to choose where their blood will be analyzed.

**Motivation** As a result of extensions to services/products within a particular process, there may arise the need to include a variety of choices (and thus alternative execution paths) in the original process definition. Although this change can be incorporated in each of the process instances by applying the Momentary Choice Insertion pattern (cf. page 365), this approach is less efficient than performing the same change at the type level. The Permanent Choice Insertion pattern allows a process definition to be extended with an alternative execution path at run-time by identifying a decision point in the process and associating a new task with it.

**Overview** Figure 295 illustrates the graphical notation for the Permanent Choice Insertion pattern. The top view depicts the execution state of several process instances populated on the basis of the same process definition before applying the pattern. For this process definition, a decision has been taken to include task E which represents an alternative to task C.

The bottom view shows that the process definition associated with all process instances has been affected by inserting the choice construct preceding with these two tasks. For the process instance that has passed task C, this change is inconsequential, while for the other two process instances the ability to choose from several execution paths has become possible.

**Context** Figure 296 illustrates the basic engine enhanced with the Permanent Choice Insertion pattern using the CPN formalism. This model adopts the structure described earlier for the Permanent Entry-Point Change pattern. The execution of the Permanent
**choice insertion** transition allows for the inclusion of an alternative path with a task that has not been foreseen during design-time into the process definition at the type level. The `change_possible()` function specifies that the choice can be inserted only if a task with no splits and joins (other than the start and end tasks) is available in the model. The `modify_m()` function picks an arbitrary task of the sequential type, adds a bypass path, creates a new task and inserts it in the bypass path introduced, and modifies the splits and joins of related synchronization tasks to the XOR-type. The `Migrate` transition can be enabled only if process instances associated with the process model with the same identifier as the model recently changed is available at the `Running instance` place. Furthermore, the `transfer_possible()` function checks whether the migration is possible, e.g., there must be no changes impacting the enabling of the currently enabled tasks. The `migrate()` function defines the strategy for migrating the existing process instances to the new process definitions.

**Implementation** Of the systems examined, only Declare and YAWL allow an alternative execution path to be inserted by adding a new task and defining scheduling constraints for it. Declare also ensures that all instances are migrated (if possible). In YAWL, an exlet can be called allowing a new worklet to be defined which is associated with a choice be-
tween several alternatives tasks. The worklet defined becomes available for other process instances and can be used for the same purpose.

**Issues** Similar issues apply to those identified for the Permanent Entry-Point Change pattern (cf. page 345).

**Solutions** See solutions identified for the Permanent Entry-Point Change pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a new task representing an execution alternative to an existing task to be added to the process definition, such that only one of these tasks can be selected at run-time. In addition to this, there must be support for process instance migration.

### 6.2.5 Flexible reordering

This group of patterns aims at achieving flexibility by establishing alternative execution ordering for the tasks in a process at run-time. Figure 297 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

<table>
<thead>
<tr>
<th>Flexible initiation</th>
<th>Flexible termination</th>
<th>Flexible selection</th>
<th>Flexible reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative entry points</td>
<td>Alternative exit points</td>
<td>Choice</td>
<td>Interleaving</td>
</tr>
<tr>
<td>Entrance skip</td>
<td>Termination skip</td>
<td>Task substitution</td>
<td>Swap</td>
</tr>
<tr>
<td>Undefined entry</td>
<td>Undefined exit</td>
<td>Late selection</td>
<td>Momentary reordering</td>
</tr>
<tr>
<td>Momentary entry change</td>
<td>Momentary exit change</td>
<td>Momentary choice insertion</td>
<td>Permanent reordering</td>
</tr>
<tr>
<td>Permanent entry change</td>
<td>Permanent exit change</td>
<td>Permanent choice insertion</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 297:** Process Flexibility Matrix: flexible reordering

In situations, where a set of predefined tasks need to be executed is available, but whose eventual execution order may vary for different process instances, it may be desirable to allow for flexible reordering of the tasks in the process in order for a suitable execution path to be selected. This group consists of four patterns: **Interleaving, Swap, Momentary Reordering** and **Permanent Reordering**. The **Interleaving** pattern allows different task orders to be included in the process definition at design-time such that, i.e. only one of them will be selected at run-time. The other three patterns are characterized by the fact that the need for an alternate execution order is identified only at run-time. In the **Swap** pattern, the possibility of deviating from the prescribed execution order can be accomplished by swapping the execution order of two tasks. The **Momentary Reordering** and **Permanent Reordering** patterns allow the desired execution order to be achieved by moving a particular task to a desired location in the process definition, thus modifying the process definition both at the instance and type level respectively.

Note that this pattern group has no pattern related to flexibility by underspecification. The reason for this is that if at design-time the need for alternative execution paths is foreseen, they can be either explicitly incorporated into the process definition at that
time (which corresponds to the *Interleaving* pattern), or the (re-)definition of the ordering relations between the tasks can be postponed until run-time as described by the *Late Selection* pattern. Effectively, any placeholder associated with a set of predefined tasks can be replaced by an ordered sequence of the tasks from the given set.

**Pattern** PF-16  **INTERLEAVING**  
**Description** A process definition specified at design-time contains two execution paths, each representing alternate execution sequences for two tasks allowing them to be executed in either order but not concurrently. The need for flexible tasks ordering in this pattern is recognized at design-time, and corresponds to the *flexibility by design* type.

**Examples**
- When diagnosing patients both a blood test and an MRI scan have to be performed. The order in which the tests are undertaken is not important, however they cannot be conducted concurrently.
- When interacting with another party, a receiver may both send and receive messages, however these operations cannot be done concurrently.

**Motivation** In situations, where a specific execution ordering of two tasks needs to be relaxed in order to allow their execution sequence to vary for different process instances, it may be necessary to incorporate alternate execution sequences for these two tasks in the process definition at design-time. The *Interleaving* pattern allows two execution paths representing alternate sequences for the given tasks to be included in the process definition at design-time such that only of these sequences can be selected at runtime for a given process instance.

**Overview** Figure 298 illustrates the graphical notation for the *Interleaving* pattern. The top view illustrates the process definition before applying the pattern. In this process definition, tasks B and C have been identified, as tasks which may execute in either order.

![Figure 298: Interleaving pattern](image)

The bottom view illustrates the process definition after applying the pattern. To allow for alternate orderings of these two tasks, a choice construct has been associated with task A. After executing this task a decision needs to be taken in regard to selecting the desired execution order of tasks B and C.

**Context** The basic process engine incorporates functionality for handling the *Interleaving* pattern. In fact, the same functions which are used to support the *Choice* pattern also provide support for the *Interleaving* pattern. The alternate order of tasks that need to be interleaved must be specified in the process definition. For instance, the process model with id m1, the start task A and the end task D, and two alternate sequences of tasks, B, C and C, B, have to be defined as follows:
Section 6.2 Catalog of process flexibility patterns

("m1", A, D, {("A",concrete,XOR,none),("B",concrete,none,none),
("C",concrete,none,none),("D",concrete,none,XOR)},
{(A,B),(A,D),(B,C),(C,B),(C,D),(B,D)})

In this definition, the types of split and join connectors associated with tasks A and D are set to the XOR-type, indicating that only one of the branches B, C or C, B would be taken.

**Implementation** Oracle BPEL PM supports this pattern by means of serializable scopes or by defining alternative branches for the `<switch>` construct. In YAWL, ADEPT1 and FLOWer, this pattern can be realized by explicitly defining branches with alternative execution sequences, one of which needs to be selected using the means of the Choice pattern (cf. page 359). In Declare, this pattern is also supported by associating exclusive choice constraints with the tasks whose execution order needs to be interleaved. In Asbru, this pattern is supported by means of the Any-Order plan. PROforma, GLIF and EON require the alternative execution sequences to be explicitly defined as branches associated with a choice construct.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a decision in regard to the selection of an execution order for two or more sequential tasks (each of which has to be executed) to be explicitly included in the process definition at design-time.

**Pattern** PF-17  SWAP

**Description** During execution, at any time after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by swapping the execution order of the currently enabled task and its successor. This has no effect on the process definition and corresponds to the flexibility by deviation type.

**Examples**
- In an emergency situation, it may necessary to swap the order of the register patient and perform triage tasks.
- Before preparing a patient for medical treatment at a private hospital, an examination by a doctor and preparation of a patient file have to be performed. Depending on the availability of the doctor, the execution order of these tasks may be swapped.

**Motivation** Where the execution order of two sequential tasks needs to be reversed at run-time and it is not possible to include an alternate path in the process definition, there is a need to deviate from the prescribed execution order by swapping the currently enabled task with its successor task. The Swap pattern allows the the execution of the currently enabled task to be postponed until the moment when the execution of the subsequent task completes.

**Overview** Figure 299 illustrates the graphical notation for the Swap pattern. The top view shows a process instance in which task B is enabled. For the given process instance task C needs to be executed first. The bottom view shows that after applying the pattern, instead of task B task C is enabled. Note that by executing task C it is not guaranteed that task B will be the task selected for execution. Deviation allows to make local decision related to execution of a currently enabled task, e.g. after executing task C any other
deviation operation could be applied. Hence, Figure 299 illustrates an example of possible execution sequence.

![Diagram showing execution sequence]

**Figure 299**: Swap pattern

**Context** Figure 300 illustrates the basic engine enhanced with the *Swap* pattern using the CPN formalism. To support flexibility in task reordering this engine is extended using a generic structure for deviations described earlier for the *Termination Skip* pattern on page 351. In addition to modifying the content of the functions, in this model the information about the initiated process instances stored in the *Running instances* place has been extended with further state information:

```
colset ProcInstStSt = product piID * ProcModel * State *State;
```

![Diagram showing engine enhanced with Swap pattern]

**Figure 300**: Engine enhanced with the Swap pattern

This information is required in order to keep track of tasks that had to be executed but the execution of which has been postponed because another task has been executed instead (as a consequence of a task swap). The *Deviate* transition allows a task that has not yet been executed and which is not currently enabled to be executed, thus deviating from the normal flow of control. The *deviation_possible()* function specifies that the *Deviate* transition can be only executed if there is an enabled task that needs to be executed, the process end task has not yet been reached and there is a task other than currently enabled task which can be executed instead. The *dev_action()* function selects an arbitrary task from the set of tasks that have not yet been executed and which are not enabled. After executing the selected task *tid*, the record about its execution is added to the *Log* place.
The `update_dev()` function does not modify the model, but updates both states associated with the process instance. The normal state contains information about subsequent tasks that become enabled after the task execution, and the negative state keeps the record of the task which had to be executed but whose execution has been postponed. This information is used by the `execTask()` function for determining what task needs to be executed next. If there are tasks in the negative state, one of these tasks is selected non-deterministically. If no tasks whose execution has been postponed can be found, e.g., the negative state is empty, a task from the normal state is selected. The `update_exec()` function updates both states after the task has been executed. If the task from the negative state has been selected, this task is removed, otherwise a new state according to the process model is calculated.

**Implementation** Only FLOWer and Declare offer direct support for this pattern. In FLOWer, a task that is not currently enabled can be immediately initiated. After completion of this task the thread of control returns back to the originally enabled task which then can be executed.

**Issues** Swapping the execution order of tasks may result in data dependencies between them being invalidated. The absence of data required for enabling the subsequent task may result in its blocking when the decision to execute it before the preceding task is made and the data dependencies have not been satisfied.

**Solutions** To loosen the dependency between tasks and allow their execution order to be switched, input variables for each of the tasks should be set to a default value.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the execution order of a task and its successor in a given process instance to be reversed without modifying the process definition.

**Pattern** PF-18  **MOMENTARY REORDERING**

**Description** During execution, at any time after process initiation, the process definition associated with a particular process instance can be modified by changing the execution order associated with a particular task. Having a task moved to another place in the process definition allows the execution of the task to be postponed until a desired moment, thus enabling an execution outcome that might not have been foreseen at design-time to be effected. The need for task reordering in this pattern is recognized at run-time. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

**Examples**

– Based on a request from a client, the travel advisor changed the order of events planned for the booked trip. Because of their late arrival, the client cannot attend the first planned social event. The ticket for this activity has been re-booked for a later date.

– A typical procedure for car-hire starts with the recording of payment details. For repeat clients, an exception can be made, where they can pick-up the car on an express basis and arrange the payment afterwards.

**Motivation** In situations, where the execution order of two sequential tasks needs to be reversed for a particular process instance at run-time and it is not possible to deviate from the order prescribed by swapping these tasks (as described in the *Swap* pattern), there is the need to temporarily change the order of these tasks in the process definition in order to achieve the desired behavior. The *Momentary Reordering* pattern allows the currently
enabled task to be moved later in the process definition in order to allow a subsequent task to be executed first.

**Overview** Figure 321 illustrates the graphical notation for the *Momentary Reordering* pattern. The top view shows two distinct process instances based on the same process definition. For the process instance, whose execution thread is depicted by the triangle, the need to interchange the order of the currently enabled task B with its successor task C is recognized.

![Diagram of Momentary Reordering pattern](image)

Figure 301: Momentary Reordering pattern

The bottom view illustrates that after applying the pattern the process definition associated with the given process instance has been modified by reordering these tasks. The thread of control in this process instance has been moved to task C, whilst the other process instance remains unaffected.

**Context** Figure 302 illustrates the basic process engine enhanced with the *Momentary Reordering* pattern using the CPN formalism. To allow for momentary reordering, the functions `update_pi()` and `change_possible()` have been modified as follows. The *Momentary task move* transition can be only executed if there is an enabled task and if the enabled task is not the end task. The `update_pi()` function selects a task without splits or joins that has not yet been executed and an arc between two other tasks which also have not been executed, where the selected task will be moved to. The selected task is deleted, tasks from the preset and postset of the removed task are reconnected, and the removed task is inserted into the selected arc.

**Implementation** Declare supports this pattern by modifying constraints defining the execution order of tasks. In ADEPT1, it is not possible to move tasks from one place to another, however two tasks can be swapped by deleting these tasks first and inserting them in the reverse order. None of the other tools investigated support this pattern.

**Issues** Similar issues as identified for the *Swap* pattern apply here too.

**Solutions** See solutions identified for the *Swap* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the process definition associated with a given process instance to be modified in order to reverse the execution order of two subsequent sequential tasks which have not yet been executed.
A task which has no splits and joins is selected for moving into another arc. The selected task is removed from the model, tasks from preset and postset of the removed task are connected, and the task is inserted into the selected arc.

\[\text{update}\_\text{pi}(\text{pid}, \text{m}, \text{st})\]

\[\text{createinst}(\text{m}, \text{pid})\]

\[\text{update}\_\text{pi}(\text{pid}, \text{m}, \text{st})\]

Momentary task move

\[\text{change}\_\text{possible}(\text{pid}, \text{m}, \text{st})\]

**Pattern** PF-19 PERMANENT REORDERING

**Description** During execution, there is the possibility to permanently modify the execution order associated with a particular task in the execution sequence. The task can be moved before or after its current position in order to hasten or postpone its execution respectively. The change performed affects all future process instances directly, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

**Examples**
- In order to improve communication between students working together in projects, the presentation and communication skills course has been moved to the beginning of the education program. The current generation of students continues with the old program, while new students participate in the updated program.
- The procedure for issuing residence permits to foreign visitors has been changed in the following way: before the application for a residence permit can be accepted, the applicant must personally visit the office and pay the fee. In the past, applicants could submit applications at any time, which required the payment to be performed only when the application has been accepted.

**Motivation** In situations, where at run-time the need to rearrange the order of tasks in a process is anticipated, a set of actions may need to be taken in order to redesign the original process definition and to adapt the execution sequence of current and future process instances. The Permanent Reordering pattern allows the order of two sequential tasks in a process to be changed by permanently moving the currently enabled task in order to postpone its execution and let its successor task execute instead.

**Overview** Figure 303 illustrates the graphical notation for the Permanent Reordering pattern. The top view shows the execution state of three distinct process instances, sharing the same process definition. At run-time execution the need to change the order of tasks B and C at the type level is recognized. At run-time execution the need to change the order of tasks B and C at the type level is recognized. The bottom view illustrates the process definition after applying the pattern. The order of tasks B and C has been swapped. For process instances where task B was enabled, the thread of control has been moved to task C. For process instances that have passed task B, this change is inconsequential.
Context  Figure 304 illustrates the basic process engine enhanced with the *Permanent Reordering* pattern using the CPN formalism. In order to permanently change the execution order of two sequential and independent tasks, the process definition stored in the *Process definition* place needs to be adjusted and all existing process instances have to be migrated. For this, the model of the basic process engine is extended with the *Permanent task move* transition, which modifies the process definition both for a given process instance and the original process definition stored in the *Process definition* place, and the *Migrate* transition which transfers the old process definition $oldm$ associated with each process instances stored in the *Running instance* place to the new process definition $newm$. The mechanism for reordering tasks in a process definition is implemented using the $moveTask(tID,inarc,m)$ function in the same way as described for the *Momentary Reordering* pattern.

**Figure 304:** Engine enhanced with the Permanent Reordering pattern

Implementation  Of all of the systems analyzed ADEPT1, Declare and YAWL support this pattern. ADEPT1 allows the tasks to be removed and to be inserted in the reverse order, providing that all existing process instances migrate to the new process definition. In Declare, the execution order between tasks can be changed by modifying the constraints associated with these tasks, and migrating all existing process instance to the updated process definition. In YAWL, it is possible to invoke an exlet to execute tasks in an alternate order and to skip the execution of these tasks in the original order when the...
execution of the exlet completes. The alternate ordering of tasks in this case is defined via a workflow, which becomes available for other process instances to use.

**Issues** The same issues apply as identified for the Permanent Entry-Point Change pattern (cf. page 345) and the Swap pattern (cf. page 371) apply.

**Solutions** See solutions identified for the above mentioned patterns.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the process definition to be permanently modified at run-time execution in order to reverse the execution order of two tasks which have not yet been executed. In addition, there must be support for process instance migration.

### 6.2.6 Flexible elimination

This group of patterns aims to facilitate flexibility by avoiding the execution of a particular task. Figure 305 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

<table>
<thead>
<tr>
<th>Flexible elimination</th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>Alternative entry points</td>
<td>Entrance skip</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative exit points</td>
<td>Termination skip</td>
<td>Undefined exit</td>
<td>Momentary exit change</td>
<td>Permanent exit change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Choice</td>
<td>Task substitution</td>
<td>Late selection</td>
<td>Momentary choice insertion</td>
<td>Permanent choice insertion</td>
</tr>
<tr>
<td>Flexible reordering</td>
<td>Interleaving</td>
<td>Swap</td>
<td></td>
<td>Momentary reordering</td>
<td>Permanent reordering</td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>Foreseen bypass path</td>
<td>Task skip</td>
<td></td>
<td>Momentary task elimination</td>
<td>Permanent task elimination</td>
</tr>
</tbody>
</table>

**Figure 305:** Process Flexibility Matrix: flexible elimination

In situations, where the execution of a particular task needs to be avoided, there may arise the need to omit the execution of a task. Depending on the moment at which the need for eliminating the execution of a particular task is realized, we distinguish four patterns: Foreseen Bypass Path, Task Skip, Momentary Task Extraction and Permanent Task Extraction. Of these four patterns, the Foreseen Bypass Path pattern is characterized by anticipating and realizing an optional bypass for a task, whose execution may need to be avoided, at design-time. In the other three patterns the need to eliminate the execution of a given task is anticipated and realized at run-time. The Task Skip pattern allows a currently enabled task to be skipped, which corresponds to deviation from the execution path prescribed by the process definition. The Momentary Task Extraction and Permanent Task Extraction patterns allow the execution of a given task to be avoided by removing it from the process definition on a temporary or permanent basis.

Note that the idea of flexible task elimination has no straightforward mapping to flexibility by underspecification. The context conditions of this pattern group assume that the execution order and the content of the task which may need to be eliminated is known and fixed at design-time, whilst flexibility by underspecification assumes that the content
of particular process fragment that is marked as placeholder cannot be precisely defined until runtime.

**Pattern** PF-20 FORESEEN BYPASS PATH

**Description** A process definition specified at design-time contains a bypass path associated with a particular task, which if taken at run-time results in the task not being executed. This pattern recognizes the need for flexible task elimination which is anticipated at design-time, and corresponds to the *flexibility by design* type.

**Examples**
- The composition of the education program for postgraduate students allows some of the obligatory courses to be skipped if they have been completed during previous study.
- At the end of a course all students are offered a questionnaire to fill in investigating possible areas for improvement. Completion of the questionnaire is optional.

**Motivation** A typical process definition specifies the order in which tasks need to be executed. In some situations, it may be necessary to allow the execution of a particular task to be executed on an optional basis. This requires flexibility in choosing whether or not to execute a given task. The *Foreseen Bypass Path* pattern allows the possibility for optionally excluding a particular task which is included into the process definition at design-time.

**Overview** Figure 306 illustrates the graphical notation for the *Foreseen Bypass Path* pattern. The top view shows the process definition before applying the pattern. According to this process definition, after executing task A always task B has to be executed. In this process, the optional execution of task B recognized at design-time.

![Figure 306: Foreseen Bypass Path pattern](image)

The bottom view shows the process definition after applying the pattern. In order to offer a choice between executing task B and bypassing it, task A is associated with a choice construct and an alternate direct path to task C is inserted.

**Context** The *Foreseen Bypass Path* pattern is supported by the basic process engine in the similar manner as by the *Choice* pattern described on page 359. The realization of a bypass is equivalent to realization of a choice construct where one of the branches has no tasks. In order to incorporate the bypass path in the process definition, two tasks performing the selection and synchronization of branches need to be identified and their connectors have to be set to the XOR-type.

**Implementation** In Declare, it is possible to bypass a task by violating an optional constraint. In AsbruView, the *If-Then-Else* plan can be used where either a desired task or an empty task can be selected. In all other systems investigated, a bypass path associated with a particular task can be realized the same way as for the *Choice* pattern, providing that one of the branches contains no tasks.
Section 6.2 Catalog of process flexibility patterns

**Issues**  Selection of a bypass path associated with a specific task may result in a subsequent task blocking due to missing data that should have been provided by the bypassed task.

**Solutions**  In order to avoid blocking of the subsequent task, mandatory input data elements for this task should be set to have a default value.

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering that allows support for optional task execution to be included in the process definition at design-time.

**Pattern**  PF-21  TASK SKIP

**Description**  During execution, at any time after a process instance has been created, there is the possibility of deviating from the execution path prescribed by the process definition by skipping the execution of a currently enabled task. This has no effect on the process definition and is only reflected in the execution trace associated with a given process instance. The need for flexible task elimination in this pattern is anticipated at run-time. It corresponds to the flexibility by deviation type.

**Examples**
- The registration for an email account requires a set of questions to be answered. Some of the questions can be skipped and filled in later after the registration of the account has completed.
- The planning of a travel route in the navigation system offers a demo of the route that can be viewed before the route is initialized. A user may skip this step and proceed immediately to entering route instructions.

**Motivation**  In some situations, after a process has been initiated, it may be necessary to postpone the execution of a given task because the mandatory input data elements required for processing are not available or because this task does not need to be executed at all. When the process definition does not contain a bypass for the task (as described in the Foreseen Bypass Path pattern) that can be taken in order to obviate the need to execute it, other means of eliminating the currently enabled task are required. The Skip pattern allows the execution of a given process instance to deviate from the execution path prescribed by the process definition by skipping the currently enabled task.

**Overview**  Figure 307 illustrates the graphical notation for the Task Skip pattern. The top view shows a process instance before applying the pattern. For this process instance the need to avoid the execution of the currently enabled task B is recognized.

![Figure 307: Task Skip pattern](image)

The bottom view shows that after applying the pattern the currently enabled task B is skipped (without modifying the process definition), and the thread of control has been moved to task C.
**Context** Figure 308 illustrates the basic process engine enhanced with the Task Skip pattern using the CPN formalism. In order to allow for the execution of a currently enabled task to be ignored, the content of the functions deviation_possible(), dev_action(), and update_dev() have been modified as follows. The dev_action() function determines the currently enabled task tid whose execution needs to be ignored. The Skip transition can only be executed if there are tasks enabled in the process instance state which have not yet been executed. The update_dev() function updates the state of the process instance by moving the thread of control to a subsequent task. Note that the process definition remains unchanged.

**Implementation** In Declare, it is possible to skip a task by violating an optional constraint. In FLOWer, there is a special operation that allows a currently enabled task to be skipped. In PROforma, it is possible to skip the execution of the prescribed task. All other systems investigated offer no support for this pattern.

**Issues** Similar issues and solutions as for the Entrance Skip pattern (cf. page 338) apply here too.

**Solutions** See solutions identified for the Entrance Skip pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip the execution of a specific task at run-time, without affecting the process definition associated with the given process instance.

---

**Pattern** PF-22  **MOMENTARY TASK ELIMINATION**

**Description** During execution, at any time after process initiation, the process definition associated with a particular process instance can be modified by removing a currently enabled task. This allows for flexible elimination of a currently enabled task, should this task no longer be required. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.
- For visitors to a football match who hold a European passport, no visa requirements apply. This modification in the customs procedure is performed on a person-by-person basis for the period of the championship competition.
- A time registration program provides a timesheet with predefined fields. A task which is included in the timesheet (for instance, planned sick leave for a doctor’s appointment), but which has not been undertaken during the reporting period, can be removed from the timesheet.

**Motivation**
When at run-time a particular task needs to be ignored, and there is no a bypass path defined in the process definition for skipping the given task, and it is not possible to deviate from the prescribed execution order, there may arise the need to temporarily modify the process definition to exclude the task. The *Momentary Task Elimination* pattern allows a particular task to be removed from the process definition in order to avoid executing the task for a given process instance.

**Overview**
Figure 309 illustrates the graphical notation for the *Momentary Task Elimination* pattern. The top view shows two distinct process instances based on the same process definition before applying the pattern. For one of these process instances the need to eliminate the currently enabled task B is recognized.

![Figure 309: Momentary Task Elimination pattern](image)

The bottom view shows that after applying the pattern the process definition associated with one of the process instances has been modified. In particular, task B has been removed from the process definition, and the thread of control associated with the given process instance has been moved forward to task C.

**Context**
Figure 310 illustrates the basic process engine enhanced with the *Momentary Task Elimination* pattern using the CPN formalism. The model adopts the structure presented earlier in the *Momentary Entry-Point Change* pattern (cf. page 343), however the content of functions `change_possible()` and `update_pi()` has been modified. The *Momentary task extract* transition can only be executed if there are enabled tasks in the process instance state which have not been executed and if the end of the process has not been reached. The `update_pi()` function selects a currently enabled task, deletes it from the process definition, and connects the task preceding it to a subsequent task by means of an arc. Once that task has been removed, the process instance state is also recalculated.

**Implementation**
In Declare, it is possible to remove a specific task from the model. ADEPT1 also allows a task to be deleted from the model, however the deleted task is visualized as disabled rather than extracted from the process definition associated with the given process instance. In YAWL, there is the possibility to invoke an exlet to remove a workitem, however when the workitem has been removed, no subsequent tasks can be
executed. Instead, it is possible to force failure or completion of the given task in order to proceed with execution of the next task. No other systems that have been investigated offer support for this pattern.

**Issues** Removing a task from the process definition may trigger blocking of subsequent tasks requiring output data provided by the task removed.

**Solutions** After removing a task, either all data dependencies have also to be removed or a default value assigned to the input data elements of subsequent tasks.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task to be deleted from the process definition associated with a particular process instance.

**Pattern** PF-23 **PERMANENT TASK ELIMINATION**

**Description** During execution, there is a possibility to *permanently* eliminate the execution of a particular task in the process by removing it from the process definition at the type level. This allows the selected task to be ignored in all future process instances and existing process instances whose execution has not passed this point yet. The change performed affects all future process instances directly, while existing process instances may require migration from the old to the new process definition, thus the pattern is of the *flexibility by permanent change* type.

**Examples**
- A template for project schedule can be updated by removing tasks which do not need to be executed any more.
- Due to centralization of data by the city hall, tax office and insurance companies, changes in family living arrangements do not need to be reported to the insurance company directly. This step is removed from the predefined procedures of the insurance company. The data is instead reported to the insurance company by the tax office.

**Motivation** In situations where after process initiation the need to eliminate the execution of a particular task for current and future process instances is recognized, there may arise a need to modify the process definition at the type level. The *Permanent Task
Elimination pattern allows a particular task to be permanently removed from the process definition such that this task will never be executed again.

**Overview** Figure 311 illustrates the graphical notation for the Permanent Task Elimination pattern. The top view shows the execution state of several process instances populated based on the same process definition. At run-time the need to eliminate task B at the type level is recognized.

![Figure 311: Permanent Task Elimination pattern](image)

The bottom view shows that the process definition after applying the pattern. Task B has been removed from the process definition. For process instances where task B was enabled, the thread of control has been moved forward to its successor task C.

**Context** Figure 312 illustrates the basic process engine enhanced with the Permanent Task Elimination pattern using the CPN formalism. The model of the engine is extended according to the structure described earlier for the Permanent Entry-Point Change pattern (cf. page 345). The functions used have been changed as follows. The Permanent task extract can only be executed if the process definition contains more than two tasks (e.g., start and end tasks). This condition is incorporated in the change_possible() function. The modify_m() function removes the selected task from the process definition m, and reconnects the tasks preceding and following it. The Migrate transition performs migration of existing process instances stored in the Running instance place only if the process instance has not reached the end. For the process instance whose process model id matches with the id of the process definition earlier modified, the migrate() function transfers the old process definition oldm to the new process definition newm according to a desired migration policy.

**Implementation** Of all systems analyzed, only Declare and ADEPT1 support this pattern. In addition to the functionality for removing a task from the model, they also provide a means of process instance migration.

**Issues** The same issues as identified for the Permanent Entry-Point Change pattern (cf. page 345) and the Momentary Task Elimination pattern apply here too.

**Solutions** See solutions identified for the above mentioned patterns.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task to be permanently removed from the process definition, and provides facilities for the process instance migration.

### 6.2.7 Flexible extension

This group of patterns aims to provide flexibility in enabling an execution path alternative to the one prescribed by the process definition by incorporating a task that has not previously been foreseen (i.e. no reordering or selection). Figure 313 illustrates the scope of
patterns presented in this subsection and their relationship to different types of flexibility.

<table>
<thead>
<tr>
<th></th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>Alternative</td>
<td>Entrance</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative</td>
<td>Termination</td>
<td>Undefined exit</td>
<td>Momentary exit change</td>
<td>Permanent exit change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Choice</td>
<td>Task substitution</td>
<td>Late selection</td>
<td>Momentary choice insertion</td>
<td>Permanent choice insertion</td>
</tr>
<tr>
<td>Flexible reordering</td>
<td>Interleaving</td>
<td>Swap</td>
<td></td>
<td>Momentary reordering</td>
<td>Permanent reordering</td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>Foreseen</td>
<td>Task skip</td>
<td></td>
<td>Momentary task elimination</td>
<td>Permanent task elimination</td>
</tr>
<tr>
<td>Flexible extension</td>
<td>Task invocation</td>
<td>Late creation</td>
<td>Momentary task insertion</td>
<td>Permanent task insertion</td>
<td></td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 313:** Process Flexibility Matrix: flexible extension

In situations, where at run-time a task needs to be executed which has not been foreseen in the process definition, it should be possible to easily incorporate the desired task in order for desired behavior to be achieved. The need for flexible extension in this group of patterns is recognized at run-time. The concept of flexible extension cannot be facilitated at design-time, because all execution alternatives that are foreseen at design-time can explicitly be included into the process definition by applying the Choice pattern (cf. page 359) as illustrated in Figure 314.

There are four flexible extension patterns: Task Invocation, Late Creation, Momentary Task Insertion and Permanent Task Insertion which describe four approaches to incorporating unforeseen tasks in order for a desired execution option to be obtained at run-time.
The Task Invocation pattern allows for deviation from the execution path prescribed by invoking a task different to the one that is currently enabled without replacing any existing task. The Late Creation pattern offers the possibility to create an alternative execution path at a particular point in the process at the last possible moment based on the need to introduce optional behavior that could not be foreseen at design-time. The Momentary Task Insertion and Permanent Task Insertion patterns allow for inclusion of a task that has not been foreseen at design-time in the process definition on a temporary or permanent basis respectively.

**Pattern** PF-24 TASK INVOCATION

**Description** During execution, at any time after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by invoking a task which has not yet been executed and completing it before the currently enabled task is executed. This allows for flexible extension of the execution options associated with a given process instance with one of the tasks contained in the process definition. Such a deviation does not affect the process definition, and corresponds to the flexibility by deviation type.

**Examples**
- Students possessing diploma’s from European institutions do not have to pass a language test in order to enroll into the masters program. In some cases however, the level of English ability may need to be assessed. For this an appointment is made with an advisor handling English tests for foreign students.
- A patient treated in the department of internal diseases has developed a skin rush. To handle this problem a doctor from the dermatological department has been called.

**Motivation** In some situations, after a process has been initiated, it may be necessary to postpone the execution of a certain task because another task that has not been previously foreseen at design-time needs to be executed first. The Task Invocation pattern allows for suspension of the execution of tasks prescribed by the process definition in order for another task to be invoked and executed first. The possibility of extending the set of tasks prescribed by the process definition in this pattern corresponds to flexibility by deviation, which has an impact only on the manner in which the given process instance is executed, and does not affect the original process definition.

**Overview** Figure 315 illustrates the graphical notation for the Task Invocation pattern. The top view illustrates a process instance where after executing task B the need to execute a task different to the one prescribed by the process definition is recognized.
Figure 315: Task Invocation pattern

The bottom view shows that after applying the pattern task E has been invoked and the thread of control from task C has been temporarily moved to task E. Note that task E is a generic representation of a task that is contained in the process definition but which has not been executed yet. The invocation of this task is performed without modifying the process definition.

Context  Figure 316 illustrates the basic process engine enhanced with the Task Invocation pattern using the CPN formalism. The model adopts the structure for deviation operations described earlier for the Termination Skip pattern on page 351. The Invoke Task transition can only be executed for a process instance whose execution has not completed yet. The dev_action() function selects a task that has not yet been executed and which will be invoked before executing the currently enabled task. The invoked task is recorded in the Log place, and the state of the process instance is updated by the update_dev() function so that the originally enabled task becomes enabled again.

Implementation  Of all of the systems analyzed, only FLOWer, Declare, and Oracle BPEL provide support for this pattern. In FLOWer, another task can be invoked at any point in the process. In Oracle BPEL PM, an exception needs to be raised and in the corresponding exception handler an invocation activity needs to be included.

Issues  None identified.
Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows a task contained in the process definition which has not yet been executed to be invoked. Execution of the main process must be suspended until the execution of the invoked task completes.

Pattern PF-25 LATE CREATION

Description A process definition created at design-time contains a placeholder which may be completed at run-time in order to allow for an extension of the process definition with a task whose content cannot be foreseen in advance. The completion of the placeholder is optional and is performed only if an additional task needs to be added at run-time. The need for flexible execution path selection in this pattern is recognized at design-time, whose actual realization is performed at run-time, thus this pattern is of the flexibility by underspecification type.

Examples

– The process of submitting electronic photos for printing starts with the uploading of photos into a photo application, selection of the print format and submission of the order. The client may submit the order immediately, or order additional products offered by the company, and only then proceed to order submission.
– The software program supplied with the navigation system offers a default voice for route directions. When defining a route, a user may also choose to record their own instructions and then move on to planning the route.

Motivation At design-time when creating a process definition it may be foreseen that an additional task may need to be added at run-time, whose content is either unknown or may vary for different process instances. The Late Creation pattern allows for the possibility of creating an additional task at the latest possible moment to be included in the process definition in the form of a placeholder. The definition of this placeholder may be completed at run-time if for a given process instance a new task needs to be added. Otherwise it can be ignored.

Overview Figure 317 illustrates the graphical notation for the Late Creation pattern. The top view shows a process definition where after executing task B there is the possibility of executing task C or creating another task and executing it before proceeding to task C. The possibility for late creation of a task is incorporated into the process definition by means of a placeholder.

The bottom view shows that after applying the pattern, a new task E is created when completing the definition of the enabled placeholder, and the thread of control is passed to this task.

Context The CPN semantics of the Late Creation pattern can be described in terms of the Choice pattern (cf. page 359) and the Late Selection pattern (cf. page 363). In order for flexibility in process extension facilitated by the Late Creation pattern to be obtained, the process definition created at design-time has to include a choice construct with two alternative branches: one representing the normal flow of tasks and another containing a placeholder task whose content can be created when needed at run-time. Figure 318 pattern shows the basic process engine modified in a similar way as has been done for the Late Selection pattern. The enabling condition for the Execute task transition has been weakened in order to allow the user to choose whether to execute a concrete task
or to define the content of an underspecified task. This condition is incorporated in the existsConcrete(pid,m,st) function.

![Diagram of Late Creation pattern]

**Figure 317:** Late Creation pattern

...
**Pattern** PF-26  MOMENTARY TASK INSERTION

**Description** During execution, at any time after process initiation, the process definition associated with a particular process instance can be modified by extending the process definition with a task that has not been foreseen. The execution of the currently enabled task is postponed until the newly identified task has been completed. The change applies only to a given process instance and has no effect on other (current or future) process instances, thus it corresponds to the _flexibility by momentary change_ type.

**Examples**
- A timesheet used for recording activities performed by an employee during the day contains a list of predefined tasks. If a non-standard task that is not included in the list needs to be performed, the timesheet for the corresponding day can be adjusted by inserting the new task.
- Due to the unexpected illness of an employee, the tasks of this employee are divided amongst other members of the group. Each employee to whom an extra task has been assigned adjust their personal planning accordingly.

**Motivation** When at run-time a desired task cannot be found in a process definition, it should be possible to introduce this task on a temporary basis. The _Momentary Task Insertion_ pattern allows the set of tasks executed by a particular process instance to be extended by temporarily inserting a task into the corresponding process definition.

**Overview** Figure 319 illustrates the graphical notation for the _Momentary Task Insertion_ pattern. The top view shows two process instances populated based on the same process definition before applying the pattern. Both process instances are in a state where task C is enabled and needs to be executed. For one of the process instances, the need to execute a task not contained in the process definition is recognized.

![Figure 319: Momentary Task Insertion pattern](image)

The bottom view shows the process definitions associated with given process instances after applying the pattern. The process instance where the new task was introduced has been extended with task E, and the execution thread has been moved backward from the previously enabled task C to task E. The other process instance remains unaffected.

**Context** Figure 320 illustrates the basic process engine enhanced with the _Momentary Task Insertion_ pattern using the CPN formalism. The process engine is extended according to the structure adopted by all patterns facilitating flexibility by momentary change (cf. page 343). To realize the functionality for extending the set of tasks associated with a particular process instance the `change_possible()` and `update_p()` functions have been
updated as follows. The `change_possible()` function specifies that a new task that previously has not been foreseen be inserted into the process definition by the Momentary task insertion transition only if for a given process instance there are enabled tasks that have not yet been executed. The `update_pi()` function creates a new task, selects an arc connecting two tasks between which the created task will be inserted, and replaces this arc with two arcs connecting the preceding and subsequent tasks to the created task.

**Implementation** Of the systems analyzed, only Declare, ADEPT1 and YAWL support this pattern. In Declare and ADEPT1, there is an explicit operation defined allowing a new task to be inserted in a model. In YAWL, at run-time execution it is possible to invoke an exlet in order for a process fragment (which was not earlier foreseen) to be executed while the main process is suspended.

**Issues** By inserting a task between two tasks in the process, any data assumptions that exist between these two tasks can potentially be disrupted and may cause the last task to block.

**Solutions** In order to solve this problem, the visibility of data used in the process has to be set to the process instance level, so that data produced at each step during the process becomes visible to other tasks.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task to be inserted into a process definition associated with a given process instance at run-time.

**Pattern** PF-27 PERMANENT TASK INSERTION

**Description** During execution, there is the possibility to permanently modify the process definition by adding a new task. This allows for functionality that has not been foreseen earlier to be incorporated in all existing and future process instances. The change performed affects future process instances directly, whilst existing process instances require their old process definition to be migrated to the new process definition, thus this pattern corresponds to the flexibility by permanent change type.

**Examples**
The introduction of electronic identity cards requires all traveling passengers to hold a passport with an electronic id. Passengers whose passport has not expired yet, continue to use it during the transition period, however when this passport expires a new one must be requested. The issuing of an electronic id is an additional step performed after the new passport has been acquired.

The takeover of one company by another requires product certification, which previously has not been done. This implies that all ongoing requests have to be terminated and redone according to the new steps dictated by the certification standard.

**Motivation**  In situations, where at run-time the need to extend the set of tasks contained in a process is anticipated, a set of actions may need to be taken in order to redesign the original process definition and to adapt the execution of current and future process instances. The *Permanent Task Insertion* pattern allows a task that has not previously been foreseen to be included in the process definition on a permanent basis.

**Overview**  Figure 321 illustrates the graphical notation for the *Permanent Task Insertion* pattern. The top view shows several process instances sharing the same process definition. At run-time execution the need to execute task E which has not previously been foreseen is recognized at the type level.

![Figure 321: Permanent Task Insertion pattern](image)

The bottom view shows that after applying the pattern, the process definition has been extended with task E. The process instance that has passed this point remains unaffected, while for other process instances the thread of control has been migrated from previously enabled task C to the newly inserted task E.

**Context**  Figure 322 illustrates the process engine enhanced with the *Permanent Task Insertion* pattern using the CPN formalism. The model of the process engine is extended with the *Permanent task insertion* and *Migrate* transition according to the structure adopted by all patterns facilitating flexibility by permanent change and described earlier on page 345. In order to insert a task that previously has not been foreseen into the process definition at the type level by means of the *Permanent task insertion* transition, functions `change_possible()` and `modify_m()` have been modified as follows. A newly created task is inserted into a selected arc by means of the `modify_m()` function by splitting the selected arc into two parts. Process instances corresponding to the process model which has been recently modified, which are stored in the *Running instance* place, may need to be migrated to the new process definition. The `transfer_possible()` function checks whether transfer is possible for the given process instance (i.e. there are enable tasks which have not yet been executed and transfer is possible). If migration needs to be performed, the `migrate()` function transfers the corresponding process instances from the old to new process definition.
Figure 322: Engine enhanced with the Permanent Task Insertion pattern

**Implementation**  Of all of the systems analyzed, only Declare and ADEPT1 support this pattern. They offer an operation for inserting a task into a process model and allow existing process instances to be migrated to the new process definition.

**Issues**  The same issues as identified for the Permanent Entry-Point Change pattern (cf. page 345) and the Momentary Task Reordering pattern apply here also.

**Solutions**  See solutions identified for the Permanent Entry-Point Change pattern.

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering that allows a task to be permanently inserted into a process definition at run-time, and provides facilities for migrating existing process instances from the old process definition to the new one.

### 6.2.8 Flexible concurrency

This group of patterns aims at achieving flexibility when executing several independent tasks concurrently, thus avoiding unnecessary dependencies between them. Although a similar outcome can be achieved by applying patterns facilitating flexible reordering, the focus of this group of patterns is not to change the order of tasks, but to achieve the desired concurrency. Figure 323 illustrates the scope of patterns presented in this subsection and their relationship to the other types of flexibility.

Often sequential processes which require that tasks be executed one after another contain a set of tasks whose execution does not depend on each other. To decrease the duration of the process and make its execution more efficient, whereas possible independent tasks should be undertaken concurrently. The flexible concurrency patterns describe different ways in which dependencies between independent tasks can be decreased by letting them run in parallel. This group contains three patterns: Parallelism, Momentary Task Parallelization and Permanent Task Parallelization corresponding to flexibility by design, momentary change and permanent change respectively. Note that no mapping of flexible concurrency to the concepts of flexibility by underspecification can be made. Flex-
Flexibility by underspecification assumes that the content of a particular process fragment is not foreseen at design-time and any necessary control-flow dependencies are not known, whilst the tasks whose execution needs to be parallelized in this case are well-defined and no dependencies between these tasks are required. The mapping to flexibility by deviation corresponds to the *Swap* pattern described on page 371, which enables the deviation from executing the currently enabled task to invoking another task to occur. In light of the interleaving semantics of CPN Tools the same realization mechanism as used for the *Swap* pattern also applies for supporting flexible concurrency. In the *Parallelism* pattern, the need for concurrent task execution is recognized at design-time and is realized by means of parallel branching. The *Momentary Task Parallelization* and *Permanent Task Parallelization* allow the introduction of concurrent behavior based on parallel branching at the instance and type levels respectively.

**Pattern** PF-28 **PARALLELISM**

**Description** A process definition specified at design-time contains a construct that allows two independent tasks to execute simultaneously or in any order. This allows for more concurrency during the execution of tasks and avoids unnecessary waiting and dependencies. The need for flexible process concurrency in this pattern is anticipated at design-time, and corresponds to the *flexibility by design* type.

**Examples**
- In high-performance computing large problems are often divided into small units which are then processed concurrently.
- Where the production of parts for car assembly involves production activities for parts which are not dependent on each other, their individual production processes can be undertaken in parallel.

**Motivation** When a set of tasks is identified at design-time which can be executed independently, it may be necessary to specify that they can be executed concurrently or in any order. The *Parallelism* pattern allows two independent tasks to execute concurrently
by introducing two branches which can be undertaken in parallel. At run-time execution these tasks may be executed in either order or concurrently.

**Overview** Figure 321 illustrates the graphical notation for the Parallelism pattern. The top view shows the process definition where task C is executed after task B has completed. At design-time it is recognized that tasks B and C are independent of each other, and there is no need to enforce the execution ordering between them.

![Figure 324: Parallelism pattern](image)

The bottom view shows the process definition after applying the pattern. In this process definition, tasks B and C have been put in parallel, i.e. they can be executed in either order or concurrently after the completion of task A.

**Context** The basic process engine (cf. Figure 325) supports the Parallelism pattern as follows. In order to specify the parallelism between two tasks B and C, which have been identified as independent tasks at design-time, and which need to be become enabled concurrently after task A has completed, the split and join connectors of associated divergence and synchronization tasks have to be set to AND. After executing task A by means of the Execute task transition, the ns() function analyzes the type of split of the executed task, and if it corresponds to the AND-split enables all outgoing subsequent arcs. Note that although theoretically these tasks can be executed concurrently, CPN Tools assumes an interleaving semantics which allows these tasks to be executed in any order but not simultaneously.

![Figure 325: Basic process engine: support for Parallelism](image)

**Implementation** This pattern is supported by all of the systems analyzed. In Oracle BPEL PM, this pattern is supported by means of the <flow> construct with several concurrent branches. In GLIF and EON, a special branching block needs to be inserted with which parallel branches have to be associated. In AsbruView, the Parallel plan can be used for this purpose. YAWL enables parallel execution by associating the AND-split
connector with a task. In PROforma, parallel execution can be enabled by incorporating the Decision construct.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows two or more tasks to be executed concurrently at run-time, based on the parallel execution paths specified for them in the process definition at design-time.

**Pattern** PF-29  **MOMENTARY TASK PARALLELIZATION**

**Description** During execution, at any time after process initiation, a process definition associated with a particular process instance can be modified by parallelizing the execution of two subsequent independent tasks. This gives flexibility in initiating these tasks allowing them to execute in any order as well as concurrently. The change applies only to a given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

**Examples**
- Due to change in planning, all activities are required to be completed earlier than usual, therefore in addition to the currently executing task an employee initiates another activity that has been planned for a later time.
- The workshop program consists of a set of presentations, followed by a feedback session where students complete evaluation forms. To shorten the workshop program, students have been asked to fill in the evaluation form while attending one of the presentations.

**Motivation** When at run-time several independent tasks have been identified, which are prescribed by the process definition to execute sequentially, it should be possible to let them run concurrently or in any order. The Momentary Task Parallelization pattern allows for concurrent execution of two independent tasks by temporarily parallelizing them for a given process instance.

**Overview** Figure 321 illustrates the graphical notation for the Momentary Task Parallelization pattern. The top view shows two process instances populated from the same process definition. For one of the process instances the need to execute enabled task B concurrently with its successor task C is recognized.

![Diagram](image-url)  

**Figure 326:** Momentary Task Parallelization pattern
The bottom view shows that after applying the pattern for a given process instance the process definition has been changed: tasks B and C have been parallelized, and an additional execution thread has been spawned off in order to enable task C.

**Context** Figure 327 illustrates the basic process engine enhanced with the Momentary Task Parallelization pattern using the CPN formalism. The model of the engine is extended using the structure adopted by all patterns facilitating flexibility by momentary change (cf. page 343). The Momentary parallelization transition can be executed in order to parallelize a currently enabled (sequential) task with a subsequent task that is yet to be executed. The change_possible() function checks whether for the given process instance stored in the Running instance place there exist an enabled task that still has to be executed, and whether there exist two adjacent tasks with no join and split connectors (these tasks should different from the start and end tasks). The update_pi() function selects an arbitrary arc between two sequential tasks (other than those being parallelized), deletes the sequential dependency between them, modifies the join and split connectors of the corresponding divergence and synchronization tasks and adds missing arcs from the divergence and synchronization tasks to the parallelized tasks. After the change the process instance state is updated. If the currently enabled task has been involved in the change, then the updated state would contain two enabled tasks.

**Figure 327:** Engine enhanced with the Momentary Task Parallelization pattern

**Implementation** Of all of the systems analyzed, only Declare supports this pattern. To allow another task to be executed concurrently with currently enabled task, the constraints associated with the task to be invoked have to be removed. Although ADEPT1 provides flexibility by change, it does not allow for the parallelization of sequential tasks.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the order of tasks in the process definition associated with a given process instance to be modified in such a way that two independent sequential tasks are parallelized, and may be executed concurrently.
Pattern PF-30  PERMANENT TASK PARALLELIZATION

Description  During execution, there is the possibility to permanently modify the process definition by transforming the sequential ordering of previously independent tasks into a concurrent structure, which allows these tasks to execute simultaneously or in any order. The parallelization of tasks is performed in the process definition at the type level on a permanent basis. The change performed affects future process instances directly, and existing process instance require migration from the old process definition to the new process definition, thus it corresponds to the flexibility by permanent change type.

Examples
- The education program at a school has been revised allowing students to follow independent courses in parallel. The previous version of the program allowed only for participation in a new course only after the previous course has been completed.
- It is common practice in hospitals to solve the health problem of one patient at a time. In order to decrease waiting times, patients in the same family are accepted together and treated concurrently.

Motivation  When at run-time the possibility for concurrent execution of independent tasks is recognized there may arise the need to restructure the original process definition by parallelizing the execution order of these tasks. The Permanent Task Parallelization pattern allows two independent tasks to be parallelized on a permanent basis in order to provide flexibility in defining the execution order for these tasks.

Overview  Figure 321 illustrates the graphical notation for the Permanent Task Parallelization pattern. The top view shows three distinct process instances populated from the same process definition. At run-time execution for the given process definition the need to relax the dependency between tasks B and C at the type level is recognized.

The bottom view shows that after applying the pattern tasks B and C have been parallelized. For the process instance that has passed these tasks this change is inconsequential. For the other two process instances, where task B has been enabled, an additional thread of control has been spawned off in order to execute task C independently of task B. Both tasks have to complete in order for task D to become enabled.

Context  Figure 329 illustrates the basic process engine enhanced with the Permanent Task Parallelization pattern using the CPN formalism. The model of the process engine is extended using the structure described earlier in the Permanent Entry-Point Change pattern on page 345.

Two sequential tasks can be put in parallel by the Permanent parallelization transition in the process definition only if conditions specified by the change_possible() function are satisfied. In particular, there must be an arc between two sequential tasks (with
no splits or joins), and these tasks must be different from the start and end tasks. The \texttt{modify}_m() function selects an arbitrary arc between sequential tasks (other than those being parallelized), identifies their synchronization tasks, sets the join and split connectors of the divergence and synchronization tasks to AND, deletes the selected arc, and adds two other arcs connecting the synchronization tasks with the tasks recently parallelized instead. After the change has been performed, it may be necessary to migrate existing process instances stored in the \texttt{Running instance} place. The \texttt{transfer\_possible()} function checks whether there mismatches between the old and new versions of the process definitions and whether the transfer is possible. The \texttt{migrate()} function performs the transfer of a process instance from the old to new process definition according to the specified migration strategy.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure329.png}
\caption{Engine enhanced with the Permanent Task Parallelization pattern}
\end{figure}

**Implementation** Of all of the systems analyzed, only ADEPT1, Declare and YAWL offer some support for this pattern. Declare allows previously sequential tasks to be concurrently enabled by removing the dependency constraints between them. Furthermore, it offers the possibility to transfer existing process instances to the new process definition. In order for two tasks to be put in parallel in ADEPT1, one of them have to be removed and inserted back in parallel with the other task. In YAWL, an exlet needs to be called in which given two tasks will be removed from the process model and inserted in the desired order by defining a new worklet.

**Issues** The same issues as identified for the \textit{Permanent Entry-Point Change} pattern (cf. page 345) also apply here.

**Solutions** See solutions identified for the \textit{Permanent Entry-Point Change} pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the execution order of two sequential tasks to be parallelized, providing that existing process instances can be migrated from the old process definition to the new one.
6.2.9 Flexible repetition

The last group of patterns aims at flexibility in repeating the execution of a particular task variable number of times. Figure 330 illustrates the scope of patterns presented in this subsection and their relationship with the other types of flexibility.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Design</th>
<th>Deviation</th>
<th>Flexibility by</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>Alternative entry points</td>
<td>Entrance skip</td>
<td>Underspecification</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative exit points</td>
<td>Termination skip</td>
<td></td>
<td>Momentary exit change</td>
<td>Permanent exit change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Choice</td>
<td>Task substitution</td>
<td>Late selection</td>
<td>Momentary choice insertion</td>
<td>Permanent choice insertion</td>
</tr>
<tr>
<td>Flexible reordering</td>
<td>Interleaving</td>
<td>Swap</td>
<td></td>
<td>Momentary reordering</td>
<td>Permanent reordering</td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>Foreseen bypass path</td>
<td>Task skip</td>
<td></td>
<td>Momentary task elimination</td>
<td>Permanent task elimination</td>
</tr>
<tr>
<td>Flexible extension</td>
<td>Task invocation</td>
<td>Late creation</td>
<td></td>
<td>Momentary task insertion</td>
<td>Permanent task insertion</td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td>Parallelism</td>
<td></td>
<td></td>
<td>Momentary task parallelization</td>
<td>Permanent task parallelization</td>
</tr>
<tr>
<td>Flexible repetition</td>
<td>Iteration</td>
<td>Redo</td>
<td></td>
<td>Momentary loop insertion</td>
<td>Permanent loop insertion</td>
</tr>
</tbody>
</table>

Figure 330: Process Flexibility Matrix: flexible repetition

In some situations, when creating a process definition it may be foreseen that a particular task may need to be executed multiple times, but the number of times it will need to execute may vary or be unknown. The flexible repetition patterns describe different ways of achieving the behavior allowing a certain task to be executed multiple times. Of the four patterns identified, the *Iteration* pattern represents a generic form of loop introduced in the process definition at design-time, which allows tasks residing in the body of the loop to be executed repeatedly until the loop termination condition is satisfied. There are different forms of iterations possible, such as those described by the control-flow repetition patterns described on page 154. However, in the context of this research we abstract from different forms of the loops and consider its generic form. The *Redo* pattern applies to situations where after executing a particular task the decision to repeat it at some later point in time is taken. The *Momentary Loop Insertion* and *Permanent Loop Insertion* patterns represent variants of the *Iteration* pattern realized at run-time by modifying the process definition at the instance and type levels respectively. Note that this group contains no patterns corresponding to flexibility by underspecification. This has to do with the fact that the content of the task and its position are known and thus do not need to be represented as placeholders.

**Pattern** PF-31 **ITERATION**

**Description** A process definition specified at design-time contains a loop which allows the execution of a given task to be repeated. Typically, the decision to enable the task in the loop or continue with the subsequent task depends on the evaluation of a data condition associated with this task. This gives flexibility in realizing execution sequences where the task embedded in the loop may need to be executed multiple times. The need
for flexible task concurrency in this pattern is anticipated at design-time and corresponds to the flexibility by design type.

**Examples**
- For a driving license a candidate is required to take a theory exam. If the first attempt is unsuccessful, a candidate may repeat the test until they finally pass it.
- The blood examination in a laboratory is repeated for a sample until all required elements have been identified.

**Motivation** In situations where it is recognized (at design-time) that a certain task may need to be executed a variable number of times, it should be possible to specify that the execution of this task should be repeated. The *Iteration* pattern provides flexibility in repeating the execution of a certain task by embedding it in the body of a loop, which can be iterated as many times as required.

**Overview** Figure 331 illustrates the graphical notation for the *Iteration* pattern. The top view shows the process definition in which each task executes only once. At design-time for this process definition the possibility of executing task *C* multiple times is recognized.

The bottom view shows that after applying the pattern task *C* has been modified in such a way that it can be executed multiple times. The number of times a given task is executed in distinct process instances may vary.

**Context** The *Iteration* pattern is supported by the basic process engine in a similar way as for the *Choice* pattern (cf. page 359). The possibility of repeating the execution of a specific task can be incorporated into the process definition at design-time by inserting a choice construct forming a self-loop. For instance, in order to specify that after executing task *C* either another instance of task *C* needs to be executed or task *D*, both the join and split connectors of task *C* have to be set to XOR, and an additional arc (C,C) needs to be added to the set of arcs associated with the process model. In Figure 332, after executing task *C* by means of the *Execute task* transition, the ns() function calculates a new process instance state. When the value of the split connector associated with a task has been evaluated, for the XOR-type only one task out of several possible options is selected non-deterministically. This means that the number of times the task will iterate is not predefined, but is determined on a random basis.

**Implementation** In each of the systems analyzed, it is possible to achieve task iteration by embedding a task in a loop (for this, the same support as for the *Choice* pattern is required). In PROforma, there exists an option to set an iteration attribute associated with a specific task to a value or to an expression, thus allowing the number of task iterations to vary. In Oracle BPEL PM, a task needs to be inserted into a <while> loop, whose enabling condition defines whether a task residing in the loop will be iterated again. In FLOWer, iteration of a specific task can be achieved by inserting this task in the body of a sequential plan.
Issues In order to define whether a given task can be repeated, a data-based condition identifying whether to repeat the task has to be defined. The number of times the execution of the given task will be repeated depends on the moment of the data condition is evaluated.

Solutions There exist two variants of task iteration: a ‘while’ loop and a ‘repeat-until’ loop. In case of the ‘while’ loop the condition is evaluated before executing the task, whilst in case of the ‘repeat-until’ loop the data condition is evaluated after the task has been completed. For implementation details see the Structured Loop control-flow pattern on page 155.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows the execution of a specific task to be repeated multiple times based on a data condition included in the process definition at design-time.

Pattern PF-32 REDO
Description During execution, after process instance initiation, there is the possibility of deviating from the execution path prescribed by the process definition by repeating the execution of a task that has recently been completed. The need to execute a particular task more often than prescribed by the process definition is anticipated at run-time. This has no effect on the process definition, and corresponds to the flexibility by deviation type.

Examples
- Due to unexpected changes in travel plans, one of the reservations had to repeated in order to change the flight to another date.
- The results of software testing showed some inconsistencies in the software design. This triggered revision of the design.

Motivation When at run-time it is recognized that the execution of a particular task did not go as well as expected, it should be possible to perform the given task again. The Redo pattern offers flexibility in deviating from the execution path prescribed by the process definition by allowing the execution of the currently enabled tasks to be deferred until the execution of previous task has been repeated.

Overview Figure 333 illustrates the graphical notation for the Redo pattern. The top view shows the process instance for which the need to repeat the previously executed task C is recognized.

The bottom view shows that after applying the pattern the thread of control has been moved in the given process instance from task D to task C, thus allowing its execution to be repeated.
**Context**  Figure 334 illustrates the basic process engine enhanced with the *Redo* pattern using the CPN formalism. The *Redo* transition allows for deviation from the normal prescribed execution order by executing a task that has already been completed (and a record of which is available in the *Log* place). The `deviation_possible()` function specifies that in order to redo a task, the list of executed tasks stored in the *Log* place should not be empty. The task whose execution needs to be redone is randomly selected by the `dev_action()` function. Note that the execution state of the given process instance and the process definition remain unchanged.

**Implementation**  Of all systems analyzed only FLOWer and Oracle BPEL PM support this pattern. FLOWer offers a special operation to redo the execution of a previously executed task. In Oracle BPEL PM the only possible way to redo a task is to raise an exception or trigger an event upon which a previously executed task will become enabled again.

**Issues**  When the execution of a specific task needs to be redone, the problem of overwriting data obtained during the first execution can occur.

**Solutions**  Depending on whether there are data dependencies between the task that needs to be repeated and its subsequent task(s), it may be necessary to roll back the execution of these (subsequent) tasks. In order to avoid overwriting data, the values provided for data elements when executing the task to be redone can be marked as ‘unconfirmed’ rather than being discarded [114].

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering
that allows the execution of a previously executed task to be redone.

**Pattern** PF-33  MOMENTARY LOOP INSERTION

**Description** During execution, at any time after process initiation, a process definition associated with a particular process instance can be *temporarily* modified by inserting a particular task in a loop. This allows the selected task to be executed more often than originally prescribed by the process definition. The need for momentary task repetition is recognized at run-time. This change applies only to the given process instance and has no effect on other (current or future) process instances, thus it is of the *flexibility by momentary change* type.

**Examples**
- A patient who arrived at the emergency care center has been given an injection in the stomach for a dog bite. The injection is required to be repeated daily for 10 days.
- During process execution there arose a need to repeat a particular task over a specific period of time. The task planning section of the configuration window has no option to repeat a particular task. To make this possible the advanced settings of the configuration window have been changed by marking the “task repetition” field.

**Motivation** When at run-time a task has been identified which needs to be executed more often than prescribed by the process definition, it should be possible to adjust the process definition in order to allow the execution of the given task to be repeated multiple times. The *Momentary Loop Insertion* pattern allows a sequential task to be incorporated into a loop in a given process instance in order for multiple subsequent executions of the given task to be achieved.

**Overview** Figure 335 illustrates the graphical notation for the *Momentary Loop Insertion* pattern. The top view shows two distinct process instances based on the same process definition. At run-time execution, the need for iteration of the currently enabled task C is recognized in one of these process instances.

![Figure 335: Momentary Loop Insertion pattern](image)

The bottom view shows that after applying the pattern the process definition associated with the given process instance has been modified by transforming task C into a repetitive task. Note that the other process instance remains unaffected.

**Context** Figure 336 illustrates the basic process engine enhanced with the *Momentary Loop Insertion* pattern using the CPN formalism. The model of the engine adopts the structure for momentary changes described earlier in the *Momentary Entry-Point Change* section.
pattern on page 343. In order to allow the execution of the currently enabled task to be repeated, it needs to be inserted in the loop by executing the **Momentary loop insertion** transition. The `change_possible()` function specifies that a loop can only be inserted in a given process instance if there exists an enabled task that still needs to be executed, and if the currently enabled task has no split or join connectors of **AND** type. The `update_pi()` function modifies the split and join connectors of the selected task to **XOR**, and adds an arc representing a self-loop to the set of arcs associated with the given process instance. Note that the `update_pi()` function can be changed in such a way that not only the currently enabled task but any other task that still needs to be executed can be inserted in the loop.

![Figure 336: Engine enhanced with the Momentary Loop Insertion pattern](image)

**Implementation** Of all of the systems analyzed, only Declare supports this pattern. Declare allows the execution of a task to be repeated by changing its cardinality constraint. In YAWL, an exlet needs to be called, which requires a new worklet containing a loop to be defined. Once the defined worklet has been defined, it becomes available to other process instances (this considered to be a change of permanent nature rather than momentary).

**Issues** Depending on the moment when the decision to repeat the execution of a particular task is made (i.e. before or immediately after it has been executed), different forms of a loop may need to be used.

**Solutions** Having completed a specific task, the only possibility to repeat its execution is to insert it in a loop of the ‘repeat-until’ type. The condition associated with this loop will be evaluated at its conclusion, allowing the execution of this task to be repeated if the loop termination condition has not been satisfied. When the need to repeat the execution of a particular task is recognized prior to task execution, both the ‘repeat-until’ and ‘while’ forms of a loop are possible. In the ‘while-do’ loop, the decision about executing the task is taken before entering the loop (cf. the **Structured Loop** control-flow pattern on page 155).

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a specific task in a process definition associated with a given process instance to be inserted into a loop in order to allow this task to be executed multiple times.

**Pattern** PF-34 **PERMANENT LOOP INSERTION**

**Description** At run-time, a process definition can be permanently modified by inserting a particular task in a loop. This allows the number of times the selected task will be
executed may vary for different process instances. The need for task repetition is recognized at run-time. The change performed affects future process instances directly, and existing process instances may require migration from the old process definition to the new process definition, thus this pattern is of the flexibility by permanent change type.

**Examples**
- According to changes in the medicine prescription policy, patients do not need to visit a doctor each time a repeat prescription is required, but can pick up their medicine at their preferred pharmacy. The repeat prescription request is processed automatically by a new system installed at the drugstore.
- As a result of a reorganization an electronic time-registration system has been introduced. Employees of this company involved in reorganization have to register their time working hours weekly using the system installed.

**Motivation** In situations, where at run-time execution the need to execute a certain task multiple times is identified, it should be possible to modify the number of times the given task is required to execute. The Permanent Loop Insertion pattern allows the execution of a sequential task to be repeated multiple times by embedding this task into a loop.

**Overview** Figure 337 illustrates the graphical notation for the Permanent Loop Insertion pattern. The top view shows three distinct process instances populated from the same process definition. At run-time execution, the need to iterate task C is recognized at the type level.

![Figure 337: Permanent Loop Insertion pattern](image)

The bottom view shows that after applying the pattern, the process definition has been modified by transforming task C into an iterative task and all existing process instances have been affected.

**Context** Figure 338 illustrates the basic process engine enhanced with the Permanent Loop Insertion pattern using the CPN formalism. The model of the engine has been extended using the structure adopted by all patterns facilitating the flexibility by permanent change (as described in the Permanent Entry-Point Change pattern on page 345). The execution of the Permanent loop insertion transition inserts a selected task in a loop. This transition can only be executed if the process description m, stored in the Process definition place, contains more than two tasks and if there is a task that has no split and join connectors of AND-type. The modify_m() function selects an arbitrary task of the sequential type and modifies its split and join connectors to XOR value. The migration of existing process instances from the old process definition m to the new process definition newm is performed using the migrate() function.

**Implementation** Of all of the systems analyzed, only Declare offers support for this pattern by allowing the cardinality constraints associated with a specific task to be modified and migration of existing process instances to the new process definition to be performed. In YAWL, an exlet needs to be called, which requires a new worklet containing a loop to be
defined. Once the defined worklet has been defined, it becomes available to other process instances.

**Issues** The same issues as identified for the *Permanent Entry-Point Change* pattern (cf. page 345) also apply here.

**Solutions** See solutions identified for the *Permanent Entry-Point Change* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a process definition to be modified by permanently inserting a specific task into a loop. In addition, there must be support for process instance migration.

### 6.2.10 Discussion

In this subsection, we give an overview of the flexibility patterns identified and examine the relationships between them. Figure 339 depicts the process flexibility matrix with 34 patterns that have been identified. Some of the cells in this matrix are empty, indicating that the mapping of the process flexibility aspect to the specific flexibility type cannot be made or is not meaningful. In particular, there is no pattern providing flexibility by extension that can be mapped to flexibility by design. The reason for this is that all alternative execution paths that have been foreseen at design-time, can be included into the process definition using the *Choice* pattern. In the column for flexibility by underspecification, there are no patterns providing flexibility in task reordering, bypassing a task, task concurrency and task repetition. This has to do with the fact that these operations apply to tasks which are known at design-time (i.e. there is no need to underspecify their content).

Amongst the seven patterns characterizing flexibility by design, only the *Alternative entry points* and *Choice* patterns require direct support in order for flexibility in process initiation and flexibility in execution path selection to be achieved. The other five patterns if not directly supported can be expressed in terms of the *Choice* pattern. For instance, to
realize the Parallelism pattern the condition for selecting one of several available branches defined for the Choice pattern needs to be relaxed, allowing all branches to be selected. The Foreseen Bypass Path pattern represents a choice between executing a specific task or not executing it at all, thus this can be implemented using the Choice pattern also. The Interleaving pattern can be expressed using the Choice pattern by explicitly defining all possible ordering sequences, and selecting only one of them. The Iteration pattern can be seen as a special kind of choice where a decision to iterate a loop or to continue with subsequent activities needs to be taken.

Patterns related to flexibility by deviation have a corresponding mapping to each of the eight aspects of process flexibility identified. In this group of patterns, direct support for the Task Skip pattern and the Task Invocation pattern is required in order for the flexibility features offered by the other six patterns to be realized. In particular, the Entrance Skip pattern can be considered as a special variant of the Task Skip pattern. In order to start the execution of a process from a task other than the nominated one, the Task Skip pattern can be iteratively applied until the desired point in the process has been reached. The Termination Skip pattern can be also seen as a variant of the Task Skip pattern. The difference between them is that the effect of the Termination Skip pattern can be achieved by iteratively applying the Task Skip pattern, which moves the execution thread to a subsequent task rather than to the end of the process immediately. Having support for the Task Invocation pattern available, a task that is not currently enabled can be invoked in order to substitute the currently enabled task (i.e. the Task Substitution pattern), or to repeat a previously executed task (i.e. the Redo pattern). Combinations of the Task Skip and Task Invocation pattern are required in order for the Swap pattern to be realized.

The group of patterns supporting flexibility by underspecification consists of four patterns. To realize these patterns, support for placeholders is required. The completion of placeholders in each of these patterns can be done according to the late modeling and late binding strategies, introduced earlier on page 326.

Patterns characterized by flexibility by momentary change and flexibility by permanent change can be mapped to each of the eight process flexibility aspects. Support for the Momentary Entry Change can be accomplished by applying the Momentary Task Elimination pattern until the desired entry-point is reached. The support for this pattern is
also required in order for flexibility in process termination offered by the *Momentary Exit Change* pattern to be achieved. The *Momentary Loop Insertion* pattern requires support for the *Momentary Choice Insertion* pattern, whereas the *Momentary Reordering* can be achieved via the *Momentary Task Insertion* and the *Momentary Task Elimination* patterns. The same relationships apply to the group of patterns supporting flexibility by permanent change.

This concludes the discussion of requirements for PAISs from the process flexibility perspective. We now move on to a comprehensive evaluation of flexibility pattern support in the selected PAISs.

### 6.3 Tool evaluations

In this section, we show how the process flexibility patterns can be used for assessing and comparing process modeling languages and enactment mechanisms employed by a wide range of PAISs. We concentrate on ADEPT1, FLOWer, YAWL and Declare because they provide support for different types of flexibility. Furthermore, we analyze the capabilities of Oracle BPEL PM that is often used for the realization of interactive processes. Changes in one of the interacting processes may impact other processes involved in the interaction, thus it is interesting to see up to which extent this tool is capable to adapt to changes in the operating environment. In the health-care environment, unpredictable situations often occur, which require a very sensitive mechanism for adapting to changes. In such dynamic and unpredictable environment flexibility is an important issue. To check the degree of flexibility offered by CIG modeling languages used in the health-care domain for encoding medical guidelines we evaluate PROforma, EON, GLIF, and Asbru.

Table 6.3 summarizes the results of evaluating the selected PAISs. The full support (marked in Table 6.3 as “+”) for each of the patterns is defined on the basis of the evaluation criteria associated with them. Typically, there must be an explicit operation available that allows the effect of the flexibility approach identified by a pattern to be easily achieved. Partial support (marked in Table 6.3 as “+/-”) indicates that there is no direct way of realizing the desired behavior, however it can be achieved by executing a number of other steps. The absence of support (marked in Table 6.3 as “-”) indicates that either the desired behavior cannot be achieved or that a workaround solution requires significant effort and is hard to realize. In addition to the overview of supported patterns, Table 6.4 illustrates the support of patterns related to each of the five flexibility types.

Oracle BPEL PM offers good support for flexibility by design. By setting the `createInstance` attribute to “yes” different `<receive>` activities in a process model can serve as alternative start tasks. The ability to put the `<Terminate>` activity in different places in a process model allows the process to be terminated once any of these activities has been executed. Support for flexible selection, flexible reordering and flexible

---

8 Of these four modeling languages only PROforma can be evaluated from the perspective of five flexibility types identified: the Tallis tool is based on PROforma semantics and provides functionality for process execution. The other three languages are analyzed only from the perspective of flexibility by design. The run-time behavior associated with flexibility by deviation, flexibility by underspecification, flexibility by momentary and permanent change cannot be assessed due to non-availability of enactment engines for these languages.
Table 6.3: Support for the process flexibility patterns in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flexible Initiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-1</td>
<td>Alt. Entry-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>FP-2</td>
<td>Entrance Skip</td>
<td>-</td>
<td>-</td>
<td>+/</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-3</td>
<td>Undefined Entry</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-4</td>
<td>Mom. Entry Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-5</td>
<td>Perm. Entry Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Termination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-6</td>
<td>Alt. Exit-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-7</td>
<td>Termination Skip</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-8</td>
<td>Undefined Exit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-9</td>
<td>Mom. Exit Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-10</td>
<td>Perm. Exit Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-11</td>
<td>Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-12</td>
<td>Task Substitution</td>
<td>+</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-13</td>
<td>Late Selection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-14</td>
<td>Mom. Choice Insertion</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-15</td>
<td>Perm. Choice Insertion</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-16</td>
<td>Interleaving</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-17</td>
<td>Swap</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-18</td>
<td>Mom. Reordering</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-19</td>
<td>Perm. Reordering</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Elimination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-20</td>
<td>Foreseen Bypass Path</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-21</td>
<td>Task Skip</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-22</td>
<td>Mom. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-23</td>
<td>Perm. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Extension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-24</td>
<td>Task Invocation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-25</td>
<td>Late Creation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-26</td>
<td>Mom. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-27</td>
<td>Perm. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-28</td>
<td>Parallelism</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-29</td>
<td>Mom. Task Parallelization</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-30</td>
<td>Perm. Task Parallelization</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flexible Repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-31</td>
<td>Iteration</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-32</td>
<td>Redo</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-33</td>
<td>Mom. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-34</td>
<td>Perm. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6.4: Support for the process flexibility patterns in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flexibility by Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-1</td>
<td>Alt. Entry-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>FP-6</td>
<td>Alt. Exit-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-11</td>
<td>Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-16</td>
<td>Interleaving</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-20</td>
<td>Foreseen Bypass Path</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-28</td>
<td>Parallelism</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-31</td>
<td>Iteration</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Flexibility by Deviation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-2</td>
<td>Entrance Skip</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-7</td>
<td>Termination Skip</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-12</td>
<td>Task Substitution</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-17</td>
<td>Swap</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-21</td>
<td>Task Skip</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-24</td>
<td>Task Invocation</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-32</td>
<td>Redo</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexibility by Underspecification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-3</td>
<td>Undefined Entry</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-8</td>
<td>Undefined Exit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-13</td>
<td>Late Selection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-25</td>
<td>Late Creation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexibility by Mom.Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-4</td>
<td>Mom. Entry Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-9</td>
<td>Mom. Exit Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-14</td>
<td>Mom. Choice Insertion</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-18</td>
<td>Mom. Reordering</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-22</td>
<td>Mom. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-26</td>
<td>Mom. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-29</td>
<td>Mom. Task Parallelization</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-33</td>
<td>Mom. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexibility by Perm.Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-5</td>
<td>Perm. Entry Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-10</td>
<td>Perm. Exit Change</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-15</td>
<td>Perm. Choice Insertation</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-19</td>
<td>Perm. Reordering</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-23</td>
<td>Perm. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-27</td>
<td>Perm. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-30</td>
<td>Perm. Task Parallelization</td>
<td>-</td>
<td>+/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-34</td>
<td>Perm. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
elimination can be accomplished in Oracle BPEL PM using the `<switch>` construct, where alternative tasks, sequences of tasks or empty tasks can be defined in each of the branches. Support for flexible concurrency is provided by the `<flow>` activity, whilst flexible repetition can be obtained by using the `<while>`-loop construct. The ability to throw an exception in the scope of a specific task, allows a compensation action to be performed. In this way, the execution of one task can be substituted with another, a previously executed task can be redone, a process instance can be terminated or another task or process can be invoked, thus providing support for four deviation patterns. Although Oracle BPEL PM does not support the concept of a placeholder, it allows the definition of credentials of a process that needs to be invoked to be postponed until run-time. These can be set dynamically based on the input provided, thus corresponding to the Late Creation pattern. This system provides no means of support for flexibility by momentary change or flexibility by permanent change, because the process definition cannot be modified at an instance nor at a type level.

ADEPT1 is characterized by its support for structured workflows with a single entry and a single exit point. This feature is used for checking soundness when defining process models and migrating process instances from an old process definition to a new process definition, obtained as the result of change. This system scores well in supporting flexibility by design, flexibility by momentary change and flexibility by permanent change. ADEPT1 allows the process definition to be modified at run-time in order to facilitate adaptation to changes in the operating environment. The basic operations that are directly supported are adding and deleting a task, these correspond to direct support for flexibility in process extension and flexibility in task elimination. Patterns such as Momentary Entry Change, Momentary Exit Change, Momentary Choice Insertion, Momentary Reordering and Momentary Parallelization cannot be realized directly, however the effect of the flexibility mechanisms they facilitate can be achieved by applying the insert and delete task operations. The desired start task in a process can be achieved by deleting tasks in the beginning of the process one-by-one. Similar steps can be performed in order for a process to be terminated earlier. To reorder two tasks, they need to be removed and re-inserted in the correct order. The same holds for task parallelization. It is however not possible to insert a task in a loop, therefore the only patterns not supported in the group of momentary and permanent change patterns are the Momentary Loop Insertion and Permanent Loop Insertion. A notable feature supported by ADEPT1 is the migration of process instances to a new process definition. This explains why patterns in the group of permanent change are supported as well as those in the group of momentary change.

FLOWer is a case-handling system offering a lot of support for deviating from the execution order prescribed by the process definition. Similar to ADEPT1, FLOWer allows only a single entry to and single exit from the process model. This explains, why the Alternative Entry Points and the Alternative Exit Points patterns are not supported. The availability of various types of plans (i.e. sequential, dynamic) provide good support for the other six patterns in the flexibility by design group. The majority of patterns facilitating flexibility by deviation are supported by FLOWer directly. It is possible to start the execution of a non-enabled task directly, skip a task, invoke a non-enabled task concurrently with a currently enabled task or rollback and repeat a previously executed task. There is the possibility to perform a termination skip by executing a skip operation at the level of a root plan. The only two deviation patterns that are supported indirectly are the Entrance Skip pattern and the Task Substitution pattern. In order to skip all tasks preceding a desired start task, a skip operation has to be applied for each of the tasks individually.
order to execute a task different to the currently enabled one, the task needs to be invoked and the skip operation needs to be applied to the originally enabled task. FLOWer offers no support for flexibility by underspecification, flexibility by momentary change and flexibility by permanent change.

Declare is a system based on the declarative approach to process specification. It offers full support for flexibility by design-time, flexibility by momentary change and flexibility by permanent change. Furthermore, it supports the majority of patterns facilitating flexibility by deviation. From this group, the Task Substitution pattern and the Redo pattern are not supported. Such a broad range of patterns are supported by means of inserting and deleting tasks and/or constraints defining the execution order of these tasks. Support for process instance migration allows process instances to be transferred from an old process definition to a newly defined one, thus facilitating flexibility by momentary and permanent change.

The CIG modeling languages analyzed offer support for almost all patterns related to flexibility by design. Of all these languages, only PROforma allows a process to be started from and ended with alternative tasks (this is possible because of its combined imperative and declarative approaches). In GLIF, various start tasks can be defined allowing a patient to enter a process at any suitable stage. The rest of the patterns facilitating flexibility by design can be directly expressed in each of the languages examined.

Based on the results of the analysis, we can conclude that the intent of the system/language analyzed influences the types of flexibility supported. Systems based on declarative approaches (i.e. Declare) score well from the perspective of flexibility by deviation, because various types of behaviors may be achieved by simply adding, removing or modifying constraints associated with tasks. Case-handling systems such as FLOWer offer an explicit set of deviation operations in order for the desired behavior to be easily achieved at run-time, without requiring an underlying process definition to be modified. Support for flexibility by momentary and permanent change is facilitated in systems where the possibility of reacting to unforeseen events has been incorporated in the system design (i.e. migration mechanisms of ADEPT1, YAWL and Declare). The ability to leave a process definition underspecified, such that it can be completed at run-time is clearly an aspect of process flexibility that is less widely supported. The fact that Oracle BPEL PM and YAWL’s worklets extension accommodate service-based approaches illustrates the applicability of concepts related to flexibility by underspecification in the service-oriented domain. From the perspective of the health-care domain, where more flexibility is required due to unpredictable nature of events, CIG modeling languages score slightly better than traditional workflow systems. In particular, their the ability to initiate a process at an arbitrary point and terminate it “on spot” are typical operations in this domain.

Tables 6.5 and 6.6 summarize the results of evaluating the offerings against the process flexibility groups and process flexibility types respectively. In these tables, the rating (+++) is given to an offering that support more than half of the patterns per a given group or flexibility type; the rating (+++) is given to an offering supporting at least half of the patterns; the rating (+) is given to an offering if it supports one pattern; the rating (+/-) is given to an offering that offers no direct support for patterns; finally, the rating (-) indicates that no patterns are supported by the offering.
Section 6.4 Related work

The need for process flexibility has been acknowledged in the workflow and process technology communities as a critical quality of effective business processes in order for organizations to adapt to changing business circumstances [121, 180, 184]. The notion of flexibility is often viewed in terms of the ability of an organization’s processes and supporting technologies to adapt to these changes [70, 203]. An alternate view advanced by Regev and Wegmann [178] is that flexibility should be considered from the opposite perspective, i.e. in terms of what stays the same not what changes. Indeed, a process can only be considered to be flexible if it is possible to change it without needing to replace it completely [179]. Hence flexibility is effectively a balance between change and stability that ensures that the identity of the process is retained [178, 181].

A series of proposals have been made to classify process flexibility based on factors which motivate it and the ways in which it can be achieved within business processes. Snowdon et al. [203] identify three causal factors: type flexibility (arising from the diversity of information being handled), volume flexibility (arising from the amount of information types) and structural flexibility (arising from the need to operate in different ways. Soffer [204] differentiates between short-term flexibility, which involves a temporary deviation from the standard way of working, and long-term flexibility, which involves changes to the usual way of working. Kumar and Narasipuram [145] distinguish pre-designed flexibility which is anticipated by the designer and forms part of the process definition and just-in-time responsive flexibility which requires an “intelligent process manager” to deal with the variation as it arises at runtime. Carlsen et al. [54] identify a series of desirable flexibility features for workflow systems based on an examination of five workflow offerings using a quality evaluation framework.

Heinl et al. [121] propose a classification scheme with distinct approaches – flexibility
Chapter 6 Process Flexibility Patterns

by selection, where a variety of alternative execution paths are designed into a process, and flexibility by adaption, where a workflow is “adapted” (i.e. modified) to meet with the new requirements. Two distinct approaches to achieving each of these approaches are recognized: flexibility by selection can be implemented either by advanced modeling (before execution time) or late modeling (during execution time) whereas flexibility by adaption can be handled either by type adaption (where the process definition is changed but individual process instances currently running are unaffected) or instance adaption where selected (or all) process instances are changed to meet with new operational requirements. Van der Aalst and Jablonski [13] adopt a similar strategy for supporting flexibility. Moreover they propose a scheme for classifying workflow changes in detail based on six criteria: (1) reason for change, (2) effect of change, (3) perspectives affected, (4) kind of change, (5) when are changes allowed and (6) what to do with existing process instances. In [200], Schonenberg et al. presented a preliminary version of the taxonomy of process flexibility, covering four types of flexibility: design, deviation, underspecification and change. This work has been elaborated in a more detail in this chapter, and for each flexibility type a set of process flexibility patterns have been defined that allow evaluation of offerings in a more precise way.

Regev et al. [179] made an initial attempt to define a taxonomy of concepts that are relevant to business process flexibility. This taxonomy has three orthogonal dimensions: the abstraction level of the change, the subject of the change and the properties of the change. Whilst it incorporates elements of the research initiatives described above, it is not comprehensive in form and does not describes the relationships that exist between these concepts or link them to possible realization approaches.

There are a variety of approaches to incorporating flexibility within a design-time process definition. Traditional process design methods [43, 146, 184] have centered on the separation of business logic from the actual application processing and utilizing constructs such as hierarchy, conditional elements and business rules within the process definition to explicitly cater for various execution scenarios that might be encountered. Whilst effective, these strategies require that all possible situations be captured a priori at design-time, an assumption that proves to be unrealistic in practice [121]. The use of exceptions [84, 195, 207] provides one means of handling expected but infrequently occurring processing errors without requiring their explicit inclusion in the process definition. Various techniques to implementing exception handling strategies in workflow systems have been demonstrated by offerings including WAMO [83], ConTracts [186], Exotica [27], OPERA [115, 116], TREX [206] and WIDE [55].

Another approach that has been investigated for embedding flexible constructs in business processes involve the augmentation of control-flow routing constructs operators based on fuzzy logic [17]. Indeed one area that offers significant opportunity for increasing the potential flexibility of a business process is the replacement of the strict graph-based structures that are generally used to describe control-flow dependencies between the tasks in a process with other means of describing these dependencies. ConDec [14, 168, 170, 171] is a declarative language that specifies control-flow dependencies using linear temporal logic expressions. Other research initiatives in this area have investigated a variety of other means of defining control-flow including the use of process grammars to specify dependencies between tasks and documents (i.e. data elements) in a process [103], the introduction of the notion of “anticipation” [110] which allows the execution of sequential tasks to overlap at the discretion of workflow users where there are not specific data dependencies between them, the inclusion of flexible elements in process definitions that describe alternate execu-
tion options, alternate task orderings and optional tasks [141] and basing control-flow on rule-based invariants that must hold during process execution [178] or constraints based on task pre and postconditions [218] that determine when individual tasks can start and complete.

The potential for increasing process flexibility by allowing deviations from the specified process definition at runtime is supported in PROSYT [68] which allows a deviation policy to be specified for a process, identifying which forms of deviation are tolerated, together with a consistency handling policy, which ensures any allowed deviations do not impact the overall correctness of the system. In the context of the WASA system, Weske [68] nominates three user-initiated operations – SkipActivity, StopActivity and RepeatActivity – that allow for deviations from normal workflow execution.

Several approaches have been proposed that support the underspecification of processes thus allowing for greater flexibility in the actual tasks initiated at runtime. Noll [160] advocates the use of low fidelity models which specify the major tasks and main sequence in a process, but leave the actual sequence of execution at the discretion of the user. This essentially corresponds to a more general notion of the case handling paradigm [16] as it allows distinct tasks in a given process instance to be undertaken by differing users. In a similar vein, Herrmann and Loser [122] advocate the inclusion of “vagueness” in socio-technical process definitions allowing concepts such as arc conditions and task ordering to be deliberately omitted from models and also supporting the inclusion of specific modeling constructs to identify aspects of the model that are incomplete or unspecified. Van der Aalst advances the notion of generic process definitions [3, 5] which allow placeholders elements (termed generic processes) to be specified in models which correspond to fragments of the overall process whose actual composition is determined at runtime. Mangan and Sadiq [149] propose an analogous scheme where a process is partially specified as a set of fragments and the actual format of the process definition undertaken for a given instance of the process is deferred to runtime at the discretion of individual users. In a subsequent paper [196], Sadiq et al. describe a flexible workflow modeling language which incorporates “pockets of flexibility” which denote regions of the process whose actual content is determined at runtime based on workflow fragments (tasks or subprocesses) and composition rules that are associated with them. The OPENflow system [117] is an example of an actual system that supports this approach to process flexibility. The issue of managing dynamic change to executing processes has been widely researched in the fields of adaptive and evolutionary workflow [57, 61, 85, 131, 134, 202, 222]. A number of significant research prototypes have been developed in this area including ADEPTflex [182], ADOME [59], CBRFlow [221], DYNAMITE [120], WASA2 [222], Declare [200] and YAWL worklets [20].

When during process execution, the process needs to be changed on an ad-hoc basis, such that all existing process instances are migrated, various kind of errors can occur, e.g., introduction of duplicate tasks, deadlocks or livelocks. This problem is known as the ‘dynamic change bug’. One of the first steps towards resolving this problem was made by Ellis et al. [86], who introduced a mathematical formalism for modeling and analyzing dynamic changes in workflow. This work was not complete, and was addressed by a number of subsequent investigations. In [10], van der Aalst et al. propose the use of inheritance-preserving transformation rules in order to prevent the occurrence of the dynamic change bug. The authors define a set of transformation rules that can be used to restrict changes in process definitions such that new process model inherits desirable properties of the old workflow process. In [6], van der Aalst describes an approach for calculating a safe change region. If a process instance is in such a change region, the transfer of the process instance
from the old process definition to the new one is postponed. A comprehensive evaluation of various approaches (both conceptual and implementation-based) to managing dynamic changes to workflow processes is presented by Rinderle et al. in [187].

As a means of comparing various approaches to process change, Weber et al. [219, 220] have proposed a set of 18 change patterns and 7 change support features, which have been gathered by empirically analyzing a selection of process models. The authors divide the 18 change patterns into adaptation patterns and patterns for changes in predefined regions. They indicate that process flexibility can be achieved either through structural process adaptations or by allowing for loosely specified models (which may be refined during process execution). Process adaptations defined at the type and instance level correspond to two types of flexibility defined in this chapter, i.e. flexibility by permanent change and flexibility by momentary change. The other approach, identified by Weber et al. is termed ‘built-in flexibility’. This approach assumes that some part of a process model are left unspecified during design-time, however they are refined at run-time when more information becomes available. This approach corresponds to the flexibility by underspecification type identified in this chapter. Weber et al. focus on changes to the control-flow perspective, whereas in this chapter we systematically define five flexibility types, two of which (i.e. flexibility by design and flexibility by deviation) are not covered in [219]. Furthermore, there is a large distinction in the approaches used for pattern identification between the work of Weber et al. and the work presented in this chapter. The change patterns of Weber et al. were identified based on an empirical analysis of models from the healthcare and automotive domains, and are limited to flexibility by underspecification and flexibility by momentary and permanent changes types. Not only were process flexibility patterns presented in this chapter derived in a more structured way, they also cover the broader scope. In fact, the majority of the change patterns by Weber et al. can be mapped onto the process flexibility patterns presented in this chapter.

Reijers et al [183] present yet another view on flexibility. The authors categorize exceptions as expected and unexpected. Expected exceptions are addressable if they are technologically solvable. They authors describe how each type of addressable exception should be handled, i.e. either by executing the main workflow in a specific way; or by executing a separate workflow. To increase the flexibility of WFMSs the authors propose extending the functionality of WFMS with case variables and preconditions, providing direct access to the workflow execution status and jump facilities (this suggestion has been implemented by Staffware).

De Moor and Jeusfeld [72] propose the ‘legitimate user-driven approach’ which enhances the acceptability of workflow changes in the context of virtual communities. Chun and Atluri in [205] propose an approach for workflows in the context of eGovernment to adapt to run-time changes. The authors classify different types of run-time changes and propose an ontology-based framework for dynamic workflow change management system which adopts the changes based on profile change, exceptions and rule change using migration rules. Van der Aalst [6] discusses the problem of case migration caused by workflow change leading to task duplication, skipping of tasks, deadlocks, and livelocks. To address this problem, the author proposes an approach for calculating a safe change region. Several approaches for adaptive process management have been proposed in [187]. Adaptive processes enable users to evolve process definitions, so that they meet new requirements [113].

In several contemporary offerings, process flexibility has played a significant role in defining the manner in which each of these offerings were developed. Staffware Process Orchestrator [101] has been developed with an approach to assign process components
(that are not known at design-time) dynamically at run-time, when sufficient amount of information becomes available. This feature is especially useful in service-oriented environment, where the selection of a service needs to be based on an event or a response from an external system. InConcert [77] allows the task structure of a process instance to be modified dynamically by a user with appropriate authorization using operations such as adding and deleting tasks, adding and deleting dependencies, and specifying task properties such as durations and conditional execution.

6.5 Summary

In this chapter, we have focused on the issue of process flexibility and defined a taxonomy consisting of five flexibility types: flexibility by design, flexibility by deviation, flexibility by underspecification, flexibility by momentary change and flexibility by permanent change. This classification shows that there are different approaches to facilitating flexibility for a user when selecting a desired execution outcome. The need for flexibility may be recognized during design-time when creating a process definition or at run-time. However, the actual decision regarding which execution path will be taken is made at run-time execution depending on the constraints posed by the operating environment. Since it is very hard to foresee all possible behaviors in advance, contemporary PAISs must be able to adapt to continuously-changing requirements. In terms of the functionality offered by PAISs, it is important to clearly identify which type of flexibility the system supports:

- Does a system allow for enactment of incomplete process definitions?
- Is it possible to deviate from the steps prescribed by the process definition?
- Is it possible to change the behavior of a specific process instance?
- Is it possible to redesign the process definition in such a way that the changes made directly become available in current and future process instances?

Since the taxonomy of process flexibility does not give sufficient details about how each of the flexibility types can be operationalized, we have defined a set of process flexibility patterns, each addressing a specific aspect of flexibility and corresponding to one of the five flexibility types. The 34 patterns reason about flexibility in process initiation, flexibility in process termination, flexibility in execution path selection, flexibility in task reordering, flexibility in task elimination, flexibility in process extension, flexibility in task concurrency and flexibility in task repetition.

The process flexibility patterns identified have a wide range of potential application areas including:

- Facilitating a better understanding of process flexibility requirements;
- Providing a precise, systematic description of flexibility aspects;
- Benchmarking of PAISs; and
- Establishing a common process flexibility vocabulary.

Process flexibility is a topic that has been attracting a lot of attention both from research and commercial observers. A clear understanding of the requirements for process flexibility is important for process designers who need to deal with non-typical behavior whilst designing process definitions; for architects and developers of information systems to define the degree of flexibility a system needs to offer and how these needs will be realized; and for users who need to have sufficient knowledge of the system and its capabilities.
from the process flexibility perspective in order to make correct decisions when adapting a process to changes in the operating environment.

Process flexibility patterns provide a precise and systematic description of wide variety of flexibility aspects. We defined a map of process flexibility requirements, where for each of eight process flexibility aspects a set of patterns corresponding to a specific flexibility type are defined. In order to distinguish between patterns, for each of them we defined a graphical notation. The purpose of this notation is to show the effect of a pattern by visualizing the process definition on the type/instance level before and after applying the pattern. In order to avoid ambiguous interpretation, for each of the patterns identified we designed a CPN diagram illustrating the behavior of a pattern in the context of a basic process engine.

As we showed in Section 6.3, the patterns identified can be used as a means of benchmarking the degree of process flexibility support in PAISs. It is interesting to note that none of the offerings analyzed provide support for all flexibility types. Patterns can be used as a tool for selecting a system supporting desired types of flexibility.

Finally, the systematic approach used for identifying process flexibility patterns helps to structure knowledge accumulated in the domain. Concepts defined in the taxonomy of process flexibility, i.e. flexibility types, flexibility groups and pattern names can enrich the domain vocabulary, and be used for improving communication between process designers, system developers, and end users.

The scope of the topic addressed in this chapter is limited to process control-flow flexibility. In order to achieve a holistic understanding of requirement for process flexibility, other relevant perspectives (such as data and resources) have to be investigated. Nevertheless, the information presented in this chapter can be seen as a first step towards a universal flexibility model (for example, this could be done using a meta- or ontological-format). There are many other possible applications of flexibility patterns. For example, issues related to process mining of patterns applied during run-time execution can be investigated.
Chapter 7

Epilogue

The goal of this thesis was to facilitate the understanding of the requirements for PAISs by describing the control-flow, service-interaction and process flexibility perspectives in a precise and systematic manner. The workflow control-flow patterns, service-interaction patterns and process flexibility patterns address the requirements for each of these perspectives. The patterns defined provide a reference point for evaluation and improvement of contemporary offerings, bring new insights to the definition of new languages and standards in the domain, and provide a language-independent manner for sharing knowledge accumulated in the domain. In addition to contributing to the conceptual foundations of PAISs, this thesis also provides a knowledge-base of proven solutions used in CPN modeling. In this chapter we describe contributions, limitations and future work (cf. Section 7.1). Furthermore, we present generic insights and observations related to the pattern research presented in this thesis (cf. Section 7.2).

7.1 Contributions, limitations, and future work

This section summarizes the contributions. Moreover, limitations and directions for future work are given.

7.1.1 Colored Petri Nets

Contributions In Chapter 3 of this thesis, we presented 33 CPN patterns which address problems that are commonly encountered when modeling dynamic systems which require the flow of control and data to be specified explicitly. There are a number of important characteristics of this work:

- **Classification**: in order to assist users in selecting an appropriate pattern, the CPN patterns identified were divided into several clusters. By mapping a problem that needs to be solved to a suitable cluster, a desired pattern can be selected.
- **Catalog-navigation**: the relationships identified between patterns are presented in the form of a relationship diagram which can be used as a means of navigation through the pattern catalog. If a chosen pattern does not fit the problem needing to be solved, based on problem similarity or problem specialization relationships to more suitable patterns may be identified.
• **Complex problems**: given that each of the patterns addresses a specific problem and in practice such isolated problems are very rare, problems of a more complex nature are decomposed into smaller parts. In order to solve the original problem, a solution of the core problem may need to be combined with other relevant solutions. The ‘can be combined with’ relationship identified between the CPN patterns specifies patterns that can be combined in order to solve a complex problem.

• **Shared knowledge-base**: in order to facilitate sharing of accumulated knowledge, the implementations of the CPN patterns have been made publicly available at [http://is.tm.tue.nl/staff/nmulyar/Repository/index.html](http://is.tm.tue.nl/staff/nmulyar/Repository/index.html).

**Limitations**

• **Catalog-completeness**: the CPN patterns have been gathered using an empirical approach based on an analysis of literature, models and expert knowledge in the field. The limited range of the source material, and the nature of the empirical approach adopted do not guarantee the completeness of the catalog. On the contrary, this collection should be considered as starting point for a broader initiative to motivate designers to share solutions that are proven in practice. As part of such a scheme, the pattern catalog would be extended with new patterns when new solutions are encountered.

• **Language-dependence**: although CPN patterns describe problems where control-flow and data interplay and the solutions can be applied when modeling a diverse range of systems, the applicability of CPN patterns is limited to the CPN community because the implementation of the patterns is CPN-language dependent. Furthermore, for the purpose of illustrating the realization of these patterns, we selected a specific tool: CPN Tools. Although as far as possible we abstract from the functionality offered by CPN Tools, the implementation of patterns in alternative CPN tools may necessitate that some adjustments are made.

**Future directions**

• **Automatic pattern discovery**: to date the identification of the patterns used in a CPN model has been performed manually. This requires a deep understanding of the core assumptions and details of implementing the CPN patterns and is consequently a difficult and error-prone process. In order to define pattern usability in a more precise way, tool support providing automatic pattern identification is required. Although each CPN pattern is supplied with a corresponding implementation diagram created in CPN Tools, for automated pattern identification the pattern definitions have to be formally defined (for example, using an ontological approach as in [214]) both syntactically and semantically.

• **User-interface support** One of the promising applications for the CPN patterns is to extend user-interfaces of CPN modeling tools with templates allowing a desired CPN pattern to be inserted into a model automatically.

### 7.1.2 Workflow control-flow

**Contributions** In Chapter 4, we analyzed the requirements for PAISs from the control-flow perspective and identified the fundamental constructs for describing the structure of
Section 7.1 Contributions, limitations, and future work

a process model, presenting them in the form of control-flow patterns. There are a number of important characteristics of this work:

- **Pattern collection**: the fundamental requirements for the control-flow perspective in PAISs have been described by 43 control-flow patterns. The definition of the original 20 patterns has been revisited: some of the definitions has been revised, some of the patterns have been decomposed into several pattern variants, and several new patterns have been added.

- **Precise definition**: the problem of ambiguity in the pattern definitions has been resolved by revising the definitions and by specifying their formal semantics using CPNs.

- **Evaluation criteria**: in order to provide an objective means of evaluating workflow offerings, for each of the patterns a set of evaluation criteria has been defined, on the basis of which a full, partial or no support rating can be determined for a given offering.

- **Reclassification**: revision of the original 20 patterns and introduction of new patterns triggered the need to reclassify the 43 patterns. Because some new patterns did not fit very well into the previous classification, the patterns have been reclassified based on their purpose rather than the manner in which they are realized. Such a classification identifies groups of similar patterns and assists users in selecting a pattern based on their intended usage.

- **Operationalization**: the control-flow patterns operate at a conceptual level, i.e. they specify recurring generic constructs relevant to process structure and enactment in an abstract sense, consequently they do not provide much guidance in regard to their actual realization. In order to describe how the control-flow constructs work, we proposed the CPC-ML modeling language which defines the operational semantics of selected patterns in a precise way.

- **Graphical notation**: CPC-ML offers a graphical notation in order to visualize and distinguish different approaches to the operationalization of the control-flow patterns. This notation can be used not only for expressing the operational semantics of the patterns, but also for illustrating the capabilities of workflow offerings at the level of individual process constructs.

**Limitations**

- **Completeness**: the majority of the control-flow patterns have been gathered using an empirical approach. In contrast to CPC-ML, which systematically describes various process constructs and possible behavioral variants, the pattern identification was not a structured process. Although the set of patterns identified may be incomplete, by configuring attributes of CPC-ML various pattern variants can be obtained.

- **Restricted scope of CPC-ML**: due to the fact that CPC-ML concentrates only on the basic functionality that can be encountered in the majority of workflow systems, it captures only a limited set of patterns variants. In order to allow for all possible pattern variants, this language would need to be extended with elements related to external triggers, multiple instances and task compositionality.
Future directions

• **User-interface support:** in order to simplify the process of creating a process definition, the user interfaces of process modeling tools can be extended with templates corresponding to the control-flow patterns.

• **Assessment tool:** the wide range of process constructs identified by the control-flow patterns can be used as an objective means of assessing current standards and systems, leading to revision or extension of the functionality associated with the control-flow perspective in these offerings. The control-flow patterns can also be used as a reference for designing an architecture for new PAISs.

• **Meaningful combinations:** of the wide range of pattern variants identified it may be helpful to identify which pattern variants can be combined in order for problems of more complex nature to be addressed than those which a specific pattern can solve.

• **Pattern discovery:** by means of process mining [78], an event log describing the execution of a specific process can be analyzed in order to identify which control-flow patterns have actually been used during execution, and how often they have been used [35]. This information could be used in order to define whether original process definition contains all patterns identified and whether it needs to be adjusted. Furthermore, the quantitative characteristics such as frequency of use can be identified.

• **Minimal set:** one of the challenging tasks related to the control-flow patterns is to identify the minimal set of patterns that have to be supported by a tool. This issue requires a balance to be found between suitability and expressiveness of the process modeling facilities offered by a given tool.

7.1.3 Service interaction

**Contributions** In Chapter 5, we analyzed the requirements for PAISs from the service-interaction perspective and presented them in the form of a configurable framework. This included service-interaction scenarios of a bilateral and multilateral nature, involving one or more messages and characterized by short and long-running conversations. There are several aspects characterizing this work:

• **Configurable framework:** a systematic and compact form of presenting service-interaction requirements in the form of five pattern families, each describing possible pattern configurations, allows the broad range of requirements in service interaction to be distinguished. The five pattern families identified contain 1602 pattern variants.

• **Meaningful combinations:** since each pattern configuration is characterized by a set of parameters that may take different values, many pattern variants can be obtained by setting the parameters to different values. Of these combinations, only meaningful pattern variants have been identified by analyzing possible combinations of values assigned to different parameters and the meaning of such combinations.

• **Graphical notation:** for each pattern family a graphical notation has been defined that can be configured in order to depict a specific pattern variant. Considering the large number of pattern variants contained in a pattern family, this graphical notation offers an intuitive means of distinguishing different pattern variants graphically.

• **Precise definition:** in order to avoid ambiguous interpretation of pattern definitions, their semantics are defined in terms of CPN diagrams.
• **Pattern-based service interaction design method:** in order to assist users in classifying service-interaction scenarios by means of pattern families, we have defined a pattern-based service interaction design method. Depending on the goal of the classification and the degree of detail required, this method defines which pattern families have to be used for the classification, how they should be configured, and which pattern families can be combined.

**Limitations**

• **Restricted scope:** the service interaction patterns concentrate mainly on the behavioral aspects of interactions between several parties from the control-flow perspective. The data used in interaction scenarios is necessary in order to describe the routing aspects rather than to define types of messages, possible content and other message-handling issues. In order to understand requirements in service interaction in more detail, the data and resource perspectives have to be considered.

• **Limited empirical validation:** unlike the control-flow, data, and resource patterns, it is less clear what systems should be evaluated.

**Future directions**

• **Benchmarking tool:** the patterns identified can be used to identify the support for different aspects of service interaction in SOA offerings. For this, configuration parameters defined in each of the pattern families can be mapped to features offered by specific systems, and the support for realizing different values for each of the configuration parameters can be identified. The requirements for service interaction described in the configurable framework can be used to assess and improve web-based standards.

• **Configurable templates:** in order to support users in modeling interactive processes, the user interfaces of the process modeling tools could be extended with templates that can be configured to allow desired pattern variants to be included in the process definition.

• **Process change impact:** the service interaction patterns concentrate only on inter-process communication, however the need for interacting with external processes is usually triggered on the basis of an internal process definition. Changes performed in the process definition such as modification of control-flow or data parameters may have an impact on interactions with external parties. Therefore it is important to investigate relationships between these perspectives and analyze how changes in processes involved in the interaction influence the overall interaction.

### 7.1.4 Process flexibility

**Contributions** In Chapter 6, we analyzed requirements for PAISs from a process flexibility perspective and systematically described them in the form of process flexibility patterns. There are several aspects characterizing this work:

• **Taxonomy of process flexibility:** five approaches to achieving process flexibility have been identified and presented in the form of a taxonomy. Such a classification can be used as a reference for describing the support for process flexibility by PAISs at a generic level.
• **Pattern catalog**: in order to determine patterns with similar intents across five flexibility types, we identified eight flexibility aspects (such as flexible process initiation, termination, etc.). Of the 34 patterns identified, a desired pattern can be selected by identifying a desired approach to process flexibility and the behavior that needs to be achieved.

• **Graphical notation**: for each pattern identified, a graphical notation illustrating the behavioral aspect of process flexibility has been defined. This notation illustrates the type and the scope of flexibility operation facilitated by the pattern.

• **Flexible process engine**: in order to illustrate the dynamic nature of the patterns, we defined a generic process engine using the CPN formalism, and for each of the patterns we showed how the given process engine needs to be extended or modified in order to allow for the desired flexibility behavior. The combination of realizations of all flexibility patterns ultimately represents a maximally flexible process engine. The use of the CPN formalism ensures precise interpretation of each of the pattern definitions.

• **Assessment tool**: the patterns identified can be used as an assessment tool for evaluating the degree of support for process flexibility aspects, and in general for classifying approaches to process flexibility adopted by specific PAISs.

**Limitations**

• **Restricted scope**: the process flexibility patterns identified concentrate only on the control-flow perspective. In order to get a complete understanding of flexibility issues, the data and resource perspectives must also be considered.

**Future directions**

• **Tool development**: the operations allowing for flexible behavior which are described in the process flexibility patterns could be used as a reference for extending the functionality of existing tools and, development of new tools.

• **Data and resource flexibility**: as the current set of process flexibility patterns concentrates only on the control-flow perspective, flexibility in data manipulation and resource assignment also needs to be investigated.

• **Universal model of flexibility**: having investigated flexibility issues in all perspectives relevant to PAISs and clearly described them, a universal model of flexibility could be obtained. One possible means to specify such a model is by applying an ontological approach.

**7.2 Reflection**

In this thesis, we have described requirements for PAISs from the control-flow, service interaction and process flexibility perspectives in the form of patterns. The patterns identified have one thing in common: they are associated with the dynamic behavior. In case of the control-flow perspective, it is the flow of control from one task to another; in case of service-interaction, it is the flow of control between processes; and in case of process flexibility, it is the ability to react to changes in external environment by choosing an appropriate execution path. Additional research may need to be performed in order
Section 7.2 Reflection

to understand the interrelation between these perspectives and to identify how changes of process definition on the inter- and intra-process level influence other perspectives relevant to PAISs.

There are a number of generic insights and observations related to the pattern research presented in this thesis:

- **Pattern format**: the selection of a pattern format is an important step in systematization of knowledge in the field and describing it systematically in the form of patterns. In this thesis, we used different formats to document the various patterns identified. The major differences between formats used for describing the CPN patterns and patterns related to PAISs are caused by the intent of the patterns and the audience the patterns are aimed at. Unlike the patterns related to PAISs, which aim to provide a generic and language-independent manner of describing requirements for PAISs, the CPN patterns are intended to support users experiencing problems while using a specific modeling language (e.g., CPNs). In order to describe problems that are experienced by CPN modelers and how these can be solved, language-specific concepts and terms have been utilized. Moreover, the CPN patterns require a more extensive pattern format which allows various solution alternatives and implementation variants to be systematically described. The control-flow patterns and process flexibility patterns adopt almost identical pattern formats, with the only exception being that the service interaction patterns are presented in form of a configurable framework in order to describe numerous pattern variants identified in a compact manner.

- **Relation ‘context’-‘pattern’-‘pattern variant’**: since a pattern expresses a relation between a problem and a solution applied in a particular context, the context conditions play an important role in identifying a correct solution for a given problem. When comparing the series of patterns identified, one can notice that multiple CPN patterns offer several solutions corresponding to a particular problem. The selection of a solution or even the selection of an implementation alternative is determined by the context conditions that have to be fulfilled. For example, the *Capacity Bounding* pattern (cf. page 71) offers means for bounding the capacity of a particular place. When objects stored in this place need to be represented as a single collection, Solution 3 needs to be applied, whereas for objects distinguished as separate entities either Solution 1 or Solution 2 is applicable.

In Section 2.2, we mentioned that no agreement on the use of a common pattern format for describing patterns has been reached. Some practitioners represent a problem that is encountered in different contexts as being separate patterns, whereas others consider them as being variants of a single pattern. The standpoint related to the role of context conditions in a pattern influences the pattern format used for documenting patterns. Unlike the service interaction patterns, where we defined a number of attributes that can be configured in order for a pattern variant related to a particular pattern family to be obtained, the control-flow patterns separately list pattern variants (such as the *Blocking Discriminator* and *Canceling Discriminator* patterns, described on pages 137 and 138). Although the process flexibility patterns identified have been presented as separate patterns, it is also possible to consider patterns belonging to a common group (e.g., flexible repetition, flexible selection, etc.) as variants of one pattern, whose context conditions depend on the moment at which the need for flexibility has been anticipated and the manner in which needs to
be realized.

- **Degree of problem granularity**: patterns described in this thesis differ in terms of problem granularity. For example, when we consider the service interaction patterns, the scope of problems addressed by them is much broader than the scope of the problems associated with the control-flow patterns. In case of the service interaction patterns, we concentrate on interaction between several processes, whereas the control-flow patterns concentrate on internal behavior of a single process. The CPN patterns are even more specialized as they concentrate on particular aspects of CPN concepts. Differences between levels of problem granularity are directly related with differences in pattern evaluations. In particular, when analyzing the extent to which the control-flow patterns are supported, we mainly evaluated a modeling language provided by an examined offering; in case of the process flexibility patterns, we analyzed the whole system (i.e. both the process modeling languages and capabilities of enactment engines); whereas in case of the service interaction patterns, we evaluated functionality available for expressing interactions between several systems.

- **Role of classification in a pattern language**: when describing patterns, we identified corresponding relationships between patterns. Together with relationships identified, patterns form a part of a pattern language. Patterns in isolation only document recurring solutions to a particular problem, whilst a pattern language can be used in order to solve problems of a more complex nature than the ones addressed by individual patterns. Patterns classification helps in selecting solutions that may need to be combined in order for a particular problem to be solved. As patterns languages are not fixed, they can be extended with new practices, providing that corresponding relationships to other patterns are clearly identified.

- **Preciseness of pattern definitions**: in order to avoid ambiguous interpretations, the semantics of all patterns is described by means of CPNs. The CPNs formalism significantly reduces the ambiguity in pattern interpretations. In case of the control-flow patterns, service interaction patterns and process flexibility patterns, the CPN diagrams are aimed at describing the semantics of patterns without dictating the particular way (e.g., language) in which these patterns need to be realized. Furthermore, as has been mentioned in Chapter 4, the control-flow patterns describe various control-flow constructs, however they do not provide sufficient details regarding their operationalization. For this purpose, we defined a formal language CPC-ML which resolves the ambiguities related to the realization of the control-flow patterns by precisely describing their behavior in a language independent manner.

- **Subjectivity of empirical evaluations**: the process of pattern identification is at some degree subjective because there is no a uniform way to define whether a particular solution corresponds to a pattern or not. Subjective elements are present in the process of empirically evaluating the support of a pattern by a particular offering. Although for the majority of patterns a set of evaluation criteria have been identified, the use of these criteria as a reference for evaluating a particular system may still result in different interpretations. This may become problematic when the semantics of the evaluated constructs have not been completely defined and implicit assumptions have been made during the system design.

In case of CPN patterns, we performed an empirical analysis of CPN models in order to identify how often these patterns are used in practice. In order to perform such analysis, both the knowledge of patterns and context conditions associated with the model analyzed are required. The process of pattern identification can be made
more precise by performing it automatically using a pattern discovery tool. Such automatic pattern identification requires explicit differentiation between fixed and variable elements of a pattern. Every pattern has a set of fixed attributes, defining the core of a pattern and a set of attributes with variable values, which are context-dependent.

- **Technological support:** in order to increase the applicability of patterns in practice, patterns could be stored in a pattern repository. There has been an attempt to design a pattern repository by Norta et al. [161], however this repository had a very limited usability. The main limitations of this repository is inability to incorporate different pattern formats, i.e. the pattern repository should provide a means for selecting an appropriate format for describing patterns of descriptive nature, and also the possibility to describe the configurable patterns of more generative nature (such as the service interaction patterns presented in Chapter 5).

Furthermore, patterns can be supported by tools of two different types: pattern discovery tools allow a (process) model to be analyzed in order to identify what patterns it comprises, and process modeling tools can include solutions or their implementations in form of templates directly accessible via a user interface for assisting users in modeling. Note that a selection of the most frequently occurring control-flow patterns has been used to extend the commercial modeling tool IBM WebSphere Business Modeler [111]. In this tool, in addition to the ability to drag-and-drop a pattern template from the editor panel, the soundness of a model is checked and tool feedback is provided to prevent users from modeling errors.

- **Approach for pattern generation:** the pattern-based approach can be successfully used to describe requirements in a particular domain. In this thesis, we used two approaches for pattern identification. The ‘bottom-up’ approach is based on empirical analysis of various sources of information. It results in identification of recurring solutions, however it does not guarantee that all possible patterns within a particular scope have been discovered. We applied this approach for identification of the CPN patterns and for revision of the control-flow patterns that have previously been gathered using the empirical approach.

In order to describe requirements for PAISs in a more systematic manner, we adopted another ‘top-down’ approach that allows various pattern variants to be derived from a generic core pattern. In particular, we analyzed the problem domain and identified a set of important (orthogonal) dimensions. For each of the dimensions identified, we determined a set of fixed and variable parameters, each associated with a set of possible values. By assigning each of the parameters to a particular value a large number of combinations can be generated. We applied this approach for specifying various options to realization of the control-flow patterns by means of CPC-ML. Furthermore, this approach was used in the configurable framework for service interaction patterns and during identification of the process flexibility patterns. Because not all pattern variants generated using this approach may appear to be meaningful, it is important to identify only meaningful combinations. Where it not possible to analytically define which of combinations are meaningful, one has to test the occurrence of each combination in practice in order to claim that they are applicable.

- **Strengths and weaknesses of the pattern-based approach:** overall, the pattern-based approach proved to be a good design choice as it allows the requirements for PAISs to be described in a language-independent way, thus making patterns accessible to the large audience without requiring any domain-specific knowledge. The problem-
solution-context structure works well for positioning a specific problem that can be encountered in a certain context and a corresponding solution. The possibility to adjust a pattern format helps in including (domain or purpose-specific) information, whereas maintaining a uniform structure throughout a pattern language. While we formalized the behavior associated with a pattern using CPNs, the pattern descriptions appear to be not precise enough when it comes to translating them directly in a language supported by a tool offering the functionality for discovering patterns in existing models. This deficiency could be solved by describing the pattern definitions using a specific language, i.e. applying an ontological approach [214]. The static and dynamic aspects of behavior associated with each of the concepts identified in a pattern language is unambiguously described using the syntax of a chosen language [71, 108]. The language-dependent description can be used directly in tools supporting this language, whereas a big drawback of this approach is that it narrows the domain of users to whom these descriptions are easy understandable.

7.3 Summary

In this thesis, we presented requirements for PAISs from the control-flow, service interaction and process flexibility perspectives by means of patterns. As mentioned earlier, this research complements the set of patterns identified by N. Russell [194] in order to get a generic understanding of requirements for PAISs, however it leaves room for future research. In particular, process flexibility needs to be further investigated in relation to data and resource perspectives. Furthermore, adaptability of processes is closely related to the concept of configurability of process models. On a more generic level, one could incorporate details of all perspectives addressed so far and define workflow architecture patterns, incorporating both service interaction and process flexibility dimensions. We hope that this thesis will facilitate the understanding of various perspectives of PAISs as well as bring new insights in the development of new systems and standards in the domain.
Appendix A

Workflow Reference Model

In this appendix, we describe the *Workflow Reference Model* defined by the WfMC. This model identifies five categories of interoperability and communication standards allowing workflow products to coexist and interoperate within a user’s environment [62], as illustrated in Figure 340. The model consists of *workflow enactment service* and five types of interfaces with external tools and applications. The workflow enactment service represents the core of the workflow management system and may consist of one or more *workflow engines*. It communicates with external tools and applications through various workflow application programming interfaces (APIs). In this model, each interface is described in terms of an abstract specification. The functionality provided by each of interfaces is as follows:

- **Interface 1** (Process definition tools) provides a generic process definition format which describes common process elements and the relationships between them. Var-

![Figure 340: WfMC Workflow Reference Model (from [62])](image-url)
ious process definition tools can be used in order to create a process definition which can be interpreted at runtime by the workflow engine(s) within the workflow enactment service.

- **Interface 2** (Workflow client applications) describes communication between a workflow engine and client applications (e.g., a worklist handler through which users interact with workflow management system).

- **Interface 3** (Invoked applications) describes an interface for invoking external applications. These could be standardized or special-purpose software tools (e.g., editors, spreadsheet programs, etc.).

- **Interface 4** (Other workflow enactment services) provides an interface for specifying process-to-process interaction with another workflow engine. This enables both environments to share a single view of the process definition objects and their attributes [62]. However, there may be compatibility issues as a consequence of differences in encoding and data formats utilized by individual applications, which must be resolved outside of the facilities provided by the interface (for instance, by exporting data from one format to another or by using a suitable mapping between process definition formats).

- **Interface 5** (Administration and monitoring tools) specifies an interface allowing administration and monitoring tools utilized by different applications to share functions for analyzing certain process parameters characterizing the run-time execution. Audit trails recording the process-relevant information can be used as performance indicators for workflow executions.

Note that not all of the interfaces specified have been utilized in practice. Although interface 1 provides a good overview of process elements common in workflow management systems, it is perceived to be without practical relevance [239]. XPDL (XML-based Process Definition Language) is a language proposed by the WfMC for exchanging process definitions, is used by over 80 products nowadays, however the practical demand for reusing workflow models is rather small [239] and different vendors tend to interpret XPDL in different ways. Interfaces 2 and 3 have been implemented by a number of vendors [239] for interacting with client and invoked applications. Interface 4, offering a description of interoperability functions, has been implemented by a number of workflow vendors. However, the majority of aspects introduced by the WfMC to support this interface are also covered by competing standards such as BPML [47], WS-BPEL [164] and WS-CDL [217].
Appendix B

Web-services stack

In this appendix, we illustrate a Web-services stack which specifies standards for establishing, managing and processing messages exchanged between services. The Web-services stack, presented in Figure 341, consists of three layers: Messaging, Service, and SOA [150]. The Messaging layer serves as a backbone on which the other two layers reside. It specifies how messages are being defined and transported. The Service layer defines the core of a service and how it can be accessed. Finally, the SOA layer addresses both service orchestration and choreography. Many important issues such as reliability, security and quality of service are also addressed in this Web-services stack, however we focus only on the key Web services standards (e.g., XML, WSDL, and SOAP) that are necessary for understanding the research presented in this thesis.

Figure 341: Web-services stack (from [82])
XML (eXtensible Markup Language) is a standard introduced by the World Wide Web Consortium (W3C) for exchanging structured documents and data [50]. By means of this standard the content of messages to be exchanged can be easily defined. WSDL (Web-Services Description Language) is a standard based on XML used to create an XML document which describes a Web service and how to access it [60]. WSDL has been designed in order to separate the abstract definition of an interface from the invocation details. It mainly describes operations which need to be called in order for a message sent to be processed, defines types of messages that can be sent and received, and if necessary can be used to describe the message body using the XML Schema specification. SOAP (Simple Object Access Protocol) is a standard used to enable the exchange of XML information between interacting parties [45]. SOAP is a transport-independent protocol which can be bound to an existing transport (for instance, HTTP (HyperText Transfer Protocol used to transfer web pages on the World Wide Web)).

The discovery layer residing between the Service and the SOA layers focuses on the definition of a service directory which can be used as a search engine in order to find a suitable service amongst those that have already been published. Universal Data Description Interface (UDDI) used for this purpose provides all necessary information for accessing a desired service. Such registries can be defined for private or public use.

Two more layers that merit further discussion are Orchestration& Composition layer related to application of services in the SOA, and Choreography& Coordination layer related to business integration. The main difference between these layers is in their view on services in an SOA: Business Process Execution Language for Web Services (WS-BPEL) [164] describes an interaction from the perspective of a single participant, whilst Web-Services Choreography Definition Language (WS-CDL) [217] defines information formats being exchanged by all participants in an interaction.

In the service-oriented domain, it has been recognized that in order to help organizations function effectively their processes may need to be reused in the form of business services which can be freely acquired and used by other business partners. In order to enable interaction between services, their interfaces should be defined according to a commonly recognized standard. In the last decade, multiple standards proposals have been proposed, some of which have been discontinued ((XLANG, BPML and WSCI) [33, 47, 167, 209]) and some of which have gained broader acceptance (BPEL4WS, WS-BPEL, BPMN) and currently serve as the basis for ongoing standardization initiatives.
Appendix C

Glossary

Active task A task whose execution changes the state of the modeled system.

Asynchronous interaction An interaction between two parties, where one party sends a request message to another party, and expects a reply message to be received back in response to the first message (note that the requestor party is not blocked and can continue processing).

Atomic task A task that cannot de decomposed into smaller parts and described in terms of other tasks.

Blocking mode A parameter associated with output ports of a task, specifying whether output ports are allowed to send messages to the corresponding output channels when their maximal capacity has been reached.

Bounded channel A channel with limited maximal capacity.

Business process A special type of process that can be defined as a set of tasks that need to be executed in a specific order by dedicated employees or other kinds of resources, processing supplied input data and producing output data, with the aim of realizing one or more business goals.

Business Process Management (BPM) A methodology that aims at continuous improvement of processes by defining, measuring and improving various process performance indicators.

Cancellation set A set of locations from which all messages are to be removed upon the task termination.

Channel A path connecting two or more tasks, used to convey messages.

Choreography A sequence of dependencies between interactions of multiple parties defined in order to implement a business process comprising multiple services.

Composite task A task execution of which results in an initiation of a sub-process.

Conversation Communication of a set of contextually related messages between two or more parties.

Configuration parameters Parameters that have to be set to a specific value from the defined range in order to for a specific pattern variant to be configured.

Control-flow pattern A three-part rule expressing a relation between a certain context (the lifecycle of a single process instance), a problem (addressing the behavioral aspects of task routing), and a solution (expressed in terms of structural entities).

Core Process Constructs Modeling Language (CPC-ML) A formally defined language which offers a graphical notation to depict individual variants of process constructs encountered in every PAIS.
Correlation A mechanism used by a party to identify a process instance for processing of a received message in the context of a particular conversation.

Customer A party, the recipient of a service or a product, which may be represented by various people and processes in different roles (e.g., user, buyer, payer, etc.).

Discrete system A system that is characterized by a certain state at each moment in time.

Dynamic attribute A pattern attribute, whose value is derived from other pattern attributes.

External channel A channel providing messages for shared access between several tasks.

External input port An input port mapped to an external channel.

Information System A particular type of a work system that processes information by performing various combinations of six types of operations: capturing, transmitting, storing, retrieving, manipulating, and displaying information.

Input port A port consuming messages from a corresponding channel.

Input selection mode A parameter defining which set of enabled input ports associated with a given task is to be selected from the input sets for message consumption.

Input sets All possible sets of input ports associated with a given task, which must be enabled for task commencement.

Interaction An act of interacting between two or more parties by means of sending/receiving of messages.

Flexibility The ability of a process to deal changes in operating environment, by varying or adapting those parts of the business process that are affected by them, whilst retaining the essential format of those parts that are not impacted by the variations.

Flexibility by design The ability to incorporate alternative execution paths within a process definition at design time allowing for selection of the most appropriate execution path to be made at runtime for each process instance depending on the circumstances encountered.

Flexibility by deviation The ability for a process instance to deviate at runtime from the execution path prescribed by the original process without altering its associated process definition. The deviation can only encompass changes to the execution sequence of tasks in the process for a specific process instance, it does not allow for changes in the process definition or the tasks that it comprises.

Flexibility by underspecification The ability to execute an incomplete process definition at run-time, i.e. one which does not contain sufficient information to allow it to be executed to completion. Note that this type of flexibility does not require the definition to be changed at run-time, instead the definition is completed by providing a concrete realization for the undefined parts as they are encountered at run-time.

Flexibility by momentary change The ability to modify a process definition at run-time such that the process definition associated with a given process instance is amended in order to realize previously not foreseen behavior.

Flexibility by permanent change The ability to modify a process definition at run-time such that all currently executing process instances are migrated to a new process definition and all new process instances utilize the new process definition.

Generic Workflow Net (GWF-net) A language-independent representation of a workflow model, which can be created using process entities encountered in any PAIS, expressed in terms of CPC-ML constructs.

Local channel A channel conveying messages between two specific tasks.

Local input port An input port mapped on a local channel associated with a given task.
Mandatory input port A port which must be enabled before the task associated with it may commence.

Mandatory output port A port which always produces a message to an output port once the corresponding task has terminated.

Mediator A party who acts as a link between two parties involved in a conversation.

Maximal capacity of a channel A parameter that defines how many messages the channel may hold at once.

Message A unit of information associated with the task input/output, which is expressed in terms of basic or complex data structure.

Message consumption mode A parameter that defines how many messages are to be consumed from the channels attached to the input ports selected for message consumption.

Migration The systematic movement of process instances from to the old process definition to the new process definition.

Minimal capacity of a channel A parameter that defines at least how many messages a channel must contain in order to make a port, consuming messages from this channel, enabled.

Momentary change An action of modifying a process definition associated with a particular process instance.

Multiple-instance task A task, execution of which results in initiation of one or more distinct task instances that run concurrently and independently of each other within the same process instance.

Optional output port A port that produces a message to an output channel if and only if a data-based condition associated with this port has been satisfied.

Orchestration The control and data flow between business services that are necessary to achieve a business process, from the viewpoint of one participant.

Output port A port producing messages to the corresponding output channel.

Output sets A set of output ports associated with a given task, each of which will produce one message at the moment of task termination.

Party An entity involved in communication with other entities by means of sending/receiving messages (e.g., a process, a service, a business unit, etc.).

 Permanent change An action of modifying a process definition at a type level, potentially impacting all existing and future process instances.

Pattern A solution for a recurring problem encountered in a certain context. Conceptually similar patterns that share a set of common pattern attributes and differ only by values these pattern attributes take are also termed as pattern variants.

Pattern catalog A set of patterns addressing problems in a particular domain, listed in the form of a catalog.

Pattern configuration A set of pattern attributes defined for a given pattern family to differentiate between different pattern variants. By setting pattern attributes to different values from the defined range various different pattern variants can be generated.

Pattern variant An instance of a pattern configuration for a particular pattern family whose configuration parameters are set to a specific value from the defined range.

Pattern family A set of pattern variants, sharing the same concepts, grouped together.

Pattern format A format selected for documenting of patterns, typically including the pattern name, problem description, solution, and consequences sections.

Port A gate through which messages are sent and received by a task.
Appendix C Glossary


Process-Aware Information System (PAIS)  A software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models.

Process definition  The representation of a business process in a form which supports modeling and/or enactment by a process execution engine.

Process instance  An executing instance of the process definition.

Process fragment  A set of one or more tasks contained in a process definition.

Process mining  A technique that allows to mine a process model representing the actual order of events on the basis of a data log.

Process model  A representation of a business process (i.e. process entities and relationships between them) using some kind of (graphical) notation.

Provider  A party that provides services/products to a customer.

Product  A good or a service that is produced.

Reply  A reply, answer, or additional message that is returned to a requestor.

Renew policy  A policy specifying who from the involved two parties in a subscription is responsible for renewing the subscription.

Request  A message sent by a requestor party to another party.

Requestor  A party issuing request messages to a (set of) other parties involved in the conversation.

Responder  A party producing a reply message to respond on the request message received.

Response period  A period of time within which a reply on the request sent is expected.

Safe channel  A channel that is able to hold at most one message at a time.

Static attribute  A pattern attribute whose value is fixed for all pattern variants derived from the pattern configuration.

Subscription  A special kind of a conversation between two parties, a provider and a customer, who are both capable of initiating and renewing the process of subscription aiming at the delivery of a certain product under accepted subscription terms.

Subprocess  A process within another process that may be executed on its own right.

Subscription initiation  A conversation held for the purpose of establishing a subscription.

Subscription renewal  A conversation held for the purpose of renewing of the established subscription.

Subscription terms  A set of rules or constraints characterizing a given subscription, which includes a subscription period, a renew policy and a product of subscription.

Synchronous interaction  An interaction between two parties, where one party sends a request message to another party, and expects a reply message to be received back in response to the first message, providing that the requestor party is blocked until the response message is received.

Task  An abstraction of an activity, characterized by a set of inputs and outputs, assigned to a certain resource.

Unbounded channel  A channel with unlimited maximal capacity.

Web-service  Any business application that has been defined using standard Web interfaces and deployed in order to communicate with other applications over a network.

Workflow  A set of interdependent tasks that occur in a specific sequence.

Work System  A system in which human participants and/or machines perform a business process using information, technology, and other resources to produce products (and/or services) for internal or external customers.
Bibliography


[99] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.


Patterns for Process-Aware Information Systems: 
An Approach Based on Colored Petri Nets

Organizations are continuously seeking ways to improve the efficiency and effectiveness of their operations. To assist in meeting these objectives, it is increasingly recognized that they require a range of techniques and technologies for managing their organizational business processes. With this need in mind, the Business Process Management (BPM) discipline has been established with the aim of developing approaches to the operationalization of business processes based on software technology. Systems which manage business processes in conjunction with a process model (either explicitly or implicitly) are typically termed Process-Aware Information Systems (or PAISs). The increasing demands of the modern business environment mean that PAISs need to be capable of supporting dynamic organizations in deploying flexible business processes that are subject to ongoing change and evolution and involve the integration of external parties, organizations and software applications.

Numerous PAIS offerings have been developed over the past decade resulting in an increasingly diverse range of approaches to modeling and enacting business process concepts. This diversity of techniques has triggered a number of initiatives aimed at establishing common standards in the BPM field. However none of the resultant standards proposals has met with widespread adoption. In an effort to develop a rigorous conceptual foundation for the domain, the Workflow Patterns Initiative adopted a pattern-based approach to identifying and describing the fundamental requirements for PAISs. The work presented in this thesis contributes to this initiative by refining the conceptual foundation for PAIS, specifically concentrating on the control-flow, service-interaction, and process flexibility perspectives. This thesis addresses these perspectives as follows.

The requirements for PAISs from the control-flow perspective are described by (1) a comprehensive set of 43 workflow control-flow patterns, which identify recurring generic constructs relevant to process structure and enactment, and (2) the Core Process Constructs Specification Language that allows different approaches to the operationalization of process constructs to be explicitly described in a language-independent way.

The requirements in service interaction are described in the form of a configurable framework, consisting of five pattern families, in total combining 1602 Service Interaction pattern variants. A graphical notation has been developed that encompasses each of the pattern families. It visualizes configuration parameters and their settings, thus providing a means to illustrate and distinguish distinct pattern variants.

The requirements for process flexibility are described by means of 34 process flexibility patterns based on five distinct flexibility types. These flexibility types distinguish the moment and the manner in which both foreseen and unforeseen behavior can be introduced into a process.

In order to avoid potential ambiguities in regard to pattern interpretation, the semantics of all patterns are formally described in the terms of Colored Petri Nets (CPNs). This modeling
technique is widely used throughout the thesis. In doing so, a set of commonly-used and recurrent constructs have been identified during the modeling of CPN diagrams. These constructs form the basis for a comprehensive CPN pattern language.
Samenvatting

Patterns for Process-Aware Information Systems: An Approach Based on Colored Petri Nets

Bedrijven zijn continue op zoek naar manieren om de efficiëntie en de effectiviteit van hun organisatiestructuur te verbeteren. Om deze doelen te halen, is het gebruikelijk om verschillende technieken en technologieën voor beheersing van bedrijfsprocessen toe te passen. Business Process Management (BPM) is een discipline die verschillende aanpakken voor ondersteuning van bedrijfsprocessen met behulp van informatiesystemen biedt. De systemen die de bedrijfsprocessen met behulp van een procesmodel ondersteunen worden ook wel Procesgerichte Informatiesystemen (of PAISs) genoemd. Door de toenemende vraag naar de beheersing van bedrijfsprocessen met behulp van IT worden in de moderne zakelijke omgeving veel eisen gesteld aan de PAISs. Omdat de bedrijfsprocessen vaak door de veranderingen in een operationele omgeving worden beïnvloed, wordt er van PAISs verwacht dat ze in staat zijn om dynamische organisaties te ondersteunen met de middelen die het mogelijk maken om de bedrijfsprocessen te wijzigen en verbeteren.

In het afgelopen decennium zijn er veel informatiesystemen beschikbaar gekomen die bedrijfsprocessen ondersteunen. Deze systemen gebruiken verschillende technieken voor de specificatie en de uitvoering van bedrijfsprocessen. Daarnaast interpreteren ze de basisconcepten op verschillende manieren, waardoor er vaak misverstanden ontstaan. Dit brede scala van technieken bracht verschillende initiatieven voor de standaardisatie van het BPM gebied teweeg. Helaas is er tot heden geen enkel standaardvoorstel gedaan dat wereldwijd geaccepteerd is. Om een sterke conceptuele grondslag in dit gebied te bouwen, heeft het Workflow Patronen Initiatief een patroongerichte aanpak geadopteerd als een manier om de fundamentele eisen voor PAISs te identificeren en te beschrijven. Het werk dat in dit proefschrift gepresenteerd wordt, voegt aan dit initiatief een aantal aspecten toe. In dit proefschrift zijn de volgende belangrijke perspectieven van PAISs verfijnd: besturing van de werkstroom, interactie tussen services, en procesflexibiliteit.

De eisen voor PAISs vanuit het besturingsperspectief worden beschreven door: (1) een uitgebreide set van 43 werkstroompatronen. Deze patronen identificeren generieke vaak terugkerende constructies die gerelateerd zijn aan de processtructuur; (2) de Core Process Constructs Specification Language die in staat is om de verschillende aanpakken voor uitvoering van procesconstructies op een taalonafhankelijke manier te beschrijven.

De eisen gerelateerd aan de service interactie zijn in de vorm van een raamwerk beschreven. Het raamwerk bestaat uit vijf patroonfamilies. Deze patroonfamilies kunnen vervolgens worden geconfigureerd om 1602 verschillende patroonvarianten te verkrijgen. Om de patronenvarianten te onderscheiden wordt er een grafische notatie toegepast. Met behulp van deze notatie kunnen de verschillende configuratieparameters duidelijk gevisualiseerd worden.

De eisen voor PAISs die aan procesflexibiliteit opgelegd zijn, worden met behulp van proces flexibiliteit patronen beschreven. Elk van deze patronen behoort tot een van de vijf groepen, die door het moment en de wijze van voorspeld gedrag onderscheiden kunnen worden.
Om de ambigüiteit die tijdens de interpretatie van patronen kan ontstaan te vermijden, wordt de semantiek van alle patronen op een formele manier beschreven met behulp van Gekleurde Petri-netten. Deze modelleringstechniek wordt regelmatig in dit proefschrift toegepast en vormt een grondslag voor de beschrijving van patronen. Bovendien hebben we patronen in Gekleurde Petri-netten zelf gedefineerd. Deze patronen beschrijven vaak terugkomende constructies die door ontwerpers tijdens het modelleren gebruikt kunnen worden.
Acknowledgements

The work presented in this thesis has been influenced by many people. First of all, I would like to thank prof.dr.ir. Wil van der Aalst, my first supervisor, for his outstanding guidance, expertise and knowledge he has kindly been sharing with me during the whole project. Secondly, I would like to express my gratitude to dr. Nick Russell, who has become my daily supervisor during the last year of the research project. Furthermore, I would like to thank each member of the Information Systems capacity group as they made it a very pleasant environment to work. Kind memories go back to the colleagues of the Queensland University of Technology, Australia, who welcomed me during the one-month stay at their group. After Wil van der Aalst has moved to the Computer Science department, his new group became almost integrated with the group of the Technology Management department. I would like to thank the colleagues of this group for their contribution in joint projects and cheerful attitude during social events.

My special thanks go to Harold Weffers, the project-manager of the post-master programm OOTI, who helped me to estimate the job market needs, resulted in the correct career choice. Business Process Management has become one of the subjects I am passionate about.

In addition, I would like to thank all members of the defense committee, i.e. prof.dr. Arthur ter Hofstede, prof.dr. Kees van Hee, prof.dr. Marlon Dumas, prof.dr. Jan Dietz and dr. Søren Christensen, who have been kind to accept the invitation to read and assess my doctoral thesis and to participate in the defence ceremony.

During my research, I have closely cooperated with dr. Mor Peleg and prof.dr. John Fox on a subject of computer-interpretable guidelines, prof.dr. Wolfgang Reisig on a subject of distributed algorithms, and prof.dr. Kurt Jensen on a subject of Colored Petri Nets, to whom I grateful. The work on service interaction patterns has been done with involvement of Lachlan Aldred, Marlon Dumas, Eric Verbeek, David Shaffer, Christian Stahl and Jan Martijn van der Werf. I thank them all for fruitful discussions. In the research on process flexibility, Ronny Mans, Helen Schonenberg, Maja Pesic and Michael Adams were involved. This cooperation has shown that team efforts can result in novel and promising ideas. Furthermore, I would like to thank Florian Gottschalk, Anke Hutzschrenreuter, Minseok Song and Boudewijn van Dongen for helping me out with Latex-hints, and Olivia Oanea for providing a template for the cover design. Next, I would like to thank Maja Pesic, Ana Karla Alves de Medeiros, Mariska Netjes, Christian Günter, Irene Vanderfeesten, and Monique Jansen-Vullers for providing their CPN models for evaluation. Additionally, I would like to thank Dutch-speaking friends and colleagues who helped me to check the translation of the thesis summary.

Last but not least, I thank to my current and former friends and family members spread all over the world (Ukraine, Russia, Netherlands and USA) for their continuous support, understanding and believe in me. This book is the reward for your love and care.

Nataliya Mulyar
Eindhoven, March 2009
Curriculum Vitae

Nataliya Alexandrovna Mulyar was born on November 11, 1978 in Bratsk, Russian Federation. After moving to Ukraine, she studied at Vinnitsa secondary school N33. Having graduated from the high-school with honors, Nataliya chose Vinnitsa State Technical University as the next step in her education path. In 1995, Nataliya enrolled in the education at the Automatics and Computer Control Systems faculty. There she obtained the Bachelor’s degree (1999), soon followed by the Master's degree (2000). Her thesis was dedicated to the research and development of means for information protection. This project Nataliya carried out under the supervision of prof. I.Ya. Haimzon. Nataliya continued the post-graduate study in this direction under the supervision of prof. A.S. Vasyura, however it had to be discontinued due to moving to the Netherlands in July 2001.

Soon after arriving to the Netherlands, Nataliya decided to deepen the knowledge of software technology, and therefore she enrolled in the 2-year postmaster program Ontwerpers Opleiding Technische Informatica (OOTI) at Stan Ackermans Institute (Technical University of Eindhoven). There she has gained practical experience in design, development, and testing of software in various software development projects. After following a set of courses and practical assignments, Nataliya did a traineeship at the Embedded Systems Institute. During her final project, she investigated whether systems used by Oce for modeling of software and hardware components can be linked together in order to obtain a single simulation environment.

In September 2004, Nataliya switched her focus from embedded systems to Business Process Management. She became a PhD candidate in the department of Technology Management of Technical University of Eindhoven, under the supervision of prof.dr.ir. Wil van der Aalst. The focus of her doctoral thesis was on improving standards in the BPM domain. In particular, she analyzed requirements for Process-Aware Information Systems from the control-flow, service interaction and process flexibility perspectives, and described them in the form of patterns.

After finishing her research at Eindhoven University of Technology, Nataliya got a position of Integral Process Advisor at Rijkswaterstaat, the implementation organization of the Ministry of Transport, Public Works and Water Management.