This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs (Project Number: 12157)

Typeset by the author with \LaTeX.
Cover Design: Paul Verspaget and Carin Bruinink, Nuenen, The Netherlands
Printed by University Press Facilities, Eindhoven

Copyright © 2014 by Debjyoti Bera

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

Author email: d.bera@tue.nl
Petri nets for Modeling Robots

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 woensdag 22 oktober 2014 om 16.00 uur

door

Debjyoti Bera

geboren te Sarenga, India
Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter: prof.dr. L.P.H. de Goey
1e promoter: prof.dr. H. Nijmeijer
2e promoter: prof.dr. K.M. van Hee
leden: prof.dr. J. Desel (FernUniversität in Hagen)
dr. P.U. Lima (Instituto Superior Técnico, U. Lisboa)
prof.dr. J.J. Lukkien
prof.dr. M.G.J. van den Brand
prof.dr.ir. P.P. Jonker (TU Delft)
Over the last few decades, robots have been successfully used in controlled environments of industrial settings as pre-programmed devices capable of carrying out boring and repetitive tasks at blazing speeds and with high precision. They have also been extensively used in hazardous environments, for instance at nuclear facilities and in military applications, as extensions of a human operator at a distance. More recently, changing demographics resulting from rapidly aging populations, particularly across Europe and Japan, have led to a tremendous rise in interest towards exploring possibilities for robots to assist people in their daily lives as means to counter the negative aspects of a declining workforce. These robots are called service robots.

There are currently many commercially available service robots such as lawn mower robots, vacuum cleaner robots, drones, milking robots, surveillance robots, surgical robots etc. Unfortunately, the breakthroughs in each of these areas have not yet lead to successful service robots capable of carrying out normal human tasks like picking and placing objects, opening doors, etc. In order to have a breakthrough, such robots must become affordable and guarantee safe and reliable operations. Although affordability can be tackled if the production is scaled up, guaranteeing safe and reliable operations requires good engineering methods.

Robotic systems consist of both hardware and software. The software part of service robots is much more larger than industry robots. So software becomes a crucial element in such systems and therefore it is important to develop good software engineering methods. One of the most promising approaches is Model Driven Development in which an abstract model of the software is modeled and analyzed before developing it. We focus on the model driven approach and use the Petri net formalism for modeling software and focus on two important properties: one the system must never get stuck and the other that it should perform its task in a timely manner.

The research conducted in this thesis contributes towards the development of safe and reliable service robots. In the future, this will enable the mass production of advanced service robots that will improve the quality of life in our society.
SUMMARY

Petri nets for Modeling Robots

The turn of the last century, has seen a rapid expansion in the scope of commercial robots, from controlled environments of an industrial setting (industry robots) to uncontrolled human environments that are dynamic and unpredictable (service robots). The rapid expansion has been primarily fueled by social demographic needs and technological breakthroughs in cheap and reliable hardware and the development of robust algorithms for sensing, interpretation, locomotion and manipulation capabilities. The rapid developments in individual capabilities have led to many commercially successful mono-functional service robots but have not been matched by a corresponding success in multi-functional service robots that operate in uncontrolled real world environments. One of the primary cause for this mismatch stems from the complexity of the underlying software subsystem and the need to provide guarantees on its behavior. In particular, we focus on behavioral properties such as safety, effectiveness, performance and reliability.

The separation of concerns is a widely adopted principle in software engineering to master design complexity and enhance reusability by decomposing a system into loosely coupled components, each realizing a coherent set of functionality. The components cooperate with each other asynchronously to achieve a goal of the system. Over the past decade, many component-based frameworks (middleware) for robots, supporting this philosophy, have been proposed. Here the emphasis is on designing components in isolation and/or reusing existing off-the-shelf components, and finally composing them over syntactically compatible interfaces. Although such approaches aid the rapid assembly of a complex system, syntactic compatibility of interfaces alone is not sufficient for guaranteeing behavioral properties of the final system. Furthermore, as concurrency is an inherent feature, their compositions become hard to analyze. Therefore, a purely component-based approach must be complemented by a formal model based approach. The latter relies on the use of models with precisely defined syntax and semantics to specify, understand and predict the behavior of a system either by model checking or by correct by construction design methods. Most existing model based approaches for software systems of robots either rely on
model checking techniques (suffers from state space explosion) and/or focus on a particular software layer and/or support only simple interaction patterns like publish-subscribe and request-reply.

In this thesis, we present a component-based architectural framework for compositional design and verification of software systems of robots. The framework focusses on control flow and is based on the mathematical formalism of Petri nets. The components of the framework may provide a service and may in turn consume services of other components. The communication between a pair of components is asynchronous and is defined by interaction patterns. An interaction pattern is a component extended with a set of clients, modeling procedures that invoke the component. The framework provides four recurring interaction patterns and prescribes a design method to derive arbitrary interaction patterns that are correct by construction. By correct we mean freedom of deadlocks and livelocks which are essential for safety and effectiveness. The framework also prescribes an incremental bottom-up composition and top-down elaboration based design method to derive the architecture of the system starting from a set of components, interaction patterns and the relationships between them. The design method guarantees correctness of the resulting system. For performance analysis by model checking, we propose a unifying and intuitive model of time in Petri nets and call it Discrete Timed Petri nets. For this model, we prescribe a finite reduction method for its infinite state space and compare its expressive power against other models of time in literature. We also give semantics to Simulink models using this model and show how they can be expressed in our architecture. For reliability analysis, we extend the model with probabilities for non-deterministic choices and call it Discrete Stochastic Petri nets. We combine all these techniques into a model driven design method and demonstrate its applicability on two real life applications robot ROSE and the ITER Remote Handling. The former is a service robot for home care of elderly people, while the latter concerns the remote maintenance of equipments in a nuclear fusion reactor.
# CONTENTS

## 1 Introduction

1.1 Brief History of Robotics ........................................... 3  
1.2 The General Challenges ........................................... 5  
1.3 The Component Based Approach ................................... 7  
1.4 Model Driven Approach ............................................. 11  
  1.4.1 Informal Approaches ......................................... 11  
  1.4.2 Formal Approaches .......................................... 12  
  1.4.3 Gaps and Challenges ........................................ 15  
1.5 Research Questions ................................................ 17  
1.6 Contributions of this Thesis ...................................... 18  
  1.6.1 Component-Based Architectural Framework ................... 18  
  1.6.2 Analysis of Timed Properties ............................... 19  
  1.6.3 Applications ................................................ 21  

## 2 Preliminaries

2.1 Basic Mathematical Notations ..................................... 23  
2.2 Graphs and Labeled Transition System ........................... 25  
2.3 Petri nets .................................................................... 26  
  2.3.1 Refinement Techniques of Petri nets ....................... 29  
  2.3.2 Colored Petri nets ............................................. 30  

## 3 Architectural Framework

3.1 Introduction .......................................................... 34  
3.2 Overview of Architectural Framework ............................ 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.3 Execution of Simulink Models</td>
<td>137</td>
</tr>
<tr>
<td>6.3 Expressing Petri nets using Simulink</td>
<td>142</td>
</tr>
<tr>
<td>6.4 Analysis</td>
<td>146</td>
</tr>
<tr>
<td>6.5 Relationship to the Architectural Framework</td>
<td>153</td>
</tr>
<tr>
<td>6.6 Conclusions</td>
<td>154</td>
</tr>
<tr>
<td>7 Design and Analysis of Robots</td>
<td>157</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>158</td>
</tr>
<tr>
<td>7.2 Modeling the Software Architecture of the ITER RH Plant</td>
<td>162</td>
</tr>
<tr>
<td>7.3 Modeling the Software Architecture of Robot ROSE</td>
<td>169</td>
</tr>
<tr>
<td>7.3.1 Robot ROSE</td>
<td>169</td>
</tr>
<tr>
<td>7.3.2 Use cases of Robot ROSE</td>
<td>171</td>
</tr>
<tr>
<td>7.3.3 Identifying Components and Interaction Patterns</td>
<td>172</td>
</tr>
<tr>
<td>7.3.4 Architectural Model</td>
<td>181</td>
</tr>
<tr>
<td>7.4 Analysis of Architectural Models</td>
<td>186</td>
</tr>
<tr>
<td>7.4.1 Modeling Time, Buffers and Resources</td>
<td>187</td>
</tr>
<tr>
<td>7.4.2 Model Checking with UPPAL</td>
<td>191</td>
</tr>
<tr>
<td>7.4.3 Structural Reduction Rules for DTPN</td>
<td>200</td>
</tr>
<tr>
<td>7.5 Conclusions</td>
<td>203</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>205</td>
</tr>
<tr>
<td>9 Outlook</td>
<td>209</td>
</tr>
<tr>
<td>Appendices</td>
<td>211</td>
</tr>
<tr>
<td>A Formal Analysis Techniques</td>
<td>215</td>
</tr>
<tr>
<td>A.1 Automata and Petri nets</td>
<td>215</td>
</tr>
<tr>
<td>A.2 Model Checking versus Correctness by Construction</td>
<td>218</td>
</tr>
<tr>
<td>A.3 Formal Component-Based Approaches</td>
<td>220</td>
</tr>
<tr>
<td>B Proofs of Lemmas and Theorems</td>
<td>223</td>
</tr>
<tr>
<td>B.1 Proofs of Chapter 3</td>
<td>223</td>
</tr>
<tr>
<td>B.2 Proofs of Chapter 4</td>
<td>229</td>
</tr>
<tr>
<td>C Modeling Behavior of Components</td>
<td>233</td>
</tr>
<tr>
<td>C.1 The Platform Controller, Switch and Joystick Driver</td>
<td>233</td>
</tr>
<tr>
<td>C.1.1 Functional Design of the Joystick Driver</td>
<td>233</td>
</tr>
<tr>
<td>C.1.2 Functional Design of the Platform Switch</td>
<td>235</td>
</tr>
<tr>
<td>C.1.3 Functional Design of the Platform Controller</td>
<td>236</td>
</tr>
<tr>
<td>C.2 The Platform Driver Component</td>
<td>239</td>
</tr>
<tr>
<td>C.3 The Planner Component</td>
<td>242</td>
</tr>
</tbody>
</table>
A robot is a programmable device that is capable of sensing its environment and physically manipulating the objects within it.

1.1 Brief History of Robotics

The concept of a robot as a mechanical device that would free a human from unpleasant, repetitive, dirty and hazardous tasks dates back many centuries. However, the word robot derives more recently from the Czech word robota (which
means forced labor) coined by the Czech playwright Karel Capek and his brother Jozef for a rather dramatic play *Rossum’s Universal Robots* in the year 1921. Later, in the year 1941, Issac Asimov coined the word robotics in his short story titled *Liar!*. In a later novel titled *Runaround*, he popularized the three laws of robotics, emphasizing on the importance of human safety in the design of robots for the service of humanity.

It was only in the year 1954 that the first digitally operated and programmable robotic arm called *Programmed Article Transfer* was developed by George Devol. A few years later in 1961, George Devol with the help of an entrepreneur named Joseph (Joe) Engelberger, setup the world’s first robot manufacturing company called Unimation and named its first line of commercial robotic arms as Unimate. The operations of this arm were controlled step by step using instructions stored on a magnetic drum. The following years, especially the late 70s and 80s saw an exponential increase in growth of robotic applications in the manufacturing industry. The definition of an *Industry Robot* was standardized (ISO 8373) by the International Federation of Robotics [350] as

*A device that is automatically controlled, reprogrammable, multipurpose manipulator programmable in three or more axes, which may be either fixed in place or mobile for use in industrial automation applications.*

The story of industry robots as tools for automation in industry has been one of great success. The capabilities developed in the area of industrial robotics since the late seventies (for eg. spot welding, spray painting, assembly of complex devices etc.) are now sufficiently mature to support the automation of most manufacturing processes. Within factories around the world, more than a million industry robots have been installed, where they perform complex feats of manipulations at blazing speeds and with very high accuracy. These developments were possible because industry robots are pre-programmed offline as single function devices and operated in carefully controlled environments under the supervision of experts.

Between the 1960s and the late 1980s, many attempts at developing fully autonomous mobile robots [8, 9, 10, 36] operating in unpredictable environment were made. It was only in 1989 that Joe Engelberger through his seminal book titled *Robotics in Service*, envisioned the first applications for robots outside the controlled environments of the manufacturing arena. This book is often considered as the birth place of service robotics. In his book, he presents a collection of almost 15 envisioned services that a robot could provide to either aid or replace humans in their daily life activities. A *service robot* has tentatively been defined by the International Federation of Robotics [350] as

*A robot that operates either semi or fully autonomously to perform services useful to the well-being of humans and equipment, excluding manufacturing operations.*
The last 20 years have seen a tremendous expansion in the deployment of service robots in areas such as personal/domestic robots (vacuuming [354, 119], lawn moving [285]), entertainment robots (toys [30], tours [263, 82]), education and training (like [146]), personal transportation [147], security and surveillance [121, 253], field robots (milking [332], agricultural [190], space [351]), inspection and maintenance robots (hull cleaning [83, 266], pipe cleaning [361, 194], wall cleaning [363],) etc. Many of these service robots have met tremendous commercial success by focusing on a limited set of functions and sharing their autonomy with a human in the loop. Although each of these examples individually demonstrate a significant progress in technologies for mobility, manipulation and sensing; these developments have not translated into equivalent successes for commercially viable service robots that try to combine these capabilities, for instance in service robots for home care [353, 59, 148, 309, 352].

Recent Developments. Over the recent years, a lot of interest has been generated around the development of service robots for home care. A crucial factor behind this interest arises from the changing demography in the EU, Japan and the US. An ageing population and a rapidly declining workforce leading to rising costs of institutional care has led to a search for solutions that would improve the quality of human life and allow an elderly to stay independently as long as possible. One widely acknowledged solution is seen in the development of a service robot capable of carrying out activities of daily living [207] in a human environment, safely effectively and reliably. Although this demand has given rise to several initiatives [353, 59, 148, 309, 352], they have only resulted so far in prototypes that operate in controlled environments of laboratories. The lack of commercial success of domestic multi-functional service robots, against all forecasts [133], stems from the complexity of hardware and software.

1.2 The General Challenges

A multi-functional service robot is a self-contained distributed system, integrating a wide array of sensors and controlling a variety of actuation devices by running concurrently, the various algorithms for sensing, control, planning and coordination that cooperate asynchronously to perform tasks in environments where humans live and work. So on the one hand, guaranteeing safety of humans is a primary requirement, and on the other hand, their commercial success depends on their effectiveness (i.e. are tasks being carried out in an efficient manner?), reliability (i.e. is the system dependable?), performance (i.e. are the tasks being completed in an acceptable amount of time?), and flexibility (i.e. is the system able to adapt to changes easily?). This presents both design time and run time challenges for a multi-functional service robot to be commercially viable.

A complex service robot is seldom built from scratch, rather they are built by reusing existing hardware and software solutions (for eg. sensing and actuation
INTRODUCTION

devices, device drivers, pattern recognition, trajectory generation algorithms, etc.) alongside custom made solutions. Often, the details about parts that are being reused are not fully known. This arises from a lack of having a formal specification of what we need to know about them. As a consequence they are treated as black boxes and their behavior as a whole is not well understood. So the integration of both existing and self-made computation methods into an operational system, while ensuing safety, effectiveness, reliability, performance and flexibility is a software design challenge. Although, the logical behavior of the various computation methods are sometimes well understood in isolation (if they are self-made and relatively unknown if using third party solutions), ensuring their correct interactions to achieve a collective goal of a composed system is not trivial. By correct we mean behavioral correctness properties like freedom of deadlocks, livelocks etc.

The runtime challenge for such robots lies in their ability to deal with a dynamic world in a safe and effective manner. The development of autonomous behavior for robots to deal with such environments has been extensively investigated and achieved only in few mono-functional systems. In fact, higher the autonomy of the robot, the more its environment needs to be adapted. This is because it is not possible to foresee and pre-program the behavior corresponding every situation that could arise in such environments. Therefore for multi-functional robots operating in uncontrolled environments, autonomy only makes sense if it is supported by natural intelligence in the form of a human in the loop. So for a service robot to be effective it must be partly autonomous or simply a controllable tool, where human interactions replaces the need for perfect sensors (that are noiseless and fault tolerant) and robust interpretation algorithms. Due to this kind of shared autonomy between a human and a robot, one of the challenges lie in getting this balance right in order to be safe and effective.

The robotics domain has long been dominated by mechanical and electrical engineers and many challenges in the context of these domains have been extensively studied and addressed. As a result many well-established model based approaches (using Simulink, Modelica etc.) exist to understand and predict the behavior of the system using an abstraction, i.e. model. However, in all these approaches, the final artifact is the software of the system (which is a discrete event system) and it is often the case that the semantics of these translations are not clearly defined. As the complexity shifts more and more towards software, providing strict guarantees about their behavior becomes a prerequisite. So a more structured approach based on formal models becomes necessary. Unfortunately, most existing design and development methods rely on informal models, simulations and test cases which are inherently incomplete. Many of these problems have been extensively investigated in the domain of computer science and have been successfully applied as software engineering principles, notably in information systems, manufacturing systems, embedded systems, communication protocols etc. In the next section, we discuss some notable advances in the application of software engineering practices to tackle the complexity problems in the software design aspect of robotics.
1.3 The Component Based Approach

The separation of concerns as a method to deal with complexity of software is one of the most fundamental principles of software engineering [282, 114, 268, 232], dating back almost 40 years. This strategy involves the identification and separation of a complex problem into smaller concerns (problems) such that each concern can be solved in isolation and then combined together to form a solution. In the software domain, the separation of concerns is achieved by modular programming principles that focus on encapsulation of operations into tightly coupled coherent units of functionality called components. A widely accepted definition of a software component is as follows:

*A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties.* [315]

A component based approach (commonly known as Component Based Software Engineering (CBSE) [166, 96, 76]) is a strategy to build complex systems by composing components over their shared interactions (interfaces) just like how electric circuits are created from individual hardware components. This facilitates the handling of complexity in the design phase and leverages the required engineering quality factors like compositionality, reuse, incremental construction, adaptability, extensibility and maintainability. In the traditional object oriented approach, subsystems interact by invoking functions or methods of other subsystems from their code. Such an interaction requires a knowledge about the functions being invoked. In contrast, components and their interactions are designed independently of their context of use. The adoption of the component based approach has led to a shift in emphasis from building systems following traditional requirement analysis, design and development towards composing software systems from a mixture of self-made and off-the-shelf components.

The realization of a component-based system requires the adoption of a component based design strategy supported by the right software component technology to implement them. By component based design strategy, we mean an architectural style or a high level design pattern describing the components of the system and their interactions. Software component technologies complement the design strategy by facilitating the incremental development of communicating components by providing an infrastructure for their deployment and asynchronous communications.

A software architecture is a specific choice of functional building blocks (software components) that cooperate as a system to perform per specification. By providing abstractions at various levels, it serves as a means to reason about the various aspects. There are many different definitions and understanding of a software architecture. The commonly accepted definition seems to be
the structure, which comprises software components, the externally visible properties of those components, and the relationships among them [28].

However, it is not only the explicitly visible properties, rather what it is that needs to be known to be able to use it. In the context of component-based design, the relationship between each pair of components in an architecture represents an asynchronous interaction. The set of all interactions supported by a component are explicitly defined by its interface(s) [96]. By separating the design of a component from its interface, an existing component can seamlessly be replaced by another component implementing the same interface, i.e. the notion of a plug and play architecture. Furthermore, a software architecture has two competing dimensions [7], namely functional and non-functional. The former deals with the structure and behavior of the architecture itself, while the latter captures the quality attributes of the system, namely performance, reliability, usability, portability etc. The design of component based systems is inevitably a balancing act between these two dimensions.

The application of component based architectures in the domain of robotics is fairly new [79, 75, 56, 258, 265]. Before the recent rise of flexible component-based architectures, layered architectures based on hierarchial decomposition were popular with autonomous robots for a long time [293].

Survey. In the sixties, an early notion of robotic architectures started out with pure symbolic planner based approaches, popularized as the sense-plan-act paradigm [262]. The main feature of this approach is that planners operate on a world model, updated by sensory information. So they do not directly interact with the sensors. As a result, these systems were slow and unable to react safely in a dynamic world. These drawbacks led to a quest for new robot control architectures based on reactive planning [125, 12]. In the late eighties, an influential contribution in this direction was made by the subsumption architecture of Brook [77], based on reactive planners. This architectural paradigm consisted of layers of interacting finite state machines (behaviors) that each close the loop from sensors directly to actuators. By means of an arbitration mechanism it was possible to create multiple interacting layers of behavior to produce more complex robots [23]. However, this approach had its limitations in that behavior composition and their optimization was a difficult task. The late eighties and nineties saw many new approaches attempting to combine the best of both reactivity and deliberation. This culminated into the popular three tier (3T) architecture consisting of functional, executive and planner layers [132, 126, 60, 300, 14, 293, 127]. In these architectures, the higher layer coordinates the actions of a lower layer according to some generated plan, maybe by an even higher layer. Many architectures based on 3T have been subsequently proposed, most notably ATLANTIS [131], Saridis’ intelligent control architecture [290], LAAS [15] etc. However, this kind of layering had a few shortfalls: (a) the responsibilities of the planner and executive layer were often overlapping, (b) the inability of the planner to directly access
the functional layer led to redundancy, and (c) no emphasis on hierarchy within each layer. The Coupled Layered Autonomous Robot (CLARAty) architecture [337] addresses these shortfalls by proposing a two layered model with an added dimension of granularity (object oriented principles) within each layer and saw the blending of declarative and procedural techniques for decision making. The two layered approach also occurs in the CIRCA [254] and ORCCAD [64] architectures, augmented with techniques for formally guaranteeing reliability. The former was based on an online approach while the later was based on offline logical and temporal verification techniques.

Despite the strong coordination support provided by layered architectures, none of them have become widely used, primarily due to their limited extensibility. In contrast, component-based software architectures are not layered, rather these architectures represent on a single layer, a network of connected components, where connections represent sets of asynchronous interactions over interfaces of components.

Development Frameworks. Over the last two decades, the realization of software architectures of robots have been well-supported by domain specific development environments, also referred to as Robot Development Frameworks (RDFs). An RDF provides a way to deal with the heterogeneity in hardware and software while simplifying software design and reducing development costs. So a developer can focus on building components in isolation, after which they are combined and integrated with other existing components. All RDFs are in essence a middleware. A widely used definition of a middleware [24] is:

A class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system.

The history of RDFs have also seen a similar evolution from layered approaches like CLARATY [338], CARMEN [246] and Player [135], towards message passing, component-based approaches like OpenRTM [21], MSRS [172], YARP [237], OROCOS [80], ROS [278], OPrOsis [173] and ORCA [267]. An extensive surveys comparing the various RDFs can be found in [118, 195].

Robot Operating System

Over the recent years, the Robot Operating System (ROS) framework has seen an exponential rise in its popularity within the service robotics community as a common platform to share knowledge through component based repositories. In many ways, the general purpose nature of ROS [53] addresses most shortcomings of its predecessors. It hides heterogeneity of hardware and software by providing a
thin and structured communication layer supporting the most recurring message-based interaction patterns in robotic systems. This communication model is made available to a component developer as a set of software libraries. In addition to a communication model, ROS also provides infrastructure capabilities for component instantiation, dynamic configuration and deployment. It comes with a number of integrated frameworks (like OROCOS [80], Player [135] etc.), libraries (like OpenCV, KDL, etc.), general purpose configurable software modules (for mobility, manipulation and vision) and tools for visualization and simulation like RViz and Gazebo. In ROS, blocks of functional code (C++/Python) can be instantiated as components (referred to as nodes).

A component performs iterative computations (at some specified rate) and it is allowed to communicate with other component(s) using one of the supported message-passing interaction patterns. Messages are strictly typed data structures and are exchanged over buffers (by send/receive actions). A buffer has an associated data type specifying the data type of messages that can be stored. Furthermore, at run-time, a ROS system can be configured dynamically, i.e. components maybe added or removed from the system at run-time. Dynamic reconfiguration is made possible by the naming and service registration server called ROS master. A ROS system has one ROS master. The ROS master acts as a central registry of information about deployed components and their interaction patterns. This allows components to discover each other. In this way location transparency is achieved. Both the arrival as well as the departure of components are registered by the ROS master. Once a component is aware of the available components and the services they provide, connection negotiation and an exchange of messages are carried out on a peer to peer basis.

The communication model of ROS is a collection of three very frequently recurring asynchronous interaction patterns in robotic systems: the publish-subscribe (PS) pattern, the remote procedure call (RPC) pattern and a pre-emptive variant of RPC. Each interaction pattern is made of a set of clients and a set of servers. The clients and servers rely on a middleware module that is responsible for (a) interactions with the ROS master, (b) client-server connection negotiation and management, (c) message transportation (over managed connection), and (d) message buffering capabilities.

**Observations.** Component-based RDFs like ROS have made it possible to rapidly assemble complex software systems using a mixture of generic off-the-shelf components and self-made components, and deploy them on robots. In the current practice of robot software development, developers tend to focus on the functional details of a component at an early stage of product development without considering how these components will collaborate with each other to achieve the objectives of the system. As a result, the behavior of the final system becomes hard to predict. This is due to the fact that the gluing together of components over shared interfaces (i.e. interface compatibility) alone is in itself not sufficient to guarantee the absence of unexpected behavior for eg. race conditions
and deadlocks, in the presence of concurrency. Furthermore, for validation of system behavior, traditional software testing techniques and simulations are commonly used. Although a certain level of dependability can be attained by these techniques, they have the disadvantage of being incomplete as only a small subset of the operating conditions and inputs can be explored. Unfortunately, none of the presented RDFs prescribe an integrated formal design method to assemble a system from a set of components such that the system behavior is guaranteed.

1.4 Model Driven Approach

The prediction of properties of a solution from its constituent parts using an abstraction of the concrete system (i.e. model) is a fundamental concept across many engineering disciplines. A model enhances the ability to understand, predict and possibly control the behavior of the system [257]. The use of models to drive the development of software systems is commonly known as Model-Driven Software Engineering (MDE) [339, 55, 35]. The model-driven approach compliments the component-based approach. Both these approaches tackle complexity but at different levels. The component based approach focuses on reuse and separates component development from system integration. The model driven approach separates the domain knowledge (robot planning and control strategies) from the implementation details (source code), promoting reuse of domain knowledge by domain experts and at the same time facilitating the identification and application of new/existing design patterns as solutions to recurring design issues by software designers. In the context of component-based systems, the model of a component and its interface(s) are treated as different concerns. A component model must specify how a component behaves in an arbitrary environment. An interface model must specify how it can be used and what it might require from its environment.

Over the last decade, the MDE approach has been widely adopted in domains where complex systems must guarantee their reliability, effectiveness, performance and safety, such as in embedded systems [318, 313, 120, 248, 226], avionics [99], automotive [129], etc. However, the MDE approach is fairly new in the robotics domain. As with other applications of software engineering, two types of modeling frameworks have received a lot of attention: informal (UML-based solutions) and formal (automata and Petri nets). The former is good for structural representation of system constituents and for its simplicity of use but ambiguous for understanding and analysis. The latter provides a rigorous mathematical framework for specifying and analyzing the behavior of a modeled system.

1.4.1 Informal Approaches

The informal MDE engineering approach for service robotics has seen two popular initiatives to standardize the robotics component market, namely the BRICS
project [81] and Object Management Groups Robotics Technology Component (OMGRTC) [97]. The latter gave rise to three well known implementations, namely [287], OpenRTM-aist [21] and OPRos [174].

The 3-View Component Meta-Model [17] is a model driven tool chain that supports component-based platform independent modeling and platform-specific code generation through model transformations. It supports three complementary views, namely structural, coordination and algorithmic and facilitates the specification of structural and behavioral variations during design time.

The GenoM3 [221] is a component generator framework for functional design of components of the LAAS architecture. It hides the large variability in available middleware solutions by providing a middleware independent component template for component specification. Furthermore, a template based approach to generate glue logic to integrate components sharing a common middleware is also provided. As a consequence, underlying middleware can be easily replaced without affecting the components themselves.

Another model-based development tool, EasyLab [27] is popular for programming sensors and actuators. The framework supports facilities for modeling, simulation, code generation and debugging and provides a consistent view of models at different levels of abstraction.

Observations. Most informal approaches to MDE for service robotics rely on informal UML-based models ([289]) with a focus on code generation. Due to the lack of well defined semantics, the behavior of the resulting system cannot be guaranteed by traditional case-based testing techniques and simulations. Therefore there is a clear need for formal models supported by tools for automatic verification. This need has been recognized long ago and extensively addressed by the formal methods community. Over the years, many model verification and behavior preserving model transformation techniques have been developed and many of these techniques have been successfully applied in the context of Discrete Event Systems (DES) [87].

1.4.2 Formal Approaches

A DES represents a class of systems (manufacturing systems, flexible production systems, transportation systems, embedded systems, robots etc.) whose state spaces are logical or symbolic and whose state evolutions depend entirely on the occurrence of discrete events (instantaneous) over unknown instants of time (also known as reactive systems [158]). As such systems are generally distributed, the treatment given to interactions between components becomes a central issue. The two fundamental interaction models of distributed systems are synchronous and asynchronous communications. In both these models, an interaction involves send and receive events. In case of synchronous communication, an interaction is assumed to be instantaneous, i.e. non-buffered and no intermediate state in which a receive is pending is modeled. An asynchronous communication, on the other
hand, decouples the send and receive with a buffer. Even though considering intermediate states yields in an increased complexity, asynchronous communication allows for a natural representation of message passing distributed systems.

The formal study of DES is rooted in the mathematical formalisms of automata theory [86, 168] and Petri nets [270, 250, 271] and to some extent process algebra [239, 167, 43]. In the Appendix A, we present a brief survey of the various extensions and applications of automata and Petri net theory to the modeling and analysis of (un)timed component-based systems. Most of these extensions are either based on networks of communicating automata (synchronous interactions) or open Petri nets (asynchronous interactions). We also discuss the two ways of guaranteeing system behavior: model checking and correctness by construction.

In the robotics domain, the use of the Petri nets and automata are the most prevalent. In early approaches, the extent to which formal techniques for system design were being applied was largely fragmented between the lower functional layer and higher decisional layers. In fact the latter has received more attention since components like planning already rely on a model.

Survey. The work in [358], presents a model-based probabilistic approach that abstracts the system into a state transition based language with a focus on dependability. This approach relies on explicitly specifying invariants of state evolutions, which were then used by a controller at run-time to execute these models. In [139], the CIRCA SSP planner for hard real time controllers is proposed. The planner synthesizes an offline controller as a timed automaton from a set of deadlines and pre/post-conditions, which can be verified by standard model checking techniques. In [63], an approach for model checking an agent programming language, AgentSpeak, for reactive planning systems is proposed. For this a toolkit called CASP (Checking AgentSpeak Programs) supporting model checking was developed. In [301], a system supporting the translation of MPL (Model Processing Language) and TDL (Task Description Language of CLARAty) into a symbolic model checking language of SMV is proposed. In [196], an approach for automatically generating correct by construction robot controllers from high level task representations in structured English is proposed. The strategy relied on the translation from structured English descriptions into a type of temporal logic and eventually into an automata. A similar approach is proposed in [360], starting from temporal logic descriptions into synthesized finite state automata based controllers that satisfy the specified temporal descriptions. In [171], R2C has been proposed as a run-time state checker tool between the functional and decisional levels of the layered architecture based on the language EXoGEN. In [64], the ORCCAD system to prove formal properties of the functional level was proposed. The development environment was based on Esterel extended with constructs to specify tasks, procedures and coordination [304]. The work presented in [19] based on synchronous languages had a similar objective. The SCADE system [115] uses the Lustre language, which combines state machines and synchronous data flow with code generation facilities.
An extended version of the automata-based *interaction system* framework (see Appendix A) resulted in the BIP (Behavior, Interaction and Priority) framework [29], which also includes timed specifications. BIP is a component-based framework and supports the formal modeling and verification of real-time systems. Here a two layered coordination mechanism is defined for its constituent components. The semantics of the BIP model are described in terms of a labeled transition system, extended with data and functions written in C. The communication model of the BIP framework relies on synchronization mechanisms.

In [8, 122] an interesting approach for the two-tiered LAAS architecture was proposed. The functional level components are specified using the model-based GenoM tool [127] and then composed into the BIP software framework. The framework supports offline verification (tools like D-Finder) using compositional [34] and incremental methods [36] to guarantee absence of deadlocks and data integrity. As the focus is on the interaction layer, the behavior of components themselves are not considered. The timed extension of the BIP model is expressed as a timed automata with support for real-time model execution capabilities. For the decisional level, temporal planners are specified in ATL [151] and then translated into components of the BIP framework. So both the functional and decisional layers are modeled using a common framework. However, the approach has a few drawbacks: (a) only a part of the functional level is checked for the absence of deadlocks. Other properties like liveness, timeliness and reliability cannot be guaranteed, (b) due to the synchronous communication paradigm between components, the BIP model cannot be naturally distributed. For timed systems this problem is even more critical. and (c) domain specific interaction patterns (like the communication model of ROS) are not available for reuse.

Similar developments have appeared in Petri nets as well, since the seminal paper [117], extensive applications have appeared in modeling and analysis of safety critical systems using stochastic Petri nets [111, 219, 279], flexible manufacturing systems [26, 31, 189, 227, 252, 255, 321] and robot controller design [128, 209, 208, 365]. There were also some contributions to structured design of robot controllers with a focus on the overall design of the system [292, 303, 302].

For the layered architectural styles in robotics, a number of authors have used Petri nets and their extensions for modeling and analysis.

In [343], a three level structure (based on the 3T architecture of [290]) for organization, coordination and execution levels were defined along with definitions for cost functions and reliability measures. Here the focus was on the coordination layer. The work of [107] combines the hierarchial approach of [343] with a notion of time based on stochastics (GSPN). In [344], alongside these concepts they also define Petri net Transducers, which are a good choice for modeling the control and synchronization of operations exhibiting concurrency and conflict.

In [211], a method to optimize robotic tasks is defined. The work in [210] goes further and consolidates the concepts of robotic task theory with a set of primitive tasks. Some results from successful empirical implementations were presented in [243]. An extension of the work in [243, 103] is carried out in [101]
with a focus on performance and reliability analysis, as well as synthesis of task plans for robots. The work in [367] presents a modeling and execution framework for tasks in (multi-)robotic systems, called Petri net Plans [369, 368]. In [199, 201, 200, 204, 202, 203, 205], methodologies for modeling supervisory control of discrete event systems as finite state automaton or Petri net is presented. Here there is a focus on multi-robot applications while using Linear Time Temporal Logic to specify and enforce coordination rules.

In [84], a modular approach based on control nets was proposed. A control net is a high-level Petri net (predicate-transition net) that are ideal for designing the control systems. The approach propagated the use of modular building blocks, each realizing a coherent set of functionalities. The work of [245] extends this approach using a popular timed extension of Petri net, namely time Petri nets to model time-outs, periodic activities, as well as synchronization and concurrency.

**Static Analysis.** Another widely used approach to verify software systems of robots is by program verification. Such techniques compute information about the behavior of an abstract representation of the program without actually executing it. The formal basis lies in the abstract interpretation framework [102], which relates how abstraction techniques like abstract domains and abstract functions can be used to analyze the static behavior of a program. There exists a plethora of tools that support this technique for static analysis of procedural languages like C/C++ and Java with well-known real life applications in aviation (like Airbus) and space missions (like Mars Path Finder, Deep Space One, etc.). There are also several tools that support an annotation-based approach [238] using pre/post conditions and loop invariants. Such techniques are useful for validating the functional behavior of components in isolation. However, for a component-based approach, such techniques have their limitations because behavioral correctness in the presence of communication cannot be guaranteed.

### 1.4.3 Gaps and Challenges

It is clear that there is a lack of a formal component-based modeling and analysis framework for software systems of robots that provides a good balance between the ease of modeling and their analyzability.

- On the one hand, existing frameworks for verification by model checking techniques suffer from the well-known state space explosion problem. On the other hand, existing correctness by construction design methods either (a) focus on specific parts of the architectural model, (b) guarantee only safety properties, and (c) focus on a communication model that is either too general or supports only simple interaction patterns like unidirectional broadcast and request-response.

- The components responsible for high level planning and coordination are often long running processes that provide a service to other components. For such components, it is very desirable to provide as part of their service,
feedback about their progress to other components as well as facilities for structured process cancellation by other components. Furthermore, as robots become multi-functional and as a consequence more complex, the participation of a human operator in decision making will become more integrated into the logical structure. As a result, complex interaction protocols will be required to support choices in the underlying control flow.

- Petri nets provide a more expressive and natural means to model, visualize and analyze asynchronous interactions and concurrency in distributed systems. Furthermore, weak termination (implies the absence of deadlocks and livelocks) is a well-studied property in Petri nets (see Appendix A). It means that a system must always be able to reach a final state from any reachable state. In combination with structural properties exhibited by subclasses of Petri nets, many interesting results exist such that weak termination is implied by their structure. These classes of nets are ideal for modeling procedures that exhibit non-deterministic choice and concurrency. In [164], the authors build upon this result to propose a correct-by-construction strategy that is very promising for two reasons: (a) it prescribes a bottom-up refinement-based strategy for incremental construction of a network of asynchronous communicating components while guaranteeing the weak termination property, and (b) the graphical representation of these formal operations makes it easy for a non-domain expert to adopt these principles during system design.

- The two popular timed variants of Petri nets for verification of real-time systems are time Petri nets [235] and interval timed colored Petri nets (ITCPN) [1]. The latter, i.e. ITCPN, with the notion of token timestamps provides a natural way to model message passing systems. As with other timed approaches, the time domain is non-decreasing, so finite reduction methods are required for model checking them. In [2], an analysis method called Modified Transition System Reduction Technique was proposed which is sound but not complete and could result in possibly infinite state spaces. A finite, sound and complete approach preserving trace equivalence was proposed in [66]. The approach relies on computing classes of reachable states called state classes as an occurrence graph. An infinite occurrence graph is then transformed into a finite one by factoring out the global time. For a sub-class of ITCPN, with deterministic fixed delays and ignoring data, a reduction method based on termination time equivalence was proposed in [197], which gives a better reduction. A similar approach to deal with time intervals is yet an open question.

- For reliability analysis, there is extensive literature on stochastic Petri nets where the execution time of transitions is exponentially distributed which leads to continuous time Markov processes [13]. For the class of Generalized Stochastic Petri Nets, an extension of time Petri nets [235], there is a famous
software tool for modeling and analysis called GSPN [349]. However, the case for discrete probabilities has received very little attention, although it has many advantages. We believe that duration of events are naturally specified as random choices within time intervals, and their reliability by associating discrete probabilities to these choices.

– The mathematical theory of designing control algorithms for dynamical systems like robots has its roots in control theory. It is common practice to use tools like Matlab/Simulink, Modelica, LabView etc. to model and validate these control algorithms. A Simulink model describes a time driven dynamic system as a set of mathematical equations defining one or more control algorithms together with their environment, that are evaluated at discrete points in time. Such models exhibit both discrete and continuous dynamics, simulated by discretizing time. These tools also provide facilities to generate code which are then embedded into a larger DES like for instance an arm controller component cooperating with an online trajectory planner component to achieve a desired task. Then the underlying time-driven dynamics (expressed as difference/differential equations) are captured by the notion of time progression in a state (i.e. the time elapsed between consecutive event occurrences) of a discrete event system. Unfortunately Simulink has only informal semantics and of course an operational semantics in the form of an implementation in software. As a result, it is not clear how a Simulink model can be incorporated into a formal framework like Petri nets.

The objective of this thesis is to provide a general framework to model and analyze both logical and timed properties of software systems of robots. To achieve this objective, a set of research questions have to addressed.

1.5 Research Questions

In this thesis, we want to develop a component-based architectural framework for compositional design and verification of software systems of robots. The framework must support the asynchronous communication paradigm and prescribe design principles to construct a network of components that are guaranteed to weakly terminate. It must also be possible to extend these models with the notion of time to analyze performance and reliability of the system. To realize this, we propose the following research questions:

R1 Components of software systems of robots exhibit recurring patterns of interactions. Therefore, can we identify and model the set of recurring interaction patterns and prove their weak termination property? Furthermore, as robots become more multi-functional and involve a human, more complex interaction patterns will be required. Therefore, can we find a design method to construct new interaction patterns guaranteeing the weak termination property? Furthermore, can we propose a correctness by construction
design method to derive a model of the system from a set of components and interaction patterns?

R2 The correct logical ordering of events in a software system is not sufficient to guarantee safety, performance and reliability of a robot. For this we must incorporate the notion of time. However, such an extension leads to an infinite state space due to a non-decreasing time domain. Therefore, can we extend the architectural framework with time constraints specifying task execution time and message time-out such that verification by model checking is possible?

R3 How does a time driven system modeled using tools such as Simulink fit into our architecture framework?

R4 In order to validate our design method, we must validate its applicability using real world examples. Robot ROSE and the remote maintenance of the ITER plant are two good examples of applications of service robots. Can we demonstrate the applicability of our architecture framework and its design strategies using these examples?

1.6 Contributions of this Thesis

In this thesis, we will use the mathematical formalism of Petri nets as they offer some inherent advantages: (a) represents a larger class of languages than the class of regular languages; (b) has an intuitive graphical notation; (c) ideal for describing asynchronous distributed systems; (d) locality of state and action concept in a Petri net allows for both top down and bottom up modeling strategies; (f) notion of time is well-studied; and (e) structural analysis techniques of Petri nets overcome the drawbacks of model checking. Furthermore, the weak termination property is well studied and there exist many classes of Petri nets that guarantee this property by their structure.

The thesis is divided into three parts. The first part introduces the architectural framework and design principles for constructing arbitrary correct by construction systems. In the second part, we introduce a timed framework and show how verification by model checking is possible. We compare this framework with other existing timed frameworks. In this part, we also study the relationship between time driven systems in the context of Simulink and our architecture framework. In the third part, we apply the techniques developed in the first two parts to real-life examples.

1.6.1 Component-Based Architectural Framework

In Chapter 3, we present an architectural framework with a focus on control flow (i.e. the logical ordering of events) and based on the two concepts: component
and interaction pattern. An interaction pattern models a pattern of communication between components. The framework provides four recurring interaction patterns (that are weakly terminating by their structure) and a design method to derive arbitrary interaction patterns that are weakly terminating by their structure. The relationships between components and their interaction patterns are captured by an architecture diagram in a concise manner. For the weak termination property, we identify a sufficient condition that the relationship between components of an architecture diagram must satisfy. To derive a weakly terminating Petri net model from an architecture diagram, we prescribe a construction method. The construction method is an incremental and bottom-up refinement strategy to insert interaction patterns into existing components of the system, so far. During construction, the behavior of existing components may be elaborated by top-down refinement techniques. Although the proofs for guaranteeing the weak termination property are formal and rigorous, the construction method itself is very simple and the design process is graphical in nature. As a result, a non-domain expert can easily grasp these design principles and use them as guidelines during system design.

The material presented in this part of the thesis is based on:


- Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf, A Component Framework where Port Compatibility implies Weak Termination, Computer Science Report, TR-11-08, Technische Universiteit Eindhoven, 2011

1.6.2 Analysis of Timed Properties

In the Chapter 4, we propose a discrete model of time in Petri nets called Discrete Timed Petri net (DTPN) and its stochastic variant called Discrete Stochastic Petri net (DSPN). The well known Interval Timed Colored Petri net model, discarding data (color) (ITPN) is a subclass of DTPN, with good tool support (see CPN Tools [347]). In both these models, the delay incurred by an event are specified as a time interval. However, the DTPN model extends the ITPN model with constructs that allow for modeling time-outs. As with every model of time,
reduction methods are necessary to transform their infinite state space (due to a non-decreasing time domain) into a finite one for analysis. For ITPN, a reduction method based on occurrence graphs and preserving trace equivalence was proposed in [46]. For DTPN, we propose a two step reduction method preserving simulation equivalence, which is a stronger notion. The novelty lies in the first step, transforming an infinite interval into a finite one. The following time reduction step is quite similar to the one proposed in [197], i.e. based on termination time equivalence of a subclass of ITPN. For DSPN, we show how Markov techniques can be applied to answer interesting questions about probability of reaching a marking, expected time to leave a marking and expected sojourn times in equilibrium. For the class of workflow nets, we show how structural techniques can be applied to reduce computational effort.

In the Chapter 5, we study the expressive power of DTPN by comparing it with other existing models of time like timed automata and timed extensions of Petri nets like timed Petri nets and time Petri nets. We show how these different models can be unified by transformations from one to the other. Then the advantages of each formalism, which can either be in the ease of modeling, ease of verification or availability of mature modeling tools, can be exploited. We consider timed automata because it is supported by the popular and mature UPPAAL toolset [356]. By expressing inhibitor arcs using subclasses of DTPN, we prove the Turing completeness of model class DTPN. For these subclasses, we also analyze their modeling comfort without claiming to be complete.

In the Chapter 6, we study the relationship between time driven systems modeled using Simulink and discrete event systems modeled using our architecture framework. As Simulink has no formal semantics, we give semantics to Simulink using DTPN and show how interesting questions about model timing validation and performance can be verified by model checking. For completeness, we also show how Petri nets can be expressed as a Simulink model.

The material presented in this part of the thesis is based on:


- Debjyoti Bera, Kees M. van Hee and Henk Nijmeijer, Relationship between Simulink and Petri nets, Computer Science Report, TR-14-06, Technische Universiteit Eindhoven, 2014
1.6.3 Applications

In the Chapter 7, we present a design strategy using all the results developed in the previous chapters and apply this strategy to the two real life examples: service robot ROSE and the ITER remote handling plant. The strategy starts with user requirements and delivers a Petri net model of the system from an architecture diagram. By adding time constraints, this model is transformed into a DTPN. As CPN Tools only provides the possibility of model checking a subclass, we show how DTPN models can be expressed as a network of timed automata as incorporated in UPPAAL. For models expressed in this way, interesting timed properties are formulated and verified by model checking. To reduce the computation effort of model checking, few behavior preserving structural reduction rules for DTPN are presented.

The material presented in this part of the thesis is partly based on:


In this chapter we introduce the basic mathematical notations that will be used throughout this thesis.

2.1 Basic Mathematical Notations

A set $S$ is a possibly infinite collection of elements. We denote a finite set by listing its elements between braces, i.e. $S = \{a, b, c\}$ denotes a set $S$ containing elements $a$, $b$ and $c$. The empty set is denoted by $\emptyset$. We call a set with one element a singleton set. We denote by $|S|$ the number of elements in the set $S$. An element $a$ belongs to a set $S$ is denoted by $a \in S$. The set operations $\cup, \cap, \setminus, \subseteq$ are defined in a standard way. We denote the powerset of a set $S$ as the set of all subsets of $S$, denoted by $\mathcal{P}(S)$. A partition of a finite set is a collection of disjoint, non-empty subsets whose union is the finite set. We denote the set of reals, rationals, integers and naturals by $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$ and $\mathbb{N}$ (with $0 \in \mathbb{N}$), respectively. We use superscripts $+$ for the corresponding subsets containing all the non-negative values, e.g. $\mathbb{Q}^+$ is the set of all non-negative rationals.
The cartesian product of two sets $A$ and $B$ denoted by $A \times B$ is the set of all ordered pairs $\{(a, b) \mid a \in A \land b \in B\}$. On the other hand, a relation $R$ between sets $A$ and $B$ is a subset of $A \times B$, i.e. $R \subseteq A \times B$. The domain of the relation is the set $\text{dom}(R) = \{a \in A \mid \exists b \in B : (a, b) \in R\}$. Its range is the set $\text{rng}(R) = \{b \in B \mid \exists a \in A : (a, b) \in R\}$. The inverse of relation $R$ is a relation $R^{-1} \subseteq \{B \times A\}$. Consider a relation $R$ defined over the same set $A$. Then $R$ is called reflexive if $\forall a \in A : (a, a) \in R$. We call $R$ irreflexive if $\forall a \in A : (a, a) \notin R$. $R$ is called a symmetric relation if $\forall a, b \in A : (a, b) \in R$ implies $(b, a) \in R$, and it is called antisymmetric if $\forall a, b \in A : (a, b) \in R \land (b, a) \in R$ implies $a = b$. If $\forall a, b, c \in A : (a, b) \in R \land (b, c) \in R$ implies $(a, c) \in R$, then we call relation $R$ as transitive. If relation $R$ is reflexive, symmetric and transitive then we call it an equivalence relation. Relation $R$ is a preorder if $R$ is reflexive and transitive. A preorder is a partial order if it is antisymmetric.

A subclass of relations is functions. A relation $f$ over two sets $A$ and $B$ is a function $f : A \rightarrow B$, if $\forall a \in A, b \in B : (a, b) \in f \land (a, b') \in f$ implies $b = b'$. We write $f(a) = b$ for $(a, b) \in f$. We lift the notion of functions to sets in a standard way. A partial function is denoted by $f : A \rightarrow B$, if $\text{dom}(f) \subseteq A$. If $\text{dom}(f) = A$ then the function is called total. We will call a total function as a function in the remainder of this thesis. A function with an empty domain is called an empty function. The notions of injection, surjection and bijection are defined in a standard way. To establish a relation between more than two sets, we define the notion of generalized cartesian product. The generalized cartesian product of a set-valued function $F$ is denoted by $\Pi_F = \{f \mid f$ is function $\land \text{dom}(f) = \text{dom}(F) \land \forall t \in \text{dom}(F) : f(t) \in F(t)\}$.

A bag over some set $S$ is a function $m : S \rightarrow \mathbb{N}$. For $s \in S$, $m(s)$ denotes the number of occurrences of $s$ in $m$. We enumerate bags with square brackets, e.g. the bag $m = [a^2, b^3]$ has an element $a$ occurring twice and element $b$ occurring thrice and all other elements have multiplicity zero. The set of all bags over $S$ is denoted by $B(S)$. We write $[]$ for an empty bag and we use $+$ and $-$ for the sum of two bags and $=, <, >, \leq, \geq$ to element wise compare bags, which are defined in the standard way. A set can be seen as a multiset in which each element of the set occurs exactly once.

A finite sequence $\sigma$ over some set $S$ of length $n \in \mathbb{N}$ is a function $\sigma : \{1...n\} \rightarrow S$. The set of all finite sequences over $S$ is denoted by $S^*$. The length of a sequence is denoted by $|\sigma|$. We represent a sequence of length $n$ by $\sigma = \langle s_1, ..., s_n \rangle$ where $s_1, ..., s_n \in S$ and $\sigma(i) = s_i$ for $1 \leq i \leq n$. The presence of an element $x \in S$ is asserted by the predicate $x \in \sigma$. If $|\sigma| = 0$, it is the empty sequence denoted by $\epsilon$. The concatenation of two finite sequences $\sigma = \langle a_1, ..., a_n \rangle$ and $\sigma' = \langle b_1, ..., b_m \rangle$ is denoted by $\sigma \circ \sigma'$ and is the sequence $\langle a_1, ..., a_n, b_1, ..., b_m \rangle$ of length $n + m$. The Parikh vector of a sequence $\sigma$, denoted by $\overrightarrow{\sigma}$, is a bag representing the number of occurrences of each element in $\sigma$. A sequence $\sigma$ is a prefix of a sequence $\gamma$ if
a sequence \( \sigma' \) exists such that \( \sigma \circ \sigma' = \gamma \). The projection of a sequence \( \sigma \in S^* \) onto a set \( Q \) is defined inductively as \( \epsilon_Q = e; (\langle s_1 \circ \sigma \rangle)_Q = (\langle s_1 \rangle) \circ \sigma |_Q \) if \( s_1 \in Q \) and \( (\langle s_1 \circ \sigma \rangle)_Q = \sigma |_Q \) if \( s_1 \notin Q \). We denote the set obtained by interleaving two sequences \( \sigma \) and \( \gamma \) by \( \sigma||\gamma \).

**Intervals and Set Refinement.** Let \( \inf(A) \), \( \sup(A) \), \( \min(A) \) and \( \max(A) \) of the set \( A \) have the usual meaning and \( \mathcal{P}(A) \) is the powerset of \( A \). We define \( \max(\emptyset) = \infty \) and \( \min(\emptyset) = -\infty \) and call a set \( \{ x \in \mathbb{Q} \mid a \leq x \leq b \} \) with \( a, b \in \mathbb{Q}^+ \) a closed rational interval. The set \( B \) is a refinement of set \( A \subseteq \mathbb{Q}^+ \), denoted by \( A \triangleleft B \), if and only if \( A \subseteq B \land \sup(A) = \sup(B) \land \inf(A) = \inf(B) \). Given a grid distance \( 1/d \in \mathbb{N} \), a lower bound \( l \in \mathbb{Z} \) and an upper bound \( u \in \mathbb{Z} \), an equidistant interval is a finite rational set \( \text{eint}(d,l,u) = \{ i/d \mid i \in \mathbb{Z} : l/d \leq i/d \leq u/d \} \).

### 2.2 Graphs and Labeled Transition System

A **graph** is a pair \( G = (V, E) \), where \( V \) is the set of vertices and relation \( E \subseteq V \times V \) called edges. Edges have directions represented by a head and a tail. Note that a graph is undirected if \( E \) is a symmetric relation. In a directed graph, a sequence \( \sigma \in V^* \) of length \( n > 0 \) is called a directed path, if \( (e_{i-1}, e_i) \in E \) for all \( 1 < i \leq n \). A non-empty path is called a cycle if \( e_1 = e_n \). Note that an acyclic graph does not contain cycles. Furthermore, a directed graph is said to be **strongly connected** if for each \( e_1, e_2 \in V \), a directed path exists from \( e_1 \) to \( e_2 \).

A **bipartite graph** is a graph whose vertices have been partitioned into two sets such that no edge exists that has a head and tail in the same set. Note that every state \( s \) is the silent action \([242]\), and \( s \) (labels), names, \( L \) otherwise.

We define the preset of a vertex \( v \in V \) as \( \cdot v = \{ u \mid (u, v) \in E \} \) and the postset as \( \cdot^v = \{ u \mid (v, u) \in E \} \). As a shorthand, we use the preset (postset) as a function such that \( \cdot u(v) = 1 \) (\( u^v(v) = 1 \)), if \( v \in \cdot u \) (\( v \in u^v \)) and \( \cdot u(v) = 0 \) (\( u^v(v) = 0 \)), otherwise.

A **labeled transition system** (LTS) is a labeled graph expressed as a tuple \( L = (S, A, \rightarrow, s_0) \) where \( S \) is the set of states (vertices), \( A \) is a finite set of action names (labels), \( \rightarrow \subseteq S \times A \cup \{ \tau \} \times S \) is a transition relation (edges) where \( \tau \notin A \) is the silent action \([242]\), and \( s_0 \in S \) is an initial state. For \( s, s' \in S \) and \( a \in A \), \( s \xrightarrow{a} s' \) if and only if \( (s, a, s') \in \rightarrow \). An action \( a \in A \cup \{ \tau \} \) is called enabled in a state \( s \in S \), denoted by \( s \xrightarrow{a} \), if there is a state \( s' \) such that \( s \xrightarrow{a} s' \). If \( s \xrightarrow{a} s' \), we say that state \( s' \) is reachable from \( s \) by an action labeled \( a \). We define \( \Rightarrow \) as the smallest relation between two states \( s \in S \) and \( s' \in S \) such that \( s \Rightarrow s' \) if \( s = s' \), or \( \exists s'' \in S : s \Rightarrow s'' \xrightarrow{a} s' \). For non-silent actions \( a \in A \), we define \( \Rightarrow_a \) such that \( s \xrightarrow{a} s' \) iff there exists \( s_1 \in S : s \xrightarrow{a} s_1 \xrightarrow{a} s' \). We lift the notion of actions to sequences.
We say that a non-empty finite sequence $\sigma \in A^*$ of length $n \in \mathbb{N}$ is a firing sequence, denoted by $s_0 \xrightarrow{\sigma} s_n$, if there exist states $s_i, s_{i+1} \in S$ such that $s_i \xrightarrow{\sigma[i+1]} s_{i+1}$ for all $0 \leq i \leq n - 1$. We write $s \xrightarrow{\sigma} s'$ if there exists a sequence $\sigma \in A^*$ such that $s \xrightarrow{\sigma} s'$ and say that $s'$ is reachable from $s$. We denote the set of all reachable states from an initial state $s_0$ by $R(L, s_0) = \{s \mid s_0 \xrightarrow{\sigma} s\}$ and the set of all firing sequences from an initial state $s_0$ by $T(L, s_0) = \{\sigma \mid \exists s \in S : s_0 \xrightarrow{\sigma} s\}$.

Behavioral Equivalence. Given two transition systems $N_1 = (S_1, A_1, \rightarrow, s_0)$ and $N_2 = (S_2, A_2, \rightarrow, s'_0)$ there exist several notions to compare their behavior. In this thesis, we will focus on four behavioral equivalences, namely trace equivalence, simulation equivalence, branching bisimulation and strong bisimulation equivalences. We call $N_1$ and $N_2$ trace equivalent if they have the the same set of firing sequences from their respective initial states. A binary relation $R \subseteq S_1 \times S_2$ is a simulation relation if and only if for all $s_1 \in S_1, s_2 \in S_2, a \in A_1, (s_1, s_2) \in R$ and $s_1 \xrightarrow{a} s'_1$ implies that there exist $s'_2 \in S_2$ and $a \in A_2$ such that $s_2 \xrightarrow{a} s'_2$ and $(s'_1, s'_2) \in R$. We say $N_2$ simulates $N_1$ and write as $N_1 \preceq_R N_2$ or simply $N_1 \preceq N_2$ if a simulation relation $R$ exists. Note that this implies trace inclusion, i.e. $T(N_1, s_0) \subseteq T(N_2, s'_0)$, where $s_0$ and $s'_0$ are initial states. If $R$ and $R^{-1}$ are both simulations, relation $R$ is called a bisimulation denoted by $\simeq$. If simulation relations $R$ and $R'$ exists such that $N_2 \preceq_R N_1$ and $N_1 \preceq_R N_2$ then $N_1$ and $N_2$ are simulation equivalent. We use the notion of branching bisimulation as defined in [137] and denote it by $\simeq_b$.

2.3 Petri nets

A Petri net is a bipartite graph consisting of two types of nodes, namely places (represented by a circle) and transitions (represented by a rectangle). We give an example of a Petri net in the Fig. 2.1. The nodes labeled $P1$ and $P2$ are called places and the node labeled $A$ is called a transition. A place can be connected to a transition and vice-versa by means of directed edges called arcs. The notion of a token gives a Petri net its behavior. Tokens reside in places and often represent either a status, an activity, a resource or an object. The distribution of tokens in a Petri net is called its marking or state. Transitions represent events of a system. The occurrence of an event is defined by the notion of transition enabling and firing. A transition is enabled if all its input places have at least one token each. When an enabled transition fires it consumes one token from each input place and produces one token in each output place, i.e. changes the state. Apart from arcs between places and transitions there are two other types of arcs, namely inhibitor and reset arcs. An enabled transition having an inhibitor arc (represented as an edge having a rounded tip from transition $A$ to place $Q1$) can fire only if the place associated with the inhibitor arc does not contain consumable tokens, i.e. place
Q1. A transition connected by a reset arc (represented as a dotted edge from transition $A$ to place $Q_2$) to a place, removes all tokens residing in that place (i.e. $Q_2$) when it fires. We will now formally define a Petri net.

Figure 2.1: Petri net with inhibitor and reset arcs

**Syntax of a Petri net.** A Petri net is a bipartite graph defined as a tuple $N = (P,T,F)$, where $P$ is the set of places; $T$ is the set of transitions such that $P \cap T = \emptyset$ and $F$ is the flow relation $F \subseteq (P \times T) \cup (T \times P)$. We refer to elements from $P \cup T$ as nodes and elements from $F$ as arcs. We denote the places of net $N$ by $P_N$, transitions as $T_N$ and similarly for other elements of the tuple. If the context is clear, we omit $N$ in the subscript. Throughout this thesis, we will assume that the set of places and set of transitions of an arbitrary Petri net is both non-empty and finite, unless otherwise explicitly mentioned.

An inhibitor/reset net is a 5-tuple $(P,T,F,\iota,\rho)$ where $P$ and $T$ are two disjoint sets of places and transitions respectively, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, and $\iota,\rho : T \to \mathcal{P}(P)$ specify the inhibitor and reset arcs, respectively.

Nets can be depicted graphically. Places and transitions are represented as circles and squares, respectively, an arc $(n,m)$ is depicted as a directed arc from node $n$ to node $m$. A dot-headed arc is drawn from place $p$ to transition $t$ if $p \in \iota(t)$, if $p \in \rho(t)$, a dashed arc is drawn between $p$ and $t$.

We will extend the notion of preset and postset to nodes of a Petri net. In case there is more than one Petri net’s in the context, we will use the name of the net in the subscript. Furthermore, we extend the notion of a path in a graph to a Petri net and call the set of all paths of a Petri net $N$ as its path space denoted by $\text{PS}(N)$. In a similar way, a Petri net is strongly connected if for any two nodes there exists a directed path from one to the other.

Two Petri nets $N$ and $M$ are disjoint if $(P_N \cup T_N) \cap (P_M \cup T_M) = \emptyset$. They are isomorphic, denoted by $N \cong \psi M$ if and only if a bijective function $\psi : P_N \cup T_N \to P_M \cup T_M$ exists such that $P_M = \psi(P_N)$, $T_M = \psi(T_N)$ and
∀(x, y) ∈ F_N ⇔ (ψ(x), ψ(y)) ∈ F_M. We write \( N \cong M \) if a bijective function \( ψ \) exists such that \( N \cong_ψ M \). Given two nets \( N_1 = (P_1, T_1, F_1, i_1, ρ_1) \) and \( N_2 = (P_2, T_2, F_2, i_2, ρ_2) \) such that \( (P_1 \cup P_2) \cap (T_1 \cup T_2) = \emptyset \), their union is the net \( (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, i, ρ) \) where \( i(t) = i_1(t) \cup i_2(t) \) and \( ρ(t) = ρ_1(t) \cup ρ_2(t) \) for all \( t \in T_1 \cup T_2 \).

**Semantics of a Petri net.** A net models behavior. The state of a Petri net \( N = (P, T, F) \) is determined by its marking which represents the distribution of tokens over places of the net. A marking \( m \) of a Petri net \( N \) is a bag over its places \( P \), i.e. \( m \in B(P) \). A transition \( t \in T \) is enabled in some marking \( m \) if and only if \( t^*(p) \leq m(p) \), for all places \( p \in P \). If \( N \) is an inhibitor net then for the enabling of transition \( t \) from marking \( m \), we also require that \( m(p) = 0 \) for all places \( p \in i(t) \). An enabled transition \( t \) from marking \( m \) may fire, resulting in a new marking \( m' \) with \( m'(p) = m(p) - t(p) + t^*(p) \) if \( p \in P \setminus ρ(t) \) and \( m'(p) = 0 \) if \( p \in ρ(t) \), and is denoted by \( m \xrightarrow{t} m' \).

We lift the notion for firing and enabledness of transitions to a firing sequence, i.e. a sequence of transitions of a Petri net. A firing sequence \( σ = (t_1, t_2, \ldots, t_n) \) of length \( n \in \mathbb{N} \) is enabled in marking \( m \) if and only if \( \exists m_1, \ldots, m_n \in B(P) : m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} m_{n-1} \xrightarrow{t_n} m_n \), and we denote it as \( m \xrightarrow{σ} m' \). We write \( m \xrightarrow{σ} m' \) and call marking \( m' \) reachable from marking \( m \), if there exists a firing sequence \( σ \in T^* \) such that \( m \xrightarrow{σ} m' \). We denote the set of all reachable markings from marking \( m \) of a Petri net \( N \) by \( R(N, m) = \{ m' | m \xrightarrow{σ} m' \} \). Furthermore, the number of occurrences of the transitions belonging to the set \( A \subseteq T \) in any arbitrary firing sequence \( σ \in T^* \) is denoted by \( \text{occurs}(σ, A) = \sum_{t \in A} σ(t) \).

A Petri net \( N \) with an initial marking \( m_0 \) is called \( k \)-bounded if for all \( p \in P, m \in R(N, m_0) : m(p) \leq k \). A safe net is a 1-bounded Petri net. We call a place \( p \in P \) safe if \( \forall m \in R(N, m_0), m(p) \leq 1 \). We define the net system of a Petri net \( N \) as a 3-tuple \( M = (N, m_0, m_f) \), where \( m_0 \in B(P) \) is the initial marking and \( m_f \in B(P) \) is the final marking.

**Weak Termination.** The weak termination property for a net system \( M = (N, m_0, m_f) \) states that \( \forall m \in R(N, m_0) : m_f \in R(N, m) \), i.e. for all reachable markings from the initial marking \( m_0 \) is reachable. If a marking does not enable any transition in the net, it is called a dead marking.

**Petri net Subclasses.** A workflow net (WFN) is defined as the tuple \( N = (P, T, F, \text{init}, \text{fin}) \), where \( (P, T, F) \) is a Petri net, \( \text{init} = \{ i \in P \mid t^* = \emptyset \} \) is a singleton set, \( \text{fin} = \{ f \in P \mid t^* = \emptyset \} \) is a singleton set, and all nodes \( n \in P \cup T \) are on a path from \( i \in \text{init} \) to \( f \in \text{fin} \). The closure of a workflow net \( N \) is a net \( \text{closure}(N) = (P, T \cup \{ i \}, F \cup \{ (i, i), (f, i) \}, \text{init}, \text{fin}) \) such that \( i \notin T \) and \( t^* = \{ i \} \). A WFN \( N \) weakly terminates if its net system \( (N, [i], [f]) \)
weakly terminates. Note that in [163] this property is called $1$-Soundness. For an overview of soundness, see [4]. Note that weak termination is an AGEF-property in $CTL^*$ and cannot be expressed in LTL.

Net $N$ is a state machine ($S$-net) [110] if and only if $\forall t \in T : |\cdot t| = |\cdot t^*| = 1$. In a state machine, a place $p$ is called a split if $p^* > 1$. Likewise, it is a join if $p^* > 1$. Net $N$ is a marked graph ($T$-net) [110] if and only if $\forall p \in P : |\cdot p| = |p^*| = 1$. A workflow net that is also a state machine is called an $S$-WFN. If it is both a workflow net and a marked graph, it is called a $T$-WFN.

### 2.3.1 Refinement Techniques of Petri nets

The refinement of places in a Petri net is a well known refinement step and has been described in various contexts [163]. This operation is formally defined as follows:

Given a Petri net $N$ and a WFN $M$ such that $N$ and $M$ are disjoint, the refinement of a place $p \in P_N$ by a WFN $M$ is denoted by $N' = N \odot_p M$, where $N' = (P, T, F)$ is a Petri net defined as $P = (P_N \setminus \{p\}) \cup P_M$, $T = T_N \cup T_M$, $F = (F_N \setminus ((\cdot p \times \{p\}) \cup (\{p\} \times p^*))) \cup F_M \cup (\cdot p \times \{i_M\}) \cup (\{f_M\} \times p^*)$, i.e. the place $p$ is replaced by a WFN such that the preset and postset of the place $p$ becomes the preset and postset of the initial and final places of the WFN, respectively. In this way, an activity modeled by a place is elaborated by the Petri net (in this case by a WFN) that is replacing this place. In the Fig. 2.2, we illustrate the place refinement operation.

![Figure 2.2: Place Refinement](image)

We will now define the two refinement operations: transition refinement and loop addition. We give an example in the Fig. 2.3.

The transition refinement of a transition in a Petri net replaces the original transition by two new transitions connected by a place in between. One of the
newly added transitions, preserves the preset of the original transition and the other, preserves the postset of the original transition. The former transition has the newly added place in its postset and the latter has it in its preset. The operation is formally defined as follows: Given a Petri net $N$, the refinement of a transition $t \in T_N$ is denoted by $N' = TR(N, t)$, where $N' = (P, T, F)$ is a Petri net defined as $P = P_N \cup \{p\}$, where $p \notin P_N$, $T = (T_N \setminus \{t\}) \cup \{t', t''\}$, where $t' \notin T_N$ and $t'' \notin T_N$, $F = (F_N \setminus \{(* t \times \{t\}) \cup (\{t\} \times t^*)\}) \cup (** t \times \{t''\}) \cup (** t' \times t^*) \cup \{(t', p), (p, t'')\}$.

The addition of a loop to a place in a Petri net, adds a transition connected to this place with bi-directional arcs. The operation is formally defined as follows: Given a Petri net $N$, the addition of a loop to a place $p \in P_N$ with a transition $t$ such that $t \notin T_N$ is denoted by $N' = LA(N, t, p)$, where $N' = (P, T, F)$ is a Petri net defined as $P = P_N$, $T = T_N \cup \{t\}$, $F = F_N \cup \{(t, p), (p, t)\}$.

### 2.3.2 Colored Petri nets

A colored Petri net (CPN) is an extension of a Petri net with data and time. We will keep the description of a CPN informal because we only use it for modeling purposes in Ch. 6. Furthermore, for performance analysis of such models we drop the notion of color and then use the resulting timed net. For a more formal definition of a CPN, please refer [177].

We give an example of a counter modeled as a CPN in the Fig. 2.4. Each place has an inscription which determines the set of token colours (data values) that the tokens residing in that place are allowed to have. The set of possible token colours is specified by means of a type called the colour set of the place (for eg.
Figure 2.4: An example of a CPN

The place `counter` has a color set of type integer denoted by `INT`). A token in a place is denoted by an inscription of the form `x^y` (see token `1^5`), interpreted as `x` tokens having token color `y`, i.e. as a multiset of colored tokens. Like in a standard Petri net, when a transition fires, it removes one token from each of its input places and adds one token to each of its output places. However, the colors of tokens that are removed from input places and added to output places are determined by `arc expressions` (inscriptions next to arcs).

The arc expressions of a CPN are written in the ML programming language and are built from typed variables, constants, operators, and functions. An arc expression evaluates to a token color, which means exactly one token is removed from an input place or added to an output place. The arc expressions on input arcs of a transition together with the tokens residing in the input places determine whether the transition is `color enabled`. For a transition to be color enabled it must be possible to find a binding of the variables appearing on each input arc of the transition such that it evaluates to at least one token color present in the corresponding place. When the transition fires with a given binding, it removes from each input place a colored token to which the corresponding input arc expression evaluates. Firing a transition adds to each output place a token whose color is obtained by evaluating the expression (defined over variables of input arcs) on the corresponding output arc. In our example, the variables `d1` and `d2` are bound to the value of the token (value 5) in place `counter`. When one of the transitions, say `add` fires, it produces a token with value `d1 + 1 = 6` in the place `counter`. Furthermore, transitions are also allowed to have a `guard`, which is a Boolean expression. When a guard is present it serves as an additional constraint that must evaluate to true for the transition to be color enabled. In our example, the guard `d1 < 10` ensures transition `add` can fire if the value of the token in place `counter` is less than 10.

In addition, tokens also have a timestamp that specifies the earliest possible consumption time, i.e. tokens in a place are available or unavailable. The CPN
model has a *global clock* representing the current model time. The distribution of tokens over places together with their time stamps is called a *timed marking*. The execution of a timed CPN model is determined by the timed marking and controlled by the global clock, starting with an initial value of zero. In a timed CPN model, a transition is *enabled* if it is both color enabled and the tokens that determine this enabling are available for consumption. The time at which a transition becomes enabled is called its *enabling time*. The *firing time* of a timed marking is the earliest enabling time of all color enabled transitions. When the global clock is equal to the firing time of a timed marking, one of the transitions with an enabling time equal to the firing time of the timed marking is chosen nondeterministically for firing. The global clock is not advanced as long as transitions can fire (eager semantics). When there is no longer an enabled transition at the current model time, the clock is advanced to the earliest model time at which the next transition is enabled (if no such transition then it is a deadlock), i.e. the firing time of the current timed marking. The time stamp of tokens are written after the @ symbol in the following form $x'y@z$, interpreted as $x$ tokens having token color $y$ carry a timestamp $z$. When an enabled transition fires, the produced tokens are assigned a color and a timestamp by evaluating the time delay interval, inscribed on outgoing arcs of a transition (see time interval $@[a, b]$). The time stamp given to each newly produced token along an output arc is the sum of the value of the current global clock and a value chosen from the delay interval associated with that arc. Note that the firing of a transition is instantaneous, i.e., takes no time.
In this chapter, we will use the formalism of Petri nets to develop a component-based architectural framework for modeling software of planning and control systems like robots. Here we focus on the control flow model of the system, i.e. only the right ordering of events without considering data. The framework is built around the concept of components and their interactions, which we call interaction patterns. In the software of planning and planning systems, we encounter recurring patterns of interactions. We identify and model the four recurring interaction patterns, namely the remote procedure call (RPC) pattern, the publish-subscribe pattern, ...
(PS) pattern, goal-feedback-result (GFR) pattern and mirrored port (MP) pattern, and prove that they are all weakly terminating by their structure. The first two also occur in other types of distributed systems, whereas the third is typical for control systems of robots. The fourth is a class of weakly terminating interaction patterns. The framework also prescribes a construction method to derive a weakly terminating Petri net model from a set of components and interaction patterns. The construction method is a combination of bottom-up and top-down stepwise refinement strategies. This way of designing systems allows a designer to focus on the functional aspects of the design, while being assured that the system derived so far is guaranteed to weakly terminate.

3.1 Introduction

The software system underlying a service robot is a complex distributed system. The complexity arises from the need to interface with heterogeneous hardware while executing concurrently a large variety of computations such as image processing, sensor data processing, planning and control tasks. The distributed nature and growing complexity of these systems warrant the need for a component based development approach. Component-based development frameworks are attractive for programming software systems of robots because they augment the advantages of the traditional programming frameworks by facilitating the loose coupling between components (asynchronous communication), promoting reuse and allowing for dynamic reconfiguration of components at run-time. Over the years a number of component based programming frameworks like OROCOS [80], openRTM [21] and Player/Stage [134] have been developed. A few of them have enjoyed great success as demonstrated by the popular Robot Operating System (ROS) [277]. ROS is a widely used component based development framework for robot systems that runs on top of the Linux operating system. To a large extent, ROS has been able to address the shortcomings of its predecessors. For this reason, it saw an unprecedented growth in popularity over the past few years as a common platform to share knowledge. To our knowledge, none of the popular programming frameworks provide an integrated formal technique to verify correctness properties of systems, like deadlock freedom, in a structured way.

The central idea behind the design of component based systems involves the construction of complex systems by integrating components. As a consequence, component based systems exhibit a high degree of concurrency. This makes their analysis a hard task. Therefore, it becomes essential to model the behavior of such systems and guarantee certain behavioral properties during design time. In the traditional (formal) design of software systems, design decisions are verified post-design by model checking techniques. Over the past years, many formal component-based frameworks relying on such an approach have been proposed, like BIP/GenoM [29], Cadena [160], SaveCCM [85] etc. Such frameworks provide a model to specify the components and their composition while relying on state
space based explorations to verify the correctness of the design. State space based explorations are generally time consuming and do not scale well to the complexities of real world models. Therefore there is a need for a design method that guarantees certain correctness properties (i.e. behavioral properties like freedom of deadlock, etc.) by construction, i.e. the designer must be able to design architectural models of the system from a given specification such that the design method guarantees certain correctness properties. We focus on one particular correctness property: weak termination, which states that in each reachable state, the system has the option to eventually terminate, which implies the freedom of both deadlock and livelock in the system.

In [141, 187], the authors focus on constructing deadlock free systems using labeled transition systems, i.e., each component is a state machine, which after composition guarantee deadlock freedom. In the context of Petri nets, the generalized soundness property [163] is a generalization of the weak termination property for workflow nets. For a class of workflow nets satisfying the state machine property [163], the generalized soundness property is implied by its structure, making them an ideal choice for modeling the control flow of components. In [5] a sufficient condition is presented to pairwise verify weak termination for tree structured compositions of components. For a subclass of compositions of pairs of components, called ATIS-nets (i.e. communicating workflow nets that support the modeling of both concurrency and choice), this condition is implied by their structure [164]. ATIS-nets are constructed from pairs of acyclic marked graphs and isomorphic state machines, and the simultaneous refinement of pairs of places [357]. However, the strict structural constraints on communication conditions enforced by these modeling frameworks, make them unsuitable for modeling commonly occurring communication patterns in robotics.

In this chapter, we develop a component-based architectural framework using the formalism of Petri nets. The framework provides a natural way of modeling a component and its interactions as an interaction pattern. Interaction patterns are modeled as a special kind of multi-workflows ([165]). These multi-workflows have so-called client and server sides, where the clients are open workflows (modeling invocation procedures) and the servers are open components (modeling cyclic communicating state machines). We identify and define four recurring interaction patterns, namely remote procedure call (RPC), publish subscribe (PS), goal feedback result (GFR) and mirrored port (MP). We show that all these patterns are weakly terminating and present a design strategy for constructing arbitrary MP patterns. To represent a system design in a concise way, a graph based architectural diagram notation is presented. For the weak termination property, we require that this graph be acyclic. To derive a Petri net model of the system from an architectural diagram, we define a construction method and prove that it guarantees the weak termination property for the resulting system. The construction method is incremental and relies on a refinement based strategy to introduce interaction patterns into the system.

It is not necessary to fully accord the rules of the construction method, as
long as the final system can be derived according to these rules, the system is ensured to be correct. In fact, our approach is a structured programming technique for distributed programs, comparable to structural programming methods for sequential programs of the late sixties [113].

This chapter is structured as follows. In Section 3.2, we present an overview of the four interaction patterns, which are modeled using Petri nets in Section 3.3. Here we prove the weak termination property for the four interaction patterns and prescribe a design strategy for arbitrary MP patterns. In the Section 3.4, we present the construction method and prove that it preserves the weak termination property. In the Section 3.5, we conclude this chapter.

### 3.2 Overview of Architectural Framework

The architectural framework presented in this chapter is based on the two concepts: components and interaction patterns.

The behavior of a component is defined by the ordering of its events (often referred to as orchestration), modeled as a strongly connected communicating state machine having an idle state. As a consequence, a component does not exhibit concurrency and has a cyclic execution path. If a component provides a unique service to other components then it is called a server component, otherwise it is a root component.

The components of this framework communicate with each other by message-passing. An interaction pattern models a recurring pattern of interaction between components. Such a pattern consists of a set of identical clients and a set of identical server components. The client of an interaction pattern models a procedure as a workflow net that defines all possible sequences of messages that can be exchanged with its server components. Due to the cyclic nature of a component, a client can be handled each time the component returns to its idle state.

A component is allowed to communicate with other server components if it has the corresponding client embedded within its structure. A client of an interaction pattern is embedded inside a component using stepwise refinement techniques that preserve the weak termination property. So a component, in addition to orchestrating its internal events, also defines the order of invocations of its embedded clients, often referred to as its choreography.

The architectural framework supports the modeling of the four classes of interaction patterns: RPC, PS, GFR and MP patterns. The first two also occur in other types of distributed systems, whereas the third is typical for control systems of robots. The fourth is a class of weakly terminating interaction patterns, where clients are mirrored copies of server state machines with constraints on their structure. By mirrored copies, we mean structurally identical copies but with inverted send and receive events. This is easily captured as a design strategy for arbitrary MP patterns. An MP pattern is suitable for modeling long running service negotiations. For example, the negotiation between a planner component
that is modeling the opening of a door and a user interface component. The user interface component facilitate the decision making process of the planner by providing inputs from a human operator.

In the remainder of this section, we will introduce the four interaction patterns.

**Publish-Subscribe Pattern**

The PS pattern is an unidirectional notification pattern supporting many to many message broadcasting, i.e., multiple clients are allowed to broadcast messages to multiple servers. Messages are published by the client on some topic. A *topic* describes a message stream by specifying its message type and name. The pattern is well suited for the broadcast of sensor data streams.

**Remote Procedure Call Pattern**

The RPC pattern comprises of a single server and multiple clients. The server models a set of procedures that can be executed one at a time. Each procedure can have one or more associated clients. A procedure waits for a request message from the client, and terminates after sending back a result message. The client is blocked until the server returns a result message. No other messages are exchanged between the server and the client.

**Goal-Feedback-Result Pattern**

The GFR pattern is a more elaborate interaction pattern. The GFR pattern comprises a single server handling many clients. The GFR pattern is well suited for long running procedures and gives the client the option to preempt the server. The arrival of a goal message at the GFR server triggers a process (generally a control algorithm) whose progress on that goal may be reported via feedback messages to the waiting client. At any moment in time, an active client waiting for a message from the server is allowed to cancel the process at the server by sending a *cancel message*. The process at the server always terminates after sending a *result message* to the client indicating the result (i.e., finished, aborted, or canceled).

As in case of the RPC pattern, the GFR client and server negotiate the connection over which messages are exchanged. Furthermore, the server enforces the *single pending goal* policy. This means that at most one pending goal is allowed. So new goals preempt the current pending goal and the client is notified.

**The MP Pattern**

The MP pattern defines a class of weakly terminating single server and multiple client interaction patterns. An MP pattern is modeled as copies of communicating state machines with mirrored directions of communication for each corresponding...
event in a client and server (i.e. send-receive and vice versa) and whose structures satisfy certain structural properties that enforce communication conditions in every path between a split and join occurring in each state machine. The structural constraint guarantees that every choice is communicated between clients and server and every path between a split and a join always requires the participation of both a server and one of the clients. As with other interaction patterns, the communication is initiated by one of the clients and the server handles requests from each client in a sequential manner.

In the next section, we will model the four interaction patterns: RPC, GFR, PS and MP patterns as a Petri net and show that each of these patterns are weakly terminating. For the class of MP patterns, we prescribe a design method to derive arbitrary MP patterns in a structured way.

3.3 Logical Model of the Architectural Framework

The architectural framework we introduce in this section is built upon Petri nets with inhibitor arcs and reset arcs, called inhibitor/reset nets. In the remainder of this chapter, we will refer to an inhibitor/reset net as a net.

3.3.1 Components and Their Interaction

Components in our architectural framework communicate asynchronously by message exchange. For this, we introduce the notion of open Petri nets (OPN) which are a subclass of classical Petri nets. OPN are ideal to model asynchronous communicating systems. This is because they have a distinguished set of interface places partitioned into input and output places that represent the interfaces of the net. A direct consequence of the interaction of an OPN with its environment results in tokens being exchanged between these places. If a transition of an OPN is associated to an interface place then it is either an input place or an output place. This structural restriction disallows a transition from consuming a token from an input place and producing a token in an output place. In this way, a distinction between a send and receive event is enforced. The net obtained by discarding the interface places of an OPN is called its skeleton. The skeleton of an OPN defines its internal behavior. By adding structural constraints to an OPN, its subclasses are derived. In the Fig. 3.1, we give an example of an OPN having three output places o1, o2, o3 and one input place o4.

**Definition 3.3.1 (Open Petri net).** An open Petri net (OPN) is defined as \( N = (P, I, O, T, F, \text{init}, \text{fin}, \iota, \rho) \), where

- \((P \cup I \cup O, T, F)\) is a Petri net, \( P, I, O, T \) are pairwise disjoint, \( I \cup O \) is the set of interface places,
Figure 3.1: An Example of an OPN: S-OWN

- $I$ is the set of input places such that $\forall i \in I : i^* = \emptyset$, $O$ is the set of output places such that $\forall o \in O : o^* = \emptyset$,
- $\forall t \in T : t \cap I \neq \emptyset \Rightarrow t^* \cap O = \emptyset \land t^* \cap O \neq \emptyset \Rightarrow t \cap I = \emptyset$, i.e. a transition connected to an input place has no output place in its postset and vice versa,
- $\text{init} = \{ i \in P \mid i^* = \emptyset \}$ is the set of initial places, $\text{fin} = \{ f \in P \mid f^* = \emptyset \}$ is the set of final places, $\iota$ and $\rho$ represent sets of inhibitor and reset arcs, respectively.

We call the net $S(N) = (P, T, F', \text{init}, \text{fin}, \iota, \rho)$ the skeleton of $N$ with $F' = F \cap ((P \times T) \cup (T \times P))$. If $S(N)$ is a WFN then $N$ is called a open workflow net (OWN). If $S(N)$ is a S-WFN then $N$ is called a state machine open workflow net (S-OWN). We denote the closure of an OWN $N$ with initial place $i \in \text{init}$ and final place $f \in \text{fin}$ as closure$(N) = (P, T \cup \{ i \}, F \cup \{ (i, i), (f, i) \}, I, O, \text{init}, \text{fin}, \iota, \rho)$ such that $i \notin T$ and $i^* = f^* = \{ i \}$.

Two OPNs are composable if and only if the only nodes they share are interface places. The composition of a set of pairwise composable OPNs is almost a pairwise union of their corresponding tuples except that the shared interface places between OPNs become the internal places of the composition. In the Fig. 3.2, we give an example. Note that the places $o1$ and $o2$ become internal places of the composition.

**Definition 3.3.2 (Composition of OPN).** Two OPNs $N$ and $M$ are composable if and only if $(P_N \cup I_N \cup O_N \cup T_N) \cap (P_M \cup I_M \cup O_M \cup T_M) \subseteq ((I_N \cap O_M) \cup (O_N \cap I_M))$. The composition of a set $S$ of pairwise composable OPNs is denoted by compose$(S) = (P, I, O, T, F, \text{init}, \text{fin}, \iota, \rho)$, where

- $P = \bigcup_{X \in S} P_X \cup (\bigcup_{X \in S} I_X \cap \bigcup_{X \in S} O_X)$,
Figure 3.2: Composition of OPNs

\[ I = \bigcup_{X \in S} I_X \setminus \bigcup_{X \in S} O_X ; \quad O = \bigcup_{X \in S} O_X \setminus \bigcup_{X \in S} I_X ; \]

\[ T = \bigcup_{X \in S} T_X, \quad F = \bigcup_{X \in S} F_X, \quad \text{init} = \bigcup_{X \in S} \text{init}_X, \quad \text{fin} = \bigcup_{X \in S} \text{fin}_X, \]

\[ \forall t \in T : \iota(t) = \bigcup_{X \in S} \iota_X(t) \land \rho(t) = \bigcup_{X \in S} \rho_X(t). \]

Note that since the shared places become internal places of a composition of OPNs, \( \text{compose}(S_1 \cup S_2) \neq \text{compose}(S_1 \cup \text{compose}(S_2)) \), where \( S_1 \) and \( S_2 \) are sets of pairwise composable OPNs.

The transitions of an OPN are distinguished into three categories depending on the direction of communication, namely send, receive and internal. A send transition contains an output place in its postset and represents the sending of a message. A receive transition has an input place in its preset and represents the receipt of a message. A transition that does not send or receive is called an internal transition and models an internal event.

**Definition 3.3.3 (Direction of communication).** The direction of communication of a transition with respect to a place in an OPN \( N \) is a function \( \lambda : T \rightarrow \{ \text{send}, \text{receive}, \tau \} \) defined as \( \lambda(t) = \text{send} \Leftrightarrow t \cap O \neq \emptyset \); \( \lambda(t) = \text{receive} \Leftrightarrow t \cap I \neq \emptyset \) and \( \lambda(t) = \tau \), otherwise, for all \( t \in T \). We call a transition \( t \in T \) a communicating transition if and only if \( \lambda(t) \neq \tau \).

A component models a process as a strongly connected state machine net. An idle place in this net indicates cyclic execution sequences. We give an example of a component in the Fig. 3.3, having three transitions \( t_1, t_2 \) and \( t_3 \) of type send, one transition \( t_4 \) of type receive and one internal transition closure expressing cyclic behavior.
Definition 3.3.4 (Component). A component is an OPN $C$ whose skeleton is a strongly connected state machine net (see Chapter 2) with a special place called idle, denoted by $v_C$. The initial marking of a component has one token in the idle place. If component $C$ is the closure of an S-OWN then its initial place is the idle place $v_C$.

In component-based systems, components communicate asynchronously by message passing in order to achieve some goal. The communication between components is defined by an interaction pattern. It is a parameterized net that can be embedded into existing components of a system. An interaction pattern has two sets, a set of identical clients and a set of identical servers. A client is an S-OWN, whereas a server is the closure of an S-OWN, i.e. is a component.

The net of an interaction pattern is the composition of clients and servers satisfying (a) for any pair of client and server, the set of input places of the client are equal to the set of output places of the server and vice-versa, (b) for any pair of client and server transitions, connected to the same interface place, have opposite directions.

Note than an interaction pattern is in fact a special multi workflow net [165], which are a generalization of classical workflow nets, having multiple initial/final pairs, whereas a single workflow net has only a single initial/final pair. We give two examples of interaction patterns in the Fig. 3.4.

Definition 3.3.5 (Interaction pattern). An interaction pattern is specified by the pair $(C, S)$, where $C$ is a set of S-OWNs called the clients, and $S$ is a set of closures of S-OWNs (components) called the servers. The net of an interaction pattern is defined as $N(C, S) = \text{compose}\{C \cup S\}$ and satisfies the following properties:
Figure 3.4: An Example of an Interaction Pattern

- \( \forall c \in C, s \in S : I_c = O_s \land O_c = I_s \), i.e. for any client and server pairs, input places of a client is equal to the output place of a server and vice-versa,

- \( \forall c \in C, s \in S : \forall x \in (I_c \cup O_c) \cap (I_s \cup O_s) : (x \in T_c \setminus T_s \land x^* \in T_s \setminus T_c) \lor (x^* \in T_s \setminus T_c \land x \in T_c \setminus T_s) \), i.e. for any client and server transition pairs that are connected to the same interface place, have opposite directions

We define two operations to embed an interaction pattern into existing components of a system, namely (a) simultaneous refinement with an interaction pattern, and (b) insertion of an interaction pattern.

**Simultaneous refinement** [165] is a refinement operation that simultaneously refines a set of places of a system with clients of an interaction pattern. As a consequence of this refinement, a new server component is added to the system (this is the server of the interaction pattern). This is a bottom-up strategy for constructing a system. In the Fig. 3.5, we give an example of simultaneous refinement. Note that simultaneous refinement is a natural extension of the refinement of a single place.

**Definition 3.3.6 (Simultaneous refinement with an Interaction Pattern).** Let \( N \) be an OPN and \( R \subseteq P_N \) be a set of places to be refined such that no \( r \in R \)
Figure 3.5: Simultaneous Refinement of an OPN with an Interaction Pattern
Figure 3.6: Insertion of an Interaction Pattern into an OPN
exists with \( r \in \iota(t) \) or \( r \in \rho(t) \) for all \( t \in T_N \). Let \((\mathcal{C}, S)\) be an interaction pattern with the net \( M = N(\mathcal{C}, S) \) such that \( N \) and \( M \) are disjoint. Let \( \alpha : R \to \init_M \times \fin_M \) be a total bijection. The refinement of places \( R \) in net \( N \) with an interaction pattern \( M \) is denoted by \( N \circ_\alpha M \), is a net \((P, T, F, I, O, \init, \fin, \iota, \rho)\) defined as

\[
P = (P_N \setminus R) \cup P_M ,
T = T_N \cup T_M ,
F = (F_N \setminus \bigcup_{q \in R} ((\alpha q) \times \{q\}) \cup ((\{q\} \times q_N^*)) \cup F_M \cup
\bigcup_{q \in R} ((\alpha q) \times \{\pi_1(\alpha(q))\}) \cup \{\pi_2(\alpha(q)) \times q_N^*)\}
I = I_N \cup I_M , O = O_N \cup O_M , \init = \init_M \cup \init_N , \fin = \fin_M \cup \fin_N ,
\forall t \in T : \iota_N(t) = \iota_M(t) \land \rho_N(t) = \rho_M(t)
\]

**Insertion** of an interaction pattern is a combination of place refinement and place fusion strategies to insert an interaction pattern into existing components of the system. Unlike simultaneous refinement, this operation does not add new components to the system.

Given a set of components, we can insert an interaction pattern. For a client we select a place within a component that will be refined by the client S-OWN, a server is inserted into a component by fusing the idle places of the original component and of the server component. Further, we disallow that a server and a client are inserted in the same component. In the Fig. 3.6, we give an example of an insertion.

**Definition 3.3.7 (Insertion of an Interaction Pattern)**. Let \((\mathcal{C}, S)\) be an interaction pattern, and let \( O \) be a set of pairwise composable components such that \( \mathcal{C} \) and \( O \), and \( \mathcal{C} \) and \( \mathcal{O} \) are pairwise disjoint. Let \( \alpha : \mathcal{C} \to \bigcup_{X \in \mathcal{O}} P_X \) be an injective function defining which place is refined by which client, and \( \beta : S \to \mathcal{O} \) be an injective function defining which server is added to which component, such that no client and server are inserted in the same component, i.e., \( \alpha(C) \notin \beta(S) \) for all \( C \in \mathcal{C} \) and \( S \in \mathcal{S} \). The **insertion** of an interaction pattern \( N(\mathcal{C}, S) \) into \( \text{compose}(O) \), denoted by \( \mathcal{O} \upharpoonright_{\alpha, \beta} (\mathcal{C}, S) \), is defined by:

\[
\mathcal{O} \upharpoonright_{\alpha, \beta} (\mathcal{C}, S) = \text{compose}(\bigcup_{X \in \mathcal{O}} \{X \in \mathcal{C} \mid (\exists S \in \mathcal{S} : \beta(S) = X) \lor (\exists C \in \mathcal{C} : \alpha(C) \in P_X)\})
\cup \{X \circ_{(\alpha(C) \to (\iota(C), f_C))} C \mid X \in \mathcal{O}, C \in \mathcal{C}, \alpha(C) \in P_X\}
\cup \{X \circ_{[v_X \to v]} S_{[v_S \to v]} \mid X \in \mathcal{O}, S \in \mathcal{S}, \beta(S) = X\}
\]

where \( N_{[p \to r]} \) denotes the renaming of place \( p \) into a new place \( r \notin P_N \).

Next, we will formalize the four interaction patterns and show that each pattern weakly terminates.
3.3.2 The GFR Pattern

The GFR pattern allows multiple clients to communicate with a single server (single server interaction pattern). A client triggers the server by sending a goal message. It accepts this goal, and it becomes the pending goal of the server. In case the server already has a pending goal, this goal is canceled, and a reject message is sent to the corresponding client.

On rejection, the corresponding client terminates. When the server accepts a pending goal, the server starts processing it. Depending on the goal, the server may send feedback messages. Finally, the server will send its result and returns to the initial state to process a new pending goal. When the client has received the accept message for a goal, it needs to process the feedback sent by the server before it may accept the result message of the server. The client is also allowed to cancel the goal at any moment after it has been activated. In case a client sends a cancel message to the server, the server will report that it has canceled by sending a result message. Fig. 3.7 presents the logical model of the GFR pattern.

The server of a GFR pattern executes a procedure in a loop. This procedure is modeled by the busy place in the server labeled $x$. Such a procedure can be further elaborated by first modeling its control flow as an S-WFN and then refining the place with it (i.e. place refinement). This is a top-down strategy for elaborating the behavior of an existing system. Note that the busy place may also be refined by clients of other interaction patterns.

**Definition 3.3.8 (GFR pattern).** We will use the notations depicted in Fig. 3.7
to define the GFR pattern consisting of a single server M and clients C = \{C_1, \ldots, C_n\}. The GFR pattern is an interaction pattern denoted by the pair \((C, \{M\})\) with an idle place \(v\).

We define the system \(\text{GFR}(C, \{M\}) = (N, M_0, M_f)\) with \(N = N(C, \{M\})\), \(M_0\) is defined by \(m \in M_0\) iff \(m(v) = 1, m(i_j) \leq 1\) for each client \(C_j\) with \(1 \leq j \leq n\) and all other places are empty, the set of all final markings \(M_f\) is defined by \(m \in M_f\) iff \(m(v) = 1, m(i_j) + m(f_j) = m_0(i_j)\) for each client \(C_j\) with \(1 \leq j \leq n\) and initial marking \(m_0 \in M_0\), and all other places are empty.

Based on the properties of the net, we show that for any number of clients, the GFR pattern is weakly terminating.

**Theorem 3.3.1 (Weak termination of the GFR pattern).** Let \((C, \{M\})\) be the GFR pattern with clients \(C = \{C_1, \ldots, C_n\}\) as defined in Def. 3.3.8. Then, the system \(\text{GFR}(C, \{M\})\) is weakly terminating.

**Proof.** (sketch) Let \(\text{GFR}(C, \{M\}) = (N, M_0, M_f)\) and let \(m_0 \in M_0\). From the structure of the GFR pattern it is easy to verify that the following place invariants hold, i.e., for any marking \(m\) reachable from \((N, m_0)\), we have

1. The single goal policy: \(m(v) + m(u) \leq 1\);
2. Only one goal is active: \(m(u) + m(v) + m(w) + m(x) + m(y) + m(z) = 1\);
3. A single result is returned: \(m(w) + m(x) + m(y) + m(a_6) + m(a_7) \leq 1\);
4. The skeleton of the clients is safe: \(\forall 1 \leq j \leq n : m(i_j) + m(b_j) + m(c_j) + m(d_j) + m(f_j) = m_0(i_j)\);
5. Each goal is accepted or rejected: \(\sum_{j=1}^{n} m(b_j) = m(a_1) + m(a_2) + m(a_3) + m(u)\);
6. A single instance is handled by the server, and the server returns to the idle state: \(m(v) + m(u) + m(a_3) + m(a_7) + \sum_{j=1}^{n} (m(c_j) + m(d_j)) = 1\);
7. For each goal there exists at most one cancel message: \(m(a_4) \leq m(a_7) + \sum_{j=1}^{n} m(d_j) \leq 1\)

Based on these invariants we conclude:

- Only one client can be active at the same time.
- All places except \(a_1, a_2\) and \(a_5\) are safe.
- If place \(a_3\) is marked then \(j \in \{1 \ldots n\}\) exists such that place \(b_j\) is not empty.

Let \(C_j\) be this active client. Then either place \(c_j\) or place \(d_j\) is marked with a single token. Places \(a_4\) and \(a_5\) do not influence the desired behavior of \(N\), as all
tokens in places $a_4$ and $a_5$ are respectively removed by transition $go$-$idle$, and by transitions $receive$-$feedback_j$ or $clean_j$.

To analyze the behavior note that subnet $N_1 = \{P_1, T_1, F_1\}$ defined by $P_1 = \{u, v, w, x, y, z, a_3, a_7\} \cup \bigcup_{j \in \{1, \ldots, n\}} \{c_j, d_j\}$, $T_1 = P_1 \cup P_1^\ast$ and $F_1 = F \cap ((P_1 \times T_1) \cup (T_1 \times P_1))$ is initially marked with a single token in place $v$. Subnet $N_1$ corresponds to the goal handling of server $M$. From the invariants it follows that net $N_1$ is a strongly connected state machine. Thus, it is always possible to reach the idle place $v$. Note that place $a_4$ only influences the order in which transitions $send$-$result$ on the one hand and $receive$-$result_j$ or $clean_j$ on the other hand fire.

Next, observe that the server autonomously may fire transition $send$-$result$ and that client $C_j$ can either fire transition $clean_j$ or transition $receive$-$result_j$. After either one of these two transitions, transition $go$-$idle$ fires, returning the subnet to its initial marking. Remark that in each non-empty cycle from and to $v$ in $N_1$, transition $go$-$active$ fires only once. Since only either one of the transitions $clean_j$ and $receive$-$result_j$ fires once, only a single token is added to place $f_j$ in this cycle.

As each initially marked client $C_j$ always receives either a reject or accept message after firing its transition $send$-$goal_j$, and by invariant 4, each client ends with a single token either in place $f_j$ or $i_j$ if and only if initially place $i_j$ was marked with a single token.

In any reachable marking for a GFR pattern, a client is either waiting to be executed, under execution, or already finished. The proof of the theorem above shows that clients that are not under execution can be added or removed from the pattern.

**Corollary 3.3.2 (Dynamic reconfiguration of the GFR pattern).** Given a GFR pattern $(C, \{M\})$ as defined in Def. 3.3.8 with $C = \{C_1, \ldots, C_n\}$, let $m$ be a reachable marking of GFR$(C, \{M\})$. Let $X, Y, Z \subseteq C$ be a partitioning of $C$ such that $C_j \in X$ iff $m(i_j) = 1$, $C_j \in Y$ iff either $m(f_j) = 1$ or the net $C_j$ is unmarked, and $C_j \in Z$ otherwise. Then GFR$(C', \{M\})$ is weakly terminating for any set of clients $C'$ with $Z \subseteq C' \subseteq C$.

### 3.3.3 The RPC Pattern

The RPC pattern consists of a single server and multiple clients (single server interaction pattern). The server of an RPC pattern comprises of a set of procedures and internal behavior. Each procedure is started on the request of a client, and will in due time send a response. A server can only execute one procedure at a time. Each client calls a single procedure and multiple clients may invoke the same procedure. Fig. 3.8 presents the logical model of an RPC pattern.

Similar to the GFR pattern, the busy places labeled $e$ and $c_i$, $i \in \{1\ldots k\}$ belonging to the server of a RPC pattern can be refined by either an S-WFN or clients of other interaction patterns. The place $e$ models the internal behavior of the server and each place $c_i$ models the internal behavior of their respective procedure.
Definition 3.3.9 (RPC pattern). We use the notations of Fig. 3.8 to define the RPC pattern as an interaction pattern consisting of the pair $(C, \{S\})$ with an idle place $v$, where $S = W \cup \bigcup_{Q \in R} Q$ denotes the server and $C = \{C_1, \ldots, C_n\}$ the set of clients, with $R = \{R_1, \ldots, R_k\}$ a non-empty set of $k$ procedures and $r : C \to \{1 \ldots k\}$ a total surjective function denoting the procedure each client invokes.

For the RPC pattern, we define the system $\text{RPC}(C, S) = (N, M_0, M_f)$ with $N = N(C, \{S\})$, $M_0$ is defined by $m \in M_0$ iff $m(v) = 1$, $m(i_j) \leq 1$ for each client $C_j$ with $1 \leq j \leq n$ and all other places are empty, and the set of all final markings $M_f$ is defined by $m \in M_f$ iff $m(v) = 1$, $m(i_j) + m(f_j) = m_0(i_j)$ for each client $C_j$ with $1 \leq j \leq n$ and initial marking $m_0 \in M_0$, and all other places are empty.

Next, we show that the RPC pattern is weakly terminating.

Theorem 3.3.3 (Weak termination of the RPC pattern). Let $(C, \{S\})$ be an RPC pattern as defined in Def. 3.3.9. Then $\text{RPC}(C, \{S\})$ is weakly terminating.

Proof. (sketch) Let $C = \{C_1, \ldots, C_n\}$, i.e., the RPC pattern consists of $n$ clients, let $R = \{R_1, \ldots, R_k\}$ be the $k$ procedures handled by server $S$, and $r : C \to \{1, \ldots, k\}$ be the total surjective function defining for each client which procedure is taken. Let $\text{RPC}(C, S) = (N, M_0, M_f)$ and let $m_0 \in M_0$.

From the structure of the RPC pattern it is easy to verify the following place invariants, i.e., for any marking $m$ reachable from $(N, m_0)$, we have:

1. The skeleton of the server is safe: $m(v) + m(q) + m(e) + \sum_{j=1}^{k} m(c_j) = 1$
2. Each procedure $R_i$, $i \in \{1, ..., k\}$ is called at most $|r^{-1}(i)|$ times, i.e., $|r^{-1}(i)|$
clients call the procedure $R_i$: $\sum_{C_j \in r^{-1}(i)} m(d_j) = m(a_i) + m(b_i) + m(c_i)$

Based on these invariants, and the observation that server $S$ is a component with
idle place $v$, it follows that a client that requests a procedure is always able to
receive a response. Hence, the system weakly terminates.

A logical consequence of the definition of the RPC pattern is that clients can
be attached and detached at runtime, as shown in the next corollary.

**Corollary 3.3.4 (Dynamic reconfiguration of the RPC pattern).** Given
a RPC pattern $(C, \{S\})$ as defined in Def. 3.3.9 with $C = \{C_1, \ldots, C_n\}$, let $m$ be
a reachable marking of RPC$(C, \{S\})$. Let $X, Y, Z \subseteq C$ be a partitioning of $C$ such
that $C_j \in X$ iff $m(i_j) = 1$, $C_j \in Y$ iff either $m(f_j) = 1$ or the net $C_j$ is unmarked,
and $C_j \in Z$ otherwise. Then RPC$(C', \{M\})$ is weakly terminating for any set of
clients $C'$ with $Z \subseteq C' \subseteq C$.

### 3.3.4 The Publish-Subscribe Pattern

The PS pattern consists of multiple servers and multiple clients (multiple server
interaction pattern). The client and servers of a PS pattern communicate over one
topic. A **topic** is modeled as a set of interface places. A **server** is a component
with one transition connected to an input place. A **client** is a OWN with one transition,
fiiring which produces copies of the same message in the input place of
each server. Fig. 3.9 presents the **logical model** of the publish-subscribe pattern.
Note that places of a PS pattern are not refifiable.

**Definition 3.3.10 (Publish-subscribe pattern).** We use the notations of
Fig. 3.9 to define the **PS pattern** as an interaction pattern consisting of the pair
$(C, S)$, where $C = \{C_1, \ldots, C_n\}$ denotes the set of $n$ clients and $S = \{S_1, \ldots, S_m\}$
denotes the set of $m$ servers. The **idle** place of each server $S_i$, $1 \leq i \leq m$ is labeled
as $v_i$.

For the publish-subscribe pattern we define the system $PS(C, S) = (N, M_0, M_f)$
with $N = N(C, S)$, $M_0$ is defined by $m \in M_0$ iff $m(v_j) = 1$ for all $1 \leq j \leq m$,
$m(i_j) \leq 1$ for all $1 \leq j \leq n$ and all other places are empty, and the set of
final markings $M_f$ is defined by $m \in M_f$ iff $m(v_j) = 1$ for all $1 \leq j \leq m$,
$m(i_j) + m(f_j) = m_0(i_j)$ for each client $C_j$ with $1 \leq j \leq n$ and initial marking
$m_0 \in M_0$, and all other places are empty.

Note that if an instance of a PS pattern has no servers, then the output places
of each client yields the empty set. On the other hand, if a PS pattern has no
clients, then all servers are dead, i.e. no transition in any client can fire.

Although the PS pattern does not satisfy the proper completion property [163],
it is easy to verify that any publish-subscribe pattern weakly terminates.

**Theorem 3.3.5 (Weak termination of the PS pattern).** Let $PS(C, S)$ be a
PS pattern as defined in Def. 3.3.10. Then $PS(C, S)$ is weakly terminating.
Proof. Let $m$ be a reachable marking of $PS(C, S)$. As for each client $C \in C$ either place $i$ or place $f$ is marked in $m$, we only need to consider the interface place $q \in I_S$ for each server $S \in S$. By the structure of $S$, we can fire transition $t$, $m(q)$ times, after which place $q$ is empty. This is possible for each server. Hence, $PS(C, S)$ is weakly terminating.

Like for the first two interaction patterns, the PS pattern can be changed at run-time, by adding or removing clients in their initial marking. In addition, the server of a PS pattern can also be added or removed if its topic place is empty.

**Corollary 3.3.6** (Dynamic reconfiguration of publish-subscribe pattern). Given a PS pattern $(C, S)$ as defined in Def. 3.3.10, let $m$ be a reachable marking of $PS(C, S)$. Let $X, Y \subseteq S$ be a partitioning of $S$ such that $S_j \in X$ iff $m(q_j) > 0$ and $S_j \in Y$, otherwise. Then $PS(C', S')$ is weakly terminating for any set of clients and servers with $C' \subseteq C$ and $X \subseteq S' \subseteq S$.

### 3.3.5 The Mirrored Port Pattern

The MP pattern consists of multiple clients and a single server (single server interaction pattern). The model of a client and server of a MP pattern is based on the notion of a port.

A port is an S-OWN with constraints on its structure. In a port, each interface place is connected to exactly one transition, and each transition is connected to exactly one interface place. This restriction disallows a transition from consuming tokens from more than one interface place. In this way, we enforce for each
interface place there exists exactly one transition that can either consume tokens from it or produce tokens into it. Furthermore, a port must satisfy the choice and leg property. A path in a port is called a leg if it is a path from a split to a join. We also consider the initial place as a split and the final place as a join. The choice and leg property requires (a) every leg in a port to have at least two transitions with different directions of communication, and (b) all transitions belonging to the postset of a place must have the same direction of communication. The former ensures that choices are communicated and the latter ensures that no port can take a path from a split to a join on its own.

We distinguish between two types of ports: A server port advertises a service and needs a startup message and terminates after sending a result message. A client port consumes a service by sending a startup message and terminates after receiving the result message.

**Definition 3.3.11 (Port).** A port $C$ is a S-OWN such that

- $\forall t \in T : |(t \cup t^*) \cap (I \cup O)| = 1$, i.e. every transition is connected to one interface place;
- $\forall x \in I \cup O : |x \cup x^*| = 1$, i.e. every interface place has one transition from a given port in either its preset or its postset;
- (Choice and Leg property): (a) $\forall \beta = \langle p_1, t_1...t_{n-1}, p_n \rangle \in PS(C) : (\|p_1^\bullet\| > 1 \lor p_1 = i_N) \land (\|p_n\| > 1 \lor p_n = f_N) : \exists t, t' \in \beta : \lambda(t) \neq \tau \land \lambda(t') \neq \tau \land \lambda(t) \neq \lambda(t')$, i.e. every leg has at least two communicating transitions having different directions of communication, and (b) $\forall t_1, t_2 \in T : t_1 \cap t_2 \neq \emptyset \Rightarrow \lambda(t_1) = \lambda(t_2)$, i.e. if two transitions share the same place in their preset, then they are of the same type.

We call port $C$ as a server port if and only if $\forall t \in i_C^\bullet : \lambda(t) = receive \land \forall t \in f_C^\bullet : \lambda(t) = send$. We call port $C$ as a client port if and only if $\forall t \in i_C^\bullet : \lambda(t) = send \land \forall t \in f_C^\bullet : \lambda(t) = receive$.

The server of an MP pattern is a server port extended with a closure transition. So it satisfies the definition of a component and as consequence, it is able to handle more than one client request. A client of an MP pattern is a client port whose skeleton is isomorphic to the skeleton of the server discarding the closure transition. Unlike other interaction patterns, all places of an MP pattern, except the initial and final places of clients and idle place of the server, can be refined by either an S-WFN (modeling a procedure) or clients of other interaction patterns. An example of an MP pattern having one client and one server is illustrated in Fig. 3.10. The client initiates the communication by sending a start message. The server notifies the client that it is going active. As soon as the client receives this notification, it also goes active. In this state, the server computes a function and sends a result to the client. The client analyses the result and sends a feedback to the server. As soon as the server has received this feedback, it has two options:
either repeat the loop or notify the client that it has completed. In the later case, the active client is forced to exit the loop to fire the finishing transition. The server eventually terminates after sending a done message which is eventually picked up by the client.

Note that in an MP pattern no inhibitor arcs and no reset arcs occur, so they are classical Petri nets denoted by the tuple \((P, I, O, T, F, \text{init}, \text{fin})\).

**Definition 3.3.12 (The MP Pattern).** The MP pattern is defined as the pair \((B, \{\text{closure}(A)\})\), where \(B\) is the set of client ports, \(A\) is a server port with an idle place \(v_A\), and \(\forall b \in B : S(b) \cong S(A)\), i.e. a client and its server have isomorphic skeletons.

We define the system \(\text{MP}(B, \{A\}) = (N, M_0, M_f)\) with \(N = N(B, \{A\})\), \(M_0\) is defined by \(m \in M_0\) iff \(m(v) = 1, m(i_b) \leq 1\) for each client \(b \in B\), and all other places are empty, the set of final marking \(M_f\) is defined by \(m \in M_f\) iff \(m(v) = 1, m(i_b) + m(f_b) = m_0(i_b)\) for each client \(b \in B\) and initial marking \(m_0\), and all other places are empty.

**Weak Termination of MP patterns**

The choice and leg property are crucial for the weak termination property. In the absence of this property, deadlocks maybe introduced. In Fig. 3.11, we illustrate...
two cases of a deadlock in an MP pattern not satisfying the choice and leg property. In the example on the left, transitions $B$ and $C$ share a place in their preset while having different communication types. In such a case, the server and the client are able to make different choices resulting in a deadlock. In the example on the right, the legs containing transitions $C$ and $E$ at the server and $C'$ and $E'$ at the client have the same communication direction. In such a case the client is able to take the leg with transitions $C'$ and $E'$ more times than the server. So even though both client and server can reach their final places, the interface places are not empty. Hence the final marking cannot be reached.

We will prove the weak termination property for a general MP pattern. We structure our proof in the following manner:

1. First, we prove that for any reachable marking, from the initial marking of an MP pattern, there exists a firing sequence in the server and a firing sequence in one of the clients such that either these firing sequences are already isomorphic or one of the firing sequence can be extended by an executable firing sequence of only receive transitions and then they are isomorphic. Furthermore, all other clients have firing sequences with only send transitions.
2. Next, we prove that an MP pattern consisting of one client and one server, if the firing sequences of the client and server are of the same length then the interface places are empty. Using this result, we prove that such a pattern is weakly terminating.

3. Finally, we prove the weak termination property for a general MP pattern.

To prove (1), observe the following properties: From the initial marking of an MP pattern it is always possible that at least one of the client fires a sequence of send transitions. The server can now fire a sequence of receive transitions enabled by one of the clients followed by a sequence of send transitions. At this stage, the choice of a client is made by the server and the chosen client can fire a sequence of receive transitions enabled by the server, resulting in a firing sequence that is isomorphic to the firing sequence of the server. Furthermore, the firing sequences of the rest of the clients contain only send transitions. We formulate this proposition in the Lemma 3.3.7.

The crux of the proof lies in the choice and leg property. The choice and leg property ensures that no port has taken a leg one more time than its corresponding partner port, and every time one of the ports makes the choice of a leg, the choice is communicated to the corresponding partner port.

**Notations.** For the sake of convenience, we lift the notion of isomorphism to firing sequences in the following way. If two Petri nets are isomorphic with respect to some bijective function $\rho$, then for every executable firing sequence in one of the nets there exists an executable firing sequence in the other net and these firing sequences are isomorphic with respect to the function $\rho$. Furthermore, we will overload $\rho$ to denote both $\rho$ and $\rho^{-1}$.

**Lemma 3.3.7.** Consider the system $MP(B, \{A\}) = (N, M_0, M_f)$ with clients $B = \{B_1, \ldots, B_k\}$, where $k \in \mathbb{N}$, and initial marking $m_0 \in M_0$. Then for any reachable marking $m \in \mathcal{R}(N, i_N)$ and an executable firing sequence $\sigma \in T_N^*: i_N \xrightarrow{\sigma} m$, there exist the firing sequences:

$$\sigma_A \in T_A^*: [i_A] \xrightarrow{\sigma_A} \text{ and } \sigma_j \in T_{B_j}^*: [i_{B_j}] \xrightarrow{\sigma_j}, \text{ with } j \in \{1\ldots k\}, \text{ such that}$$

- $\sigma_A = \sigma|_A$ and for all $j \in \{1\ldots k\}$, $\sigma_j = \sigma|_{T_{B_j}}$;

- $\exists l \in \{1\ldots k\}: \forall j \in \{1\ldots k\}: j \neq l \land \forall t \in \sigma_j: \lambda(t) = \text{send}$, i.e. all firing sequences of clients, except for the chosen client $B_l$, contains only send transitions;

and one of the following conditions hold:

- $\sigma_A \cong \sigma_l$, i.e. firing sequences of the server and chosen client are isomorphic,
- \( \exists \sigma' \in T^*_i : \sigma_A \cong \sigma_1 \circ \sigma' \) where, \( \forall t \in \sigma' : \lambda(t) = \text{receive} \), i.e. the firing sequence of the chosen client can be extended such that it is isomorphic with the firing sequence of the server;
- \( \exists \sigma' \in T^*_A : \sigma_1 \cong \sigma_A \circ \sigma' \) where, \( \forall t \in \sigma' : \lambda(t) = \text{receive} \), i.e. the firing sequence of the server can be extended such that it is isomorphic with the firing sequence of the chosen client.

Proof. See Appendix B (Lemma B.1.1).

Next, we consider an MP pattern with one client and one server and show that if the firing sequences of the client and server are of the same length, then the interface places of this pattern are empty.

Due to the previous lemma, if the firing sequences of the client and server are of equal length then they are isomorphic. So for every send transition that has fired in the firing sequence of the server, a corresponding receive transition has fired in the firing sequence of the client and vice versa.

Lemma 3.3.8. Consider a system \( \text{MP}\{(B), \{A\}\} = (N,M_0,M_f) \) with initial marking \( [i_B,i_B] \in M_0 \). For a reachable marking \( m \in R(N,[i_N]) \) and an executable firing sequence \( \sigma \in T^*_N : [i_N] \xrightarrow{\alpha} m \), if \( |\sigma|_A = |\sigma|_B | \) then \( m(p) = 0 \) for all \( p \in I_A \cup O_B \), i.e. empty interface places.

Proof. Let \( |\sigma|_A = |\sigma|_B | = n \), where \( \sigma_A = \sigma|_{T_A} \), \( \sigma_B = \sigma|_{T_B} \). From Lemma 3.3.7 with one client, i.e. \( k = 1 \), we conclude that \( \sigma_A \cong \sigma_B \), and by Definition 3.3.5 (interaction pattern), for all \( i \in \{1 \ldots n\} \) satisfies \( \lambda(\sigma_A(i)) = \text{send} \) iff \( \lambda(\sigma_B(i)) = \text{receive} \) and vice versa.

So for every transition that produces a token on an interface place there exists a corresponding isomorphic transition that consumes it, which leads us to conclude \( m(p) = 0 \) for all \( p \in I_A \cup O_A \), since interface places were empty in the initial marking.

Using the above result in conjunction with the fact that the skeleton of the server weakly terminates, we will show that an MP pattern with one client and one server, weakly terminates.

Lemma 3.3.9. The system \( \text{MP}\{(B), \{A\}\} = (N,M_0,M_f) \) weakly terminates.

Proof. Consider a reachable marking \( m \in R(N,[i_N]) \) and a firing sequence \( \sigma : i_N \xrightarrow{\alpha} m \).

By Lemma 3.3.7, with one client, i.e. \( k = 1 \), there exists \( \sigma' : \sigma_A \cong \sigma_B \circ \sigma' \) such that \( m \xrightarrow{\sigma'} m' \). Let \( \tilde{\sigma} = \sigma \circ \sigma' \). Note that \( \tilde{\sigma}_A \cong \tilde{\sigma}_B \). By the Lemma 3.3.8, \( m'(p) = 0 \) for all \( p \in I_A \cup O_A \), i.e. the interface places are empty.

Since the skeleton of server \( A \) weakly terminates, there is a firing sequence \( \alpha \) in \( S(A) \) such that \( m_A' \xrightarrow{\alpha} [f_A] \). As the client \( B \) has an isomorphic skeleton, there exists an executable firing sequence \( m'_B \xrightarrow{\rho(\alpha(1)) \circ \cdots \circ \rho(\alpha(n))} [f_B] \) in \( S(B) \).
We will now build an executable firing sequence $\beta$ out of $\alpha$ in the following manner. For all $i \in \{1,..|\alpha|\}$, if $\lambda(\alpha(i)) = \text{send}$ then $\beta(2i-1) = \alpha(i) \land \beta(2i) = \rho(\alpha(i))$, otherwise $\beta(2i-1) = \rho(\alpha(i)) \land \beta(2i) = \alpha(i)$. By Lemma 3.3.8, the interface places are empty, so $m' \xrightarrow{\beta} [f_A, f_B]$. \hfill $\Box$

Due to the closure transition at the server, it is straightforward to prove the weak termination property for an MP pattern.

**Theorem 3.3.10.** The system $MP(B, \{A\}) = (N, M_0, M_f)$ weakly terminates.

**Proof.** Consider a reachable marking $m \in R(N, [i_A, i_{B_1}, \ldots, i_{B_k}])$ and a firing sequence $\sigma \in T_N^* : [i_A, i_{B_1}, \ldots, i_{B_k}] \xrightarrow{\sigma} m$.

By the lemma 3.3.7, either the firing sequences $\sigma_A$ and $\sigma_l$ are isomorphic, or there exists $l \in \{1 \ldots k\}$ such that either (a) $\sigma_l$ can be extended by an executable firing sequence $\tilde{\sigma} \in T_{B_l}^*$ such that $\sigma_A \cong \sigma_l \circ \tilde{\sigma}$, or (b) $\sigma_A$ can be extended by an executable firing sequence $\tilde{\sigma} \in T_A^*$ such that $\sigma_l \cong \sigma_A \circ \tilde{\sigma}$. Let $m \xrightarrow{\tilde{\sigma}} m'$ and let $\sigma' = \sigma_l \circ \tilde{\sigma}$.

Consider only the client $B_l$. By the lemma 3.3.8, we know that the interface places are empty. By the lemma 3.3.9, we know that a firing sequence $\sigma'' \in T_A^* \cup T_{B_l}^*$ exists such that $m' \xrightarrow{\sigma''} [f_A] + [f_l]$. Since server $A$ has a closure transition $t \in T_A$ that is enabled, we may fire it and then we have the step: $[f_A] + [f_l] \xrightarrow{t} [i_A] + [f_l]$, i.e. a marking with server in idle and a client with a token in its final place is reachable.

So server $A$ can handle one more client, say some $j \in \{1 \ldots k\} : j \neq l$, and then by the Lemma 3.3.9 and due to the closure transition, there is a firing sequence leading to the marking $[i_A] + [f_l] + [f_j]$. So in $k$ steps $[i_A, f_1, f_2, \ldots, f_k]$ is reachable. \hfill $\Box$

A logical consequence of the definition of the MP pattern is that clients can be attached and detached at runtime, as shown in the next corollary.

**Corollary 3.3.11** (Dynamic reconfiguration of the MP pattern). Given a MP pattern $(C, \{S\})$ with $C = \{C_1, \ldots C_n\}$, let $m$ be a reachable marking of $MP(C, \{S\})$. Let $X, Y, Z \subseteq C$ be a partitioning of $C$ such that $C_j \in X$ iff $m(i_j) = 1$, $C_j \in Y$ iff either $m(f_j) = 1$ or the net $C_j$ is unmarked, and $C_j \in Z$ otherwise. Then $MP(C', \{M\})$ is weakly terminating for any set of clients $C'$ with $Z \subseteq C' \subseteq C$.

**Designing MP patterns**

Next, we present a general strategy to derive arbitrary MP patterns. First, the interaction between a client and a server of an MP pattern is modeled as a single S-WFN describing the order of events. Each event modeled as a transition represents a communication event of type: send-receive or receive-send, but labeled as receive or send, respectively. The former represents the sending of a message from a client to server, and the latter represents the receipt of a message by a client from a
Furthermore, we require that (a) the transitions in the postset of an Input S-WFN: M, No. of Clients: N
Output Instance of an MP pattern

1. Label M as a "server" and assign a communication direction to each transition, as specified by its label;
2. Make a copy of the server and reverse the communication direction of each transition (i.e. change send to receive and vice versa). Label the copy as "client";
3. Make N − 1 copies of the client and connect corresponding transitions of server and clients;
4. Add a closure transition to the server;

Algorithm 1: Construction of an MP pattern

initial place are labeled as receive, (b) the transitions in the preset of a final place are labeled as send, and (c) every event in the postset of a split is of the same type and every path from a split to a join has at least two different communication events. In this way the choice and leg property is satisfied by the final MP pattern. Once the S-WFN has been defined in this way, the MP pattern can be derived using the Algorithm 1. In the Fig. 3.12, we illustrate the derivation of an MP interaction pattern with an example.
3.4 The Construction Method

In this section, we present a construction method to derive weakly terminating systems starting from an architectural diagram. An architectural diagram gives the blue print of the system. From this diagram we proceed in a bottom up manner consisting of two phases: In the first phase, all RPC, GFR and MP patterns are introduced in the right order by successive applications of simultaneous refinement. In the second phase, successive insertions of the PS patterns are carried out. During the first phase, it is always possible to elaborate the behavior of a system by top-down refinements with S-WFNs modeling procedures. We will first describe the architecture diagram.

3.4.1 Architectural Diagram

The Fig. 3.13 describes a graphical notation to specify the component architecture of a system. The notation is similar to component models like SCA [3], Koala [264] and UML [150] but extended with interaction patterns and component types.

An architectural diagram is a directed graph with components as nodes and interaction patterns as edges. To guarantee weak termination, the graph must be acyclic, without taking into account the PS pattern. It is easy to check that a cyclic path indicates a deadlock in the system: to handle its clients, the server needs to finish itself, using a client that can only finish if the server itself finishes, which clearly is a deadlock.

A component has a type: being either a GFR, a RPC, a MP or a root component. A GFR or RPC or MP component is a server of the corresponding interaction pattern. All other components are referred to as root components. A component is denoted by a solid rectangle, and labeled with its name and type. Clients and servers of an interaction pair are depicted by an arc with arrow heads both at its foot and head. Clients communicating to the same server are connected to the same arrow head. A filled arrow head indicates the RPC pattern, an unfilled arrow head the GFR pattern and the double arrowhead the MP pattern. A server of an RPC pattern has multiple procedures. For each procedure, an explicit incoming arrow head is drawn. A client and server of a PS pattern are denoted by a pair of circles that are connected by a directed arrow from the client to the server. The clients of GFR, RPC and MP patterns can be contained (embedded) in servers of RPC, GFR, MP patterns or root components. So we do not explicitly show this relationship. However, only for the case of MP patterns, clients may contain clients of other RPC, GFR or MP patterns. We denote this containment relationship by a dotted directed edge from child to its containing parent. For an example, see the user interface component.

The architectural diagram in the Fig. 3.13 describes a navigation system of a robot. In this diagram, there are five root components, one GFR pattern, one MP pattern, three RPC patterns and seven PS patterns. Note that in the user interface component, there is one client of an RPC pattern that is contained...
within a client of an MP pattern. In the Fig. 3.14, we show the corresponding Petri net models. Next, we will discuss how to combine them into a Petri net model of the system.

To construct a net from an architectural diagram, we use Algorithm 2. The method prescribes a structured way to add components to the system in a bottom-up manner. We start by defining root components, i.e. components that do not provide a service. Next we identify each interaction pattern in the diagram and define their clients and servers. Then the method progresses in two steps: first all the GFR, RPC and MP patterns are added to the system by simultaneous refinement, carried out in the right order of nesting as specified by the architectural diagram. Note that in each refinement step, one new server (component) is added to the system. Finally, each of the PS patterns are inserted into the system. A server of GFR, RPC or MP pattern and clients of MP pattern can be further refined with a client of another interaction pattern or an S-WFN defining some orchestration behavior. All places of an S-WFN and root components are refinable.

In the Fig. 3.15, we show the result of applying the construction method to the set of Petri net ingredients identified in the Fig. 3.14. We do not require to design
Figure 3.14: Ingredients of the Architectural Diagram
Input: Architecture Diagram

Output: Logical Model of a Weakly Terminating System

1. Identify root components and define their behavior;
2. The initial system is the union of all basic components;
3. Identify interaction patterns and the components they occur;
4. while not all RPC, GFR or MP patterns have been added in the right order of nesting do
   5. Choose an RPC, GFR or MP pattern such that for all clients there exists components, and in case clients are contained within another client, then their parents exist in the system so far;
   6. Identify the set of places to be refined (one for each client) and perform a simultaneous refinement with the chosen pattern, thereby creating a new server;
   7. If necessary, refine a place with an S-WFN;
5. end
6. while not all publish-subscribe patterns have been added do
   8. Choose a publish-subscribe pattern for which a set of refinable places (one for each client) and a set of idle places (one for each server) already exists in the system so far;
   9. Perform an insertion of the chosen pattern (see Def. 5);
10. If necessary, refine a place with an S-WFN;
7. end

Algorithm 2: Construction Method

We will now prove that the construction method always results in a weakly terminating system. For this we must first show that simultaneous refinement of a weakly terminating system with one of the interaction patterns RPC, GFR or MP, always results in a weakly terminating system.

Lemma 3.4.1 (Simultaneous Refinement Preserves Weak Termination).
Consider an OPN $\mathcal{O}$ that is weakly terminating. Let $R \subseteq P_{\mathcal{O}}$ be the set of places to be refined with either a GFR or RPC or MP interaction pattern $(B, \{A\})$ with the net $M = N(B, \{A\})$ such that $\mathcal{O}$ and $M$ are disjoint and $\alpha : R \to \text{init}_M \times \text{fin}_M$ is a total bijection. Then $N = \mathcal{O} \circ_{\alpha} M$ is weakly terminating.

Proof. Let $m_0$ be an initial marking of net $N$. Consider a reachable marking $m \in R(N, m_0)$ with a firing sequence $\sigma \in T^*_N$, i.e. $m_0 \xrightarrow{\sigma} m$. We consider two cases:

- Suppose the firing sequence $\sigma$ has either not marked any of the clients in $B$ or if a client $b \in B$ is marked then it has a token in its final place.
Figure 3.15: Petri net Model of the System: Weak Termination Guaranteed!
– Suppose none of the clients in $B$ are marked. By definition of simultaneous refinement, marking $m$ is almost a marking of $O$, except the token in the idle place of the server $A$. Let marking $m_O$ be the projection of marking $m$ on OPN $O$, while discarding the idle place $i_A$. Since the net $O$ is weakly terminating, there exists a firing sequence $\sigma' \in T^*_O$ such that the final marking $m_f$ of $S(O)$ is reachable, i.e. $m_O \xrightarrow{\sigma'} m_f$.

– Suppose there exists a client $b \in B$ with a token in its final place and all other clients are unmarked. By definition of simultaneous refinement, there exists a transition $r \in R$ such that $\alpha(r) = (i_b, f_b)$. This means that in the marking $m$, if we rename $f_b$ to $r$ (for all $b \in B$ that are marked), and then take the projection of this marking on OPN $O$, while discarding the idle place $i_A$, we have a reachable marking of net $O$ and we denote it by $m_O$. Since the net $O$ is weakly terminating, there exists a firing sequence $\sigma' \in T^*_O$ such that the final marking $m_f$ of $S(O)$ is reachable, i.e. $m_O \xrightarrow{\sigma'} m_f$.

Suppose none of the transitions $t \in \sigma'$ mark a place $r \in R$. Then $\sigma$ is also an executable firing sequence of net $N$ and then the final marking $[m_f] + [i_A]$ of net $N$ is reachable.

Suppose there exists a transition $t \in \sigma'$ that marks a place $r \in R$. Then we will construct a firing sequence $\beta$ out of $\sigma'$ that takes us to the final marking $[m_f] + [i_A]$ of net $N$. Let $t$ be the first transition in $\sigma'$ that marks a place $r \in R$. Let $\sigma' = \sigma'_1 \circ (t) \circ \sigma'_2$ and let $\alpha(r) = (i_b, f_b)$ for some $b \in B$.

By the Theorem 3.3.1, 3.3.3 and 3.3.10, we know that $M$ is weakly terminating. So there exists a firing sequence $\sigma'' : [i_b] + [i_A] \xrightarrow{\sigma''} [f_b] + [i_A]$. Let $\beta_1 = \sigma'_1 \circ (t) \circ \sigma'' \circ \sigma'_2$. By repeating this construction for every $t \in \sigma'_2$ that marks a place $r \in R$, we have constructed the firing sequence $\beta$ of net $N$ such that $m \xrightarrow{\beta} [m_f] + [i_A]$.

– Suppose the firing sequence $\sigma$ has marked at least one client $b \in B$ such that its final place is not marked. By the definition of simultaneous refinement, the interaction pattern $M$ is a subnet of net $N$, and each client of $M$ has exactly refined one place $r \in R$ of net $O$. This means there is one entry and one exit for each client (i.e. entry via its initial place and exit via its final place) and the server is interacting only with its clients. Therefore the net $M$ has the same behavior as that of being a subnet of the net $N$. So for each marked client $b \in B$ having a token that is not in its final place, there exists a firing sequence that takes it to its final marking $f_b$ and the server returns to its idle state $i_A$, due to the closure transition. This is the previous case.

$\Box$
Using the above lemma in conjunction with the behavior of a PS pattern and the insertion operation, we show that the construction method results in a weakly terminating system.

Theorem 3.4.2 (Preservation of Weak Termination). The construction method described in the Algorithm 2 preserves weak termination.

Proof. The construction method starts with a set of root components. From [163, 164], we know that each root component is weakly terminating.

Suppose we have performed a sequence of simultaneous refinements with GFR, RPC and MP patterns and the system derived so far is weakly terminating. Let the next pattern to be added to the system be either a GFR, RPC or MP pattern. Then by the Lemma 3.4.1, the resulting system is yet weakly terminating.

After all RPC, GFR and MP patterns have been introduced, the PS patterns are added. From Thm. 3.3.5, we know that a PS pattern is weakly terminating. The insertion of a PS pattern on top of an already weakly terminating system does not inhibit or extend the behavior of the original system. The only difference is that now clients of a PS pattern can put tokens in the topic input places of server transitions that are inserted in RPC or GFR or MP servers. These transitions are connected to the idle place of servers with bi-directional arcs. Since the whole system so far was weakly terminating, it is able to reach the state with the idle place of a server marked. But then these inserted transitions can consume the tokens of the topic input places.

In order to guarantee weak termination for systems at run-time, components can only be added or removed when they are in their initial state. Also interaction pattern can be changed at run-time. A client can be inserted in a component if the place to be refined is empty. It can be removed if it is not marked. Introducing a new RPC, GFR or MP pattern, which is in fact the addition of a new component, can be done at any point in time, as long as in the other components the places to be refined with a client are empty. To remove the server of a PS pattern we must first ensure that its topic place is empty. To remove a server of an RPC, GFR or MP pattern, its set of clients should be empty and it must be idle.

3.5 Conclusions

In this chapter we presented a method to design software systems of robots that are weakly terminating by construction. The method is based on familiar step-wise refinement principles. We used Petri nets to model components as strongly connected state machines and the four interaction patterns as multi-workflows. A nice feature of our method is that it allows dynamic reconfiguration. Petri nets were a good choice because the systems we model are component-based with asynchronous communication and the weak termination property is well-studied within the framework of Petri nets. Although weak termination at the level of
control flow is an important sanity check for systems, it is only a starting point for system verification. Data manipulation by transitions can destroy the weak termination property. We have to check this as well, but this can be done locally. Robot systems are typical real time systems, and the weak termination property we considered abstracts from time. In the next chapter we will extend our architectural framework with time to be able to make statements about the time needed to reach a final state from an arbitrary state.
In the last chapter, we proposed an architectural framework for modeling software of planning and control systems like robots. We also guaranteed the weak termination property at the control flow level, for architectural models that could be derived using the construction method. However, the safety, reliability and performance of such systems depends not only on the right ordering of events but also on their right timing. For this we extend our architectural framework with the notion of time.

First, we give an overview of the different approaches to time in Petri nets. In literature, most attention is paid to models where the time is expressed by delaying transitions and for the stochastic case to continuous time models with exponential enabling distributions. We will focus on discrete models where time is expressed
by delaying tokens and the probability distributions are discrete, because these model classes have some advantages. We call the former model class as discrete timed Petri nets (DTPN) and the latter as discrete stochastic Petri nets (DSPN). For model class DTPN, we present a finite reduction strategy for its infinite state space. So verification by model checking is possible. For model class DSPN, we show how Markov techniques can be used to answer interesting questions. For a subclass, we show how structural analysis techniques can be applied to reduce computational effort.

### 4.1 Introduction

Over the last 25 years, the modeling and analysis of time has been studied extensively in the context of Petri nets as well as of other models of concurrency, since time has an influence on the behavioral properties of a system. Most of the Petri net models with time are extensions of classical Petri nets, so if we discard the time aspects we obtain a classical Petri net. There are many different approaches to model time such as: delaying of instantaneous transitions, duration of transitions, aging or delays in tokens, timers and clocks.

The earliest papers seem to be [235] which introduces delaying of transitions and [280] introducing duration of transitions. These model classes are often referred to as Merlin time (or Time Petri Nets) and Timed Petri Nets respectively. Many authors have contributed to these models with results on expressiveness, e.g. [71, 89, 175], and on model checking [49, 48, 288, 10, 188]. An important reason for extending models with time features is due to the possibility for performance and reliability analysis (c.f. [294, 294, 281, 136]). While performance analysis is concerned with the extremities of behavior (like “will every service request be processed within 5 time units?”), reliability analysis is concerned with “average” behavior or “normal” behavior, i.e. behavior within bounds with a specified probability (like “will 90% of service requests be processed within 5 time units?”). To make reliability analysis possible, non-deterministic choices in models should be endowed with a probability. There is a very extensive literature on stochastic Petri nets where the execution time of transitions is exponentially distributed, resulting in continuous-time Markov processes, see e.g. [13, 224, 154]. For the class of generalized stochastic Petri nets (GSPN), having instantaneous transitions, there is a popular software tool for modeling and analysis called GSPN [349]. This class is in fact the stochastic extension of Merlin time, so the approach with delaying transitions. Different approaches proposed for modeling time dimension in Petri nets all have their own merits and weaknesses. Some of the approaches are questionable from the modeling point of view but are handy for analysis purposes, while others are good for modeling but bad for analysis. The application domain also has an impact on the right choice of the way to model time. Two extreme application domains are logistic systems (e.g. the business chain from cows to the dairy products) and embedded control systems (e.g.
robots). Not only are the time units different but the type of questions are also different.

In this chapter, we focus on the approach with token timestamps since we have the feeling that this class did not obtain enough attention although it is a powerful class both for modeling and for analysis. Extending tokens with timestamps is studied in a number of works, see [162, 2, 176, 46, 197, 65], and due to a nice tool support (ExSpect [348] and CPNTools [347]), there are multiple examples of industrial applications of this model of time in Petri nets, often called *interval timed Petri nets* (ITPN). We restrict our focus to the discrete time setting and call the corresponding model class *discrete time Petri nets* (DTPN). The ITPN model is a simplification of it. Since timestamps of tokens are non-decreasing, model checking the class of DTPN directly is not possible. So we propose a reduction strategy to derive a finite transition system that is simulation equivalent to the reachability graph of a DTPN, if the underlying untimed Petri net is bounded. Next to DTPN, we consider a stochastic variant of it called *discrete stochastic Petri nets* (DSPN) that can be seen as an alternative to the GSPN model. This class has discrete time, i.e. finite or countable time domain and discrete probability distributions over the choices and it encompasses several well-known subclasses. We show how Markov techniques can be applied to DSPN to answer questions like the probability of reaching a subset of states, the expected time to leave a subset of states, the expected sojourn time in equilibrium. For the subclass of workflow nets, we consider structural techniques that reduce the computational effort.

### 4.1.1 Related Work

Time in tokens was first considered in [294]. Next references occur in [162] and [176]. In particular, the ExSpect [348] and CPN-tools [347] use this notion of time. Newly produced tokens obtain a timestamp by some computation or simply by selection from a given set (often an interval) belonging to an output arc of a transition. The first approach for analysis of models with time in tokens was proposed by [1] and this model was called *interval timed Petri nets* (ITPN). In [1], a reduction method is introduced where tokens obtain a time interval. However, the reduction method has two weaknesses: (1) the reduced model could have more behavior (i.e. more untimed firing sequences) than the original model, which makes the method sound but not complete, (2) the state space could be infinite which allows only for analysis until a certain end time.

In [46] and [66] another approach is taken. First they reduce the model into a model with a bounded time window by subtracting the event times from tokens (we do it in this paper in a similar but different way). Secondly they cluster timed markings into state classes that depend on the initial marking and a specific untimed firing sequence. All markings in these state classes have the same untimed marking and they only differ in timestamps. The timestamps of these state classes can be characterized by logical formulae. As the state classes are dependent on
the firing sequences: If a transition \( t \) can fire from a marking \( m \) belonging to a state class \( A \) to a marking \( m' \), i.e. \( m \overset{t}{\rightarrow} m' \), then \( m' \) belongs to state class \( B \) and it is said that \( A \overset{t}{\rightarrow} B \). So for all markings \( m' \in B \) there must be a marking \( m \in A \) such that \( m \overset{t}{\rightarrow} m' \). It is shown that the formula of state classes are only dependent on the bounds of time intervals and simple arithmetic operations on these bounds. Therefore the number of different formulae is finite. If the underlying Petri net model is bounded then the number of state classes is bounded and so the system can be analyzed. The languages of the original model and state class model are the same.

Since these computations are quite complex, in [65], a simplification was given by introducing clocks for all tokens and introducing a new kind of state classes based in the untimed marking and the clock difference matrices. So all timed markings with the same untimed marking, and with timestamps that satisfy the property that the difference between two timestamps is less than or equal to the value in the difference matrix, belong to the same state class. In [65], the update rules are presented for these classes. Like in the original method, the computations are done by extending firing sequences until no new state classes are found. Although this approach is computationally more attractive than the original one, there are some flaws. First of all the update rules are not correct as illustrated by the following example (see fig. 4.1). According to the proposition 4.3 in [65], for pairs of old tokens \( d \) and \( d' \) (i.e. tokens that were not created during the creation of the marking), we have the following inequality \( D'(d, d') \leq D(d, d') \). But in the example of fig. 4.1, we observe that \( D'(2, 3) > D(2, 3) \). In fact this implies that the maximal difference between pairs of old tokens is no longer bounded by the corresponding value in the delay matrix!

Secondly, there is no proof that the new state classes (captured by difference
matrices, as proposed in [65]) are the same as the original state classes which implies that there is no proof of language equivalence. In [197], another approach is followed. Here only a fixed choice is made for newly produced tokens. They also have a reduction to a finite state space. In this chapter, we have a more straightforward approach than [46] and more in line with [197]. However, we have besides output intervals also (fixed) input delays and we prove simulation equivalence which is stronger than language equivalence.

This chapter is structured as follows: In Section 4.2, we give an overview of the five important time models in Petri nets. We do not consider coloring of tokens, although most of the results can be extended, with some technical effort, to colored Petri nets. We also do not consider continuous Petri nets (cf [283]) because the underlying untimed net is not a classical Petri net any more for this class of nets. In Section 4.3, we present the model class DTPN and a state space reduction method to analyze DTPN models in a finite way. In Section 4.4 we consider the stochastic version of DTPN and we show how classical Markov techniques can be used to analyze them. In particular, we consider three questions: the probability of reaching a set of states, the expected time of reaching a set of states and the equilibrium distribution. For these methods we need the whole state space, like in model checking. However, for workflow nets there are also structural techniques that do not need the whole state space. We will show how these techniques can be combined with the others. We conclude the chapter in the Section 4.5.

4.2 Overview of Petri nets with time

In this section we describe several options for extending Petri nets with time. To make the syntax uniform, we use the same definition for different classes of timed Petri nets; in this definition we add delay sets to the arcs of a classical Petri net. The semantics of these delay sets differ significantly in different model classes, and additional constraints on delays will be imposed in certain cases.

Definition 4.2.1 (Syntax of a timed Petri net). A timed Petri net (TPN) is a tuple \((P, T, F, \delta)\), where \((P, T, F)\) is a Petri net, \(\delta : F \to \mathcal{P}(\mathbb{R})\) is a function assigning delay sets of non-negative delays to arcs.

We consider \(\delta(p, t), \ (p, t) \in F\), as an input delay for transition \(t\) and \(\delta(t, p), \ (t, p) \in \cap F\), as an output delay for transition \(t\). We distinguish multiple subclasses of TPNs using different combinations of the following restrictions of the delay functions:

**In-zero** Input arcs are non-delaying, i.e. for any \((p, t) \in F\), \(\delta(p, t) = 0\).

**Out-zero** Output arcs are non-delaying, i.e. for any \((t, p) \in F\), \(\delta(t, p) = 0\).

**In-single** Input delays are fixed values, i.e. for any \((p, t) \in F\), \(|\delta(p, t)| = 1\).

**Out-single** Output delays are fixed values, i.e. for any \((t, p) \in F\), \(|\delta(t, p)| = 1\).
**In-fint** Input delays are finite rational sets.

**Out-fint** Output delays are finite rational sets.

**In-rint** Input delays are closed rational intervals.

**Out-rint** Output delays are closed rational intervals.

**Tr-equal** For every transition, the delays on its input arcs are equal, i.e. \( \forall t \in T, (p_1, t), (p_2, t) \in F, \delta(p_1, t) = \delta(p_2, t) \)

**Pl-equal** For every place, the delays on its input arcs are equal, i.e. \( \forall p \in P, (p, t_1), (p, t_2) \in F, \delta(p, t_1) = \delta(p, t_2) \).

There are several dimensions on which different models of time for Petri nets differ semantically, such as:

- **Timed tokens vs untimed tokens**: Some models extend tokens, and thus also markings, with the time dimension, e.g. with time stamps to indicate at which moment of time these tokens become consumable or/and till which time the tokens are still consumable. A transition may fire if it has consumable tokens on its input places. Other models keep tokens untimed, meaning in fact that tokens are always consumable. The time semantics is then captured by time features of places/transitions only.

- **Instantaneous firing vs prolonged firing**: In some models, the firing of a transition takes time, i.e. the tokens are removed from the input places of a transition when the firing starts and they are produced in the output places of the transition when the firing is finished. In an alternative semantics, a potential execution delay is selected from a delay interval of a transition, but the transition firing is instantaneous.

- **Eager/urgent/lazy firing semantics**: In the eager semantics, the transition that can fire at the earliest moment is the transition chosen to fire; with the urgent semantics the transition does not have to fire at the earliest moment possible, but the firing may become urgent, i.e. there is a time boundary for the firing; in the lazy semantics the transitions do not have to fire even if they lose their ability to fire as a consequence (because e.g. the tokens on the input places are getting too old and thus not consumable any more).

- **Preemption vs non-preemption**: Preemption assumes that if a transition gets enabled and is waiting for its firing for a period of time defined by its
delay and another transition consumes one of the input tokens of \( t \), then the clock or timer resumes when \( t \) is enabled again, and thus its firing will be delayed only by the waiting period left from the previous enabling. The alternative is called non-preemption.

- **Non-deterministic versus stochastic choices** in the delays or order of firing.

We introduce the notion of a *timed marking* for all timed Petri nets. We assume the existence of a *global clock* that runs “in the background”. The “current time” is a sort of cursor on a time line that indicates where the system is on its journey in time. We give the tokens a unique *identity* in a marking and a *time stamp*. The time stamps have the meaning that a token cannot be consumed by a transition before this time.

**Definition 4.2.2 (Marking of a TPN).** Let \( I \) denote a countably infinite set of identifiers. A marking of a TPN is a partial function \( m : I \rightarrow P \times Q \) with a finite domain. For \( i \in \text{dom}(m) \) with \( m(i) = (p, q) \) we say that \((i, p, q)\) is a token on place \( p \) with time stamp \( q \). We denote the set of all markings of a TPN by \( M \) and we define the projection functions \( \pi, \tau \) as 

\[
\pi((p, q)) = p \quad \text{and} \quad \tau((p, q)) = q.
\]

The semantics of the different model classes are given by different transition relations. However, they have a commonality: if we abstract from the time information, the transition relation is a subset of the transition relation of the classical Petri net. This requirement to the semantics is expressed in the following definition.

**Definition 4.2.3 (Transition Relation of TPN).** A transition relation \( \rightarrow \subseteq M \times T \times M \) of a TPN satisfies the following property: if \( m \xrightarrow{t} m' \) then:

- \( \text{dom}(m') \cap \text{dom}(m \setminus m') = \emptyset \). (identities of new tokens differ from consumed tokens),
- \( \{\pi(m(i)) | i \in \text{dom}(m \setminus m')\} = t \) and \( |m \setminus m'| = |t| \) (consumption from the pre-places of \( t \)), and
- \( \{\pi(m'(i)) | i \in \text{dom}(m \setminus m)\} = t \) and \( |m \setminus m| = |t'| \) (production to the post-places of \( t \)).

Given a marking \( m_0 \in M \), a sequence of transitions \( \langle t_1, \ldots, t_n \rangle \) is called a *firing sequence* if \( \exists m_1, \ldots, m_n \in M : m_0 \xrightarrow{t_1} m_1 \ldots \xrightarrow{t_n} m_n \). We denote the finite set of all firing sequences of a TPN \( N \) from a marking \( m \in M \) by \( FS(N, m) \).

There are three main classes of TPN, based on the time passing scheme chosen:

\[\text{\textsuperscript{1}}\text{By extending the time domain to } Time \times Time \text{ with } Time \subseteq \mathbb{R} \text{ we could talk about “usability” of tokens, with tokens having the earliest and the latest consumption times.}\]
– **Duration of firing (M1)**

Time is connected to transitions (Tr-equal). As soon as transitions are enabled, one of them is selected and for that transition one value is chosen from the common input delay interval. Then the tokens are consumed immediately and new tokens for the output places are produced after the delay. The global clock moves till either the end of the delay or to an earlier moment where another transition is enabled. This is the Timed Petri Net model.

– **Delaying of firing (M2)**

Time is connected to transitions (Tr-equal). As soon as transitions are enabled, for each of them a delay is selected from the input delay interval. One of the transitions having the minimal of those delays is chosen to fire after the delay has passed. If more than one transitions have the same delay then one of them fires and this choice is non-deterministic. This is often called Merlin time and race semantics.

– **Delaying of tokens (M3)**

Time is connected to tokens (Out-rint, In-single). This is the DTPN model where a transition can fire at the time of the maximal timestamp of its input tokens. One of the earliest enabled transitions will fire. The input delay is added to the timestamp in order to determine if it is consumable or not.

Timed automata are another important class of timed models (see [33]). For model classes M1 and M2, it is possible to translate them to timed automata [213, 88]. Some models of class M3 (DTPN) can also be translated to timed automata, but because of eagerness the translation is only possible for the subclass (Tr-equal, In-single, Out-rint). We will study the relationship between a DTPN and a timed automata in the next chapter.

Generalized stochastic Petri nets (GSPN) is a popular class of stochastic Petri nets. It belongs to the class of M2, i.e. race semantics with non-preemption. In general, non-preemption is a strange property from a practical point of view, because independent parts of a system are “working” while waiting for the firing of their transitions, and in case the input tokens of some transition \( t \), being in preparation to its firing, are taken by another transition, the preparation work done by \( t \) is lost. However, preemption and non-preemption coincide in case of exponentially distributed delays: if a transition with a delayed firing is interrupted, the rest-distribution in case of preemption is the same as the original waiting time. Therefore these models can be transformed into a continuous time Markov process and the analysis techniques for Markov processes can be applied. However, exponential distributions are very restrictive for modeling in practice. It is possible to approximate an arbitrary continuous distribution by a phase type distribution, which is a combinations of exponential distributions, but this is not a nice solution to model arbitrary transition delay distributions, because it blows up the state space and it disturbs the preemption property. Therefore, in Sec-
tion 4.4, we will introduce a stochastic version of the class of $M3$, which we call $DSPN$.

To compare model classes we will define an equivalence between classes.

**Definition 4.2.4 (Equivalence of model classes).** A model class $I$ of timed Petri nets is included in model class $J$, denoted by $I \subseteq J$, if for each net system $(X,x)$ of class $I$, there exists a net system $(Y,y)$ of class $J$ such that $(X,x) \simeq_b (Y,y)$. Model classes $I$ and $J$ are equivalent, denoted by $I \sim J$ iff $I \subseteq J \land J \subseteq I$.

If for each net system $(X,x)$ of class $I$, there exists a net system $(Y,y)$ of class $J$ such that $FS(X,x) \subseteq FS(Y,y)$ then we denote it by $I \subseteq_{fs} J$.

### 4.3 Discrete Timed Petri Nets

In this section, we will first define the semantics of DTPN (Out-rint, In-single), i.e. model class $M3$.

In a DTPN, there are two ways to enforce time progression (a) by associating a closed rational interval as a delay set to outgoing arcs from transitions, and (b) by associating fixed point delay as a singleton delay set to each input arc to a transition. The former specifies how long a newly produced token must wait before it becomes available for consumption. This is useful for modeling task execution time in a system. The latter specifies how much more an available token in the corresponding place must wait before it can be consumed by the corresponding transition. This is useful for specifying a message time-out.

By the semantics of a DTPN we mean the firing rule for transitions from a marking. For this we define the two concepts; enabling time of transitions in a marking and firing time of a marking. The enabling time of a transition in a marking is the earliest possible time it can fire, determined completely by the tokens in its pre-places and delays on its input arcs. A DTPN has eager semantics, i.e. the transition with the earliest enabling time will fire at its enabling time, called the firing time of a marking. The firing of an enabled transition is instantaneous, i.e. no time can progress during the consumption and production of tokens. The produced tokens have their timestamps increased by a value that is chosen from the delay set on its outgoing arc.

As timestamps of a DTPN are non-decreasing, its state space cannot be analyzed directly. To make verification by model checking possible, we consider the two subclasses of DTPN: sDTPN (Out-single, In-single) and fDTPN (In-single, Out-fint) and show that DTPN, sDTPN and fDTPN are equivalent. Then we show how a sDTPN can be transformed by reducing the time component, into a strongly bisimilar labeled transition system called rDTPN. For rDTPN we show that it has a finite reachability graph if the underlying Petri net is bounded. So rDTPN can be used for model checking and since it is equivalent to the general DTPN, we are able to model check them as well. In the following sections, we will
formally prove our results on the relationship between rDTPN, sDTPN, fDTPN and DTPN.

Note. For practical applications, the knowledge of these relationships must be used to develop model checking methods to automate the verification process of DTPN models (see Sec. 4.3.5). So it is not necessary for a system architect to understand the underlying mathematical proofs in order to be able to analyze them.

4.3.1 Semantics of DTPN

In order to define the firing rule of a DTPN we introduce the notion of an activator which models an event of a DTPN. For a transition \( t \) in a marking \( m \) the activator is a minimal subset of the marking that enables this transition. Like in classical Petri nets, an activator has exactly one token on every input place of \( t \) and no other tokens. In addition, each token of an activator corresponds the earliest timestamp token in that place.

**Definition 4.3.1 (Activator).** Consider a TPN with the set of all markings \( \mathbb{M} \). A marking \( a \in \mathbb{M} \) is called an activator of transition \( t \in T \) in a marking \( m \in \mathbb{M} \) if

- \( a \subseteq m \), \( a \) coincides with \( m \) on \( \text{dom}(a) \),
- \( \{ \pi(m(i)) \mid i \in \text{dom}(a) \} = \bullet t \) and \(|a| = |\bullet t| \) (\( t \) is classically enabled), and
- for any \( i \in \text{dom}(a) : \tau(m(i)) = \min\{\tau(m(j)) \mid j \in \text{dom}(a) \land \pi(a(j)) = \pi(a(i)) \} \) (\( i \) is the oldest token in that place; FIFO assumption).

We denote the set of all activators of a transition \( t \) in a marking \( m \) by \( A(m,t) \).

The enabling time of a transition \( t \) is the earliest possible time this transition can fire. The firing time of a marking \( m \) is the earliest possible time one of the transitions, enabled by this marking, can fire. We are assuming an eager system.

The time information of a DTPN is contained in its marking. This makes it possible to consider the system only at the moments when a transition fires. For this reason, we do not represent time progression as an explicit state transition.

**Definition 4.3.2 (Enabling time, Firing time).** For a DTPN let \( a \) be an activator of \( t \in T \) in marking \( m \). The enabling time of transition \( t \) by an activator \( a \) is defined as

\[
\text{et}(a,t) = \max\{\tau(a(i)) + \delta(\pi(a(i)), t) \mid i \in \text{dom}(a)\}
\]

The firing time of a marking \( m \) is defined as

\[
\text{ft}(m) = \min\{\text{et}(a,t) \mid t \in T, a \in A(m,t)\}
\]
The firing time for a marking is completely determined by the marking itself!

The firing of a transition and its effect on a marking of a DTPN is described by the transition relation (see Fig. 4.2). Produced tokens are ‘fresh’, i.e. they have new identities and their timestamps are increased by a value chosen from the delay set associated with the corresponding outgoing arc.

**Definition 4.3.3 (Transition Relation of DTPN).** The transition relation of a DTPN satisfies Def. 4.2.3 and moreover, for any \( m \xrightarrow{t} m' \):

- there is an \( a \in A(m, t) \) such that \( a = m \setminus m' \land ft(m) = et(a, t) < \infty \), and
- \( \forall i \in \text{dom}(m' \setminus m) : \tau(m'(i)) = ft(m) + x \) with \( x \in \delta(t, \pi(m(i))) \).

![Diagram of Firing Rule of a DTPN](image)

**Figure 4.2:** Firing Rule of a DTPN

Since delays on arcs are non-negative, a system cannot go back in time.

**Lemma 4.3.1 (Monotonicity).** Let \( N \) be a DTPN with the set of all markings \( \mathbb{M} \). Then \( \forall m, m' \in \mathbb{M}, t \in T : m \xrightarrow{t} m' \Rightarrow ft(m) \leq ft(m') \).

**Proof.** Consider two markings \( m, m' \in \mathbb{M} \) such that \( m \xrightarrow{t} m' \). If there are no enabled transitions in marking \( m' \), then \( ft(m') = \infty \), and thus \( ft(m) \leq ft(m') \).

If some transition \( t' \) is enabled in marking \( m' \), its activator contains either only tokens which were already present in \( m \), or it contains (some) tokens produced by transition \( t \). In the first case, \( ft(m) \leq ft(m') \) by the definition of the firing time applied to marking \( m \) and the definition of transition relation of DTPN. In the second case, the time stamps of the activator produced by \( t \) are at least \( ft(m) \), so the enabling time of \( t' \) is at least \( ft(m) \), implying \( ft(m) \leq ft(m') \).

\( \square \)
As the system is eager, an enabled transition fires instantaneously and a new marking is reached (created) at the firing time of its predecessor marking. We call this the \textit{creation time} of the new marking. The \textit{creation time} of a marking is not unique while the \textit{firing time} is unique. This is because the creation time of a marking depends on the firing sequence that led to the marking. In the Fig. 4.3, we illustrate with an example the creation time of marking \( m'' \) and its relationship to firing sequences leading to marking \( m''' \), starting from marking \( m' \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_3.png}
\caption{Example of Creation Time}
\end{figure}

Closed rational delay intervals on outgoing arcs of transitions of a DTPN implies an infinite number of choices for timestamp of each produced token. In order to reduce infinite delay intervals to finite ones, we introduce the refinement of a DTPN, which is in fact a \textit{refinement} of output delay intervals. Note that input delays are singletons and each singleton is a refinement of itself.

\textbf{Definition 4.3.4 (Refinement of a DTPN).} Consider two DTPNs, \( N_1 = (P,T,F,\delta) \) and \( N_2 = (P,T,F,\delta') \). Then \( N_2 \) is a refinement of \( N_1 \) denoted by \( N_1 \triangleright N_2 \) iff \( \forall (x,y) \in F : \delta(x,y) \triangleleft \delta'(x,y) \).

The refinement of a DTPN might introduce new firing sequences in the system. Due to this property, we are able to show that a refined DTPN can simulate its original DTPN with identical markings.

\textbf{Theorem 4.3.2 (Simulation by refinement).} Consider two DTPN’s \( N_1 \) and \( N_2 \) such that \( N_1 \triangleright N_2 \). Then \( \forall m \in M : (N_2,m) \preceq (N_1,m) \) w.r.t identity relation.

\textit{Proof.} Since a refined DTPN can make the same choices as the original one, it can simulate the original one. The proof is trivial. \qed

Note that trace inclusion is a consequence of the above lemma, i.e. \( \forall m \in M : FS(N_1,m) \subseteq F(N_2,m) \). Furthermore, we have a similar result for the untimed
version $\overline{N}$ of a DTPN $N$: $\forall m \in M : (\overline{N}, \overline{m}) \preceq (N, m) \land FS(N, m) \subseteq FS(\overline{N}, \overline{m})$

where $\overline{m}$ is the untimed version of $m$.

### 4.3.2 Relationship between sDTPN, fDTPN and DTPN

In this section, we will show that for every DTPN there exists a fDTPN that is simulation equivalent and for every fDTPN there exists a sDTPN that is bisimilar.

By the Thm 4.3.2, we have already for a DTPN $N_1$ and a fDTPN $N_2$ such that $N_2 \prec N_1$, $N_1 \preceq N_2$, i.e. $N_1$ simulates $N_2$.

So we must address the two questions: Is there an fDTPN such that $N_2 \preceq N_1$? and for this fDTPN, Is there a sDTPN $N_3$ such that $N_2$ and $N_3$ are bisimilar?

For both these questions the answer is yes.

To prove the first question, we first introduce a function $\varphi$ assigning to a DTPN a fDTPN, called its proxy, which has a finite grid on each delay interval. This is only interesting for output delays, but we define it for all delay intervals. The grid distance is the smallest value such that all the bounds of delay sets and timestamps in the initial marking are multiples of it.

**Definition 4.3.5 (Proxy fDTPN).** A DTPN $N = (P, T, F, \delta)$ with an initial marking $m_0$ has a proxy fDTPN $\varphi(N)$ with the same initial marking such that $\varphi(N) = (P, T, F', \delta')$ where:

- $A = \{ \tau(m(i)) \mid i \in \text{dom}(m_0) \} \cup \{ \min(\delta(x, y), \max(\delta(x, y)) \mid (x, y) \in F \}$

- $\forall (x, y) \in F : \delta'(x, y) = \{ i/d \mid i \in \mathbb{Z} \land \min(\delta(x, y)) \leq i/d \leq \max(\delta(x, y)) \}$, where $d = \min\{ k \in \mathbb{N} \mid \{ k.a | a \in A \} \subseteq \mathbb{N} \}$ and $1/d$ is the grid distance.

**Figure 4.4: An Example of a Round-Off Relation**

Suppose $\delta(x, y) \in [0, 3.5]$

Grid distance: $1/d = 1/2$
The grid distance is the least common multiple of the denominators of the non-zero elements of $A$ expressed as non-reducible elements of $Q$.

Next we introduce the round-off of a marking, which is obtained if the timestamps are round-off to a grid. We give an example in the Fig. 4.4.

**Definition 4.3.6 (Round-off Relation).** For the set of all markings $\mathbb{M}$ of a DTPN: $\forall m, \bar{m} \in \mathbb{M}: m \sim \bar{m}$ iff $\text{dom}(m) = \text{dom}(\bar{m}) \land \forall i \in \text{dom}(m): \tau(m(i)) = \tau(\bar{m}(i))$ and $\forall i \in \text{dom}(m): \exists k \in \mathbb{Z}, d \in \mathbb{N}:

\[
(\tau(m(i)) \in [k/d, k/d + 1/2d] \land \tau(\bar{m}(i)) = k/d) \lor \\
(\tau(m(i)) \in (k/d + 1/2d, (k + 1)/d) \land \tau(\bar{m}(i)) = (k + 1)/d)
\]

As a consequence, the round-off relation preserves the order of timestamps.

**Notation.** We will assert that a marking $\bar{m} \in \mathbb{M}$ is a round-off of a marking $m \in \mathbb{M}$ by the predicate $\text{ro}(m, \bar{m})$.

**Corollary 4.3.3.** Let $N$ be a DTPN with the set of all reachable markings $\mathbb{M}$ from an initial marking $m_0$. Let $\bar{N} = \varphi(N)$ be its proxy $fDTPN$ with the set of all reachable markings $\bar{\mathbb{M}}$ from the same initial marking. If two markings $m \in \mathbb{M}, \bar{m} \in \bar{\mathbb{M}}$ then $\forall i, j \in \text{dom}(m):

\[
\tau(m(i)) = \tau(m(j)) \Rightarrow \tau(\bar{m}(i)) = \tau(\bar{m}(j))
\]

\[
\tau(m(i)) < \tau(m(j)) \Rightarrow \tau(\bar{m}(i)) \leq \tau(\bar{m}(j))
\]

\[
\tau(m(i)) > \tau(m(j)) \Rightarrow \tau(\bar{m}(i)) \geq \tau(\bar{m}(j))
\]

As a consequence of the preservation of the order of timestamps it is possible to show that any DTPN can be simulated by its proxy w.r.t. the round-off relation.

**Theorem 4.3.4 (Simulation by proxy).** Let $N$ be a DTPN with the set of all reachable markings $\mathbb{M}$ from an initial marking $m_0$. Let $\bar{N} = \varphi(N)$ be its proxy $fDTPN$ with the set of all reachable markings $\bar{\mathbb{M}}$ from the same initial marking. Then $(\bar{N}, m_0) \leq (N, m_0)$ with respect to the round-off relation.

**Proof.** Consider two markings $m \in \mathbb{M}, \bar{m} \in \bar{\mathbb{M}}: m \sim \bar{m}$. Suppose $m \xrightarrow{t} m'$. We will prove that (1) $\exists \bar{m}' \in \bar{\mathbb{M}}: \bar{m} \xrightarrow{t} \bar{m}'$ and (2) $m' \sim \bar{m}'$.

(1) Since $m \sim \bar{m}$ by Corollary 4.3.3, the order of timestamps is preserved in marking $\bar{m}$ and hence the same transition $t$ is enabled and $\exists \bar{m}' \in \bar{\mathbb{M}}: \bar{m} \xrightarrow{t} \bar{m}'$.

(2) Since $m \xrightarrow{t} m'$ and $m \sim \bar{m}$, there exists a token with identity $i \in \text{dom}(m)$ such that $\tau(m(i)) + \delta(\pi(m(i)), t) = \text{ft}(m)$ and $\tau(\bar{m}(i)) + \delta(\pi(\bar{m}(i)), t) = \text{ft}(\bar{m})$ and $\text{ro}(\tau(m(i)), \tau(\bar{m}(i)))$. Hence $\text{ro}(\tau(m(i)) + \delta(\pi(m(i)), t), \tau(\bar{m}(i)) + \delta(\pi(m(i)), t))$ holds, i.e. $\text{ro}(\text{ft}(m), \text{ft}(\bar{m}))$. Note that $|\text{ft}(m) - \text{ft}(\bar{m})| \leq 1/2d$. 


Since \( \text{ro}(ft(m), ft(\bar{m})) \), \( \exists k \in \mathbb{Z} : ft(m) \in [k/d, (k+1)/d] \) and we may write \( ft(m) = k/d + v \), where \( 0 \leq v \leq 1/d \).

For the production of tokens (see Def. 4.3.3), each fresh token is assigned a timestamp \( ft(m) + x \), where the value \( x \) is chosen from an associated delay set \([l, u]\). So on the grid, there exists \( i, w \in \mathbb{Z} \) such that \( ft(m) + x = (k+i)/d + w \).

Since \( ft(m) = k/d + v \), we may rewrite as \( x = i/d + w - v \).

We will show that \( \exists \bar{x} \in [l, u] : \text{ro}(ft(m) + x, ft(\bar{m}) + \bar{x}) \).

- Suppose \( ft(m) < ft(\bar{m}) \) and \( 0 \leq w < 1/2d \). Then \( 1/2d < v \leq 1/d \) and \( -1/d \leq w - v < 0 \). Hence \( l \leq (i - 1)/d \leq x < i/d \leq u \). Since \( |ft(m) - ft(\bar{m})| \leq 1/2d \), we may choose \( \bar{x} = (i-1)/d \) and then \( \text{ro}(ft(m) + x, ft(\bar{m}) + \bar{x}) \).

- Suppose \( ft(m) < ft(\bar{m}) \) and \( 1/2d < w < 1/d \). Then \( 1/2d < v \leq 1/d \) and \( -1/2d \leq w - v < 1/2d \). Hence \( l \leq (i - 1)/d + 1/2d < x < i/d + 1/2d \leq u \). Since \( |ft(m) - ft(\bar{m})| \leq 1/2d \), we may choose \( \bar{x} = i/d \) and then \( \text{ro}(ft(m) + x, ft(\bar{m}) + \bar{x}) \).

- Suppose \( ft(m) > ft(\bar{m}) \) and \( 1/2d < w < 1/d \). Then \( 0 < v \leq 1/2d \) and \( 0 < w - v < 1/d \). Hence \( l \leq i/d < x < (i+1)/d \leq u \). Since \( |ft(m) - ft(\bar{m})| \leq 1/2d \), we may choose \( \bar{x} = (i+1)/d \) and then \( \text{ro}(ft(m) + x, ft(\bar{m}) + \bar{x}) \).

- Suppose \( ft(m) > ft(\bar{m}) \) and \( 0 < w \leq 1/2d \). Then \( 0 < v \leq 1/2d \) and \( -1/2d < w - v < 1/2d \). Hence \( l \leq i/d - 1/2d < x < i/d + 1/2d \leq u \). Since \( |ft(m) - ft(\bar{m})| \leq 1/2d \), we may choose \( \bar{x} = i/d \) and then \( \text{ro}(ft(m) + x, ft(\bar{m}) + \bar{x}) \).

Hence \( m' \sim \bar{m}' \).

The opposite is not true, i.e. a DTPN is not simulating its proxy with respect to the round-off relation as illustrated by the example in Fig. 4.5. The grid distance \( 1/d = 1/2 \). From the initial marking with tokens in places \( P_1 \) and \( P_2 \) with zero timestamps, a marking \( m \) with tokens in places \( P_3 \) and \( P_4 \) with timestamps 11/16 and 10/16, respectively is reachable. In the proxy \( fDTPN \), a
marking $\vec{m}$ is reachable such that $m \sim \vec{m}$, i.e. with tokens in places $P3$ and $P4$ with timestamps equal to one. From marking $m$ only transition $b$ is enabled but from marking $\vec{m}$ both transitions $a$ and $b$ are enabled.

However, by Theorem 4.3.2 we know that a DTPN simulates its proxy w.r.t. the identity relation. As a result, we have the following corollary.

**Corollary 4.3.5 (Simulation Equivalence).** A DTPN $N$ and its proxy $fDTPN \bar{N} = \phi(N)$ are simulation equivalent and therefore $FS(N,m_0) = FS(\bar{N},m_0)$.

Next, we prove the second question, i.e. for any fDTPN does there exist a sDTPN such that they are bisimilar?. We do so by constructing a sDTPN, given a fDTPN in the following way: replace every transition of an fDTPN with copies of the transition (i.e. each copy has the same preset and postset as the original one), one for each possible combination of output delays (from finite delay sets on outgoing arcs) on its outgoing arcs. We give an example in the Fig. 4.6. Formally this can be defined using the generalized cartesian product.

**Figure 4.6: Construction of an sDTPN from an fDTPN**

**Definition 4.3.7 (Reduction of fDTPN to sDTPN).** Let $N = (P,T,F,\delta)$ be a fDTPN. For each $t \in T$ let $A_t = \prod_{t \in T} \delta(t,p)$ be the generalized cartesian product of all its delay sets. Then, the corresponding sDTPN is the tuple $\text{construct}(N) = (P,T',F',\delta')$, where

- $T' = \{t_x | t \in T \land x \in A_t\}$
- $F' = \{(p,t_x) | (p,t) \in F \land x \in A_t\} \cup \{(t_x,p) | (t,p) \in F \land x \in A_t\}$
- $\forall p \in P : \forall t \in T : \forall x \in A_t : \delta'(p,t_x) = x(p)$
- $\forall p \in P : \forall t \in T : \forall x \in A_t : \delta'(t_x,p) = x(p)$
Theorem 4.3.6 (Bisimulation of fDTPN and sDTPN). Let $N$ be an arbitrary fDTPN. Then, $N \simeq \text{construct}(N)$.

Proof. (sketch). By the construction of the net $\text{construct}(N)$, for every transition in the net $N$, there exists a set of copies of this transition in the net $\text{construct}(N)$ having the same delay on each input arc and sharing the same preset and postset. So for every transition enabled in a marking of the net $N$, there exists a set of enabled copies of this transition from the same marking in the net $\text{construct}(N)$.

By the definition of generalized cartesian product, for every combination of token timestamps that can be produced by firing a transition in the net $N$, there exists exactly one enabled copy of this transition in the net $\text{construct}(N)$, that is also able to produce the same combination of token timestamps.

So $N$ and $\text{construct}(N)$ are bisimilar w.r.t. identity relation over markings. □

Note that if we extend the definition of a DTPN to outgoing arc delays having open rational intervals then the proxy fDTPN does not preserve the simulation property. So it is essential to include the bounds of an interval while specifying delays in a DTPN. In the Fig. 4.7, we show with an example the violation of the simulation property when bounds are not included.

So we have our result, for any DTPN there exists a proxy fDTPN that is simulation equivalent and for any fDTPN there exists a sDTPN that is bisimilar.

4.3.3 Analysis of sDTPN

Due to the result on the equivalence between DTPN, fDTPN and sDTPN, it is sufficient to consider sDTPN’s for the analysis of DTPN’s. However, as time is non-decreasing, the reachability graph of a sDTPN is usually infinite. But there
is a finite time window that contains all relevant behavior. This is because new tokens obtain a timestamp bounded by a maximum in the future, i.e. the maximum of all maxima of output delays and the timestamps of tokens earlier than the current time minus an upperbound of the input delays are irrelevant, i.e. they can be updated to the current time minus this upper bound. So we can reduce the time frame of a DTPN to a finite time window. This is done by defining a reduction function that maps the timestamps into this window. We introduce a labeled transition system for an sDTPN that is strongly bisimilar to it. Therefore we call this labeled transition system rDTPN, although it formally is not a DTPN.

We denote by $\delta^+_i$ the maximal incoming arc delay, i.e.

$$\delta^+_i = \max\{\delta(p,t)|(p,t) \in (P \times T) \cap F\}$$

and by $\delta^+_o$ the maximal outgoing arc delay, i.e.

$$\delta^+_o = \max\{\delta(t,p)|(t,p) \in (T \times P) \cap F\}$$

A reduced marking is obtained by subtracting the firing time from the timestamp of each token in the marking of a sDTPN, with a lower bound of maximum incoming arc delay $-\delta^+_i$. We define this time reduction as a reduction function.

Definition 4.3.8 (Reduction function). Consider a sDTPN with the set of all markings $M$. The reduction function $\alpha : M \rightarrow M$ is defined $\forall m \in M : dom(\alpha(m)) = dom(m)$ as

$$\forall i \in dom(m) : \pi(\alpha(m)(i)) = \pi(m(i)) \wedge \tau(\alpha(m)(i)) = \max\{-\delta^+_i, \tau(m(i)) - ft(m)\}$$

The set of all reduced markings of a sDTPN is defined as

$$\overline{M} = \{m \in M | \alpha(m) = m\}$$

Note that the reduction function is either (1) reducing the timestamp of tokens by the firing time, or (b) mapping timestamps less than or equal to $-\delta^+_i$ to $-\delta^+_i$.

Corollary 4.3.7. Let $N$ be a sDTPN with the set of all markings $M$. Then $\forall m \in M, t \in T :$

- $\forall i \in dom(m) : \tau(\alpha(m)(i)) \geq \tau(m(i)) - ft(m)$
- $A(\alpha(m), t) = \{a | a \in A(m, t)\}$

The firing time of a reduced marking is zero.

Lemma 4.3.8. Consider a sDTPN with the set of all markings $M$. Then $\forall m \in M : ft(\alpha(m)) = 0$. 
Corollary 4.3.9. Let

\[ \text{Lemma 4.3.10. Consider a sDTPN with the set of markings } \mathbb{M}. \text{ Let } m, m' \in \mathbb{M}: m \xrightarrow{t} m'. \text{ Then } \exists \tilde{m} \in \mathbb{M}, \tilde{a} \in A(\alpha(m), t): \alpha(m) \xrightarrow{\tilde{a}} \tilde{m} \text{ and } \text{ft}(\tilde{m}) = \text{ft}(m') - \text{ft}(m) \text{ and } \alpha(m') = \alpha(\tilde{m}). \]

Proof. Note that \( \alpha(m) \in \mathbb{M} \).

\[
\text{ft}(\alpha(m)) = \min_{t \in T} \min_{a \in A(\alpha(m), t)} \max_{i \in \text{dom}(a)} \{ \tau(\tilde{a}(i)) + \delta(\tilde{a}(i)), t) \}
\]

By Def. 4.3.8 and Corollary 4.3.7, we may write

\[
= \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \max\{ -\delta^T_i, \tau(a(i)) - \text{ft}(m) \} + \delta(\pi(a(i)), t) \}
\]

\[
= \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \max\{ -\delta^T_i + \delta(\pi(a(i)), t), \tau(a(i)) - \text{ft}(m) + \delta(\pi(a(i)), t) \} \}
\]

\[
= \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \max\{ -\delta^T_i + \delta(\pi(a(i)), t), \tau(a(i)) + \delta(\pi(a(i)), t) \} - \text{ft}(m) \}
\]

By definition of \( \delta^T_i \) the term

\[
\max_{i \in \text{dom}(a)} \{ -\delta^T_i + \text{ft}(m) + \delta(\pi(a(i)), t) \} \leq \text{ft}(m)
\]

and for all \( t \in T \) and \( a \in A(m, t) \) the term

\[
\max_{i \in \text{dom}(a)} \{ \tau(a(i)) + \delta(\pi(a(i)), t) \} \geq \text{ft}(m)
\]

So we may write

\[
= \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \max_{i \in \text{dom}(a)} \{ \tau(a(i)) + \delta(\pi(a(i)), t) \} - \text{ft}(m) \} = 0
\]

As a consequence of Lemma 4.3.8, the reduction function \( \alpha \) is idempotent.

Corollary 4.3.9. Let \( N \) be a sDTPN. Then \( \forall m \in \mathbb{M} : \alpha(\alpha(m)) = \alpha(m) \).

We will establish an important result that relates the behavior of a marking and its reduced marking. Given a marking with an enabled transition, the same transition is also enabled in its reduced marking and the new marking created by firing this transition from both these markings have the same reduced marking. We call the marking created by firing the enabled transition of a reduced marking as an intermediate marking. We illustrate this relationship in the Fig. 4.8.
The above lemma can be extended to compute the time taken to reach a marking by an executable firing sequence of a sDTPN, using only its intermediate markings.

**Corollary 4.3.11.** Let $N$ be an sDTPN and $(N, M, \rightarrow, m_0)$ be its timed transition system (i.e. labeled transition system). Let $n \in \mathbb{N} : n > 0$ and markings $m_0, m_1 \ldots m_n \in M$. Let $\sigma = (t_1; \ldots; t_n)$ be an executable firing sequence of net $N$ such that $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} m_n$. Then there exist markings $\tilde{m}_0, \tilde{m}_1, \ldots, \tilde{m}_n \in M : \tilde{m}_0 = m_0 \land \forall i \in [1, n] : \alpha(m_{j-1}) \xrightarrow{t_j} \tilde{m}_j$ and

$$ft(m_n) = \sum_{j=0}^{n} ft(\tilde{m}_j)$$

The relationship between two reduced markings, one reachable from the other, is given by the reduced transition relation. An rDTPN is a transition system over reduced markings of a sDTPN, related by the reduced transition relation.

**Definition 4.3.9 (Reduced transition relation, rDTPN).** Let $N = (P, T, F, \delta)$ be an sDTPN with the set of all reachable markings $\mathbb{M}$ and the set of all reduced markings $\overline{\mathbb{M}}$. Its reduced DTPN called rDTPN is a labeled transition system $\rho(N) = (\overline{\mathbb{M}}, T, \xrightarrow{\sim}, \overline{m}_0)$, where $\sim \subseteq \overline{\mathbb{M}} \times T \times \overline{\mathbb{M}}$ is the reduced transition relation defined as $\forall \overline{m}, \overline{m}' \in \overline{\mathbb{M}} : \overline{m} \xrightarrow{\overline{t}} \overline{m}' \iff \exists \tilde{m} \in \mathbb{M} : \tilde{m} \in A(\tilde{m}, t) \land \tilde{m} \xrightarrow{t} \tilde{m}' \land \alpha(\tilde{m}) = \overline{m}'$ and $\overline{m}_0 = \alpha(m_0)$ is an initial marking.

For a given sDTPN, its timed labeled transition system and reduced DTPN are strongly bisimilar w.r.t. reduction relation. Due to Lemma 4.3.10, the time relation in the bisimulation is implicit.

**Theorem 4.3.12 (Bisimulation sDTPN and rDTPN).** Consider an sDTPN with its timed transition system $N = (\mathbb{M}, T, \rightarrow, m_0)$ and its rDTPN $\rho(N) = (\overline{\mathbb{M}}, T, \xrightarrow{\sim}, \overline{m}_0)$. Then $(N, m_0) \sim (\overline{N}, \overline{m}_0)$. 

---

Figure 4.8: Markings reachable from an abstract marking
Proof. Let $R \subseteq M \times \overline{M}$ be a relation defined as
\[
\forall m \in M, \overline{m} \in \overline{M} : (m, \overline{m}) \in R \Leftrightarrow \alpha(m) = \overline{m}
\]

1. Let $m \overset{t}{\rightarrow} m'$ and $(m, \alpha(m)) \in R$. Then, by Lemma 4.3.10, a marking $\overline{m} \in \overline{M}$ exists such that $\alpha(m) \overset{t}{\rightarrow} \overline{m}$ and $\alpha(m') = \alpha(\overline{m})$. By the Def. 4.3.9, we get $\alpha(m) \overset{t}{\rightarrow} \alpha(m')$.

2. Let $\overline{m} \overset{t}{\rightarrow} \overline{m}'$ and $(m, \overline{m}) \in R$, i.e. $\alpha(m) = \overline{m}$. By the Def. 4.3.9, there exists a marking $\overline{m} \in \overline{M} : m \overset{t}{\rightarrow} \overline{m}$ and $\alpha(\overline{m}) = \overline{m}'$. We will first prove that transition $t$ is enabled in marking $m$. Note that since $\alpha(m) = \overline{m}$, we have by Corollary 4.3.7 that an activator $a \in M$ exists such that $\alpha(a) = \overline{a}$. Hence,
\[
\text{ft}(\alpha(m)) = \max_{i \in \text{dom}(\overline{a})} \{\tau(\overline{a}(i)) + \delta(\pi(\overline{a}(i)), t)\}
\]
\[
0 = \max_{i \in \text{dom}(a)} \{\max\{\tau(a(i)) - \text{ft}(m), -\delta^\uparrow_i\} + \delta(\pi(a(i)), t)\}
\]
\[
0 = \max_{i \in \text{dom}(a)} \{\max\{\tau(a(i)) + \delta(\pi(a(i)), t),
- \delta^\uparrow_i + \text{ft}(m) + \delta(\pi(a(i)), t)\}\} - \text{ft}(m)
\]
Since $-\delta^\uparrow_i + \text{ft}(m) + \delta(\pi(a(i)), t) \leq \text{ft}(m)$ and for all $a \in A(m, t)$ the term
\[
\max_{i \in \text{dom}(a)} \tau(a(i)) + \delta(\pi(a(i)), t) \geq \text{ft}(m)
\]
Hence,
\[
\text{ft}(m) = \max_{i \in \text{dom}(a)} \{\tau(a(i)) + \delta(\pi(a(i)), t)\}
\]
So transition $t$ is enabled in marking $m$ and a marking $m' \in M$ exists such that $m \overset{t}{\rightarrow} m'$. By Lemma 4.3.10, we know $\alpha(\overline{m}) = \alpha(m')$. Hence we have proven $\alpha(m') = \overline{m}'$.

\[\square\]

4.3.4 Finiteness of rDTPN

The number of different timestamps in the reachability graph of the rDTPN is finite. This is observed in several papers (see [46]). To see this we consider first only timestamps in $\mathbb{Z}$. Since we have finitely many of them in the initial marking and the only operations we execute on them are: (1) selection of the maximum, (2) adding one of a finite set of delays and (3) subtracting the selected timestamp
with a minimum. So the upper bound is the maximal output delay and the lower bound is zero minus the maximal input delay. Hence, we have a finite interval of \( \mathbb{Z} \) which means finitely many values for all markings. In case we have delays in \( \mathbb{Q} \) we multiply with the lcm of all relevant denominators, like in Def. 4.3.6 and then we are in the former case.

**Theorem 4.3.13.** The set of timestamps in the reachability graph of a rDTPN is finite, and so if the underlying Petri net is bounded, the reachability graph of the rDTPN is finite.

**Proof.** Consider the first case where all time stamps and delays are in \( \mathbb{Z} \). Note that all operations we perform are:

- taking the maxima or minima of time stamps, i.e. selecting one of them.
- adding one of the finite set of delays.
- subtracting a selected time stamp from others with a min. delay of \(-\delta_i^+\).

With induction we will show that all time stamps stay in \([-\delta_i^+ , e]\), where \(e\) is the maximum of \(\delta_1^+, \delta_2^+, \ldots\) and the maximal time stamp is in the initial marking. (We assumed in the initial marking there are no negative time stamps).

So in the initial marking all time stamps are in \([0, e]\). Suppose in an arbitrary marking \(m\) they are in \([-\delta_i^+ , e]\), then we compute \(ft(m)\) which is the selection of one of the time stamp, possibly increased with a delay. This time is subtracted from the remaining time stamps but with a lower bound of \(-\delta_i^+\). So they stay in \([-\delta_i^+ , e]\). Now consider the newly produced tokens. They have a time stamp in the interval \([0, \delta_1^+]\). So all time stamps are in \([-\delta_1^+ , e]\). So we have finitely different time stamps.

Now consider the case where the initial time stamps and the delays are in \(\mathbb{Q}\). This is a finite set, so the set of denominators has a least common multiple say \(l\).
Figure 4.10: Applying the Reduction Method
Now we may multiply all time stamps of the initial marking and all delays with \( l \) to obtain natural numerators with a common denominator \( l \). In all operations we may extract division by \( l \), for e.g. \( \text{max}(\frac{a}{l}, \frac{b}{l}) = \frac{\text{max}(a, b)}{l} \) or \( \frac{a}{l} - \frac{b}{l} = \frac{a-b}{l} \). So we have in fact only computations with integers between two bounds. Only at the end of a computation we divide by \( l \). So we are in the former case: a finite set of time stamps and delays; say \( E \).

Now consider the underlying Petri net system \((P, T, F, \bar{m}_0)\), where \( \bar{m}_0 \) is derived from \( m_0 \) by deleting the time stamps. The maximal number of tokens in all reachable markings is bounded by some \( b \in \mathbb{N} \). Since the firing rules of transitions in DTPN’s are restrictions, the set of reachable markings in the reduced DTPN’s can never have more tokens than the underlying Petri net system. Hence it is bounded. Since the number of possible time stamps is bounded \( \forall m \in M : |\{i \in \text{dom}(m) | \tau(m(i)) \in E\}| = |\text{dom}(\bar{m})| \leq b. \)

Using Corollary 4.3.11, given a path in the reachability graph of a rDTPN, we are able to compute the time required to execute this path in the original DTPN.

### 4.3.5 The Reduction Method

The results of Section 4.3.2 and Section 4.3.3 (see Fig. 4.9) can be used to define a reduction method that transforms a given DTPN into its rDTPN. There are two main steps in this method: interval reduction and time reduction. The interval reduction step transforms a DTPN into its proxy fDTPN, followed by a transformation to sDTPN by multiplying transitions. The transformation from a DTPN to its proxy fDTPN is carried out by replacing time intervals on each output arc of transitions by a finite set of equidistant values (separated by grid distance) from the interval. The time reduction step transforms the timed transition system of a sDTPN into a finite rDTPN using the reduction function. Due to the Theorem 4.3.13, only a finite number of applications of the reduction function on markings reachable from the initial marking of a sDTPN are necessary to derive its rDTPN. We give an example of an application of this method in the Fig. 4.10. Note that for practical applications by software architects, this method must be automated.

### 4.4 Discrete Stochastic Petri Nets

In this section, we endow the transition firings of a DTPN with probabilities. This is useful for analyzing the reliability of a designed system. We do this only for DTPN with (In-single, Out-fint) and we assign a probability distribution to these intervals. Additionally, we should have a stochastic mechanism to choose a transition from all enabled transitions. We do this by assigning a non-negative weight to all transitions and draw an enabled transition \( x \) with probability \( w(x)/\sum_{y:\text{enabled}} w(y) \). If there are no priorities for transitions, we may
choose all weights to be equal. Note that we could introduce stochastics for the two other main classes, $M_1$ and $M_2$ in a similar way. For the class $M_1$, we first select a classically enabled transition with the weights and then a duration from a finite distribution associated with the transition. For the class $M_2$, we select for all classical enabled transitions a delay from a finite distribution associated with the transition and then we select with the weights one of them having the minimal delay.

**Definition 4.4.1 (DSPN).**

A DSPN is a 6-tuple $(P,T,F,\delta,w,\phi)$, where

- $(P,T,F,\delta)$ is a DTPN with (In-single, Out-fint).
- $w : T \rightarrow \mathbb{R}^+$ a weight function, used to choose one of the simultaneously enabled transitions,
- $\phi$ is a function with domain $F \cap T \times P$ and for $\forall (t,p) \in F$:
  \[
  \phi(t,p) : \delta(t,p) \rightarrow [0,1] \text{ such that } \sum_{x \in \delta(t,p)} \phi(t,p) = 1, \text{ so } \phi(t,p) \text{ assigns probabilities to } \delta(t,p).
  \]

For analysis, we consider two transformations to derive a Markov chain for a DSPN. The first transformation is given in Def. 4.3.7, where for each value of a finite output delay interval, a transition is introduced with a one point output delay, as in Fig. 4.6. Here transition $t$ has two output arcs with delay sets, one with $\{2,5\}$ and the other $\{3,6\}$. Let the probabilities of these intervals be $(p_1, p_2)$, with $p_1 + p_2 = 1$ and $(q_1, q_2)$ with $q_1 + q_2 = 1$. So $w(t_1) = w(t).p_1.q_1, w(t_2) = w(t).p_1.q_2, w(t_3) = w(t).p_2.q_1,$ and $w(t_4) = w(t).p_2.q_2$.

This transformation blows up the model and gives unreadable pictures, but it is only for automatic processing. Now we have a model of type sDTPN and we can forget the probabilities $\phi(...)$ because all output delays are singletons. So we only have to deal with the weight function $w$. By Theorem 4.3.6, we know that this model is strongly bisimilar (discarding the probabilities) with the original one so we can deal with this one. It is obvious by the construction that the probabilities over the delays of produced tokens are the same as well. So after these transformations we can consider a DSPN as a 5-tuple $(P,T,F,\delta,w)$. We illustrate this concept in the Fig. 4.11.

The next transformation concerns this sDTPN model into the reduced labeled transition system (rDTPN) as in Theorem 4.3.12 which is strongly bisimilar with the sDTPN model. The weights can be transferred to this rDTPN model because the underlying Petri net has not changed. We call this new model class rDSPN. Remember that if the underlying Petri net is bounded, then rDSPN has a finite reachability graph.

We will now add two values to an arc in the reachability graph of the rDSPN, representing a transition $m, m' \in M : m \xrightarrow{} m'$: (1) probability of choosing this arc and (2) the sojourn time in a marking $m'$ if coming from $m$. Remember that
for each marking the firing time is uniquely determined, but the sojourn time depends on the former marking. The sojourn time can be computed during the reduction process as expressed by Lemma 4.3.10.

**Definition 4.4.2 (Transition probability and sojourn time).**

The transition probability $Q : \mathbb{M} \times \mathbb{M} \rightarrow [0, 1]$ satisfies:

$$Q_{m,m'} = \sum_{x : m \xrightarrow{\omega} m'} w(x) / \sum_{y : \exists m'' : m \xrightarrow{\omega} m''} w(y).$$

For $m, m' \in \mathbb{M} : r(m, m') = \text{ft}(m') - \text{ft}(m)$ is the sojourn time in marking $m'$ if coming from $m$.

The transition probability contains all information of the reachability graph. Finally we are able to define the Markov chain that is determined by the reachability graph of the rDSPN endowed with the transition probabilities.

**Definition 4.4.3 (Markov chain).**

Let a rDSPN $(P, T, F, \delta, w)$ be given and let the $Q$ be the transition probability over the state space. Then the Markov chain of the rDSPN is a sequence of random variables $\{X_n | n = 0, 1, \ldots\}$, where $X_0 = m_0$ the initial marking and $X_n$ is marking after $n$ steps, such that:

$$P[X_{n+1} = m' | X_n = m, X_{n-1} = m_{n-1}, \ldots, X_0 = m_0] = Q(m, m')$$

for arbitrary $m_0, \ldots, m_{n-1} \in \mathbb{M}$. 

**Figure 4.11: Computing the Weight Function**
The Markov property is implied by the fact that only the last marking before the transition firing is taken into account. Since a marking and an enabled transition determine uniquely the next state, we can also consider another stochastic process \( \{Y_n| n \in \mathbb{N}\} \), where \( Y_n \in T \), which is a stochastic firing sequence. For a firing sequence \( \sigma = (t_1, ..., t_n) \) with \( m_0 \xrightarrow{t_1} m_1, ..., \xrightarrow{t_n} m_n \) we have

\[
P[Y_1 = t_1, ..., Y_n = t_n|X_0 = m_0] = P[X_1 = m_1, ..., X_n = m_n|X_0 = m_0].
\]

So we can compute the probability for each finite firing sequence.

Markov chains are often endowed with a cost structure which is a function assigning to a pair of successive markings a real value, called cost function. Then we can express the total expected cost when starting in marking \( m \) as:

\[
E[\sum_{n=0}^{N} c(X_n, X_{n+1})|X_0 = m].
\]

Here, \( N \in \mathbb{N} \) or \( N = \infty \). In particular we will use the sojourn times as “cost”. In fact we may associate a semi-Markov process to rDSPN, because the sojourn times themselves are random variables, but in our case they are completely determined if we know the former state. The Markov chain \( \{X_n| n = 0, 1, \ldots\} \) is then the embedded Markov chain of the semi-Markov process [286]. We will use the function \( v : M \rightarrow \mathbb{R} \), which is usually called the value function for Markov processes (see [275]): \( v(m) = E[\sum_{n=0}^{N} c(X_n, X_{n+1})|X_0 = m] \) for cost functions \( c \).

We will use the Markov chain to answer three types of important questions. We will use the cost function to express the questions and we use the Markov property to translate our questions into Bellman equations (see [286] and [275]).

- Probability of reaching a subset,
- Expected time to leave a subset,
- Expected sojourn times in equilibrium.

Note that all these questions concern sequences of markings or equivalently firing sequences. So they belong to LTL (see [273]).

**Probability of reaching a subset**

Let \( A, B \subseteq M \) be a subsets of the state space, \( A \cap B = \emptyset \). We are interested in the probability of reaching \( A \) from \( B \). Here we choose \( c(m, m') = 1 \) if \( m \in B \wedge m' \in A \) and \( c(m, m') = 0 \) otherwise. Further we stop as soon as we reach \( A \). Then

\[
\forall m \in B : v(m) = \sum_{m' \in A} Q_{m,m'} + \sum_{m' \in B} Q_{m,m'} v(m').
\]

If \( B \) is finite, this can be computed, even if the underlying Petri net is unbounded. For example if \( B \) is the set of \( k \)-bounded markings (i.e. markings with at most \( k \)
Expected time to leave a subset $A$ of the state space. Here we use a cost function $c(m, m') = r(m, m')$, the sojourn time in $m'$, if $m, m' \in A$ and $c(m, m') = 0$ elsewhere. If $m \in A$ then

$$v(m) = \sum_{m' \in A} Q_{m,m'}(r(m, m') + v(m')).$$

This can be computed even if the underlying Petri net is unbounded but $A$ is finite.

Expected sojourn times in equilibrium. We restrict us to the case where the underlying Petri net is bounded. $\mathbb{P}[X_n = m'|X_0 = m] = Q^n(m, m')$ where $Q^n$ is the $Q$ to power $n$. The limit of averages exists: $\pi(m') := \lim_{N \to \infty} \sum_{n=0}^{N} Q_{m,m'}^n$ and it satisfies

$$\pi_{m_0}(m') = \sum_{m \in M} \pi_{m_0}(m).Q_{m,m'}.$$

We now assume the system is a strongly connected component (i.e. the Markov chain is irreducible), which implies that the limit distribution $\pi$ is independent of the initial state $m_0$. Further, we know that the expected time spent in a marking $m$ depends on the former marking, so the expected time of being in marking $m'$ is:

$$\sum_{m \in M} r(m, m').\mathbb{P}[X_{n-1} = m|X_n = m'],$$

which can be rewritten using Bayes rule to:

$$\sum_{m \in M} r(m, m').\mathbb{P}[X_n = m'|X_{n-1} = m].\mathbb{P}[X_{n-1} = m]/\mathbb{P}[X_n = m'].$$

Thus, the expected sojourn time in some arbitrary marking is obtained by multiplying with $\mathbb{P}[X_n = m']$

$$\sum_{m \in M} r(m, m').Q_{m,m'}.\mathbb{P}[X_{n-1} = m].$$

This formula could also be derived immediately as the expected sojourn time in the next marking. For $n \to \infty$, this converges either by the normal limit or limit of averages to:

$$\sum_{m \in M} r(m, m').Q_{m,m'}.\pi(m).$$

If we want to solve these equations using matrix calculations, we need to compute the transition matrix of the reachability graph. However, we can also use the method of successive approximations to approximate these values in an
iterative way using only two functions (vectors) over the state space. As an example, the probability of reaching a set $A$ from a set $B$ we set: $\forall m \in B : v_0(m) = 0$ and
\[
\forall m \in B : v_{n+1}(m) = \sum_{m' \in A} Q_{m,m'} + \sum_{m' \in B} Q_{m,m'} v_n(m').
\]

According to [6] we can derive for specially structured workflow nets the distribution of the throughput time of a case (i.e. the time a token needs to go from the initial to the final place) analytically in case of DTPN with (In-zero, Out-finet). Models of this class can be built by transition refinement, using the patterns displayed in Fig. 4.12. Pattern 1 is a sequence construction. Pattern 2 is an iteration where we have arc weights $q$ and $1 - q$ for the probability of continuing or ending the loop. Pattern 3 is the parallel construction. Pattern 4 is the choice, which has also arc weights $q$ and $1 - q$ representing the probabilities for the choices. In Fig. 4.12 the intervals $[a, b], [c, d]$ indicate the finite probability distributions. In order to be a model of this class, it must be possible to construct it as follows. We start with an initial net and we may replace all transitions $t$ with $|t| = 1$ using one of the four rules. There should be a proper parse tree for a net of this class. We associate to all transitions with output delay sets a random variable; for the initial net the random variable $U$ with distribution on $[a, b]$ and similarly random variable $Y$ and $Z$ for the patterns.

If we have such a net, we can apply the rules in the reversed order. If we have at some stage a subnet satisfying to one of the four patterns, with the finite distributions as indicated, we can replace it by an initial subnet with a “suitable” distribution on the output delay interval. For the initial subnet we have a random output variable $U$. For the sequential construction (rule 1) we have two independent random variables $Y$ and $Z$ with discrete distributions on $[a, b]$ and $[c, d]$ respectively. So $U = Y + Z$ and the distribution of $U$ is the convolution of the distributions of $Y$ and $Z$. For the parallel construction (rule 3) we have $U = \max(Y, Z)$ which is the product distribution, i.e. $P[U \leq x] = P[Y \leq x] P[Z \leq x]$. For the choice (rule 4) it is a mixture of two distributions, $P[U \leq x] = P[Y \leq x] q + P[Z \leq x] (1 - q)$. The most difficult one is the iteration (rule 2), since here we have the distribution of $U := \sum_{n=0}^N (Y_n + Z_n)$ where $N$ is a geometrically distributed random variable with distribution $P[N = n] = q^n (1 - q)$ indicating the number of iterations and $Y_n$ and $Z_n$ are random variables from the distributions on $[a, b]$ and $[c, d]$ respectively. All these random variables are independent. The distribution of $U$ can be derived using the Fourier transform (see [6]). This is an approximation, since the domain of $U$ is infinite, even if $Y_n$ and $Z_n$ have finite domains. However, we can cut the infinite domain with a controllable error.

Thus, we are able to reduce a complex DSPN. This method is only applicable if the original net is safe, otherwise different cases can influence each other and so the independency assumptions are violated.
4.5 Conclusions

In this chapter we reviewed the most studied Petri nets with time and stochastics. There are many model classes that have been proposed and it is difficult for a model engineer to understand the differences and to decide the strong and weak points of each model class. Some are strong for theoretical purposes others for modeling. One class did not get much attention in literature: the class with timestamps for tokens. We call this class DTPN and showed how we can analyse this class with model checking, in case the underlying Petri net is bounded. The DTPN class can easily be extended to deal with stochastics as we have shown. Here we have the advantage that we can apply arbitrary finite distributions which has the advantage above the GSPN model that requires exponential distributions only. The analysis of stochastic behavior is based on Markov chains and so it is similar to the approach in GSPN.
# CHAPTER 5

## EXPRESSIVENESS OF DTPN

In this chapter, we compare model class DTPN with other existing models of time like timed automata and timed extensions of Petri nets and show how they can be unified by transformations from one into the other. As a consequence, the advantages of each formalism, either in the ease of modeling, ease of verification or availability of mature modeling tools, can be exploited. By expressing inhibitor arcs using subclasses of DTPN, we prove the Turing completeness of model class DTPN. For these subclasses, we also analyze their modeling comfort.

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction .................................. 98</td>
</tr>
<tr>
<td>5.2 Relationship to Timed Automata ............... 99</td>
</tr>
<tr>
<td>5.2.1 Timed Automata ................................. 99</td>
</tr>
<tr>
<td>5.2.2 Clock DTPN .................................... 101</td>
</tr>
<tr>
<td>5.2.3 cDTPN and its Timed Automaton .............. 103</td>
</tr>
</tbody>
</table>
| 5.2.4 Relationship between cDTPN and its Timed Auto-
  maton .............................................. 106 |
| 5.3 Expressiveness of DTPN ......................... 111 |
| 5.3.1 Expressing inhibitor nets with sDTPN ........ 112 |
| 5.3.2 Modeling ODO with IDO ....................... 119 |
| 5.3.3 Modeling IDO as ODO ......................... 121 |
| 5.3.4 Simulating timed Petri nets and time Petri nets .. 124 |
| 5.4 Conclusions .................................... 127 |
5.1 Introduction

The rapid developments in the field of embedded platforms where safety, performance and reliability depend to a large extent on timing characteristics has motivated a lot of interest in development of formal modeling and analysis techniques that have a notion of time. Over the last 40 years, many models of time have been proposed. Most of these models were suggested as timed extensions of well known untimed models like automata and Petri nets. The most studied models in literature include networks of timed automata (extends the model of communicating finite state machines with a finite number of real valued clocks) and timed extensions of Petri nets (with time constraints on places, transitions and arcs) like Merlin’s time Petri nets, timed Petri nets, timed arc Petri nets and interval timed Petri nets. For both time and timed Petri nets many extensions have been proposed that consider time relative to places (P-Timed/Time Petri nets), transitions (T-Timed/Time Petri nets) and arcs (A-Timed/Time Petri nets). Both P-Timed Petri nets and T-Timed Petri nets are expressively equivalent \cite{295, 272} and these two classes are included in T-Time Petri net and P-Time Petri net \cite{272} but themselves are incomparable \cite{185}.

Most of these models were developed more or less independently of each other. Each model has its own advantages either in terms of modeling, analysis or the availability of tools. The recent availability of efficient verification tools like UPPAAL for timed automata, has generated a lot of interest in transformations between the different models of time in Petri nets (especially time Petri nets) and timed automata, c.f. from timed automata to time Petri nets \cite{152, 39, 50}, from time Petri nets to timed automata \cite{152, 212, 104, 100, 298}, from timed arc Petri nets to timed automata \cite{307}, from time Petri nets to timed arc Petri nets \cite{90, 72} and others \cite{20, 214, 88, 70}. Most of these papers refer to timed automata with clocks as incorporated in the UPPAAL-toolset \cite{356} described in \cite{33}. Remark that in \cite{40}, it was shown that a bounded Merlin time Petri net is a strict subset of timed automata w.r.t timed bisimulation but equally expressive w.r.t timed language equivalence.

In this chapter, we will first compare model class DTPN and timed automata as incorporated in the UPPAAL toolset. We identify subclasses of DTPN that are branching bisimilar and trace equivalent with a timed automata. As a result, efficient verification techniques already developed for the UPPAAL toolset can be exploited. Next, we show that model class DTPN is Turing complete by expressing inhibitor arcs using two sub-classes of DTPN IDO \textit{(in-single, out-zero)} and ODO \textit{(in-zero, out-single)}. By simulating one with the other, we show that both these subclasses have the same expressive power but not the same modeling comfort, i.e. the ease of modeling. Finally, we also show how time Petri net and timed Petri net can be expressed using a DTPN. Remark that for a sub class of DTPN, the popular CPN tools can be used for modeling and analysis.

This chapter is structured as follows: In the Section 5.2, we present the relationship between subclasses of DTPN and timed automata. In the Section 5.3,
we show by modeling inhibitor arcs, that two subclasses of DTPN are Turing complete. We also show how we can transform models from one subclass into the other and how model classes $M_1$ and $M_2$ can be expressed using model class DTPN. Furthermore, we discuss their weaknesses and their strength from the modeling point of view, without claiming to be complete. In the Section 5.4, we present our conclusions.

5.2 Relationship to Timed Automata

There exist many tools to model and verify properties of a timed system, most notably UPPAAL. In UPPAAL, a timed system is modeled as a timed automaton but whose clock valuations range over natural numbers.

In this section, we will show how a DTPN can be expressed as a timed automaton. This gives the software architect, the possibility to use the freely available UPPAAL toolset for verification of system behavior. However, not all DTPN’s can be expressed as a timed automaton. In fact, only a subclass of DTPN’s that are safe and whose synchronizing transitions have the same delay on each incoming arc (as in the Merlin time model (Tr-equal)), can be expressed as a timed automata. Note that we have only trace equivalence and not branching bisimilarity, since DTPN’s make choices of token timestamps (for tokens being produced) earlier than timed automata by specifying timestamps in the future.

In order to make the transformation to UPPAAL understandable, we introduce a sub-class of DTPN called a clock DTPN (cDTPN). We will show that an arbitrary DTPN satisfying the delay requirement on synchronizing transitions can be transformed into a cDTPN by the arc refinement operation and by adding a set of marked places called timer places connected with bi-directional arcs to a subset of transitions. Note that the criteria for safe nets can be relaxed but we do not consider it.

We will first introduce the syntax and semantics of a timed automaton.

5.2.1 Timed Automata

A Timed Automaton (c.f. [33]), is a finite automaton (a graph consisting of a finite set of locations and labeled edges) extended with a set of clocks modeled as real-valued variables. All clocks of the system are initialized to zero and then increase synchronously at the same rate. The behavior of a timed automaton is restricted by adding constraints on clock variables (guards on edges). Location invariants are used to force an automata to progress, i.e. leave the location before the invariant becomes false. An edge $e$ between a location $l_1$ and $l_2$ is denoted by $e = (l_1, C, t, R, l_2)$, where $C$ is specifying the clock constraint that must be satisfied for the edge to be enabled, $t$ is the label associated with edge $e$ and $R$ is the set of clocks to reset when this edge is taken.
Definition 5.2.1 (Clock Constraints). Consider a finite set \( C \) of real valued variables. A clock constraint is a conjunction of predicates of the form \( x \triangleleft n \) or \( x - y \triangleleft n \) for \( x, y \in C, \triangleleft \in \{\leq, <, =, >, \geq\} \) and \( n \in \mathbb{N} \). The set of clock constraints is denoted by \( B(C) \).

Definition 5.2.2 (Syntax of a Timed Automaton). A timed automaton is a tuple \((L, T, C, E, I)\), where \( L \) is a finite set of locations, \( T \) is a finite set of transition labels, \( C \) is a finite set of clocks, \( E \subseteq L \times B(C) \times T \times \mathcal{P}(C) \times L \) is the set of edges, and \( I : L \rightarrow B(C) \) assigns clock constraints to locations called invariants. We restrict location invariants to clock constraints of the form \( x \leq n \) or \( x < n \), where \( x \in C \) and \( n \in \mathbb{N} \).

A location along with a valuation of all clocks of the system defines a state of a timed automaton.

Definition 5.2.3 (State). Let \( A = (L, T, C, E, I) \) be a timed automaton. The state of a timed automaton is a pair \((l, v)\), where \( l \in L \) and \( v : C \rightarrow \mathbb{R}_{\geq 0} \) is a valuation function that assigns to each clock a non-negative real number. If the valuation of all clocks satisfies a clock constraint \( g \in B(C) \), then we write \( v \in g \).

The set of all states of a timed automaton is denoted by \( S \) and the initial state by \( s_0 = (l_0, v_0) \) satisfying \( \forall c \in C : v_0(c) = 0 \).

The state of a timed automaton changes when an enabled transition is fired. There are two types of transitions between states. A delay transition does not cause a change in location but allows the time in clocks to increase. A discrete transition causes a change in location if the clock valuations satisfy the guard condition associated with an edge corresponding the discrete transition. Once a discrete transition has fired, a subset of clocks will be reset to zero.

Definition 5.2.4 (Semantics of a Timed Automaton). The semantics of a timed automaton \( A = (L, T, C, E, I) \) is defined as a transition system having two types of transitions between states defined in the following way.

- A delay transition relation \( d \vdash \subseteq S \times \mathbb{N} \times S \) is defined as \( \forall (l, v) \in S, d \in \mathbb{N} : (l, v) \xrightarrow{d} (l, v') \) if \( \forall c \in C : v'(c) = v(c) + d \) and \( v \in I(l) \) and \( v(c) + d' \in I(l) \) for all \( d' \leq d \).

- A discrete transition relation \( \rightarrow \subseteq S \times T \times S \) is defined as \( \forall (l, v), (l', v') \in S, t \in T : (l, v) \xrightarrow{t} (l', v') \) if \( \exists (l, g, t, y, l') \in E, v' \in I(l'), v \in g \) and \( v'(x) = 0 \), if \( x \in y \) and \( v'(x) = v(x) \), otherwise.

Note that \( (l, v) \xrightarrow{d} (l, v') \xrightarrow{d'} (l, v'') \) implies \( (l, v) \xrightarrow{d+d'} (l, v'') \).

Next, we will define a sub class of DTPN and call it clock DTPN (cDTPN).
5.2.2 Clock DTPN

A clock DTPN (cDTPN) is a safe DTPN with constraints on its syntax: (a) all incoming arcs belonging to a synchronizing transition (more than one pre-place) must have the same delay, (b) if a place has more than one incoming arc then all incoming arcs have zero delays, and (c) if a place has at least one incoming arc with a non-zero delay then there exists exactly one outgoing arc and this arc is connected to a transition with zero delay and this transition has exactly one pre-place. Note that the requirements (b) and (c) can be induced on an arbitrary DTPN by the arc refinement operation (see Fig. 5.1). It is easy to check that arc refinement preserves branching bisimulation.

![Figure 5.1: An example of a cDTPN](image)

Furthermore, a subset of marked places called timer places are associated with synchronizing transitions and transitions having at least one non-zero delay on outgoing arcs (called interval transitions). A timer place associated with a synchronizing transition is connected with bi-directional arcs to each transition that produces a token in the pre-place of the synchronizing transition. In this way, the token timestamp of the timer place is capturing the timestamp of the latest token in the pre-place of a synchronizing transition. A timer place associated with an interval transition is connected to itself with bi-directional arcs. In a similar way, the timestamp of the token in the timer place is reflecting the time of firing of an interval transition. We give an example of the transformation of a DTPN to cDTPN in Fig. 5.1.

**Notation.** Throughout this section, we will require safe DTPN’s. Hence, we
simplify the definition of a marking a bit. The timed marking of a safe DTPN assigns to each place, the timestamp of a token in that place. If a place has no token then we indicate it with infinity.

Definition 5.2.5 (Simplification of Marking Definition). The marking of a safe DTPN \( N \) is a function \( \hat{m} : P \rightarrow \mathbb{N}^\infty \) defined for all \( p \in P \) such that

\[
\hat{m}(p) = \infty, \text{ if } \neg \exists i \in \text{dom}(m) : \pi(m(i)) = p
\]

\[
\hat{m}(p) = \tau(m(i)), \text{ if } \exists i \in \text{dom}(m) : \pi(m(i)) = p
\]

Note that \( \hat{m} \) is defined for all places since the net is safe. When the context is clear, we will denote a marking \( \hat{m} \) by \( m \).

Formally, a cDTPN is defined as follows.

Definition 5.2.6 (cDTPN). A cDTPN is a tuple \( N = (P, T, \delta, m_0) \), where

1. \( N \) is a safe net and \( \forall t \in T : *t \neq \emptyset \land *t \neq \emptyset \), i.e. transitions have a non-empty preset and postset,

2. \( P_{\text{timer}} \subset P \land P_{\text{int}} \subseteq P \land P_{\text{timer}} \cap P_{\text{int}} = \emptyset \), where \( P_{\text{timer}} \) is the set of timer places and \( P_{\text{int}} = \{ p \in P \setminus P_{\text{timer}} \mid \exists t \in *p : \delta(t, p) \neq \{0\} \} \) is the set of interval places.

3. \( \forall t \in T_{\text{sync}} : \forall p, p' \in *t : \delta(p, t) = \delta(p', t) \), where \( T_{\text{sync}} = \{ t \in T \mid *t \cap P \setminus P_{\text{timer}} \mid > 1 \} \) is the set of synchronizing transitions, i.e. all incoming delays to a sync transition have equal delays.

4. \( \forall p \in P : *p \cap P_{\text{preint}} \neq \emptyset \Rightarrow |*p| = |*p'| = 1 \land \forall t' \in *p' : \delta(p, t') = \{0\} \), where \( T_{\text{preint}} = *P_{\text{int}} \) is the set of interval transitions, i.e. a place has an interval transition in its preset then this place has a preset and postset of size one with zero incoming delay.

5. \( \forall t \in T_{\text{postint}} : |*t| = |*t'| = 1 \land \forall p \in *t : \delta(t, p) = \{0\} \), where \( T_{\text{postint}} = P_{\text{int}} \) is the set of post-interval transitions, i.e. successor transitions of an interval transition have a preset and postset of size one with zero outgoing delays.

6. \( |T_{\text{sync}}| + |T_{\text{preint}}| = |P_{\text{timer}}| \) and there exists two injective functions \( \lambda : T_{\text{sync}} \rightarrow P_{\text{timer}} \) and \( \hat{\lambda} : T_{\text{preint}} \rightarrow P_{\text{timer}} \) with \( \text{rng}(\lambda) \cap \text{rng}(\hat{\lambda}) = \emptyset \).

7. \( \forall t \in T_{\text{sync}} : \lambda(t) = \lambda(t) = \lambda(t) \) and \( \forall t \in T_{\text{preint}} : \hat{\lambda}(t) = \hat{\lambda}(t) = \hat{\lambda}(t) \) and \( \forall p \in P_{\text{timer}} : \forall t \in *p, t' \in *p : \delta(t, p) = \delta(p, t') = \{0\}, \) i.e. timer places associated with a synchronizing transition are connected with bidirectional arcs (zero delays) to transitions that produce at least one token in the pre-place of this transition, and timer places associated with an interval transition are connected to themselves with bidirectional arcs (zero delays).
∀p ∈ P_{timer} : m_0(p) = 0 and ∀p ∈ P_{int} : m_0(p) = ∞, i.e. in the initial marking, all timer places have each one token with timestamp zero and all interval places are empty.

Note that \( T_{sync} \cap T_{postint} = \emptyset \) and \( T_{preint} \cap T_{postint} = \emptyset \). However, the sets \( T_{sync} \) and \( T_{preint} \) are not required to be disjoint. Given a transition \( t \in T_{postint} \) we will denote its predecessor transition \( t' \in \bullet(t) \cap T_{preint} \) by \( \text{int}(t) \). Note that \( \forall t \in T_{postint} \), the set \( \bullet(t) \) is a singleton.

By the definition of a cDTPN, the latest token arriving in a pre-place of a synchronizing transition \( t \) is updating the timer place \( \lambda(t) \). So the timestamp of the token in the timer place \( \lambda(t) \) is equal to the maximum of tokens in the pre-place of transition \( t \). In case of an interval transition, the token in the timer place is updated with the firing time of the marking that is enabling the interval transition. We express these structural properties in the following lemma.

**Lemma 5.2.1.** Consider a cDTPN \( N \) with an initial marking \( m_0 \). Let the set of all reachable timed markings from the initial marking be \( \mathcal{M} \). Then \( \forall m \in \mathcal{M} : \forall t \in T_{sync} : m(\lambda(t)) = \max\{m(p) | p \in \bullet(t)\} \) and \( \forall m, m' \in \mathcal{M}, t \in T_{preint} : m \xrightarrow{t} m' \Rightarrow m'(\lambda(t)) = ft(m) \).

### 5.2.3 cDTPN and its Timed Automaton

A cDTPN can be expressed as a timed automaton. The untimed reachability graph of a cDTPN defines the underlying automaton where markings correspond to locations and transitions between markings correspond to edges between locations labeled by the corresponding transition. Note that more than one edge can be labeled with the same transition. Each place of a clock DTPN is modeled as a clock variable in a timed automaton.

The guard condition of an edge is a clock constraint of the form \( X \leq Y \), where \( X \) is a clock and \( Y \) is a natural number. If an edge is associated with a synchronizing transition, then \( X \) is the clock corresponding the timer place of this transition and \( Y \) is the delay on any incoming arc from a non-timer place. If an edge is associated with a post interval transition, then \( X \) is the clock corresponding the pre-place (interval place) of this transition and \( Y \) is the minimum of the delay set associated with an incoming arc to this interval place. For all other transitions, \( X \) is the clock corresponding the pre-place and \( Y \) is the delay on incoming arc from this pre-place. Note that a guard belonging to an edge is specifying the enabling condition of that edge.

Eagerness in a DTPN means that as soon as a transition is enabled in a marking, it must fire. In a timed automaton, eagerness can be expressed by a location invariant that restricts the maximum delay that can occur in this location (delay transition of a timed automaton). This enforces a location to be left before
its invariant evaluates to false. The invariant of a location is computed by taking the conjunction of the negation of guards belonging to outgoing edges from this location, with the following exception: If a guard has a clock constraint containing a minimum over a delay set then we replace it with a maximum over the same delay set. The set of clocks to be reset on each edge are the clocks that correspond post-places of the transition (labeled on this edge) in the cDTPN. In this way, we are able to transfer the notion of transition enabling and urgency to a timed automaton. Note that eagerness is not expressed for edges labeled with delay transitions of a cDTPN.

**Notations.** Note that we will write $\overline{\neg(X \leq Y)}$ to denote $(X \geq Y)$. Furthermore, we extend the notion of postset and preset of a node in a Petri net to locations and edges in a timed automata. We will sometimes refer to an edge of a timed automaton by its label (corresponding a transition in a cDTPN). Note that we identify clocks in a timed automaton with places in a cDTPN.

**Definition 5.2.7** (Timed Automaton of a cDTPN). Let $N = (P, T, F, \delta, m_0)$ be a cDTPN. Let $\bar{M}$ be the set of all reachable untimed markings from marking $m_0$
(ignoring timestamps) of the underlying Petri net \((P, T, F)\). Two guard functions 
\(G : T \to B(P)\) and \(\bar{G} : T_{\text{postint}} \to B(P)\) satisfy \(t \in T\) and \(p \in \bullet t\) such that

\[
G(t) = (p \geq \delta(p, t)), \quad \text{if } t \in T \setminus (T_{\text{postint}} \cup T_{\text{sync}}),
\]

\[
G(t) = (\lambda(t) \geq \max_{p \in \bullet t}\{\delta(p, t)\}), \quad \text{if } t \in T_{\text{sync}},
\]

\[
G(t) = (\lambda(\text{int}(t)) \geq \min(\delta(\text{int}(t), p))), \quad \text{if } t \in T_{\text{postint}},
\]

\[
\bar{G}(t) = (\hat{\lambda}(\text{int}(t)) \geq \max(\delta(\text{int}(t), p))), \quad \text{if } t \in T_{\text{postint}}.
\]

The timed automaton of clock DTPN \(N\) is denoted by \(\text{TA}(N) = (\bar{M}, T, P, E, I)\), where

Edges: \((m, G(t), t, t^\bullet, m') \in E \iff \exists m, m' \in \bar{M}, t \in T : m' = m - t^\bullet + t^\bullet\)

Invariants: \(\forall m \in \bar{M} : I(m) = \bigwedge_{\forall t \in T \setminus T_{\text{postint}}} \neg G(t) \land \bigwedge_{\forall t \in T_{\text{postint}}} \neg \bar{G}(t)\)

We will first explain the intuition behind a timer place with an example described in the Fig. 5.2. Consider the state with tokens in places \(P_3, P_4, U_1, U_2\) and \(U_3\). From this state two transitions are enabled of which one transition has two pre-places, namely \(P_3\) and \(P_4\). So the invariant of this state becomes \((P_3 \leq 3 \lor P_4 \leq 3) \land P_3 \leq 7\). However, as disjunctions are not allowed in clock constraints, this strategy does not work. Note that the problem arises because the guard belonging to an edge of a synchronizing transition \(t\) is a conjunction of \(\bullet t\) terms. To solve this problem, we must model the guard belonging to an edge associated with a synchronizing transition using a single clock. This is only possible if we restrict each synchronizing transition to incoming arcs having equal delays. Note that in a cDTPN, we require for each synchronizing transition, the corresponding timer place is connected with bi-directional arcs to all transitions that produce a token in the pre-place of the synchronizing transition. Hence the timestamp of token in the timer place now represents the maximal timestamp of tokens in the pre-places of the synchronizing transition. So the guard of a synchronizing transition can be expressed as a predicate over a clock corresponding the timer place. In the Fig. 5.2, the invariant of state with tokens in places \(P_3, P_4, U_1, U_2\) and \(U_3\) translates into \(P_3 \leq 7 \land U_1 \leq 3\).

Note that in a cDTPN, if a non-timer place has a preset larger than one, then all incoming arcs have zero delays. Due to this requirement, we are able to compute locally, the delay incurred by a token in a place before it becomes available. Furthermore, if a non-timer place has an interval transition in its preset then the only outgoing arc from this place to a post-interval transition has a zero delay. Due to this requirement, we are able to express the guard of a post-interval transition \(t\) as a clock constraint over a single clock \(\hat{\lambda}(\text{int}(t))\).

Note. The translation of a DTPN into its corresponding timed automaton must be automated for it to be useful for practical purposes.
5.2.4 Relationship between cDTPN and its Timed Automaton

In this section, we will show that a sub class of cDTPN with no interval delays on outgoing arcs from transitions is branching bisimilar to its timed automaton. However, for a general cDTPN we have only trace equivalence.

First, we define the relationship between a timed marking of a cDTPN and a state of a timed automaton as the state relation.

**Definition 5.2.8 (State Relation).** Let $M$ be the set of all timed markings of a cDTPN $N$. Let $S$ be the set of all states of a timed automaton $TA(N)$. Let $mct : M \to N$ be a function defined as $mct(m) = \max\{m(p) \mid p \in P \setminus P_{\text{int}}\}$, i.e. the maximal consumable token of marking $m$. The state relation $\beta \subseteq M \times S$ is defined as $(m, (l, v)) \in \beta$ if and only if

$$l = \{p \mid m(p) \neq \infty\} \land \exists b \in [mct(m), ft(m)] : \forall p \in l \setminus P_{\text{int}} : v(p) = b - m(p)$$

Note that in a DTPN, choices about the timestamps of produced tokens are made when a transition fires, but in a timed automaton these choices are made in the state reachable after taking the edge labeled with this transition because of the guard and invariant associated with this state. So a DTPN is making a choice earlier than a timed automaton. Since transitions with delays on outgoing arcs are producing tokens in interval places and these tokens may have a timestamp in the future, we discard the clocks corresponding interval places in the state relation. Furthermore, if all incoming arcs of a cDTPN have zero delays then for all markings $m$ reachable from an initial marking $m_0$, we have $mct(m) = ft(m)$.

Given a set of reachable markings of a cDTPN, the state space of its timed automaton is the set of all related states.

**Definition 5.2.9 (State Space).** Let $N = (P, T, F, \delta, m_0)$ be a cDTPN. Let $M$ be the set of all reachable markings from the initial marking $m_0$. The state space of a timed automaton $TA(N)$ is the set

$$S = \{(l, v) \in S \mid \exists m \in M : (m, (l, v)) \in \beta\}$$

Next, we show for a marking of a cDTPN and a state of its timed automaton, related by the state relation, the maximal time progression (modeled by the delay transition) that can occur in that state depends on the firing time and the maximal consumable token of the marking. Note that a delay transition of a timed automaton is treated as a silent ($\tau$-labeled) step.

**Lemma 5.2.2 (Enabling of Delay Transitions).** Let $N$ be a cDTPN with the set of all reachable markings $M$ from the initial marking $m_0$. Let $S$ be the state space of timed automaton $TA(N)$. Then $\forall m \in M, (l, v) \in S : (m, (l, v)) \in \beta$ for some $b \in [mct(m), ft(m)]$ implies

$$(l, v) \xrightarrow{d} (l, v + d) \land (m, (l, v + d)) \in \beta, \text{ where } d = ft(m) - b$$
Proof. Suppose a delay transition \( d = \text{ft}(m) - b \) cannot occur from state \((l, v)\). This implies that for some delay \( d' < d : (l, v) \xrightarrow{d'} (l, v + d') \) and \( \exists t \in T \) that is enabled in state \((l, v + d')\).

- Suppose \( t \in T \setminus (T_{\text{sync}} \cup T_{\text{postint}}) \). Then \( \exists p \in \bullet t \) and by the Def. 5.2.7

\[
\begin{align*}
v(p) + d > v(p) + d' &= \delta(p, t) \\
&= m(p) + \delta(p, t) - m(p) \\
&= \text{et}(m, t) - m(p) \\
&\geq \text{ft}(m) - m(p)
\end{align*}
\]

Hence \( v(p) + d > \text{ft}(m) - m(p) \). Since \( d = \text{ft}(m) - b \), we may write \( v(p) + \text{ft}(m) - b > \text{ft}(m) - m(p) \). This implies \( v(p) + m(p) > b \). But \( v(p) + m(p) = b \). This is a contradiction.

- Suppose \( t \in T_{\text{sync}} \). Then we replace \( p \) by \( \lambda(t) \) and arc delay \( \delta(p, t) \) by \( \max\{\delta(p, t) \mid p \in \bullet t\} \) and then the arguments are the same as for the preceding case.

- Suppose \( t \in T_{\text{postint}} \). Then we replace \( p \) by \( \hat{\lambda}(\text{int}(t)) \) and arc delay \( \delta(p, t) \) by \( \max\{\delta(\text{int}(t), p) \} \) and then the arguments are the same as for the preceding case (using the lemma 5.2.1: \( m(\hat{\lambda}(\text{int}(t))) + \max\{\delta(\text{int}(t), p) \} = \text{et}(m, t) \)).

The result on enabling of delay transitions and the definition of the state relation are used to show that a timed automaton of a cDTPN simulates the cDTPN w.r.t. the state relation.

**Theorem 5.2.3 (TA simulates cDTPN).** Let \( N \) be a cDTPN with the set of all reachable marking \( \mathbb{M} \) from an initial marking \( m_0 \). Let \( S \) be the set of all states of \( TA(N) \). Then \( TA(N) \) simulates (branching) \( N \) w.r.t the state relation.

Proof. Let \( m \in \mathbb{M} \) and \((l, v) \in S \) such that \((m, (l, v)) \in \beta \) for some \( b \in [\text{mct}(m), \text{ft}(m)] \) (see def. 5.2.8). Suppose \( \exists m' \in \mathbb{M}, t \in T : m \xrightarrow{t} m' \).

- First assume \( b = \text{ft}(m) \).

  - Suppose \( t \in T \setminus (T_{\text{sync}} \cup T_{\text{postint}}) \). Then \( \exists p \in \bullet t : \text{ft}(m) = m(p) + \delta(p, t) \wedge v(p) = \text{ft}(m) - m(p) \). This implies \( v(p) = \delta(p, t) \).

    By the def. 5.2.7, the guard \( G(t) \) is the predicate \( p \geq \delta(p, t) \) and the invariant \( I(l) \) is a conjunction of predicates containing the predicate \( p \leq \delta(p, t) \). Hence there exists an enabled edge labeled with transition \( t \) and a state \((l', v') \in S \) such that \((l, v) \xrightarrow{t} (l', v') \).
Lemma 5.2.5

automaton, i.e. only for the case with no output delays. But if a cDTPN satisfies 
P reachable timed markings from an initial marking 
m. Then 
N of a timed automaton TA

In general, the converse is not true, i.e. a cDTPN does not simulate its timed automaton. As a consequence, the set of all executable traces of a cDTPN are included in the set of executable traces of its timed automaton.

Corollary 5.2.4 (Trace Inclusion). Let 
N be a cDTPN with the set of all reachable markings 
M from an initial marking 
m0. Let 
(l0, v0) be the initial state of a timed automaton 
TA(N) with state space 
S. Then

\[
\forall \sigma \in T^* : m_0 \xrightarrow{\sigma} \exists \tilde{\sigma} \in (T \cup N)^* : (l_0, v_0) \rightarrow_\sigma \land \tilde{\sigma}|_T = \sigma
\]

In general, the converse is not true, i.e. a cDTPN does not simulate its timed automaton. But if a cDTPN satisfies 
P int = \emptyset, then it can simulate its timed automaton, i.e. only for the case with no output delays.

Lemma 5.2.5 (cDTPN simulates TA). Let 
N be a cDTPN such that 
P int = \emptyset. Then 
N can simulate (branching) 
TA(N) with respect to the state relation.

Proof. Note that 
P int = \emptyset \Rightarrow T_{\text{preint}} = \emptyset \land T_{\text{postint}} = \emptyset. Let 
M be the set of all reachable timed markings from an initial marking 
m0 and 
S be the state space of 
TA(N). Let 
m \in 
M and 
(l, v) \in 
S such that 
(m, (l, v)) \in \beta.

Now assume 
b < ft(m). Then by the lemma 5.2.2, for 
d = ft(m) - 
b, we have 
(l, v) \xrightarrow{d} (l, v + d) and 
(m, (l, v + d)) \in \beta and then we are in the former case.

\[
\exists (l', v') \in S, t \in T : (l, v) \rightarrow_t (l', v'). \text{ Then by the Def. 5.2.7, transition } t \text{ is classically enabled.}
\]

If 
t \notin T_{\text{sync}} then by the state invariant and guards of 
TA(N), \exists p \in \bullet t and 
v(p) = \delta(p, t). Note that the state invariant guarantees that no edge corresponding a transition can fire earlier. Hence by the state relation 
v(p) =
\[ b - m(p), \text{ where } b \in [\text{mct}(m), \text{ft}(m)]. \] So \( b = m(p) + \delta(p, t) \), but this means that \( b \geq \text{ft}(m) \). Hence \( b = \text{ft}(m) \) and \( t \) is also enabled in the marking \( m \).

If \( t \in T_{\text{sync}} \), then we reuse the preceding arguments after replacing place \( p \) by \( \lambda(t) \) and arc delay \( \delta(p, t) \) by \( \max\{\delta(p, t) | p \in \bullet t\} \).

Hence \( \forall p \in P : m(p) \neq \infty \Rightarrow v(p) = \text{ft}(m) - m(p) \) and \( \exists m' \in M : m \xrightarrow{t} m' \).

By the Corollary 5.2.6, \( (l, v) \xrightarrow{d} (l, v + d) \). Then by the lemma 5.2.2, for \( d = \text{ft}(m) - b \), we have \( (l, v) \xrightarrow{d} (l, v + d) \) and \( (m, (l, v + d)) \in \beta \) and then we are in the former case.

\[ \square \]

As a consequence, we have our result on branching bisimulation.

**Corollary 5.2.6 (Branching Bisimulation).** Let \( N \) be a cDTPN such that \( P_{\text{int}} = \emptyset \). Then \( N \) and \( TA(N) \) are branching bisimilar with respect to the state relation.

In the Fig. 5.3, we show why in general there is no bisimulation between a cDTPN and its timed automaton. This is because the choices of timestamps of produced tokens (lying possibly in the future) are made by the transition producing them, while in the corresponding timed automaton, the choice is made in the state reachable after taking an edge labeled with this transition.

However, we still have trace equivalence between a general cDTPN and its timed automaton for finite traces.

**Theorem 5.2.7 (Trace Equivalence).** Let \( N \) be a cDTPN with the set of all reachable timed markings \( M \) from the initial marking \( m_0 \). Let \( (l_0, v_0) \) be the initial state of a timed automaton \( TA(N) \) such that \( (m_0, (l_0, v_0)) \in \beta \) and \( \text{mct}(m_0) = 0 \) and \( \forall p \in l_0 : v_0(p) = 0 \). Then the following holds:

\[ \begin{align*}
\text{(a)} \forall \sigma \in T^* : m_0 \xrightarrow{\sigma} : \exists \bar{\sigma} \in (T \cup \mathbb{N})^* : (l_0, v_0) \rightarrow_{\bar{\sigma}} \land \bar{\sigma}[T] = \sigma \\
\text{(b)} \forall \bar{\sigma} \in (T \cup \mathbb{N})^* : (l_0, v_0) \rightarrow_{\bar{\sigma}} : \exists \sigma \in T^* : m \xrightarrow{\bar{\sigma} \land \sigma} = \bar{\sigma}[T]
\end{align*} \]

**Proof.** By the Corollary 5.2.6, (a) holds.
Figure 5.3: Why a cDTPN with interval delays cannot simulate its timed automaton?

Consider a trace $\bar{\sigma} \in (T \cup N)^*$ such that $\bar{\sigma} = \langle d_0, t_1, d_1, t_2, \ldots, t_n \rangle$ and

$$(l_0, v_0) \xrightarrow{d_0} (l_0, v_0 + d_0) \rightarrow_{t_1} (l_1, v_1) \xrightarrow{d_1} (l_1, v_1 + d_1) \rightarrow_{t_2} \cdots \rightarrow_{t_n} (l_n, v_n)$$

We will prove by induction that

$$\exists m_1, \ldots, m_n \in M : \forall k \in \{1, \ldots, n\} :$$

$$(m_k, (l_k, v_k)) \in \beta \land d_0 = ft(m_0) \land d_k = ft(m_k) - ft(m_{k-1}) \land m_{k-1} \xrightarrow{t_k} m_k \land$$

$$t_k \in T_{preint} \Rightarrow \forall p \in t_k^* : m_k(p) = \sum_{i=0}^{j} d_i$$

where, $j \geq k$ is the first index in $\bar{\sigma}$ such that $t_j \in p^*$ or $j = n$.

Note that the choice of the value to be picked from a delay set corresponding an outgoing arc in a cDTPN is made by looking ahead in the firing sequence $\bar{\sigma}$ of $\text{TA}(N)$ to find out the time the token must be consumed in the cDTPN. If this cannot be determined by looking ahead in the firing sequence $\bar{\sigma}$, then we choose the value to be picked as equal to the sum of all delay transitions in the firing sequence $\bar{\sigma}$.

Consider $k = 1$. Note $(m_0, (l_0, v_0)) \in \beta$ and $\text{mct}(m_0) = 0$ and $\forall p \in l_0 : v_0(p) = 0$. By the lemma 5.2.2, $(m_0, (l_0, v_0 + d_0)) \in \beta$, where $d_0 = ft(m_0)$. 
We have in the timed automaton \((l_0, v_0 + d_0) \rightarrow_{t_1} (l_1, v_1)\). Since \(\forall p \in P_{\text{int}} : m_0(p) = \infty\), transition \(t_1 \notin T_{\text{postint}}\). Hence, we may use the arguments for transition enabling as in the lemma 5.2.5 and then transition \(t_1\) is enabled in the marking \(m_0\).

If \(t_1 \in T \setminus T_{\text{preint}}\) then \(\exists m_1 \in M : m_0 \xrightarrow{t_1} m_1\). If \(t_1 \in T_{\text{preint}}\) then we apply the construct of the induction hypothesis to determine for each \(p \in t_1^* : m_1(p) = \sum_{j=0}^{i} d_i\), where \(j \geq 1\) is the first index such that \(t_j \in p^*\) or \(j = n\).

By the arguments of the lemma 5.2.5, \((m_1, (l_1, v_1)) \in \beta\) holds.

Suppose the statement holds for all \(k \leq i\). Consider \(k = i + 1\). Then by the induction hypothesis \((m_i, (l_i, v_i)) \in \beta\) and \(d_i = ft(m_i) - ft(m_{i-1})\) and \(ft(m_i) = \sum_{j=0}^{i} d_j\). By the lemma 5.2.2, the relation \((m_i, (l_i, v_i + d_i)) \in \beta\) holds.

We have in the timed automaton \((l_i, v_i + d_i) \rightarrow_{t_{i+1}} (l_{i+1}, v_{i+1})\).

If \(t_{i+1} \in T \setminus T_{\text{postint}}\) we may use the arguments for transition enabling as in the lemma 5.2.5 and then transition \(t_{i+1}\) is enabled in the marking \(m_i\). If \(t_{i+1} \in T_{\text{postint}}\) then \(m_i(p) \neq \infty\), where \(p \in t_{i+1} \cap P_{\text{int}}\). By the induction hypothesis there is a greatest \(x \leq i\) such that \(t_x \in p^*\) and so \(t_x \in T_{\text{preint}}\) and \(m_i(p) = \sum_{j=0}^{i} d_j\). Since \(ft(m_i) = \sum_{j=0}^{i} d_j = m_i(p)\), transition \(t_{i+1}\) is enabled in the marking \(m_i\).

Next we show that the state relation holds. If \(t_{i+1} \notin T_{\text{preint}}\) then \(\exists! m_{i+1} \in M : m_i \xrightarrow{t_{i+1}} m_{i+1}\). If \(t_{i+1} \in T_{\text{preint}}\) then we apply the construct of the induction hypothesis to determine for each \(p \in t_{i+1}^* : m_{i+1}(p) = \sum_{j=0}^{i} d_i\), where \(j \geq i + 1\) is the first index such that \(t_j \in p^*\) or \(j = n\).

By the arguments as in the lemma 5.2.5, \((m_{i+1}, (l_{i+1}, v_{i+1})) \in \beta\) holds.

\[\square\]

### 5.3 Expressiveness of DTPN

In this section, we will study the expressive power of DTPN using the two variants of sDTPN, one satisfying In-single, Out-zero (input delays only (IDO)) and the other satisfying In-zero, Out-single (output delays only (ODO)). Recall that sDTPN is simulation equivalent with a general DTPN. First, we show that sDTPN with IDO and sDTPN with ODO are capable of modeling inhibitor arcs, which means that they are Turing complete. Hence the model class DTPN is Turing complete. We prove this claim by constructing for an arbitrary Petri net with inhibitor arcs, an sDTPN model with IDO and an sDTPN model with ODO that is branching bisimilar to it. Thus, we show that the model classes of DTPN with IDO and ODO have the same expressive power. That does not mean however that they have the same modeling comfort, i.e. the ease of modeling. In order to explore this, we construct for an sDTPN with IDO, an sDTPN model with ODO that is bisimilar and vice versa. Finally, we also show how different model classes
(i.e. \(M_1\) and \(M_2\)) can be expressed using model class \(M_3\).

**Note.** The results on expressivity of DTPN and its subclasses are useful for understanding their modeling power. From a system design perspective, the results on simulating model classes \(M_1\) and \(M_2\), open up the possibility to analyze them as a DTPN.

### 5.3.1 Expressing inhibitor nets with sDTPN

In this section, we will describe the construction of IDO and ODO nets, given an inhibitor net and prove that they are branching bisimilar.

We will first consider the case of sDTPN's with IDO. We start with a construction. Given an inhibitor net \(N\) we construct an sDTPN \(N'\) with IDO as in Fig. 5.4. We add a place called \(\text{Tick}\) and we replace each inhibitor arc of \(N\) by a simple sub-net consisting of one inhibitor place \(S\) and one inhibitor transition \(T\) (which will be the silent transition). The extra place \(\text{Tick}\) is connected to all original transitions with input and output arcs. \(\text{Tick}\) always contains one token. \(S\) is connected with input and output arcs to inhibitor transition \(A\) and also with \(T\). Further \(T\) is connected with an input and an output arc to \(Q\) the inhibitor place. All original arcs between a place and a transition get input delay 0. All input arcs from place \(\text{Tick}\) get a delay 2, the input arc from \(S\) to \(T\) gets a delay 1 and from \(S\) to \(A\) a delay 2. In the initial marking there are tokens in \(S\) and \(\text{Tick}\) with timestamps 0. Let the initial marking of \(N\) be \(m_0\), then the initial marking \(m'_0\) of \(N'\) has for each untimed token in \(m_0\) a timed token in \(m'_0\) in the same place with a timestamp 0. Note that all transitions, except for the ones that were added during the construction may have more input and output arcs. We will formally define the construction as a refinement of inhibitor arcs.

**Notation.** Given a DTPN \((P,T,F,\delta)\) with a timed marking \(m \in \mathcal{M}\), we will denote the set of all enabled transitions by \(\text{enabled}(m) = \{ t \in T \mid \text{et}(m,t) \leq \text{ft}(m) \}\) and assert the existence of at least one token in a place \(p \in P\) in a marking \(m \in \mathcal{M}\) by the predicate \(\text{marked}(m,p)\). To retrieve the number of tokens in a place having a particular timestamp in a given marking, we will define a function, \(\nu : \mathcal{M} \times P \times \mathbb{Q} \rightarrow \mathbb{N}\) that satisfies \(\nu(m,p,q) = | \{ i \in \text{dom}(m) \mid \pi(m(i)) = p \land \tau(m(i)) = q \} |\). To retrieve the number of tokens in a place, irrespective of their timestamps, we overload this function as follows: \(\nu : \mathcal{M} \times P \rightarrow \mathbb{N}\) is defined as \(\nu(m,p) = \sum_{q \in \mathbb{Q}} \nu(m,p,q)\). Note that since all places \(p \in P_s\) and tick are safe, we will use the notion of marking as in the Def. 5.2.5. A transition \(t \in T \setminus T_s\) is called an inhibitor transition if \(\exists p \in P : (t,p) \in I\), otherwise, we call it a non-inhibitor transition. Furthermore, given a DTPN with a marking \(m \in \mathcal{M}\) and a transition \(t \in T\), we will assert that transition \(t\) is classically enabled by the predicate \(\text{ce}(m,t)\) iff \(\forall p \in \bullet t : \exists i \in \text{dom}(m) : \pi(m(i)) = p\).
Definition 5.3.1 (Inhibitor Arc Refinement with Input Delays). Let $N = (P, T, F, \iota)$ be an inhibitor net with an initial marking $\bar{m}_0$. Let $I = \{(t, p) \in T \times P \mid p \in \iota(t)\}$. Let $\text{tick} \notin P$ be a tick place. Let $T_s = \{\tau_{tp} \mid (t, p) \in I\}$ be the set of silent transitions such that $T \cap T_s = \emptyset$. Let $P_s = \{\pi_{tp} \mid (t, p) \in I\}$ such that $P \cap P_s = \emptyset$. The inhibitor arc refinement with input delays only of net $N$ is a DTPN with IDO denoted by $\psi_i(N) = (P', T', F', \delta)$ with an initial marking $m_0$, where

$$
P' = P \cup \{\text{tick}\} \cup P_s
$$

$$
T' = T \cup T_s
$$

$$
F' = F \cup \{(t, \pi_{tp}) \mid (t, p) \in I\} \cup \{(\pi_{tp}, t) \mid (t, p) \in I\} \cup
\{(\pi_{tp}, \pi_{tp}) \mid (t, p) \in I\} \cup
\{(\tau_{tp}, \iota(t)) \mid (t, p) \in I\} \cup
\{((t, \text{tick}) \mid t \in T\} \cup \{((t, \text{tick}) \mid t \in T\}
$$

The function $\delta$ is defined as $\forall (t, p) \in I : \delta(\pi_{tp}, t) = 2 \land \delta(\pi_{tp}, \tau_{tp}) = 1$ and $\forall t \in T : \delta(\text{tick}, t) = 2$ and all other arcs in $F'$ have a zero delay. Furthermore, the initial marking $m_0$ satisfies $\forall p \in P : \bar{m}_0(p) = \nu(m_0, p)$ and $\forall (t, p) \in I : \nu(m_0, \pi_{tp}) = \nu(m_0, \text{tick}) = 1$ and $\forall i \in \text{dom}(m_0) : \tau(m_0(i)) = 0$.

Note that since all places $p \in P_s$ and tick are safe, we will use the notion of marking as in the Def. 5.2.5.
Observations.
- \( \forall m \in M : m(\text{tick}) \) is even, since in the initial marking the token in tick has a timestamp zero and all outgoing arcs have a fixed delay of value two. Note that incoming arcs to tick have a zero delay.
- \( \forall m \in M, t \in T \setminus T_s : t \in \text{enabled}(m) \Rightarrow \text{ft}(m) = 2k \) for some \( k \in \mathbb{N} \), i.e. non-silent transitions fire only at even time points. This is because
  - all transitions \( t \in T \setminus T_s \) such that \( \neg \exists p \in P : (t, p) \in I \) are delayed only by the token in place tick whose timestamp is always even.
  - all transitions \( t \in T \setminus T_s \) such that \( \exists p \in P : (t, p) \in I \) are delayed by the tokens in places tick and \( \pi_{tp} \). The only way transition \( t \) can fire at \( 2k + 1 \) is if \( m(\pi_{tp}) = 2k + 1 > m(\text{tick}) \). So if transition \( t \) is delayed by a token in \( \pi_{tp} \) until \( 2k + 1 \) then place \( p \) was marked at \( 2k - 1 \) and only at time \( 2k \) place \( p \) could become unmarked, thereby updating the token in the tick place to \( 2k \). Hence transition \( t \) can only fire at \( 2k + 2 \) and not at \( 2k + 1 \).
- If a marking \( m \in M \) and an inhibitor transition \( t \in T \setminus T_s \) such that \( (t, p) \in I \) for some \( p \) satisfy \( \text{ce}(m, t) \) and \( \neg \text{marked}(m, p) \) and \( m(\text{tick}) = 2k \), then \( m(\pi_{tp}) \leq 2k \). Since if \( m(\pi_{tp}) = 2k + 1 \) then transition \( \tau_{tp} \) fired at \( 2k + 1 - 2 = 2k - 1 \) and transition \( t \) will fire at \( 2k + 1 \). But the latest time that place \( p \) could become unmarked by a non-silent transition is at \( 2k - 2 \) and then \( m(\pi_{tp}) = 2k \) which means \( t \) will fire at \( 2k \), which is a contradiction to the conclusion that \( t \) will fire at \( 2k + 1 \).

Next, we will consider the case of sDTPN’s with ODO, which is slightly more complex. We start with a construction shown in Fig. 5.5. For each inhibitor arc, like the arc between an inhibitor transition \( A \) and a place \( Q \), we replace this arc by a sub-net with four places \( S_1, S_2, S_3 \) and \( S_4 \), and two transitions \( T_1 \) and \( T_2 \) with double arcs (input and output) between \( S_1 \) and \( A \), \( S_2 \) and \( A \), \( S_1 \) and \( T_1 \), \( S_2 \) and \( T_2 \) and between \( T_1 \) and \( Q \) and \( T_2 \) and \( Q \). Further \( T_1 \) is the only input transition and \( T_2 \) the only output transition for \( S_4 \). Similarly \( T_2 \) is the only input transition and \( T_1 \) the only output transition for \( S_3 \). We call \( T_1 \) and \( T_2 \) the inhibitor transitions and \( S_1, \ldots, S_4 \) the inhibitor places. Like in the former case, all transitions of the original model are connected with double arcs with a new place called tick. All original output arcs get a delay 0, the output arcs from \( T_1 \) to \( S_1 \) and from \( T_2 \) to \( S_2 \) and all non-inhibitor transitions connected to place tick get an output delay of value 2. The output arcs from \( T_1 \) to \( S_4 \) and from \( T_2 \) to \( S_3 \) get a delay 1. Places tick, \( S_1, S_2 \) and one of the places \( S_3 \) or \( S_4 \) are marked with one token in the initial marking. All tokens have initially a timestamp 0 except for the token in place tick having a timestamp 2.

**Definition 5.3.2 (Inhibitor Arc Refinement with Output Delays).** Let \( N = (P, T, F, \iota) \) be an inhibitor net with an initial marking \( \bar{m}_0 \). Let \( I = \{(t, p) \in \)
\[ T \times P \mid p \in \iota(t) \}. \] Let \( \text{tick} \notin P \) be a tick place. Let \( T_s = \{ \tau_{tp}^j \mid (t, p) \in I \land j \in [1, 2] \} \) be the set of silent transitions such that \( T \cap T_s = \emptyset \). Let \( P_s = \{ \pi_{tp}^j \mid (t, p) \in I \land j \in [1, 4] \} \) such that \( P \cap P_s = \emptyset \). The inhibitor arc refinement with output delays only of net \( N \) is a DTPN with ODO denoted by \( \psi_o(N) = (P', T', F', \delta) \) with an initial marking \( m_0 \), where

\[
\begin{align*}
P' &= P \cup \{ \text{tick} \} \cup P_s \\
T' &= T \cup T_s \\
F' &= F \cup \{ \tau_{tp}^1, \pi_{tp}^4 \} \cup \{ \pi_{tp}^4, \tau_{tp}^2 \} \cup \{ \tau_{tp}^2, \pi_{tp}^3 \} \cup \{ \pi_{tp}^3, \tau_{tp}^1 \} \cup \\
&\quad \{ (t, \pi_{tp}^j, \tau_{tp}^j) \mid (t, p) \in I \land j \in [1, 2] \} \cup \{ (\pi_{tp}^j, \tau_{tp}^j) \mid (t, p) \in I \land j \in [1, 2] \} \cup \\
&\quad \{ (\pi_{tp}^j, \tau_{tp}^j) \mid (t, p) \in I \land j \in [1, 2] \} \cup \{ (\tau_{tp}^j, \pi_{tp}^j) \mid (t, p) \in I \land j \in [1, 2] \} \cup \\
&\quad \{ (\pi_{tp}^j, \pi_{tp}^j) \mid (t, p) \in I \land j \in [1, 2] \} \cup \{ \text{tick}, t \mid t \in T \} \cup \{ (t, \text{tick}) \mid t \in T \}.
\end{align*}
\]

Function \( \delta \) is defined as follows: \( \forall (t, p) \in I : \delta(\tau_{tp}^1, \pi_{tp}^1) = \delta(\tau_{tp}^2, \pi_{tp}^2) = 2 \land \delta(\pi_{tp}^1, \pi_{tp}^1) = \delta(\tau_{tp}^2, \pi_{tp}^3) = 1 \); \( \forall t \in T : \delta(t, \text{tick}) = 2 \), and all other arcs in \( F' \) have a zero delay. Furthermore, the initial marking \( m_0 \) satisfies \( \forall p \in P : \nu_0(p) = \nu(p_0, p) \) and \( \forall (t, p) \in I : \nu(m_0, \pi_{tp}) = \nu(m_0, \text{tick}) = 1 \) and all tokens have a timestamp zero except for the token in place tick having a timestamp two.

**Figure 5.5:** Modeling Inhibitor with ODO
Observations.

- \( \forall m \in \mathbb{M} : m(\text{tick}) \) is even, since in the initial marking the token in tick has a timestamp two and all incoming arcs have a fixed delay of value two. Note that outgoing arcs from tick have a zero delay.

- \( \forall m \in \mathbb{M}, t \in T \setminus T_s : t \in \text{enabled}(m) \Rightarrow ft(m) = 2k \) for some \( k \in \mathbb{N} \), i.e. non-silent transitions fire only at even time points. This is because
  - all non-inhibitor transitions \( t \in T \setminus T_s \) are delayed only by the token in place tick whose timestamp is always even.
  - all inhibitor transitions \( t \in T \setminus T_s \) for some \( (t,p) \in I \) are delayed by the tokens in places tick, \( \pi_1^{tp} \) and \( \pi_2^{tp} \). The only way transition \( t \) can fire at \( 2k + 1 \) is if \( \max(m(\pi_1^{tp}), m(\pi_2^{tp})) = 2k + 1 > m(\text{tick}) \). So if transition \( t \) is delayed either by the token in \( \pi_1^{tp} \) or \( \pi_2^{tp} \) until \( 2k + 1 \) then place \( p \) was marked at \( 2k \) and only at time \( 2k \) place \( p \) could become unmarked, thereby updating the token in the tick place to \( 2k + 2 \). Hence transition \( t \) can only fire at \( 2k + 2 \) and not at \( 2k + 1 \).

- If a marking \( m \in \mathbb{M} \) and an inhibitor transition \( t \in T \setminus T_s \) for some \( (t,p) \in I \) satisfies \( \text{ce}(m,t) \) and \( \neg \text{marked}(m,p) \) and \( m(\text{tick}) = 2k \) then \( \max(m(\pi_1^{tp}), m(\pi_2^{tp})) \leq 2k \). Since if \( \max(m(\pi_1^{tp}), m(\pi_2^{tp})) = 2k + 1 \) then transition \( \tau_1^{tp} \) or \( \tau_2^{tp} \) fired at \( 2k + 1 - 2 = 2k - 1 \) and transition \( t \) will fire at \( 2k + 1 \). But the latest time that place \( p \) could become unmarked by a non-silent transition is at \( 2k - 2 \) and then \( \max(m(\pi_1^{tp}), m(\pi_2^{tp})) = 2k \) which means \( t \) will fire at \( 2k \). This is a contradiction.

Since the case of ODO is more complex than IDO, we will first prove the inhibitor property for the case of ODO.

**Theorem 5.3.1.** Let \( N \) be an inhibitor net and \( \varphi_o(N) \) be its DTPN with ODO having an initial marking \( m_0 \).

Then \( \forall m \in \mathcal{R}(N, m_0), (t,p) \in I \) satisfies

(i) \( \text{marked}(m,p) \Rightarrow t \notin \text{enabled}(m) \)

(ii) \( \neg \text{marked}(m,p) \land \text{ce}(m,t) \Rightarrow et(m,t) = ft(m) \lor \text{enabled}(m) \cap (T \setminus T_s) = \emptyset \)

**Proof.** Consider a marking \( m \in \mathcal{R}(N, m_0) \) and \( (t,p) \in I \). There are three cases to consider.

- Suppose \( ft(m) = \infty \). Then \( t \notin \text{enabled}(m) \). If \( \neg \text{marked}(m,p) \) then silent transitions \( \tau_1^{tp} \) or \( \tau_2^{tp} \) cannot fire, so transition \( t \) cannot be delayed. Hence \( t \) is not classically enabled.

- Suppose \( ft(m) = 2k + 1 \) for some \( k \in \mathbb{Z} \). Since transition \( t \) can fire only at even time points, \( t \notin \text{enabled}(m) \). However transition \( \tau_1^{tp} \) or \( \tau_2^{tp} \) can fire, in which case \( \text{marked}(m,p) \).
Suppose $r(t, m) = 2k$ and $t$ is classical enabled. We consider four cases:

- If marked($m, p$) and $p$ became marked at time $2k$. Then only silent transitions $t' \in T_s$ can fire.
- If marked($m, p$) and $p$ became marked latest at time $2k - 2$. Then one of the silent transitions $\tau_{1p}$ or $\tau_{2p}$ might have fired is at $2k - 2$ and at $2k - 1$, thereby delaying transition $t$ until $2k + 1$.
- If $\neg$ marked($m, p$) and ce($m, t$) and the place $p$ became unmarked at time $2k$ by firing a transition $t' \in T \setminus T_s$. Then the place tick has a token timestamp $2k + 2$, so only silent transitions that are enabled may fire at time $2k$.
- $\neg$ marked($m, p$) and ce($m, t$) and the place $p$ became unmarked latest at time $2k - 2$ by firing a transition $t' \in T \setminus T_s$. This means that the latest firing time of a transition $\tau_{1p}$ or $\tau_{2p}$ is at most $2k - 2$. So $t$ is delayed until time $2k$.

We have a similar result for IDO.

**Theorem 5.3.2.** Let $N$ be an inhibitor net and $\varphi_i(N)$ be its DTPN with IDO having an initial marking $m_0$.

Then $\forall m \in R(N, m_0), (t, p) \in I$ satisfies

(i) marked($m, p$) $\Rightarrow t \notin$ enabled($m$);

(ii) $\neg$ marked($m, p$) $\land$ ce($m, t$) $\Rightarrow$ et($m, t$) = $f_l(m) \lor$ enabled($m$) $\cap$ ($T \setminus T_s$) = $\emptyset$.

The proof is similar to the proof of theorem 5.3.1, since the only difference is that the token in the places $P_s$ are always ready for consumption, because tokens in these places have a timestamp at most equal to the current time of the system. So for each arc $(t, p) \in I$, one silent transition $\tau_{tp}$ and a place $\pi_{tp}$ is sufficient to delay transition $t$ by two time units in each time step, if the place $p$ is marked. So we replace occurrences of silent transitions $\tau_{1p}$ and $\tau_{2p}$ with $\tau_{tp}$ in the proof of theorem 5.3.1.

For the proof of branching bisimulation between an inhibitor net and its ODO/IDO nets, we define the relationship between their markings.

**Definition 5.3.3 (Marking relation for ODO).** Let $N$ be an inhibitor net with an initial marking $m_0$ and $N' = \psi_\omega(N)$ be its DTPN with initial marking $m'_0$. Let $M = R(N, m_0)$ be the set of all reachable untimed markings of $N$ and $M' = R(N', m_0)$ be the set of all reachable timed markings of $N'$. The marking relation for ODO denoted by $\sim_o \subseteq M \times M'$ satisfies $m \sim_o m'$ iff $m' \in M' \land \forall p \in P_N : m(p) = \nu(m', p)$.

By replacing $N' = \psi_\omega(N)$ in the Def. 26, we define the marking relation for IDO denoted by $\sim_i$. 


Theorem 5.3.3 (Branching Bisimulation - ODO). Let $N$ be an inhibitor net. Then $N$ and $\psi_o(N)$ are branching bisimilar with respect to $\sim_o$.

Proof. (sketch) If a transition $t$ in $N$ such that $\iota(t) = \emptyset$ is enabled then the same transition is also enabled in $\psi_o(N)$ because of the marking relation but with a possible delay induced by the token in the tick place.

If a transition $t$ in $N$ such that $\iota(t) \neq \emptyset$ is enabled then all places in $\iota(t)$ are unmarked, and due to the theorem 5.3.1, transition $t$ can also fire in the net $\psi_o(N)$ (possibly delayed).

If an inhibitor transition $t$ in the net $\psi_o(N)$ is enabled, then due to the theorem 5.3.1, all inhibitor places of $t$ are unmarked and so $t$ can fire in the net $N$ as well. For all other enabled non-silent transitions in the net $\psi_o(N)$, the same transitions are also enabled in the net $N$ because of the marking relation. If a silent transition $\tau \in T_s$ is enabled in the net $\psi_o(N)$ then the firing of such a transition is not changing the distribution of tokens in places but only delaying its corresponding inhibitor transition.

Theorem 5.3.4 (Branching Bisimulation - IDO). Let $N$ be an inhibitor net. Then $N$ and $\psi_i(N)$ are branching bisimilar with respect to $\sim_i$.

The proof is similar to the theorem 5.3.3, since by the theorem 5.3.2, the net $\psi_i(N)$ is having the desired inhibitor behavior and firing silent transitions is not changing the distribution of tokens in the net.

As a consequence, model class $M3$ can express inhibitor arcs.

Corollary 5.3.5. The model class $M3$ is Turing complete.

Next, we will consider a timed variant of inhibitor arcs that takes into consideration the timestamp of tokens and we call it a timed inhibitor arc. This means that a timed inhibitor arc allows its connected transition(s) to be enabled only if the connected place is empty or has tokens with timestamps greater than the current time. We call this class of nets as timed inhibitor nets.

Definition 5.3.4 (Timed Inhibitor nets). A timed inhibitor net is a tuple $(P,T,F,\delta,\iota)$, where $(P,T,F,\delta)$ is a DTPN and $\iota : T \to P$ is the set of inhibitor arcs.

Definition 5.3.5 (Transition Relation of a Timed Inhibitor net). The transition relation of a timed inhibitor net satisfies Def. 4.3.3 and moreover, for any inhibitor transition $t \in T$ and markings $m, m' \in M$ such that $m \xrightarrow{t} m'$ is satisfying $\forall p \in \iota(t) : \neg \exists i \in \text{dom}(m) : \pi(m(i)) = p \land \tau(m(i)) \leq \text{ft}(m)$.

An obvious question arises: Using the construction techniques described above, is it possible to simulate an inhibitor arc in a timed inhibitor net? The answer is no! This is because the interleaving of non-silent transitions caused by the token the place tick is destroying the simulation property. We give an example in the
Fig. 5.6, where the firing sequence \( \langle A; C; D; B \rangle \) is not possible in the system with place tick. However for two cases we are able to simulate timed inhibitor arcs:

- For inhibitor transitions connected to a safe place, we use the well-known construct as in the Fig. 5.7. All delays on arcs with \( Q \) are 0. (In case \( P \) is bounded by some \( n \) we need multiple arcs or arc weights).

- In case place \( P \) is not bounded and the net system satisfies in each step of every executable firing sequence that the time is strictly increasing, then we may reuse the construction defined for simulating untimed inhibitor arcs after dropping the place tick and choosing a value for \( \varepsilon \) such that \( \varepsilon < \Delta \), where \( \Delta \) is the smallest arc delay of the system as shown in the Fig. 5.7. Note that (a) if place \( R \) at time \( X \) has a token with timestamp less than or equal to \( X \) then transition \( A \) is inhibited, (b) if transition \( A \) is enabled at \( X \) then it might be delayed until \( X + \varepsilon \).

In the next section, we will compare the two variants of sDTPN, namely IDO and ODO, by giving a construction to translate one variant into another while preserving branching bisimilarity.

### 5.3.2 Modeling ODO with IDO

In the Fig. 5.8, we present an example transformation of output delays into input delays. The transformation is in fact just a refinement of outgoing arcs, where for each outgoing arc from a transition, the corresponding output delay becomes the input delay of the newly introduce silent transition. We will formally define the refinement of outgoing arcs.

**Definition 5.3.6 (Output Arc Refinement of a DTPN).** Let \( N = (P, T, F, \delta) \) be a DTPN. Let \( \bar{F} = F \cap (T \times P) \) be the set of all outgoing arcs. Let \( \bar{P} \) be the set of intermediate places such that \( \bar{P} \cap P = \emptyset \) and \(|\bar{P}| = |\bar{F}|\). Let \( \bar{T} \) be the set of \( \tau \)-labeled transitions such that \( \bar{T} \cap T = \emptyset \) and \(|\bar{T}| = |\bar{F}|\). Let \( \lambda : \bar{F} \to \bar{P} \times \bar{T} \) be a function that assigns to every outgoing arc, an intermediate place and a \( \tau \)-labeled transition. We define two standard projection functions \( \pi_1 \) and \( \pi_2 \) over \( \bar{P} \times \bar{T} \).
such that $\pi_1(q, \tau) = q$ and $\pi_2(q, \tau) = \tau$. The output arc refinement of a DTPN $N$ is denoted by $\gamma_o(N) = (P', T', F', \delta')$, where

- $P' = P \cup \bar{P}$,
- $T' = T \cup \bar{T}$,
- $F' = (F \setminus \bar{F}) \cup \{(t, \pi_1(\lambda(t, p))) \mid (t, p) \in \bar{F}\} \cup \{(\pi_1(\lambda(t, p)), \pi_2(\lambda(t, p))) \mid (t, p) \in \bar{F}\} \cup \{(\pi_2(\lambda(t, p)), p) \mid (t, p) \in \bar{F}\}$,
- $\forall (x, y) \in F \setminus \bar{F} : \delta'(x, y) = \delta(x, y)$, i.e. original arcs have the same delay.
- $\forall (t, p) \in \bar{F} : \delta'(\pi_1(\lambda(t, p)), \pi_2(\lambda(t, p))) = \delta(t, p)$, i.e. move the delay from output arc to input arc of the silent transition.
- $\forall (t, p) \in \bar{F} : \delta'(t, \pi_1(\lambda(t, p))) = \delta'(\pi_2(\lambda(t, p)), p) = 0$, i.e. all other arc delays in the newly introduced subnet are equal to zero.

**Definition 5.3.7 (Marking Relation ODO-IDO).** Let $N = (P, T, F, \delta)$ be a DTPN with the set of all reachable timed markings $M$ from an initial marking $m_0$. Let DTPN $\gamma_o(N)$ have the set of all reachable timed markings $\bar{M}$ from
an initial marking $m'_0$. A Marking Relation ODO-IDO, $\phi \subseteq M \times \bar{M}$ is defined for all $m \in M$, $\bar{m} \in \bar{M}$: $(m, \bar{m}) \in \phi$ if and only if $\forall p \in P, x \in Q$ such that

$$\nu(m, p, x) = \nu(\bar{m}, p, x) + \sum_{t \in N^p} \nu(\bar{m}, \pi_1(\lambda(t, p)), x - \delta(t, p))$$

**Theorem 5.3.6.** Let $N$ be a DTPN. Then $\gamma_o(N)$ and $N$ are branching bisimilar with respect to $\phi$.

We sketch the proof of branching bisimilarity with respect to the relation $\phi$: For each token in a place of the original model, there exists a token in the arc refined net, either in the corresponding place having the same timestamp, or in the pre-place of a silent transition that is producing a token in the corresponding place and having a timestamp less by a factor of the input delay of that silent transition. This means, when a token in a place becomes available in the original model, a token with the same timestamp is also available in the corresponding place of the arc refined net, otherwise a silent transition is enabled, firing which leads us to the previous case. Hence the same transitions can be fired in both markings.

### 5.3.3 Modeling IDO as ODO

We use Fig. 5.9 to explain the construction. Consider a subnet consisting of a place $P$ having transitions $A_1, \ldots, A_n$ in its postset with incoming delays $x_1, \ldots, x_n$. Note that place $P$ may have arbitrary additional input arcs and transitions $A_1, \ldots, A_n$ may have arbitrary additional input and output arcs. We order them,
if possible, such that their input delays \( x_1, \ldots, x_n \) are strictly non-decreasing. We replace all the arcs between place \( P \) and transitions \( A_1, \ldots, A_n \) by a subnet having one transition \( T_0 \) (silent transition) that consumes the tokens from \( P \) and puts them in \( a_1, \ldots, a_n \) with corresponding output delays \( x_1, \ldots, x_n \), respectively. The rest of the subnet is about garbage collection, i.e. as soon as one of the transitions \( A_1 \ldots A_n \) can fire the others should become disabled. If \( A_n \) fires it just consumes the tokens for all the others because they are ready for consumption. But if \( A_1 \) fires then it can’t consume the tokens for the others, so it triggers silent transitions \( T_2, \ldots, T_n \) by putting tokens in the places \( S_2, \ldots, S_n \). Note that transitions \( T_2, \ldots, T_n \) are able to fire without delay induced by arcs. Note that the net derived in this way is a timed inhibitor net and that timed inhibitors can be simulated by a DTPN for the two cases of bounded and unbounded places as presented before.

It is straightforward to formalize this construct and we omit a formal proof of the following theorem.

**Theorem 5.3.7.** Let \( N \) be a DTPN. Then \( \gamma_i(N) \) and \( N \) are branching bisimilar.

We will sketch the proof using the Fig. 5.9. For each token in the place \( P \) of the original net, either there exists a token in place \( P \) of the constructed net having the same timestamp, or there exists one token in each place \( a_1, \ldots, a_n \) having timestamps increased by delays \( x_1, \ldots, x_n \), respectively (by firing silent
transition $T_0$). If a transition $A_k$ for some $k < n$, is firing, then it is consuming one available token from each place $a_1, \ldots, a_k$ and producing one token each in the place $S_{k+1}, \ldots, S_n$, thereby disabling transitions $A_{k+1}, \ldots, A_n$ (due to inhibitor arcs) until silent transitions (garbage collectors) $T_{k+1}, \ldots, T_n$ have fired. If silent transition $T_0$ is firing again while transitions $A_{k+1}, \ldots, A_n$ are inhibited, then one token is produced in each place $a_k, \ldots, a_n$ having timestamps greater than equal to the maximal timestamp of tokens in that place. As garbage collector $T_k$ is delayed only by the token in the place $a_k$, transition $T_k$ is consuming the earliest token in the place $a_k$. So the correct tokens are collected. In case $A_n$ has fired, then it is able to consume all tokens from places $a_1, \ldots, a_n$, so does not need to enable any garbage collector. This means for every token produced by the transition $T_0$, one token is consumed by a transition $A_k$ and the rest are consumed by their respective garbage collectors as soon as the token becomes available. So if a transition $A_k$ for some $k \in \{1, \ldots, n\}$ is enabled in the original net, then the same transition must be enabled in the constructed net as well, because the token in place $a_k$ is available for consumption and the rest marking is the same. Furthermore, the garbage collectors are ensuring that the extra tokens produced by transition $T_0$ for each token in place $P$ are eventually consumed.

**Modeling time-outs.** Singleton input delays are handy for modeling of *time outs*, which are an essential construct in distributed systems, c.f. Fig. 5.10. The transition $y$ (receiver) is supposed to fire if $z$ (sender) has sent a message. If the delay $\delta(z, q)$ incurred due to sending is taking more than $d_1$ time units then transition $x$ (time-out) will fire. Input delays are also handy for garbage collection. Suppose transition $x$ has fired and afterwards transition $z$ then $w$ (garbage-collector) will remove the token produced by $z$ after $d_2$ time units. On the other hand, output delays, particularly with finite delay sets are handy for modeling stochastic behavior as seen in the previous chapter. For these reasons we consider $M_3$ as the best model for practice.
Simulating timed Petri nets and time Petri nets

In this section, we will establish the following relationship between model classes: $M_1 \subseteq M_3$ and $M_2 \subseteq_{fs} M_3$, i.e. (a) for every model of type timed Petri nets, there exists a model of class DTPN that is branching bisimilar, and (b) for every model of class Merlin time, there exists a model of class DTPN that is trace equivalent to it.

Simulating Timed Petri nets $M_1$ with Timed inhibitor nets $M_3$.

For model class $M_1$, we have the property $M_1 \subseteq M_3$. To verify this we construct for each model of $M_1$ a model of $M_3$ that is branching bisimilar. There are two cases, one where a transition may be firing concurrently with itself, and one where this is excluded. The constructions are displayed in Fig.5.11. Transition $t$ is refined by a silent transition $x$ and place $y$. The transition delay for $t$ in $M_1$ is transformed into an output delay of $x$ in $M_3$. Place $z$ with initially one token is used for the second case. It is straightforward to verify that the models are branching bisimilar.

Simulating Merlin time $M_2$ with Timed inhibitor nets $M_3$.

For model class $M_2$ we have $M_2 \subseteq_{fs} M_3$. To verify this, we construct for a model of class $M_2$, a model of class $M_3$ having ODO that is trace equivalent.

Assumption. Like most Merlin time proposals, we will assume that transitions cannot be enabled more than once at the same time, i.e. for all reachable markings, there is at least one place in the preset of a timed transition with at most one token. So it suffices to assume that at least one pre-place of a timed
transition is safe.

**Construction.** We will explain the construction using Fig. 5.12. We replace the incoming arcs to transition $T$ in the original net by a subnet consisting of four silent transitions: silent transition $T_0$, two cancel transitions and one garbage collector. Only the token in place $S_4$ is delayed by a selection of delay from the interval $[x, y]$. All other arcs have zero delays.

Note that transition $T_0$ is enabled only when the current time equals the maximum of the timestamp of tokens in $P$ and $Q$. So when transition $T_0$ fires the earliest tokens in the places $P$ and $Q$ are updated with the enabling time of $T_0$. As only the tokens older than the current time are updated with the current time, the enabling of transitions in the postset of places $P$ and $Q$ remain unchanged.

Furthermore, when transition $T_0$ is firing, a token with a delay (chosen from interval $[x, y]$) is produced in the place $S_4$ and a token without this additional delay is produced in the place $S_2$. The token in place $S_3$ is ensuring that only one enabling of transition $T$ is considered at a given time. If transition $T_0$ has fired already then $T_0$ cannot fire again before either $T$ has fired or one of the cancel transitions have fired. A cancel transition will fire if either the place $P$ or $Q$ (input places of $T$) becomes empty while there is still a token in the place $S_2$ (this token is available since its creation). Firing a cancel transition will return
the token back to the place $S_3$ (allowing $T_0$ to be enabled again in the next time step) and will produce a token also in the place $S_1$ in order to trigger the garbage collector to consume a delayed token from place $S_4$. When there are more than one delayed token in the place $S_4$, then the garbage collector will fire until the place $S_1$ has become empty and then transition $T$ is enabled.

There is an underlying order: If there is more than one token in the place $S_4$, then by the eagerness of transition firing, the garbage collector will consume the tokens that have the lowest time stamp in place $S_4$, so transition $T$ will fire with the token having the highest timestamp. So it is possible that transition $T$ may fire with a token from an earlier enabling as sketched by the following example:

Let $[x, y]$ be the interval $[3, 9]$. Suppose tokens arrive in place $S_4$ at times: 1, 2, 3 (i.e. creation time of tokens) with timestamps: 10, 5, 12, respectively. Suppose that transition $T$ is still enabled and there are two tokens in place $S_1$. Then the garbage collector will fire at 5 and 10, so $T$ fires at 12, which corresponds to the last enabling!

Now suppose tokens arrive at 1, 2, 3 with timestamps: 4, 11, 6. Suppose again that transition $T$ is still enabled and there are two tokens in place $S_1$. Then the garbage collector fires at 4 and 6 and transition $T$ at 11 which is not the last enabling!

**Observation.** Suppose tokens arrive in place $S_4$ at times $X_1$ and $X_2$ such that $X_1 < X_2$ with chosen delays $Y_1$ and $Y_2$ such that $Y_1 > Y_2$ (from interval $[X, Y]$), respectively. Then the enabling time of transition $T$ is $\max\{X_1 + Y_1, X_2 + Y_2\}$. Consider the two possibilities: (a) If $X_1 + Y_1 < X_2 + Y_2$ then the garbage collector fires at $X_1 + Y_1$ and transition $T$ fires at $X_2 + Y_2$, and (b) If $X_1 + Y_1 > X_2 + Y_2$ then $T$ fires at $X_1 + Y_1$. Observe that $X_2 + Y_2 < X_1 + Y_1 < X_2 + Y_1$, which means that transition $T$ fires at $X_2 + d$, where $d = Y_1 - X_2 + X_1$, and $Y_2 < d < Y_1$. Hence $d$ is a value in the interval of $[X, Y]$ and the property $Y_1 - X_2 + X_1 > Y_2$ iff $Y_1 + X_1 > X_2 + Y_2$ holds.

Since in the Merlin time semantics, delay is an arbitrary value chosen from an interval, it is straightforward to check that a Merlin time model is able to simulate any choice of delays made by its constructed DTPN model. However, when more than one enabling is considered at different time points, the constructed DTPN
model is not able to simulate its Merlin time model as choices made by silent transition $T_0$ during a previous enabling may interfere with the choice made by $T_0$ in the current enabling (as described in the case (b)). However using a lookahead strategy on firing sequences similar to the proof of theorem 5.2.7, we are able to prove trace equivalence.

**Theorem 5.3.8.** A time Petri net (Merlin time (MT)) and its constructed ODO model are trace equivalent.

**Proof. (sketch only).** We will use the notations of Fig. 5.12. Consider a sequence of transitions $\sigma = \langle e_1, d_1, e_2, d_2,\ldots, e_n, d_n, e_{n+1}, T \rangle$ that are causing a timed transition $T$ to be enabled and disabled $n$ times before eventually transition $T$ fires. Let $t_1, t_1 + s_1, t_2, t_2 + s_2,\ldots, t_n + s_n, t_{n+1}$ be the absolute time points corresponding each occurrence of a transition in the sequence $\sigma$.

- If the sequence $\sigma$ belongs to the MT model. Then we are able to execute the sequence $\sigma$ in the ODO model in the following way: In each enabling of transition $T$ caused by a transition $e_i$, for some $i \in \{1,\ldots,n\}$, silent transition $T_0$ is choosing a delay $\delta_i \in [X,Y]$ such that $\delta_i = \min(X, s_i)$ and $\max\{t_i + \delta_i \mid i \in \{1,\ldots,n\}\} \leq t_{n+1} + \delta_{n+1}$, where $\delta_{n+1}$ is chosen as $t_{n+2} - t_{n+1}$. These choices are feasible since $s_i \in [0,Y]$, because otherwise the sequence could not have occurred in the MT model and since $\forall i \in \{1,\ldots,n\} : t_i + s_i \leq t_{i+1}$, so $\forall i \in \{1,\ldots,n\} : t_i + s_i \leq t_{n+1}$ and since $\delta_i = \min(X, s_i)$, we may write $\forall i \in \{1,\ldots,n\} : t_i + \delta_i \leq t_i + \min(X, s_i) = \min(t_i + X, t_i + s_i) = \min(t_i + X, t_{i+1}) \leq t_{i+1} + X \leq t_{n+1} + X \leq t_{n+1} + \delta_{n+1}$.

- If the sequence $\sigma$ belongs to the ODO model. Then the timed transition $T$ will fire in the ODO model at $\max\{t_i + \delta_i \mid i \in \{1,\ldots,n+1\}\}$ and in the MT model the delay for the last enabling should be $\max\{t_i + \delta_i \mid i \in \{1,\ldots,n+1\}\} - t_{n+1}$ which is possible since $\delta_{n+1} \leq \max\{t_i + \delta_i \mid i \in \{1,\ldots,n+1\}\} - t_{n+1} \leq Y - t_{n+1} = Y$.

Note that for modeling of systems in practice, it is often possible to model synchronization by instantaneous transitions with zero input and zero output delays. Activities that take time are modeled by a \textit{begin} transition $X$ and an \textit{end} transition $Y$ as in Fig. 5.13. In this case, all model classes are equal since if $X$ fires at time $t$ then $Y$ will fire at $t + a + b$, where $a \in \delta(X, P)$ and $b \in \delta(P, Y)$.

### 5.4 Conclusions

In this chapter, we have identified a subclass of DTPN that can be transformed into a timed automaton that is suitable for model checking by the UPPAAL
toolset. However, the transformation is preserving language equivalence when output delays are considered and branching bisimulation when output delays are omitted. We considered several subclasses of DTPN and showed that they all have the same expressive power (Turing completeness, because they can express inhibitor arcs) but that some have better modeling comfort, i.e. they are easier for modeling. We also extended model class DTPN with a timed variant of inhibitor arcs (called the timed inhibitor net) that is also taking into account the availability of tokens and show that only in two cases, we may replace a timed inhibitor arc by a subnet modeled as a DTPN and yet preserve the simulation property. For modeling output delays seem to be the natural way to express that some activity takes time while input delays are handy for modeling time outs. If model engineers stick to the convention that synchronization actions should not take time and that time consuming activities should be modeled with one start event and one stop event, then most different time models boil done to the same!
CHAPTER 6

RELATIONSHIP TO SIMULINK SYSTEMS

Contents

6.1 Introduction .................................................. 130
6.2 Expressing Simulink Models using Petri nets ................. 131
   6.2.1 Signals, Blocks and System ................................. 131
   6.2.2 Modeling in Simulink .................................. 136
   6.2.3 Execution of Simulink Models ......................... 137
6.3 Expressing Petri nets using Simulink .......................... 142
6.4 Analysis ....................................................... 146
6.5 Relationship to the Architectural Framework ..................... 153
6.6 Conclusions .................................................... 154

Matlab/Simulink is a tool that is widely used to design and validate control algorithms for embedded control systems using numerical simulation. A Simulink model of a control system typically defines one or more control algorithms together with a model of its environment. Such models exhibit both discrete and continuous dynamics, simulated by discretizing time. The toolset also provides features to generate an implementation from these models. However, Simulink lacks a formal semantics. As a result, the behavior of a model during simulation and that of its generated implementation is not fully understood. In this chapter, we give a formal semantics to Simulink using the CPN formalism (sub-class of DTPN after discarding data), by describing how Simulink models can be expressed as a
CPN. We also show how Petri nets can be simulated in Simulink. For performance analysis of a Simulink model, we show how its corresponding CPN model can be used. We conclude by showing how Simulink models expressed as a CPN fits in our architectural framework.

6.1 Introduction

The use of Matlab/Simulink [355] is a de facto standard in the design of embedded control systems. A Simulink model describes a time driven dynamic system [87] as a set of mathematical equations, evaluated at discrete points in time. In general, a model of a control system consists of a set of controllers and its environment (also referred to as a plant). The goal is to define a mathematical model of a controller, given a model of its environment and a set of requirements that a controller must satisfy. Such models may contain both discrete (difference equations) and continuous parts (differential equations). An exact solution of such a model is in many cases not computable, therefore it is approximated by numerical simulation methods, i.e., by discretizing time into time steps, whose minimum is bounded by the resolution of the system. The results obtained from numerical simulation are used as a reference to validate the behavior of both the model and the implemented system (described in lower level languages like C).

An embedded control system is a discrete event system (DES) [87] whose state evolution depends entirely on the occurrence of discrete events (instantaneous) over time, like "button press", "threshold exceeded" etc. The underlying time-driven dynamics (expressed as difference and/or differential equations) of an embedded control system are captured by the notion of time progression in a state (i.e., the time elapsed between consecutive event occurrences).

On the one hand, the formal specification and verification of a DES is well studied in the context of automata theory [86, 169] and Petri nets [270, 250]. Both these formalisms have been extended with the notion of time [20, 333] (e.g., Timed Automata, time Petri nets) and support model checking of timeliness properties. On the other hand, Simulink has only informal semantics and of course an operational semantics in the form of an implementation in software. So it is not clear how a Simulink model can be incorporated into a formal framework.

Many attempts have been made to address this shortcoming by proposing translations of Simulink models into existing formal frameworks (see [11, 109, 317, 316]). However, most of these proposals restrict themselves to a subset of Simulink constructs and do not formally capture the behavior of a simulation in Simulink. It is only in [67], that a formal operational semantics of Simulink is defined. Unlike other approaches that focus on formalizing the solution method of equations encoded by a Simulink model, the semantics described in [67] captures the behavior of the simulation engine itself, describing the outcome of a numerical simulation.
In this chapter, we give a formal semantics to Simulink models using the Colored Petri nets (CPN) [178] formalism. A CPN is an extension of a Petri net with time and data. This makes them an ideal choice for modeling the behavior of Simulink systems. Furthermore, if we discard the data aspect (i.e. the color) then it is a subclass of DTPN. So the techniques developed in the Ch. 4 can be applied to verify their behavior by state space exploration. Currently, the modeling and analysis of a CPN is well supported by the popular CPN Tools [347]. Furthermore, we also consider how a Petri net can be expressed as a Simulink model. For performance analysis, we show how a CPN model expressing a Simulink model can be used. By relating the notion of time progression in a marking of a DTPN to the behavior of a CPN model of a Simulink system, we show how Simulink systems can be expressed in our architectural framework.

This chapter is structured as follows: In Sec. 6.2, we discuss basic concepts of Simulink and show how a Simulink model can be expressed as a CPN model. In Sec. 6.3, we show how a Petri net can be expressed as a Simulink model. In Sec. 6.4, we show how performance properties of a Simulink model can be verified. In Sec. 6.5, the relationship between Simulink models and models of our architectural framework is discussed. In Sec. 6.6, we present our conclusions.

6.2 Expressing Simulink Models using Petri nets

In this section, we describe the basic concepts underlying a Simulink model, namely signals, blocks and system. For these concepts, we give a formal semantics using the CPN formalism. First, we show how a Simulink block can be modeled as a CPN, which we will call a P-block. A few simple rules describe how a set of P-blocks can be connected to construct a system, which we will call a P-system. The blocks of a Simulink system are executed in a certain order called the block execution order. We show how the block execution order of a Simulink system can be modeled as a CPN, on top of an existing P-system. The resulting system describes fully the behavior of a simulation in Simulink and we call it a C-system.

6.2.1 Signals, Blocks and System

Signals and Blocks are the two basic concepts of a Simulink model. A signal is a step function in time, representing the behavior of a variable. So only at discrete points in time, the value of a variable may change. Furthermore, a signal in Simulink can have one of the following types: integer, real, complex or its multi-dimensional variants.

A block defines a relationship between a set of signals and possibly a variable representing its state (state variable). If a block has a state variable then it is a stateful block, otherwise it is a stateless block. The set of signals belonging to a
A block are either of type: input or output. A block can have zero or more inputs and at most one output and one state variable.

A stateless block defines an output function $f : I^n \rightarrow O$, where $I$ is the set of inputs, $O$ is the set of outputs and $n$ is the number of inputs of the block. A stateful block defines an output function $f : I^n \times S \rightarrow O$ and an update function $g : I^n \times S \rightarrow S$, where $I$ is the set of inputs, $O$ is the set of outputs, $S$ is the set of state variables and $n$ is the number of inputs of the block. The output function of a stateful block must be evaluated before its update function is evaluated. The result of evaluating the update function of a stateful block (say unit delay block as in the Fig. 6.1), updates its state variable $S$ which is expressed by $S'$, so $S' = g(X, S)$. Furthermore, the initial condition of a stateful block is specified by the initial value of its state variable.

A block also has an associated sample time that states how often a block should repeat its evaluation procedure. The sample time of a block must be a multiple of the time resolution of a system called the ground sample time.

A Simulink model is a directed graph where a node corresponds a block repre-
ented as either a triangle, rectangle or circle and a directed edge between blocks corresponds to a signal. A signal models the data dependency between the output of one block and the input of another block. An output from a block can be connected to input of one or more other blocks. Note that the output of a block can be split into two or more copies which serves as an input to multiple blocks. However, the output signals of two or more blocks cannot join to become one input for another block. We call a network of blocks connected over signals in this manner as a system. If all blocks of a system have the same sample time then we call it an atomic system. A subset of blocks belonging to a system and having the same sample time is called an atomic subsystem.

A mathematical model of a dynamic system is a set of difference, differential and partial differential equations. Such equations can be modeled as a system in Simulink. A simulation of such a system solves this set of equations using numerical methods. The state of a system is defined as the valuation of all its output variables (signals) and state variables. So from an initial state of the system the behavior in the state space is computed in an iterative manner over discrete time steps bounded by the ground sample time. A block having a sample time equal to the ground sample time is called a **continuous block**. All other blocks are **discrete blocks** and have a sample time that is equal to an integer multiple of the ground sample time. For continuous integration, the whole simulation is repeated several times within one ground sample time (depending on the numerical integration method, i.e. solver type) to get a better approximation and detect zero crossing events [67]. Note that discrete blocks change their outputs only at integer multiples of the ground sample time which implies that the input to continuous integration blocks remains a constant. However, we neglect the numerical refinement of the ground sample time and consider only one integration step per ground sample time.

The output and update functions of a block can be programmed in Simulink using a low level programming language such as C. However, some commonly used constructs are available as predefined configurable blocks in Simulink. In the Fig. 6.1, we give a few examples of commonly used blocks in Simulink consisting of five stateful blocks (unit delay, discrete derivative, discrete integrator-A, discrete integrator-I, continuous integrator) and three stateless blocks (gain, add/sub and switch). We assume a ground sample time of \( h \) time units and \( k \in \mathbb{N} \).

The **unit delay** block is a stateful block having one input signal \( X \), a state variable \( S \) and one output signal \( Y \). A unit delay block serves as a unit buffer by delaying the current value of its input signal by one sample time, i.e. \( h.k \) time units in the following way: The output function \( f \) assigns the current value of its state variable to its output \( Y \). The update function \( g \) copies the current value of its input signal to its state variable. After every \( h.k \) time units, the output function is evaluated and then its update function.

The **gain** block is a stateless block that produces as its output the current value of its input signal multiplied by some constant (specified as a block parameter). The **add/sub** block is a stateless block that produces as its output the
sum/difference of the current value of its input signals. The *switch* block is a stateless block that produces as its output either the current value of signal $X$ or $Y$ depending on the valuation of the boolean expression defined over signal $C$.

The *continuous integrator* block is a stateful block that receives as its input (signal $X$) the derivative of the state variable $S$ (i.e. the rate of change of valuation of the state variable). The output function $f$ assigns the current value of its state variable $S$ to its output $Y$. The update function $g$, updates the value of its state variable by integrating the product of the derivative and the ground sample time. The *discrete integrator-I* block is similar to the continuous integrator block. The only difference is that the sample time of the block is an integer multiple of the ground sample time. The *discrete integrator-A* block accumulates the sum of values of its input signal and updates its state variable with this value. The output function of this block copies the current value of its state variable to its output. Note that Simulink blocks such as *transfer function* and *state space* are similar to continuous integrator blocks.

**Modeling Simulink Blocks.** In the Fig. 6.2, we show with an example (add block) how a stateless block can be modeled as a P-block. A signal is modeled as a place having one colored token (see places $X$, $Y$ and $Z$). The valuation of this colored token is a step function over time, i.e. its value maybe updated by a transition at discrete points in time. A stateless block has two transitions *compute* and *done* connected over a shared place *busy*. The former transition has an incoming arc from each of its places modeling its inputs and the latter is connected with bi-directional arcs to the place modeling its output. The availability of the token in place *enable* concerns the sampling rate. The incoming arc to the place enable, specifies the sample time of the block. In our example, the add block has a sample time of 3, since the token in the place enable is delayed by 3 time units, each time the compute transition fires. When a *compute* transition fires, it consumes one token each from its inputs ($X$ and $Y$) and produces one token in the place *busy* having a color value determined by evaluating its output function. For an add block, the output function $f(X,Y) = X + Y$. The execution time of the compute transition (called the *block execution time*) is modeled as a delay along the outgoing arc from the *compute* transition (see delay of 1 time unit along the incoming arc to the place *busy*). Note that it is also possible to specify the block execution time as a time interval. When the token in the place *busy* becomes available, transition *done* consumes this token and produces back one token in each of the inputs of the block (places $X$ and $Y$ with color value unchanged) and updates the color value of the token in its output with the computed result from the output function. This way of modeling a P-block is good for understanding its behavior. However, for a compact representation, the model of a P-block can be simplified by merging together the transitions *compute* and *done* into one transition. In the Fig. 6.3, we show a simplification of the add block. However, such a model can only be used if the block execution time is specified as a fixed point delay. This is because a time interval specifying a block
execution time allows different choices of timestamps for tokens produced in the input and output places of the block. For the remainder of this section, we will consider block execution times as a fixed point delay and use the simplified model of a P-block to present our concepts.

The P-block of a *stateful block* has all the concepts of a stateless block, and in addition is extended in the following way: (a) add a special place state containing one colored token representing the state of the block and connect it with bi-directional arcs to the block’s transition specifying the output function, (b) define the update function along the outgoing arc to place state. In the Fig. 6.4, we give an example of an unit delay block modeled as a stateful P-block. The update function $g(X, S) = X$ is updating the state variable and the output function $f(X, S) = S$ is updating the value of the token residing in the output place of the block.
6.2.2 Modeling in Simulink

In the Fig. 6.5, we present an example of a cruise control system modeled in Simulink. The model is an adaptation of the example in the paper [67]. We consider this example because it is simple and covers all relevant modeling aspects of Simulink. The system comprises of three main parts: (a) the plant (the device being controlled), (b) the PI (proportional-integral) controller, and (c) the safety mechanism. The plant approximates continuous behavior whereas the other parts are discrete. The safety mechanism ensures that the velocity of the plant does not exceed a specified limit.

The plant models continuous behavior of a vehicle whose speed $v$ and position $x$ is given by the equation: $m\ddot{x} = -bv(t) + u(t)$ and $\dot{x} = v(t)$, where $m$ is the mass, $b$ is the friction coefficient, $\dot{x}$ and $\dot{v}$ are the derivatives of $x$ and $v$ w.r.t. time $t$ and
$u(t)$ is the power of the engine over time. The above equation is implemented in the Simulink model (see Fig. 6.5) consisting of one sum block, two gain blocks that multiply their input by a factor $1/m$ and $-b/m$ and one continuous integrator block labeled $dv$. All blocks of the plant have a sample time equal to the ground sample time $h$ of the system.

A standard PI controller is modeled with its two parts labeled proportional and integral. The goal of the PI controller is to produce as output, the power demand $u(t)$ such that the the velocity of the plant $v(t)$ converges to the desired velocity, produced as output of the switch block, as quickly as possible. Note that the unit delay block acts as a unit buffer and serves for discrete integration (Euler method). The remaining blocks define the safety mechanism. The switch block monitors the current velocity of the plant. If it exceeds some specified threshold, then it produces as its output, the output of the limit block, otherwise the output of the set-point block is chosen. All blocks of the PI controller and Safety are discrete and have a sample time of $h.k$ time units, where $k$ is an integer.

Modeling Simulink System. Given a set of P-blocks, we model a P-system by fusing the input places of blocks with the output place of blocks, representing the same signal (i.e. having the same place label). Note that two or more blocks are not allowed to have the same output place because in Simulink, blocks have only one output and this is modeled in P-blocks with a unique output place. To keep the figure readable, we consider only the integral part of the PI controller (with $h = 1$ and $k = 10$) and model it as a P-system in the Fig 6.6.

6.2.3 Execution of Simulink Models

The blocks of a Simulink model are executed in a sorted order. To determine this sorted order, we distinguish between two type of blocks, namely independent blocks and dependent blocks. If the output function of a block is defined over only its state variable (i.e. does not depend on its inputs), then we call it an independent block, otherwise we call it a dependent block. The order in which independent blocks are evaluated is not important. However, the order in which the output of dependent blocks are evaluated is important because the output of a dependent block must be computed only after all other blocks that produce an input to this block have been evaluated. This kind of dependency induces a natural ordering between blocks of a Simulink model which can be represented as a directed graph (blocks as nodes, dependency between blocks as directed edges) representing the order in which blocks of a Simulink model must be evaluated, i.e. a directed edge from block $A$ to block $B$ indicates that block $A$ must be evaluated before block $B$. We call this graph as the dependency graph of a Simulink system. For the example presented in the Fig. 6.5, the dependency graph between blocks is shown in the Fig. 6.7. The block sorted order is a sorted sequence of blocks whose ordering satisfies the dependency graph of a Simulink system. The sorted order of a Simulink system is determined by the simulation engine before the start.
Figure 6.6: P-System: Integral Part (PI Controller)
The simulation engine of Simulink executes the contents of a Simulink model (blocks) according to the block sorted order. Each execution of the block sorted order is called a simulation step. In each simulation step, every block occurring in the model is visited according to the block sorted order and the following condition is checked for each block: if the time elapsed since this block’s last execution equals its sample time, then the block produces its output, otherwise the block execution is skipped, i.e. the block’s functions are not reevaluated. The time that is allowed to elapse between each simulation step is a multiple of the ground sampling time called the step size. The value of the step size for a given Simulink model can be either specified explicitly or it is determined by the simulation engine such that the bounds on approximation errors of integrator blocks are within some specified threshold.

The simulation engine of Simulink is able to execute a model under two different simulation modes, namely fixed step solver and variable step solver. In each of these modes, the Simulink tool provides a choice of various commonly available solvers for ordinary differential equations. If a fixed step solver is chosen then a simulation step occurs every step size time units. If a variable step solver is chosen then a simulation step occurs at the earliest time instant when at least one block in the model exists such that the time elapsed since its last execution is equal to its sample time. The logic underlying the variable step solver mode follows:

Given the initial conditions of a Simulink system: initial condition of stateful blocks, current simulation time: $t = 0$, end time: $t_f$, ground sample time: $h$, initial step size: $h.k$, where $k$ is an integer, block sorted sequence: $\sigma = \langle b_1; \ldots; b_k \rangle$, where $\{b_i \mid i \in 1\ldots k\}$ is the set of $k \in \mathbb{N}$ blocks, the simulation loop of a variable step is as follows:

- Loop until $t \leq t_f$
  - Evaluate output function of all blocks in the order specified by $\sigma$;
  - Evaluate update function of all stateful blocks;
– Detect zero crossing events (see [67]);
– Update $t = t + l$; Determine the next simulation step size $l$ (an integer multiple of $h$);

In the fixed step solver mode, the simulation step size is a constant and zero crossing cannot be detected.

**Modeling Simulink Control Flow.** In the previous section, we have seen how an arbitrary Simulink system can be expressed as a P-system. The enabling of P-blocks in a P-system is determined by their sample time (i.e. by the availability of token in the place *enable*). This means that in each simulation step of the model, multiple P-blocks with the same sample time can become enabled. In CPN semantics, the choice of executing a P-block is done in a non-deterministic manner. However, in a Simulink simulation, the blocks of a Simulink model that can produce their output at a simulation step (determined by their sample times), are evaluated according to their block sorted order.

Consider the P-system of the integral part of the PI controller in the Fig 6.9. This system has four blocks, namely one sum block (sum2), two gain blocks (gain-h and gain-ki) and one unit delay block. For this system, one of the block sorted order satisfying the dependency graph of the Fig. 6.7, is the sequence: $(unit delay; gain-h; sum2; gain-ki)$. This block execution order of P-blocks of a P-system is modeled by first adding a *trigger mechanism* to each P-block and then connecting the trigger mechanism of blocks according to the block sorted order. We call the resulting system a *C-system*.

The trigger mechanism is modeled on top of a P-block by adding an input
Figure 6.9: C-System: Integral Part (PI Controller)
place trigger (labeled trig), an output place acknowledge (labeled ack) and a timed inhibitor transition (inhibitor arc that accounts for the availability of token in place enable) called skip. If the token in place trigger is available and the token in place enable is unavailable, then the skip transition is enabled due to the timed inhibitor arc. Firing the skip transition does not progress time and the execution of the block is skipped in the current time step. If the skip transition is disabled (due to an available token in place enable), then the block transition (unit delay) fires at its enabling time and produces a token in the place acknowledge, delayed by the execution time of the block (see inscription \textit{CF@1}). In both cases, one token is produced in the place acknowledge. In the Fig. 6.8, we give an example of a P-block, extended with a trigger mechanism.

Next, we model the block execution order of a system by introducing a new transition called the glue transition between the acknowledge and trigger places of successive blocks as they occur in the sorted order. The construction is carried out in the following way: For each block occurring in the block execution order, we add one glue transition that consumes a token from the place acknowledge of the preceding block and produces a token in the place trigger of this block. In this way, the C-system defines a simulation run of the system. In order to allow for more than one run of a simulation at a rate specified by the step size: (a) we add a transition labeled closure that consumes a token from the place acknowledge belonging to the last block in the sorted order and produces a token in the place trigger corresponding the first block in the sorted order, and (b) we initialize the place acknowledge of the last block occurring in the block sorted order with a token having a timestamp zero. Furthermore, to this closure transition, we connect a place called GST with bidirectional arcs and having one token. On the incoming arc to the place GST, we associate a delay corresponding the step size of the simulation (multiple of ground sample time). As a result, a simulation run can only occur once every step size time units. If the model has continuous blocks then the step size must equal the ground sample time. To simulate a variable step solver, the step size must be equal to the least sample time of all blocks in the system. In the Fig. 6.9, we describe the C-system of the integral part of the PI controller having a step size of 10 time units.

### 6.3 Expressing Petri nets using Simulink

In this section, we will express the semantics of a Petri net as a Simulink system by modeling a place and a transition as an atomic subsystem. The underlying strategy is simple: The place subsystem keeps track of the number of tokens and informs all successor transitions about this value. When a transition subsystem either consumes/produces a token, it indicates the predecessor/successor place subsystem about the occurrence of this event.

In the Fig. 6.10, we give an example of a Petri net consisting of one place $P$, and two transitions $t_1$ (pre-transition) and $t_2$ (post-transition), and show how the
place \( P \) of a Petri net can be modeled as an atomic subsystem (place subsystem). The place subsystem has two input signals (signals: \( a \) and \( b \)) and one output signal (\( b' \)). The signals \( a \) and \( b \) are assumed to be updated by the pre and post transitions \( t1 \) and \( t2 \), respectively and the signals can either have a value of 0 or 1. A change in the value of either of these signals indicates that a transition has altered the number of tokens in the place. The output signal \( b' \) is an input signal to transition \( t2 \) and has a data type integer, whose value represents the number of tokens in a place subsystem. We call signals \( a \) and \( b \) as indication signals and signal \( b' \) as a token signal.

The place subsystem has three stateful blocks (two monitor blocks and a unit delay block) and one stateless block (place logic). All the four blocks have the same sample time equal to \( y \) time units. The input monitor block receives an indication signal from a pre-transition and the output monitor block receives an indication signal from a post-transition. The two monitor blocks compare the current value of their input signal with the value recorded (i.e. state: \( x \)) from the previous time step. If they are equal then the block produces an output: false, otherwise it produces an output true. The state of an unit delay block corresponds the number of tokens in the place in the current time step. At every time step, the unit delay block produces as its output, the current value of its state variable. The initial state of the unit delay block corresponds the initial number of tokens.

Figure 6.10: Modeling a Place in Simulink
in the place. The \textit{place logic} block defines a function to compute the number of tokens in the current time step depending on the output of monitor blocks \texttt{aout} and \texttt{bout} and the output of the unit delay block \texttt{S}. If either \texttt{aout} or \texttt{bout} is true, then one of the transitions must have changed the number of tokens in the place and the value of signal \texttt{b'} must be updated (i.e. either increased or decreased by one), otherwise \texttt{b'} remains unchanged, i.e.

\[
\text{PL}(aout, bout, S) = \begin{cases} 
S + 1, & \text{if } aout == \text{true and } bout == \text{false} \\
S - 1, & \text{if } aout == \text{false and } bout == \text{true} \\
S, & \text{otherwise}
\end{cases}
\]

If a place subsystem has more than one pre-transition, then we add for each pre-transition, one indication signal (input) and a corresponding input monitor block whose output is connected to the place logic block. The logic underlying the place logic block is extended to handle an additional input condition. If a place subsystem has more than one post-transition, then we add for each post-transition, one indication signal (input) and a corresponding output monitor block whose output is connected to the place logic block. Additionally, the token signal (output) is split as an input for each post transition.

In the Fig. 6.11, we give an example of a Petri net consisting of one transition \texttt{T} and two places \texttt{p1} (pre-place) and \texttt{p2} (post-place), and show how transition \texttt{T} of a Petri net can be modeled as an atomic subsystem (transition subsystem).
Figure 6.12: Connecting Places and Transitions
The transition subsystem has one input signal $b'$ (token signal) as an output from pre-place $p1$ and one output signal $b$ (indication signal) that is split as an input to both its pre-place $p1$ and post-place $p2$. The current value of signal $b'$ is representing the number of tokens in the transition’s pre-place $p1$. The signal $b$ has either value 0 or 1.

The transition subsystem has two blocks having the same sample time of $z$ time units. The unit delay block has a state either 0 or 1 and in each time step, produces as its output the state of the block in the previous time step. The output of the unit delay block is connected to the transition logic block. The transition logic block defines a function that produces as its output a value of 0 or 1 depending on the value of the token signal $b'$ (from a pre-place) and the output signal of the unit delay block. If the value of signal $b'$ is greater than zero, then it means there are enough tokens and transition $T$ can fire. The firing of a transition is indicated by a change in the value of the indication signal $b$ such that it has a value that is not equal to the output of the unit delay block. The function is defined as follows:

$$TL(b', c) = \begin{cases} 
1, & \text{if } c == 0 \text{ and } b' > 0 \\
1, & \text{if } c == 1 \text{ and } b' \leq 0 \\
0, & \text{if } c == 1 \text{ and } b' > 0 \\
0, & \text{if } c == 0 \text{ and } b' \leq 0
\end{cases}$$

If a transition subsystem has more than one pre-place, then we add for each pre-place: one token signal as an input to the transition logic block, and split the indication signal (output) as an input to the pre-place subsystem. The logic of the transition logic block is extended to change its value if and only if all of its inputs have a value greater than zero. If a transition subsystem has more than one post-place, then we split for each post-place, the indication signal (output) as an input to the post-place subsystem.

In the Fig. 6.12, we give a simple example of a Petri net (consisting of two places $P$ and $Q$ and one transition $T$), and its equivalent Simulink model by connecting the two place systems with the transition subsystem over their shared signals. Note that in a network of places and transitions connected in this manner, the place subsystems must produce their outputs faster than the transition subsystems. Furthermore, if more than one transition share the same pre-place then, they must have different sample times.

### 6.4 Analysis

In this section, we discuss how timed properties of a C-system can be verified by model checking techniques using a simple example. In the Fig. 6.13, we present an atomic subsystem modeling Euler integration with two blocks: unit delay and sum, both having a sample time equal to 10 time units and a simulation step size
of 15 time units. The execution time of the unit delay block lies in the interval of [1, 3] time units and the execution time of the sum block is 2 time units. For analysis purposes, we modify the C-system with one additional transition `complete` and one place `init` (initialized with a token) with no delays between them.

If we consider the notion of colored tokens in a state, then analysis is possible only for a finite future (partial state space). If we drop the notion of color from a CPN, then we obtain a subclass of Discrete Timed Petri nets (DTPN) [38, 333]. So for analysis of a C-system, we will ignore color and discard places modeling signals and states. As the state space of a DTPN is infinite because (a) time intervals on outgoing arcs leads to an infinite choice of timestamps for newly produced tokens, and (b) time progression is non-decreasing; model checking is not directly possible. For this a reduction method is proposed that reduces the state space of a given DTPN into a finite one. The reduction method progresses in two stages: interval and time reduction.

The interval reduction step, replaces each time interval on outgoing arcs of a DTPN with a set of finite values in the following way: For each time interval specified along an outgoing arc, construct a set containing its lower bound, upper bound and a finite number of equidistant points between the two bounds such that the points are separated from each other by a so called grid distance. The grid distance is the least common multiple of the denominators of all non-zero elements of the set consisting of all token timestamps in the initial marking, and the lower and the upper bounds of all time intervals expressed as non-reducible
148 RELATIONSHIP TO SIMULINK SYSTEMS

Figure 6.14: sDTPN of C-System

elements of the set of rational numbers. The resulting net is called an fDTPN (preserves simulation equivalence). An fDTPN can be transformed into a DTPN with outgoing arcs having only singleton delays by making copies of transitions and associating each of their outgoing arcs with a unique element from the finite set of delays associated with that arc. The resulting net is called an sDTPN (preserves strong bisimulation).

In our example, the C-System (DTPN) has a grid distance equal to 1, so the time interval $[1,3]$ associated with the outgoing arc from the unit delay transition is replaced by the finite set $S = \{1,2,3\}$ and the delay of 2 time units on the outgoing arc from the sum transition is replaced by the singleton set $\{2\}$. The resulting system is an fDTPN. Next, the choice of delays represented by the set $S$ on an outgoing arc from one unit delay transition is simulated by three copies of the unit delay transition (same preset and postset), each having a unique delay from the set $S$ specified along its outgoing arc. The resulting sDTPN is shown in the Fig. 6.14. Note that transitions unit delay 1, unit delay 2 and unit delay 3 simulate the behavior of the transition unit delay (compute).

The time reduction step, transforms a given sDTPN into a finite transition system (rDTPN) by reducing each timed marking of an sDTPN (concrete marking) into a so called abstract marking. An abstract marking represents a class of concrete markings by aggregating them based on relative distances between
Figure 6.15: rDTPN

Transition Labels

- t1: closure
- t2a: unit delay 1 (compute)
- t2b: unit delay 2 (compute)
- t2c: unit delay 3 (compute)
- t3: unit delay (done)
- t4: glue 1
- t5: sum
- t6: complete
Figure 6.16: rDTPN of sDTPN with Step Size: 10
timestamp of tokens. An abstract marking of a concrete marking is obtained by subtracting the timestamps of all tokens in the marking by its firing time. This means the firing time of an abstract marking is zero. It turns out that for any bounded sDTPN there exists a finite transition system (of abstract markings), called an rDTPN (preserves strong bisimulation).

As all places have at most one token in any reachable marking, we identify a token in a place by the name of that place and define a marking as a function that assigns to each token identity, a timestamp. A marking is represented as a vector of timestamps corresponding the ordered set of places: \([S.ENABLE; U.ENABLE; U.TRIG; BUSY; U.ACK; S.TRIG; S.ACK; INIT; GST]\). In the Fig. 6.15, we present the rDTPN of our sDTPN. Consider the reachable concrete markings \([11, 10, -,-,-,-,3,15]\) and \([26, 25,-,-,-,-,-,18,30]\) from the initial marking of the sDTPN in the Fig. 6.14. From both these markings, the earliest enabled transition is the closure transition at times 15 and 30, respectively. So their abstract marking is \(L14 : [(-4), (-5), -,-,-,-,(-12),0]\).

**Performance Analysis.** To analyse a C-system, we must be able to compute the upper and lower bounds on the time required to reach a marking \(m\) from its initial marking \(m_0\), i.e. timed reachability.

First, we show how to compute the time taken to execute a sequence of transitions \(\sigma = \langle t_0; \ldots ; t_n \rangle\), starting from the initial marking \(m_0\) and leading to marking \(m\). Consider our sDTPN (see Fig. 6.14) and its rDTPN (see Fig. 6.15). As all token timestamps are zero in the initial marking \(m_0\), the initial marking \(l_0\) of the rDTPN is an abstract marking of \(m_0\). Due to bisimulation equivalence between an sDTPN and its rDTPN, the same sequence \(\sigma\) exists from initial marking \(l_0\), leading to a marking \(l\) such that it is an abstract marking of \(m\).

Then for each transition \(t \in \sigma\), leading from an abstract marking \(l'\) to the abstract marking \(l''\), compute the relative delay of marking \(l''\) in the following way: Consider the marking \(l'\) as a concrete marking of the sDTPN. From this marking, fire the enabled transition \(t\), leading to a marking, say \(\tilde{m}\). The firing time of marking \(\tilde{m}\) (i.e. the earliest enabled transition) is the relative delay of abstract marking \(l''\). Note the relative delay of initial marking \(l_0\) is always zero.

As an example, consider the marking \(L2\) reachable from marking \(L1\) over transition \(t2a\). Consider the marking \(L1\) as a concrete marking of sDTPN. Firing transition \(t2c\) from this marking leads to marking \([0,10,-,1,-,-,-,15]\) (in the sDTPN) with a firing time 1. So the relative delay of \(L2\) is 1.

The time taken to execute \(\sigma\) is the sum of all relative delays of all abstract markings visited by the transitions of this sequence. As an example, consider the marking \(L12\) reachable from \(L0\) by the sequence of transitions \(\langle t1;t2b;t3;t4;t5 \rangle\). All the abstract markings visited by this sequence, except for \(L3\) and \(L12\) have a non-zero relative delay of 2 time units each. So the time taken to execute this sequence is 4 units.

Next, the lower and upper bounds on the time required to reach marking \(m\) from marking \(m_0\) in the C-system corresponds the minimum and maximum of
execution times of all possible transition sequences starting from marking $l_0$ and leading to marking $l$, respectively. A desirable property of a well-defined C-system is formulated as: the upper bound of a run of the block execution order must never be greater than the simulation step size. As an example, consider the set of shaded abstract markings called home markings in our rDTPN. These markings correspond the completion of a block execution order (token in the place acknowledge of the sum block). So every path starting from an initial marking or a home marking and leading back to a home marking represents a run of the block execution order. For this, the upper and lower bounds of the time can be computed by the techniques described so far. For our example, this turns out to be in the interval of [3, 5] time units. Note that we have chosen a trivial example to present our concepts. For a multi-rate system, the analysis and the type of questions become more interesting.

**Model Validation.** An rDTPN can also be used to validate the timed specification of a Simulink model. The sample time of a block of a Simulink system determines whether its output is computed during a simulation run (i.e., occurs in a simulation run). So blocks having the same sample time must occur during the same simulation run. Note that the unit delay and sum blocks have the same sample time. From its rDTPN, we verify that in every path between home markings, one of the unit delay and sum transition always occurs. If the simulation step size of the sDTPN is decreased to 10 time units then there exist paths between home markings where the skip transition of the sum block occurs because the block is not ready yet. This behavior is undesirable and can be observed in the rDTPN shown in the Fig. 6.16. Observe that the firing sequences $\langle t_1; t_2; t_3; t_4; t_5; t_6; t_1; t_2; t_3; t_4 \rangle$, $\langle t_1; t_2; t_3; t_4; t_5; t_6; t_1; t_2; t_3; t_4 \rangle$ and $\langle t_1; t_2; t_3; t_4; t_5; t_6; t_1; t_2; t_3; t_4 \rangle$, lead to a marking where the skip transition is enabled.

**With Control Logic.** Other interesting features of Simulink like decisional constructs (if-then-else and switch) and loop constructs (for and while) can also be verified using the above techniques. The four constructs can be represented in the C-system by allowing for a non-deterministic choice between triggering mechanisms of blocks. In the Fig. 6.17, we give two examples.

**With Communication Channel.** In many Simulink systems, messages may have to be exchanged over one or more communication channels (supported by UDP/TCP send-receive blocks). Such a message passing mechanism is easily modeled as a Petri net, which can in turn be expressed as a Simulink system as a network of place and transition subsystems. In the Fig. 6.18, we give an example of a Petri net modeling a communication channel with buffer size 5. For a C-system with a communication channel, interesting properties like buffer utilization and system performance in the presence of communication delays may be analyzed.
6.5 Relationship to the Architectural Framework

A Simulink model is a time driven system, i.e., systems where the evolution of states are a consequence of time progression. The progression of time in our discrete event architectural framework (extended to a DTPN) occurs in a marking, quantified as the difference between the firing time and creation time of this marking. The firing time of a marking is determined by the timestamps of tokens in that marking. A token resides in a place and depending on its timestamp and the current model time, it may either be available or unavailable for consumption. An internal place of a component represents an activity. If this place is marked
with a token then it is an ongoing activity.

So it is natural to capture the time driven dynamics of a Simulink system by refining a place representing these dynamics with its corresponding Simulink model. For this, the place refinement strategy may be used, but only after structural modifications to the Simulink model have been carried out. We describe these modifications below.

- Discard the closure transition connecting the trigger place of the first Simulink block and the acknowledge place of the last Simulink block. We will call the former as the initial place and the latter as the final place of the Simulink model.

- Transform the CPN model into a DTPN model by discarding the data (color) aspects.

In this way, we have transformed a CPN model of a Simulink system into a workflow net. However, only the busy places of GFR or RPC servers and all places of MP servers and root components can be chosen for refinement with a Simulink model. If the busy place of a GFR server has been refined in such a way, then it is possible for the Simulink model to execute in a cyclic manner (until it either succeeds or aborts), while giving feedback about its progress and having the possibility to be canceled. This is not the case for the busy place of an RPC server. To model a cyclic execution of a Simulink model, we must first elaborate the busy place of the RPC server by adding a loop structure (using the well-known loop addition and transition refinement techniques). We may then choose a place in this loop to refine with a Simulink system. The same strategy may be applied to root components and MP server components, if a cyclic execution of a Simulink model is desired.

6.6 Conclusions

Simulink is a graphical way of modeling a set of difference and differential equations whose result is simulated by discretizing time. We have shown how the
simulation behavior of Simulink models can be expressed using the formalism of CPN. As models of this formalism can be model checked, the safety and performance properties of a Simulink model can be guaranteed. We also showed how Petri nets can be simulated in Simulink. So the two formalisms are formally equivalent in their expressive power. However, the modeling comfort is different. Petri nets are a better choice for modeling discrete events like message passing, whereas Simulink has more built-in facilities for numerical approximation of differential equations, than existing tools for Petri nets. Furthermore, the formal study of DES is deeply rooted in the theory of Petri nets and automata theory. So the specification of both discrete and continuous parts of a Simulink model using the same formalism, gives a better understanding of their behavior within the context of DES.

Future Work. Another widely used modeling feature of the Simulink toolset is Stateflow (special block in Simulink). It is an extension of Harel’s statecharts [159] and is commonly used to specify discrete event control logic. Its many features include state hierarchy (i.e. nesting), concurrency etc. The Simulink toolset provides a nice integration between the control-oriented view of Stateflow and dataflow oriented view of Simulink blocks. However, Stateflow is a complex language with informal semantics. It has numerous complicated and often overlapping features that are not very well understood [156, 291]. As a consequence, unexplained runtime failures could occur. To counter some of these issues, a number of programming guidelines have been proposed and some of them are widely used. These guidelines restrict the modeling freedom to a subset of features. However, they have no formal basis. As future work, we intend to give semantics to Stateflow using the CPN formalism, thus combining the work presented in this chapter and consequently our architectural framework.
In this chapter, we demonstrate the applicability of our architectural framework to software systems of robots. For this we present our design experiences from the International Thermonuclear Experimental Reactor Remote Handling (ITER RH) Plant and Remotely Operated SErvice (ROSE) robot. The former is a maintenance facility of the envisioned nuclear fusion reactor ITER, while the later is a tele-operated service robot (TSR) for home care of elderly and disabled people.
For the ITER RH plant, we first present a brief background of the ITER project, its envisioned objectives and challenges. At its current state, the modular decomposition of system functionality has been documented in various design documents. We use these informal textual descriptions, in consultation with extensive stakeholder meetings, to make a first proposal for a formal software architecture of the ITER RH plant.

For robot ROSE, we first present its envisioned benefits to society. Then we present a set of use cases that ROSE must realize to achieve its goals. From this set of use cases, we show how a set of components and their interaction patterns can be identified and represented as an architecture diagram. Then we use our construction method to derive a Petri net model of the system. For a few interesting components, we describe their functional design in more detail and show how this can be represented in the model. For verification of timed properties, we extend the Petri net model to a DTPN and show how UPPAAL can be used for model checking them. To reduce computation effort of the verification process, four behavior preserving structural reduction rules for DTPN are prescribed.

7.1 Introduction

Remote Handling. Remote handling is a form of tele-operation. Tele-operation is probably the oldest form of robotics. Tele-operation enables a person, called operator, to act remotely as if the operator was on the spot, by for instance copying the manipulations of the operator at a distance. Two examples are the Da Vinci and Zeus systems [314] for medical surgery. Remote handling is a commonly used term to describe the handling of hazardous materials in toxic environments of nuclear reactors by human operators at a remote location. By handling we mean manipulation of artifacts in the environment using robotic arms or cranes, either for operational purposes or maintenance purposes. We focus on the latter, i.e. remote handling of reactor equipment parts and tools to carry out maintenance procedures.

ITER. The current state of energy production in the world faces ever increasing demands and suffers from the two main concerns: the exhaustion of energy resources and global warming resulting from emissions of greenhouse gases. As a result, the search for alternative and sustainable solutions is increasingly being pursued. One of the solution lies in fusion energy. Although, the use of fusion energy overcomes the drawbacks of other traditional power generation methods, the process of generating usable energy from nuclear fusion is very complex and lies on the cutting edges of technology.

Since the mid-eighties, an international effort has been undertaken to develop the technologies and processes underlying the use of fusion energy as a future power source. A concept version of such a power plant called the ITER (International Thermonuclear Experimental Reactor) project is currently under con-
Remote Handling in ITER. Due to radiation and other toxic material present inside the reactor, human access is very limited. So for the maintenance of equipments in the reactor, the remote handling strategy is adopted. In the context of ITER, the remote handling approach is adopted for maintenance purposes. For this elaborate maintenance procedures (also called operation sequence descriptions (OSDs)) are defined and executed by two four-person teams, each managed by a supervisor, by means of a system called the ITER Remote Handling System, from a remote location called the Remote Handling Control Room (RHCR). To reduce maintenance costs and ensure the safety of equipments and humans, these OSDs must be well-defined and analyzable. At its current state, the design of the ITER Remote Handling (ITER RH) system is at its conceptual stage, described as a block diagram representing a modular decomposition of the system (see Fig. 7.1), supported by informal descriptions about the roles of each module. The lines between modular blocks represent the type of communication network between them. Although, such diagrams capture structural concepts like modularity, they do not specify the behavior of the various modules and their interactions. In this chapter, we use existing design documents and knowledge gained through extensive stakeholder meetings to propose an architectural model.
Service Robots. The rise of rapidly aging societies and a falling workforce is a serious demographic problem facing many countries in Europe and Japan. As the cost of institutional care is rising, the elderly and disabled people mostly want to stay at home longer and keep their independence longer. This need is being recognized, resulting in a spurt of initiatives focussed on the development of innovative solutions for assisted living across the world. The task of assisting elderly and disabled people constitutes of activities of daily living (ADL) such as preparing a meal, cleaning the table, taking out garbage, etc. One way of providing assistance in ADL tasks is by automation in the form of service robots.

Service robots are a relatively new branch of robotics. It is only over the last few years that the development of service robots for assisted living has received a lot of interest. There are many well known examples of such robots, which include the PR2 by Willow garage [59], HERB (Home Exploring Butler) by Intel [309], etc. A service robot is distinct from industry robots and experimental humanoids. Unlike an industry robot which operates in a completely controlled environment that is often designed for them, service robots are intended to perform tasks normally done by humans (like ADLs) in unstructured and unpredictable human environments, while being safe. However, they are not required to accomplish these tasks in the same way as humans; neither do they have to look like a human.

Service robots are equipped with a mobility platform (wheeled or legged), one or more manipulators (arms) and sensing systems (like camera, laser etc.), controlled using visual servo-ing (vision based control) techniques to navigate and manipulate objects in their environment. Such techniques integrate sensing, task planning, coordination and device control capabilities. A service robot must be capable of (a) recognizing objects and locations of interest, (b) generating plans to manipulate objects or move to desired locations safely, and (c) carrying out the safe execution of generated plans by coordinating the control of hardware devices while guaranteeing successful recovery from failures. The generation of trajectories and their execution is easily automated. However, technologies available for object recognition and analysis are largely experimental. Moreover, pre-programming autonomy to deal with unpredictable situations is a hard problem. Some well known examples of such situations include the local minima problem, the detection of false positives, etc. For this reason, we believe that a human operator has to stay in the loop to assist a robot in carrying out its task effectively and quickly. A service robot that relies on human interactions for its decision making to achieve its tasks is called a Tele-operated Service Robot (TSR).

TSR Types. A TSR is a tele-operated system consisting of a master and a slave. The human from a distance by means of the master controls the slave to carry out tasks (services) typically in uncontrolled environments. The slave is in fact the robot that is carrying out the commands given by the master. The
master also known as the *cockpit* consists of a set of devices and interfaces that facilitate the operator in controlling the slave. In a basic TSR, the operator has to demonstrate the actions to be executed by the slave precisely, maybe at a different scale (for eg. controlling a robotic arm with a joystick). In an advanced TSR, the operator has a high-level command language in which he can order a complex task for the service robot with a simple command. Such a command can be given to the slave by means of advanced input devices such as gloves, joysticks with haptic feedback or by voice recognition. An even more advanced TSR is able to learn behavior from past behavior, which is actually programming by example or by an operator training which is a form of supervised learning. TSRs have a wide range of applications that include the building industry to carry and place heavy objects, or in the security sector to perform hazardous operations such as bomb disposal. Note that the ITER RH system is a basic TSR. Advanced notions of TSR are not being considered for remote handling at ITER due to safety concerns. Robot ROSE is an example of an advanced TSR with a goal of carrying out ADL tasks to assist elderly and disabled people, while being safe and guaranteeing some measure of efficiency and reliability.

**Robot ROSE.** Robot ROSE is envisioned to replace a human care giver at the local site by allowing remote operations. The care givers operate from a care center and they are able to help elderly people with simple household activities like warming a meal or taking out the garbage, by remotely commanding robot ROSE at the local site. To achieve its goal, ROSE is equipped with a four wheel steer and drive holonomic platform, two 6-DOF (Degrees of Freedom) manipulators and a wide array of sensors (camera, ultrasound ring, laser range scanners etc.). However, in order to compete with a human care giver, ROSE must be able to perform tasks rapidly, which means that the operator should be able to give a simple command to perform a complex task. So ROSE also has autonomy in task performance, but under the guidance of a human operator from a cockpit, i.e. it is semi-autonomous. The cockpit consists of a user interface and joystick based devices to control the robot at varying levels of autonomy. On the one hand, joystick based devices provide a way to individually actuate the different hardware modules of the robot, and on the other hand, a user interface provides a way to trigger complex autonomous behavior like for instance goal based navigation, find, grab and place objects, open door etc. If during the execution of autonomous behavior, an unforeseen situation is encountered, then the operator at the cockpit is able to aid the robot in its recovery by either taking over control of the robot using joystick devices or by suggesting corrective actions via the user interface. In Fig. 7.2, we present the context of robot ROSE.

The software system of ROSE is a complex distributed system that integrates a wide variety of self-made and off-the-shelf software components as a software stack consisting of hardware drivers at the lowest level, continuing up through controllers, perception, planning, coordination and beyond. In this chapter, we apply our design method (presented in the Chapter 3) to derive a Petri net based
architectural model of the software system of ROSE, starting from a set of functional requirements, specified as use cases. We also show how the behavior of each component can be further elaborated by stepwise refinements with workflow nets (modeling procedures). By construction, this architectural model is weakly terminating, so we have the right ordering of events. To analyze the right timing of events, expressed as safety and performance properties, we extend it to a DTPN. Since we will use UPPAAL for verification, we show how architectural models (DTPN) can be expressed as a network of communicating timed automata. Large models lead to an exponential blow-up of the state space. As a consequence, verification becomes costly and time consuming. To some extent, structural reduction techniques for Petri nets are a well-known strategy to cope with this problem. So for a DTPN, we propose four behavior preserving reduction rules based on transition/place fusion and subnet elimination strategies.

This chapter is structured as follows: In the sections 7.2 and 7.3, we model the software architectures of the ITER RH Plant and robot ROSE using our architectural framework and construction method. In the section 7.4, we show how the performance characteristics of architectural models can be verified by commonly available model checking tools. In the Section 7.5, we conclude this chapter.

### 7.2 Modeling the Software Architecture of the ITER RH Plant

The ITER RH system is a sub-system of ITER, containing software and hardware components responsible for carrying out maintenance-related planning, their execution, monitoring and analysis. We will focus on the software aspects of maintenance execution and monitoring. The maintenance tasks of ITER are described
as procedures called Operation Sequence Descriptions (OSDs). In essence, they are workflow nets [330] (see A.5) An OSD is a partially ordered set of tasks. Each task consists of an ordered set of sub-tasks, which in turn consists of an ordered set of actions. An action is carried out by a human operator or by a software system under the guidance of a human operator. As safety is of primary concern, the automation of RH tasks according to these descriptions will be very limited. As maintenance processes become more well-understood and predictable, automation may be applied in an incremental fashion at a later stage. For now, the maintenance processes of equipments in the RH Plant are envisioned to be manual processes, carried out by two teams of operators and a supervisor, using haptic devices and aided by graphical interfaces to the equipment and process management systems.

At its current state, the modules of the ITER RH Plant are shown in the Fig. 7.1. Each box in this figure represents a coherent set of functionalities. All modules, except for CODAC (Control, Operation and Data Acquisition), CIS (Central Interlock System) and CSS (Central Safety System) belong to the context of the ITER RH Plant. The modules of the ITER RH Plant are broadly classified into the following two layers:

- **The supervisory control system:** This sub-system contains modules for equipment management, process management and plant safety. The module labeled EMS is the Equipment Management System, responsible for the management of almost 245 pieces of ITER RH equipments. The module labeled RH Supervisor System is responsible for maintenance planning, execution, monitoring and analysis of OSDs. The RH Plant Controller maintains the local operational states of the RH plant and records the global states of the ITER system. It also carries out transformation on information being sent/received to/from CODAC. The PSH module is the Plant System Host, which acts as a gateway for information exchange between the ITER RH plant and its parent ITER system. The modules labeled PESS (Push-button Emergency Stop System), PIS (Plant Interlock System) and PSS-OS (Plant Safety System Operational Safety), concern the safety modules of ITER RH.

- **Equipment control system:** This sub-system contains software modules for planning and control of equipments required for carrying out maintenance activities. For each such device, there is one control system consisting of a high level control module and a low level control module. An example of such a device is the crane, responsible for hoisting heavy equipment. The low level control module receives commands from its high level control module and in turn provides feedback about its progress. The high level control system is either the driver component of operator devices like a haptic joystick or an (semi-)autonomous trajectory planning system. The control system of each equipment periodically relays its current state to
the RH Plant controller. These control systems also listen for and react to safety critical commands from the safety modules of the supervisory control system.

In the Fig. 7.3, we present our proposal for a formal component-based architectural description of the ITER RH software system. The architecture further decomposes some of the modules identified in the Fig. 7.1 into one or more components. The interaction patterns between components have been identified based on functional descriptions described in a textual format and from information gained during stakeholder meetings. We discuss each component of our architectural diagram and how they collaborate over interaction patterns to achieve their goals in the following paragraphs using the Fig 7.3:

- **Plant System Host (PSH):** The PSH component sits between the CODAC and RH Plant Controller (RHPC) and acts as a gateway for information exchange between the modules of the ITER RH plant system and the CODAC system. The main functionalities of this component include
  - Handle commands from CODAC and dispatch them to the RHPC (see pattern P1).
  - Keep track of the state of the ITER plant (GOS: Global Operating State) being published by the CODAC system (see pattern P2).
  - Monitor the state of the RH plant (COS: Common Operating State) and update this periodically to the CODAC system (see pattern P3).
  - Publish alarms generated by the RH plant to the CODAC system (see pattern P5).
  - Transfer relevant logs from the RH plant to the CODAC system (see pattern P4).
  - Publish incoming software events from the CODAC system to the modules of the ITER RH plant (see pattern P2).

- **RH Plant Controller (RHPC):** The main functionalities of the RHPC component include
  - Maintain the states of all ITER RH equipments as a set of Finite State Machines (see pattern P9).
  - Maintain the states of all relevant equipments outside the scope of the ITER RH plant (see pattern P16).
  - Publish commands arriving from CODAC after translating them into C&C commands (ITER RH system format) for equipment control systems (see pattern P11).
  - Provide facilities for equipment control systems to look up states of other equipments (see pattern R6).
Figure 7.3: Architecture Diagram of the ITER RH System
Subscribe to updates and Monitor the states of RH maintenance processes (see pattern P7) and the states of equipments (see pattern P15).

Publish events, alarms and other operational parameters of the RH plant to the CODAC system (see patterns P3, P4, P5).

Create and Publish alarms (see patterns P8, P17, P18) for components of the RH plant based on C&C and PIS (see pattern R5b).

Provide facilities for supervisory components to request CODAC permissions (see pattern R1).

Note that RHPC is not responsible for actual control of equipments.

RH Operator GUI: This component provides a user interface for the RHPC component and supports the following features:

- Provides an intuitive snapshot of the state of the ITER RH plant on a graphical layout of the RH facility (see pattern P17).
- Under certain modes (granting administrative rights), it is possible to update some of the RHPC parameters (see pattern R5a).

Equipment Management System (EMS): The EMS module has three components: two GUI components and one EMS application server. The EMS application server supports the following features:

- Create new tool and equipment IDs (See pattern R2).
- Equipment usage logging (See pattern R2).
- Equipment Scheduling (See pattern R2).
- Equipment Lifecycle and Maintenance (See pattern R2).
- Equipment Spare Parts Management (See pattern R2).
- Check in/ out equipment during maintenance (See pattern R3).
- Publish updates for the two GUI components (see pattern P6).

The two GUI components provide an interface for an operator to carry out equipment planning and management.

RH Supervisor System (RHSS): The RHSS is the core module of the two RHCR rooms in the ITER RH facility. Each RHCR has up to 6 work cells. For each RHCR, there is one supervisor leading a team of at most six operators, one in each work cell. The main objective of the RHSS module is to support the supervisor (one supervisor per RHCR) in the coordination of concurrent RH maintenance procedures (OSDs) among the operators of an RHCR (up to six operators per RHCR). The task of creating an OSD is a manual process and not automated by the system. Furthermore, the
planning of equipments and resources is done by the supervisor on demand basis through EMS.

The components of the RHSS module (a) UI components: SS GUI (up to 2 copies) and OMS GUI (up to 12 copies, one per work cell) (b) Core Components (see the Fig 7.4): Communication, OSD Master and OSD Slave (up to 12 copies). Note that the OSD Master and Slave components have each one communication component and one OSD component. The latter models a long running maintenance procedure (lasting days/weeks), whose progress is triggered by acknowledgements from the responsible operator. By the structure of a component, this implies that an OSD component is not able to return to its idle state to check for messages from other components. So we model the interactions of the OSD component with their UI components by a dedicated communications component, which runs in a separate thread and shares data with their corresponding OSD component.

The supervisor coordinates the RH maintenance processes (OSDs) through
the SS GUI. The SS GUI provides the supervisor with an overview of all currently active and planned OSDs, as well as relevant parameters stored in the communications component (see pattern P18). An OSD Master component defines an OSD as a workflow. Its execution is triggered by the supervisor via the SS GUI (see pattern R8). For each OSD Master component, there is a copy called the OSD Slave component. These two components synchronize with each other over every action of an OSD. Therefore they are ideally modeled as an MP pattern (see pattern M1). An OSD Slave component deliberates the individual actions of a sub-task to the appropriate OMS GUI (based on the type of RH operator) (see pattern P19). When the action, indicated on the OMS GUI has been completed by an operator, an indication on the OMS GUI is made, which is then received by the communication component of the corresponding OMS Slave and updated to a shared variable. When the OMS Slave component notices this update, it sends a notification to the corresponding OMS Master component (as part of the M1 pattern). When the OMS Master component receives this notification, it synchronizes with the OMS Slave component. The communication component of the OMS Slave component periodically publishes the state of its OMS Slave component to the corresponding OMS GUI (see pattern P19).

During the maintenance process, the supervisor is responsible for locking and unlocking parts of equipments that are foreseen to be used by an operator to carry out the actions of planned sub-tasks. The communication component provides a facility to accept a lock/unlock request (see pattern R4) from the SS GUI, which in turn requests the EMS application server over an RPC pattern (see pattern R3). For certain sub-tasks of the workflow, permissions from the CODAC system must also be acquired. This is realized by the communication component over the client of an RPC pattern that invokes the RHPC (see pattern R4 and R3). Furthermore, the communication component periodically dispatches updates about the status of its maintenance processes to the RHPC (see pattern P7). It also listens in turn for updates (from the CODAC system or alarms generated by the RH system) being published by the RHPC (see pattern P8).

- PIS, PSS and PESS: The PIS and PSS components subscribe for safety critical commands being published by the CIS and CSS components. The PESS component is a hardware device that is triggered by a human operator. All three components are able to publish safety critical commands to all equipment control systems (see pattern P10). They also notify the RHPC component about any safety related updates and in turn receive the current state of the RH system (see pattern R5b).

- Equipment Control Systems: We distinguish between three types of equipment control systems based on the type of interaction pattern between the
high level control and low level control: GFR based Control System, RPC based Control System and PS based Control System. Both the high level and low level control system of an equipment

- Subscribe to safety critical commands published by either PIS, PSS or PESS (see pattern P10).
- Publish their status to the RHPC (see pattern P9)

The high level control system can also query the RHPC about the state of other equipments (see pattern R9). It is also possible for a high level control system of one component to send commands to low level control systems of other equipments.

7.3 Modeling the Software Architecture of Robot ROSE

In this section, we derive the software architecture of robot ROSE, starting from a set of use cases. First, we identify the set of components and their interaction patterns and represent the relationships between them as an architectural diagram. From this diagram, it is straightforward to apply our construction method (see ch. 3) to derive a Petri net model of the system. Recall that this model is weakly terminating.

7.3.1 Robot ROSE

Robot ROSE consists of two main parts: the robot and the cockpit. The robot consists of hardware, electronics and software. In the Fig. 7.5, we present our
robot ROSE. The main hardware and electronic components of the robot are:

- One custom-made four wheel steer and drive holonomic platform.

- One Hokuyo UBG-04LX-F0 laser scanner used for mapping, localization and collision free navigation.

- Two 6-DOF Exact Dynamics iArms each having a 1-DOF gripper to manipulate objects in its environment.

- One Xbox Kinect camera mounted on a pan-tilt neck system for visual feedback, 3D collision-free navigation, object tracking and depth estimation.

- One Wifi router to support communications between cockpit and robot.

Each of these hardware components comes with an embedded controller that provides a software interface that accepts/produces digital signals (text-based commands/feedback) and generates/reads analog signals and/or vice versa.

The cockpit consists of one or more joystick based devices to command the platform, arms and neck; a graphical user interface to view camera images and sensor feedback, along with a set of options to command autonomous behavior like navigation, grabbing and placing objects, opening a door etc. In the Fig. 7.6, we present a snapshot of the graphical user interface (GUI) of ROSE with the different camera views, a map view and whole array of options to select autonomous behavior in tasks. The robot and the cockpit are connected to each other over a wireless network.
7.3.2 Use cases of Robot ROSE

After extensive workshops with both care givers and elderly, a number of representative use cases (functional requirements) were selected such that a good balance could be maintained between the technical possibilities and the wishes of the care givers and elderly. We will now briefly describe the eleven selected use cases.

– **UC1**: Move platform, arms and neck. An operator must be able to control the platform, pan-tilt neck system, left and right arms in a safe and intuitive manner, using one or more joystick based devices available at the cockpit.

– **UC2**: View camera image. An operator must be able to view the image currently being captured by the robot’s camera system, on the cockpit’s GUI.

– **UC3**: Create a map of the current location. An operator must be able to create a 2D map of an unknown location by driving the robot around using a joystick. The current map must be displayed at all times on the GUI;

– **UC4**: Navigate to an indicated location: An operator must be able to indicate a location on the map that the robot must navigate to, while avoiding static map obstacles and dynamic obstacles that could be presented by the environment. If the robot is unable to plan a path then it must abort. At all times, the operator must be able to cancel the currently running goal via the GUI;

– **UC5**: Grab a selected object with left or right hand. An operator must be able to select an object from the camera image that the robot must grab with either of its arms. If the object is out of reach, then the robot must try to navigate to within reach and then try to grab the object. If the object is still out of reach, then it must either abort or task the operator for suggestions. At all times, the operator must have the option to cancel this task, in which case the robot must be able to abort in a safe manner.;

– **UC6**: Grab a selected object with both hands. This use case is almost similar to UC5, except that the robot is required to carry out the ask with both its hands, for instance carrying a tray.

– **UC7**: Place object at a selected location. An operator must be able to indicate a location on the live camera image, where the robot must place the object it is currently holding. If the robot is not holding an object then it must inform the operator about the situation and abort. At all times, the operator must have the option to cancel this task, in which case the robot must be able to abort in a safe manner.
– **UC8**: *Open/Close door*. An operator must be able to indicate to the robot the location of the door handle, the door hinge and if the door opens towards or away from it. Once these indications have been given to the robot, it must be able to open the door around the door hinge safely. At all times, the operator must have the option to cancel this task, in which case the robot must be able to abort in a safe manner.

– **UC9**: *Pour contents on an indicated location*. An operator must be able to indicate to the robot the location where the contents of the object currently being held by the robot must be poured onto. At all times an operator must be able to cancel the task, in which case, the robot must be able to safely retract the pouring task without causing a spillage.

– **UC10**: *Twist open/close an indicated cap*. An operator must be able to indicate to the robot the cap of an object it is currently holding and the direction (clock-wise or anti-clockwise) it can be opened or closed. Note that for this task, the robot requires both its hands, one for holding the object and the other for manipulating the cap. At all times, the operator must have the option to cancel this task, in which case the robot must be able to abort in a safe manner.

– **UC11**: *Push/Pull a selected object forward/backward*. An operator must be able to indicate to the robot an object that must be pushed forward by a certain distance. If the object is out of reach then it tries to get in range by moving towards it. If the object is within reach then it uses one of its hands to push the object forward by the specified distance. An example where this functionality can be useful is the turning on/off the light switch. To pull an object, the robot must first grab the object and then pull it towards itself by the specified distance. If the object is out of reach or too big to grab, then the operator is notified and the task must be aborted. At all times, the operator must have the option to cancel this task, in which case the robot must be able to abort in a safe manner.

The eleven use cases cover a good cross section of the various technical challenges that need to be overcome in order to easily extend the capabilities of the robot to other similar as well as complex use cases. For instance if ROSE is able to grab the breakfast items then it is also able to grab the remote control of the TV. Furthermore, if ROSE is able to realize all the above use cases, then it is able to prepare a meal on a tray from items in the kitchen (UC4, UC5, UC6, UC7, UC9), warm it in the microwave (UC8, UC10, UC11) and bring it to an elderly (UC4) at the right time.

### 7.3.3 Identifying Components and Interaction Patterns

From the eleven use cases, a few components of the system that are crucial to all use cases are already evident.
Driver Components

Each hardware device like the joystick, the platform, the arm, the neck system, the camera and the laser scanner must have a dedicated driver component. A driver component reads data from the device and publishes them on one or more topics. If a driver component is connected to a passive device (sensors) like camera, laser scanner or joystick then it only reads data from the device and publishes them. If the driver component is connected to an active device (actuators) like platform or arms then it subscribes to one or more topics that broadcast command messages, published by their components. Often, the raw data read/written by a driver component from/to its device requires some sort of processing and transformation using the knowledge about the kinematic structure of the device. We explain this in the context of the various driver components.

The platform driver component reads the steer and drive encoders of each wheel and from this the speed of each wheel and its steer angle is computed and published. The driver component also computes and publishes odometry. Odometry is a cumulative method to determine the position and orientation of the platform, relative to its starting location, using the knowledge about the number of wheel rotations and their steer angles. The arm driver component reads the current values of the seven joint encoders and compares these values with the values from the last reading, to compute the current angle and velocity of each joint of the end-effector and publish them. The camera driver component transforms raw left and right camera images into rectified images in various formats, which are then used to construct a 3D data set called a point cloud. The rectified
images and the 3D point cloud are published on different topics to other components of the system. The laser scan driver component transforms raw range data into a scan line in the coordinate frame of the laser scanner and publishes it. The neck driver component of the pan-tilt neck system publishes raw joint angle data of its two actuators (pan-tilt motors). In addition, the driver components of platform, arms and neck, i.e. actuated devices, subscribe to topics broadcasting commands, published by other components of the system. As driver components are not servers of RPC, GFR or MP interaction patterns, they are modeled as root components, i.e. they do not provide a service.

Controller Components

Actuated devices like the platform, the arms and the neck also require a dedicated controller component to control their movements based on the last received goal. For this purpose, the use of feedback-based control algorithms, like the Proportional-Integral-Derivative (PID) scheme, is very common. Then a subscription to feedback being published by its driver and goal from higher level planners is necessary. Whenever a goal specifying the desired state of the actuated device is received, it is compared against the current state (interpreted from feedback data). If the desired and actual state of the device are not equal then a sequence of commands are published to the corresponding driver component, such that the actuated device converges on its goal as soon as possible.

The platform controller component subscribes to feedback (velocities and steer angles published by the platform driver) and goal. The goal specifies the desired linear velocity and radius of curvature of the platform from which the desired velocity of each wheel and their steer angles are derived. Then the PID control scheme of the platform controller publishes a sequence of commands (control actions) to the platform driver until the desired values and the current values converge or reach within some specified threshold. The neck controller component is similar to the platform controller, except that the goal and feedback subscriptions specify
instead the desired joint angles and current joint angles (of the pan and tilt motors), respectively. Note that both the platform and neck controller components are root components.

The arm controller component subscribes to two types of goals. One specifies the desired pose of the end-effector (arm gripper) as a 6D cartesian coordinate and the other specifies the desired velocity of the end-effector (arm gripper). In both cases, inverse kinematic techniques are used to derive the required joint angles and joint velocities, respectively, to achieve the current goal. The arm controller keeps track of the current joint angles and velocities by subscribing to feedback from the arm driver. Similar to the platform controller, the arm controller realizes a PID control scheme that publishes a sequence of commands (control actions) until the desired values and the current values become equal. For the two goal types, two PID control schemes are defined. Depending on the last received goal type, the corresponding control scheme is activated. Note that an arm controller component is also a root component.

**KT-Component**

The knowledge of the kinematics and current configuration of the robot must be stored and made available to other components of the system. For this we specify a dedicated component and call it the *kinematic transformer* (KT) component. The KT component stores the current configuration of the robot as a tree of coordinate frames and the transformational relationships between pairs of parent-child frames. The kinematic tree is kept up-to-date by subscribing to data being published by driver and localization components. This knowledge is used by the KT component to provide coordinate look-up and transformation services as procedures of an RPC server.

**Cockpit, Image and Goal Processing Components**

At the cockpit, all relevant information published by the robot, i.e. camera images, map, robot location etc. must be displayed. Furthermore, an operator must be able to select objects visible on a camera image or make an indication about a location on the map.

For this, we define a new component and call it the *GUI component*. A selection on the camera image is made by drawing a bounding box over the region of interest. A location on a map is indicated by a mouse click on the map. Everytime a selection or an indication is made, it must be analysed and transformed into a goal for the rest of the system to interpret and realize. The functionality required to construct such a goal requires a new component.

For this, we define a new component and call it the *Use Case Goal Generator Component* and design it as an RPC server having one procedure. The RPC procedure at the server accepts a selection or an indication sent by a corresponding client, embedded in the GUI component. The received information is then trans-
formed into a goal depending on the chosen use case. The choice of an use case is made by an operator by means of menus and/or button events. If an indication arrives when UC4 (navigation) has been chosen then it is directly sent as a goal, otherwise the indication is discarded. If a selection arrives when any of the use cases: UC5 through UC11 is chosen, then the selection must be interpreted in order to construct a goal. This is because, the choice of the selected use case determines the contents of the goal. For instance for UC5, one selection of the object is sufficient to compute a grasp point for the arm. This is not the case for UC6, where two selections need to be made by the operator to compute two grasp points, one for each arm. This is also not the case for UC8, where in addition to the two selections (the door handle and the door hinge), an operator must also specify the direction of door opening/closing. At this point, it is evident that we require a component to analyse a camera selection and compute a grasp point for an arm, depending upon the chosen use case and the progress of the goal construction, so far.

So we define a new component and call it the `depth estimation component`. The depth estimation component, subscribes to the camera driver component for image and point cloud data. The execution of the depth estimation component is triggered by the arrival of a goal containing the coordinates of the bounding box (selected region), the currently chosen use case and the progress of the goal.
construction, so far. First, the 2D coordinates of the bounding box are mapped onto 3D coordinates of the point cloud. Next, depending upon the chosen use case and the progress of the goal construction, so far, grasp poses are estimated for the selected region using well known grasp planning techniques. If grasp poses could be found then they are sent as a response, otherwise a failure indication is sent. This mechanism is easily realized by designing the depth estimation component as an RPC server. So the construction of a goal by the use case goal generation component, for a chosen use case, may require one or more selections or indications from the GUI (RPC pattern), and may in turn may require one or more invocations of the depth estimation component (RPC).

At this point, the use cases UC1 and UC2 have already been realized. The driver component of a joystick for controlling the platform/arms/neck transforms the movements of the joystick into goals that are published to the platform/arms/neck controller component. Furthermore, the GUI component subscribes to camera images being published by the camera driver.

**Mapping Components**

For use case UC3, we require two new components called the mapping component and the map server component. The map server component maintains a copy of the current 2D map of the robot’s environment. The copy of this map can either be updated with another map, or a copy of it can be provided to other components that request for it. We realize the map server component as an RPC server with two defined procedures: update map and retrieve map. The mapping component, subscribes to the laser scan driver, camera driver (for point cloud) and platform driver (for odometry) components to maintain a copy of the latest laser scan and point cloud data. With these inputs, well established SLAM (simultaneous localization and mapping) techniques may be used to construct a map of the environment by driving the robot around using a joystick (UC1). While the map is being constructed, an approximation of the current location of the robot is also being computed and published to the KT component. Furthermore, the mapping algorithm (SLAM [116]) of the mapping component is started using a start request and stopped using a stop request. When a start request is received, the copy of the map at the map server is updated with an empty map. When a stop request is received, the mapping component updates the copy of the map at the map server with its current copy. Such a mechanism is realized by first designing the mapping component as an RPC server component having one procedure to receive both start and stop requests. Then an RPC client is nested within this procedure to invoke the update map service provided by the map server.

**Navigation Components**

For use case UC4, we require three new components: localization, planner and costmap manager.
The localization component subscribes to the laser scan driver, camera driver, platform driver and map server components. When started up for the first time, the localization component also requires an initial estimate of the robot's current location on the map. This feature is realized by designing the localization component as an RPC server. The last received odometry, point cloud and laser scan data are overlayed on a map (retrieved once from the map server) and interpreted in order to determine an estimate of the current location of the robot, using well established probabilistic pattern matching techniques. This estimate of the current location of the robot is published to the KT component. Recall that the mapping component also carries out such a localization of the robot while constructing the map. For this reason, the mapping and localization components must never be active at the same time. This means that it must be possible to start and stop the localization component. This is easily achieved by adding another procedure to the RPC server to handle start and stop requests.

The costmap manager component subscribes to the laser scan driver and camera driver components. With these inputs, the costmap manager component maintains a configurable, fixed size, cost-based, dynamic rolling window map of
the environment relative to the robot’s current location, called its costmap. Note that the robot’s location can be obtained by invoking the KT component. Furthermore, each pixel of a costmap is assigned a value that encodes, how close that pixel is to an obstacle. Note that such a costmap is dynamic because the value of a pixel is updated, everytime a new information about it has been detected. For performance reasons, the costmap manager shares its copy of the costmap with the planner component. However, we do not explicitly model the data sharing mechanism in our architectural diagram.

The planner component, uses this copy of the costmap, overlayed on a map, in order to plan a path to a desired goal from the current location of the robot. The desired goal specifies a location and orientation on the map that the robot must navigate to, while avoiding obstacles. The current location and orientation of the robot on the map can be found by invoking the RPC server of the KT component. The planner component achieves its objectives using two types of path planners: a global planner and a local planner. The global planner uses the map, the current location and orientation of the robot; to generate a complete 2D trajectory (path) from the current location of the robot to the desired location of the robot, using the well known $A^*$ algorithm. The local planner uses the global plan, the current costmap and the current location and orientation of the robot; to generate the next command for the platform controller. To allow for the cancelation of an ongoing goal and to provide a facility to view the progress of the planner, so far, the planner component is designed as a GFR server.

Planning, Coordination and Safety Components

For all the remaining use cases we require three more components: arm manager, platform switch and macro module.

The arm manager component is triggered by a goal containing a desired 6D pose for the end-effector of an arm. The current pose of the end-effector of an arm can be found by invoking the KT component. The component also maintains a copy of the latest point cloud by subscribing to the camera driver component. Using all this information and the knowledge about the kinematic structure of the arm, the arm manager component generates a collision free 3D trajectory for the arm to follow while moving towards its goal. Once the trajectory has been generated, poses from the trajectory are peeled and published as cartesian goals to the arm controller component. For each cartesian goal that is published, the arm manager waits for the end effector to reach within some specified distance from this cartesian goal and only then the next pose of the trajectory is peeled and published as a cartesian goal. This process continues until either the entire trajectory has been traversed by the arm, or some unforeseen situation was encountered. To allow for the cancelation of an ongoing goal and to provide a facility to view the progress of an ongoing goal, so far, the arm manager is designed as a GFR server.

We require the platform switch component for two reasons: (a) Both the navigation system and the joystick driver publish goals to the platform controller.
So we require a component that can switch between the two goals, depending on the context, i.e. navigation mode or manual mode. (b) In the manual mode, a safety mechanism is required to stop the platform, whenever a potential collision is detected by one of the sensors of the robot. Both these requirements can be fulfilled by designing the platform switch component as an RPC server: containing one procedure to carry out a context switch between manual and navigation modes; along with subscriptions to the laser scan driver, camera driver, navigation system and joystick driver; and a publisher that publishes goals to the platform controller component. If the platform switch component is operating in the manual mode then it also monitors and interprets the latest copy of its laser scan and point cloud data in order to detect potential collisions. If a potential collision is detected, then a goal commanding the platform controller to stop the platform is immediately published.

We now have almost all the components required to realize use cases UC4 through UC11, but miss a component that defines their orchestration, i.e. the right order of invocation of the various components of the system.

For this we define the *macro component* as a set of procedures, one for each use case. Each procedure defines the order of invocation of one or more server components (arm manager, navigation, neck controller) in order to realize the corresponding use case. Note that the order of invocation is in fact the ordering of clients of server components that are being invoked by that procedure. Furthermore, to allow for the cancelation of a use case or to view the progress of an
ongoing procedure, we design the macro component as a GFR server with a set of pre-defined procedures. When a goal sent by a corresponding client of the use case goal generator component arrives, one of the procedures of a macro component is triggered. The goal of a macro component specifies all the relevant details about carrying out the desired use case, which includes the use case number and the coordinates of the destination that the robot must navigate to, and/or the set of grasp poses required to manipulate an object using one or both of its arms.

So we have identified all the components and their interactions required to realize use cases UC1 through UC11. Next we will show the corresponding architectural diagram of the system and derive its corresponding Petri net model. For a few interesting components, we will also discuss their functional design.

7.3.4 Architectural Model

From the set of identified components and their interactions, it is straightforward to connect them together to obtain an architectural diagram of the system, as shown in the Fig. 7.12. The components of the architecture can be classified into four layers, namely the driver layer, the controller layer, the planning and coordination layer and the user interface layer. The first two layers represent low level control components, i.e. the components generate control actions for a given goal and/or publish the data generated by hardware devices. The third layer represents high level planning components like the navigation system and arm manager, and high level coordination components like the macro component. The fourth layer represents user interface components which facilitate either the selection of a use case and goal construction for the macro component or direct control of one or more hardware devices using one or more joystick based devices.

Operational Description

When the robot is started up in a new environment, an operator first selects UC3 using the use case goal generator component. This selection is sent as a goal to the macro component, which in turn sends a request to turn off the localization component, a request to start the mapping component and a request to set the platform switch to manual mode. Then the operator is able to use the joystick to drive the robot around, in order to construct a map of its environment. Recall that the user is always presented with the latest camera image, the current status of the map and the location of the robot on it. Once the location has been sufficiently mapped, the operator cancels the execution of UC3.

At this point, an operator may choose one of the use cases UC1, UC4, UC5, UC6, UC7, UC8, UC9, UC10 and UC11 to carry out ADL tasks in the mapped environment. For UC4, an operator first selects this use case using the use case goal generator component, which then awaits a map indication from the GUI component in order to construct a navigation goal for the macro component. We
Figure 7.12: Architectural Diagram of Robot ROSE
Figure 7.13: Defining the Orchestration of the Macro Component
will now briefly discuss how one of the remaining use cases are realized by the architectural model.

Suppose UC5 was selected. Then the use case goal generator component waits for a bounding box from the GUI component. Once this selection is received, then the depth estimation component is invoked to compute a grasp point for the arm, which is sent back as a response. If a grasp point could not be found, then the operator is notified and the execution of the use case is terminated. If a grasp point was found, then it is published along with the information about the use case as a goal to the macro component.

When a macro component receives this goal, the procedure corresponding UC5 is triggered. First, the goal is transformed (by invoking the KT component) into the coordinate frames of both arms to determine which of them is better suited to carry out the task. If both the arms are out of reach, then the goal is transformed into the coordinate frame of the platform and published as a goal to the navigation system. The navigation system generates a path to this goal and publishes a sequence of commands that drive the robot to the desired location and orientation.

If the navigation system aborts for some reason, then the macro aborts UC5 as well. If the navigation task was successful then the macro component checks if the arms are now able to reach the grasp point. If they are still not reachable by the arms then the macro aborts UC5 and notifies the reason to the operator via the use case goal generator component. If the grasp point is reachable by one of the arms, then the macro component generates a goal for the arm manager component, which in turn generates a collision-free trajectory to grasp the desired object. The contents of this trajectory are published as a sequence of cartesian goals for the arm controller to realize.

Note that an operator is always able to cancel an ongoing use case at the macro component via the use case goal generator component, which in turn cancels an ongoing goal at the navigation system and/or the arm manager(s).

All other use cases are realized using the same principle, only that they may vary in the complexity of goal construction and its corresponding procedure in the macro component. For instance UC8, requires three selections on the camera image and the corresponding procedure at the macro component defines a complex concurrent coordination of the platform and an arm to achieve safe and reliable opening or closing of an arbitrary door.

**Deriving the Petri net Model**

We sketch the transformation of the architectural model into a Petri net model using the construction method described in the Ch. 3.

First we classify the components of the system into root components, RPC components, GFR components and MP components. The root component of the system include all the driver components, the controller components, the GUI component and the costmap component. The RPC components of the system in-
clude the platform switch component, the map server component, the localization component, the mapping component, the KT component, the use case goal generator component and the depth estimation component. The GFR components of the system include the planner component, the two arm manager components and the macro component.

Next, we construct Petri net models of all the root components. Then, we identify a set of places from existing components (such that no two places belong to the same component) and simultaneously refine them with clients of either an RPC, GFR or MP patterns. The order of selection of these patterns are defined by the architecture diagram. i.e. an RPC, GFR or MP pattern can be selected for refinement if the place(s) into which the client(s) of the chosen pattern are to be refined, already exist. If such a place does not exist, then it means either (a) at least one other interaction pattern must be refined into the system, or (b) a place in the existing system must be refined by a state machine workflow net. Recall that each refinement with RPC, GFR or MP patterns introduces a new component into the system, i.e. bottom-up construction. This refinement step is repeated for all the remaining RPC, GFR and MP patterns.

If two or more clients of interaction patterns (each client belonging to a unique interaction pattern) need to be added to the same component, then we must first define their choreography by a state machine workflow net and then choose places from this net to refine with clients of interaction patterns. Recall that a state machine workflow net can express non-deterministic choices but not concurrency. Consider for instance the macro component (GFR server). The macro component has four RPC clients and three GFR clients using which all eleven use cases are realized. For some use cases like simultaneous grab and open/close door, two GFR clients may need to be executed concurrently. To realize this requirement, the control flow logic is defined as a weakly terminating ST-workflow net. Then the busy place of the GFR server (macro component) is refined with this net. During the construction of the Petri net model, the places labeled $W$, $X$, $Y$ and $Z$ are refined with clients of RPC patterns and the places labeled $P$, $Q$ and $R$ are refined with clients of GFR patterns. We show this step in the Fig. 7.13.

Once all the GFR, RPC and MP patterns have been refined into the system, we carry out an insertion of all the publish subscribe patterns, on top of the existing system. In this manner, we have derived a Petri net model from an architecture diagram and this model is weakly terminating.

In the Appendix C, we give a few interesting examples of how computational aspects of a component can be elaborated in a top-down manner by place refinements with state machine workflow nets (specifying control flow/orchestration). We consider components like the platform driver, platform controller, planner and macro.
7.4 Analysis of Architectural Models

In this section, we extend our architectural model expressed as a Petri net with the notion of time and analyze them.

A Petri net model of an architectural diagram is extended to a DTPN by adding closed rational time intervals on arcs from transitions to places and point delays on arcs from places to transitions. The former is used to specify the execution time of a task, while the latter is used to specify message time-outs. Furthermore, each component of the architectural model executes at a pre-defined frequency called its loop rate, which specifies how often a component must return back to its idle state, i.e. a token in the idle place. This feature is easily modeled by associating a unique place marked with a token and connected with bi-directional arcs to the closure transition of the component such that the loop rate is specified along the arc from that place to the closure transition.

For analysis we consider the two possibilities, CPN Tools and UPPAAL. For a subclass of DTPN with no delays on arcs from places to transitions, we may use CPN tools to generate its reduced state space (i.e. rDTPN). However, at its current state a lot more needs to be done for it to be useful for practical applications by system architects. So we consider timed automata as incorporated in the UPPAAL toolset. Note that UPPAAL is not preferred over CPN Tools. First, channel synchronization is not natural for modeling asynchronous message passing. Second, only a subclass of DTPN can be expressed as a timed automata (see Sec. 5.2).

For a background on syntax and semantics of a timed automata, please refer Chapter 5. For a compact representation, systems modeled in UPPAAL are specified as a network of timed automata, which are a composition of timed automata over shared clocks and transitions. The UPPAAL modeling language extends the semantics of a network of timed automata with synchronous communication channels, urgent locations, data, etc.

Each communication between the automata of a network is modeled as a synchronous communication channel. A synchronous communication channel defines two types of events, namely send and receive. This means that for a synchronous communication channel labeled c, two types of edges are defined, one labeled c! (modeling a send event) and the other labeled c? (modeling a receive event). Both send and receive events are blocking events, i.e. an automaton in a location with an outgoing edge corresponding a send event is blocked until a synchronization by another automaton over its corresponding receive event is available, and vice versa.

In UPPAAL, two types of synchronous communication channels are defined, namely binary synchronization and broadcast channels. The former is a pairwise synchronization between an enabled send event and an enabled receive event. If more than one synchronization pairs are enabled, then a non-deterministic choice between them is made. The latter is a synchronization between one send
event and multiple receive events. All enabled receive events must participate in this synchronization. Unlike binary synchronization, broadcast channels are non-blocking, i.e. even if no receive event is enabled, then the edge corresponding the send event can be taken.

Furthermore, the notion of an urgent synchronization channel is defined. This is a binary synchronization channel where delays cannot occur if a synchronization pair is enabled.

Lastly, a location indicated as urgent does not allow the progression of time, i.e. delay transitions cannot occur. For a more detailed introduction to the modeling aspects of UPPAAL, please refer [33].

### 7.4.1 Modeling Time, Buffers and Resources

In this section, we extend our architectural model to a DTPN, expressing task execution times and message time-outs. For analysis in UPPAAL, we show how DTPN models can be expressed as a network of timed automata extended with communication channels. To make analysis more interesting, we allocate resources to these channels.

**Modeling Task Execution Time**

In our architectural model, transitions model events and internal places model an ongoing task (activity). The occurrence of an event corresponds the firing of its transition. An event triggers a task whose progress is given as a measure of time and we call this the task execution time, specified as a closed rational interval $[X, Y]$ on an outgoing arc from a transition $A$, modeling the event.
Figure 7.15: Modeling Timeouts

Using the results of Chapter 5, such a model can be expressed as a timed automaton. For this, we must first transform it into a cDTPN and then into a timed automata. We show this transformation in the Fig. 7.14. For conciseness, we do not explicitly model the timer place.

Modeling Time-out

Time-out is a recurring pattern in message passing systems. In our architectural framework, this is useful for the non-blocking behavior of receive transitions, i.e. transitions that wait for a message. A timeout is modeled as an internal transition as a choice between a receive transition such that the input arc to the internal transition specifies the timeout value. As a consequence, it cannot be transformed into a cDTPN. However, if the receive event is specified as part of an urgent synchronization channel, then it is possible to express the semantics of a timeout as a timed automata. In the Figure 7.15, we illustrate with an example, a timeout pattern modeled as a DTPN and its corresponding timed automata. Here the time-out transition is enabled if no message for the receive event has arrived within 2 time units. In the context of analysis, a timeout transition must never be enabled because its enabling indicates the late arrival of messages in a system. So if timeout transitions are never enabled, then the behavior is identical to the original system (without any timeout transitions).

Modeling Shared Places of Interaction Patterns

The shared interface places between the clients and servers of an interaction pattern act as placeholders for messages, also known as buffers. As places in a Petri net can have be unbounded, they model infinite buffers. However, for analysis of
DTPN, the boundedness requirement for places is essential. As a consequence, the arrival and departure of tokens from an interface place can be expressed as a communicating automaton, where the state indicates the buffer size and an edge between states is either a send event (departure of tokens) or a receive event (arrival of tokens), each from a different synchronization channel. The terminal state indicates a full buffer. In this state, an enabled receive event leads to itself and only an enabled send event can cause a change in state. To enforce progress, we specify all channels as urgent. For analysis, it must never be possible to reach a state where any of the buffers is full. So if the buffers are never full, then the behavior of the network channel and its corresponding place are identical.

In the Fig. 7.16, we give an example of a PS pattern having one client $C_1$, one server $S_1$ and a common interface place $R$. The corresponding timed automaton modeling the publisher $C_1$, the network channel $CH_1$ corresponding place $R$ and subscriber $S_1$ is illustrated. The network channel automaton $CH_1$ models a buffer of size two having the three states: empty, not empty and full. The size of the buffer is determined by model checking until the state labeled full is no longer reachable. The send event ($queue\_out[1]!!$) and receive event ($queue\_in[1]??$) of the
network channel, results in a change of state, except in the state labeled full. In this manner, an asynchronous message exchange between clients and servers of an interaction pattern is expressed as a network of timed automata.

**Modeling Network Resources**

To make the analysis more interesting, we restrict the enabling of receive and send events of a network channel to the availability of network resources. Consider an example of two network channels $CH1$ and $CH2$ sharing one network resource $NR$ as shown in the Fig. 7.17. A network resource is modeled as an automaton with three states idle, busy and done. In the idle state, the automaton waits for one of the network channels to synchronize over its lock event (modeled as the channel $qr\_lock$) leading to the state busy. The automaton stays in the busy state for a finite amount of time (specified by a time interval $[1, 2]$ using guards and invariants) after which the automaton reaches its final state. Here it synchronizes

**Figure 7.17: Channel with Finite Resources**
over its unlock event (modeled as the channel qr_unlock) with the waiting network channel and returns to its idle state.

To model an additional resource, an additional instance of a resource automaton must be instantiated. Further, the states of all network channels having outgoing lock and unlock events are extended to provide a non-deterministic choice of lock and unlock events over all available network resources.

So a network resource introduces delays for communicating transitions and this delay depends on how many of these transitions are competing for the same resource at that given moment.

7.4.2 Model Checking with UPPAAL

In this section, we show how components communicating over PS, RPC and GFR interaction patterns, extended with task execution times, message time-outs, network channels and network resources; are modeled and analyzed using UPPAAL.

Publish-Subscribe Pattern

In the Fig. 7.18, we present an architectural diagram consisting of three root components (C1, S1, S2) communicating over a PS pattern (1 client and 2 servers) and its corresponding network of timed automata. For readability, we do not explicitly model network channels and resources but mention their synchronizing events. The client embedded in component C1, publishes messages (as two successive send events) to its two servers embedded in components S1 and S2 (fusion of idle places), modeled by synchronizing events (queue_in[1] and queue_in[2]) of the two network channels CH1 and CH2. We do not model the PS pattern using the broadcast channel because it is non-blocking. The two network channels CH1 and CH2 compete for one resource automaton NR which induces a delay in the interval [1, 2]. When the components S1 and S2 are in their respective idle state, they have the possibility to synchronize (queue_out[1] and queue_out[2]) with their corresponding network channels (modeling the consumption of the published messages) leading to the state received. From this state, the server goes back to its idle state after a finite delay from the specified interval [1, 2].

Each component of the system operates at some specified rate called its loop rate, which specifies how often the component must return back to its idle state. We model this feature by adding a new timer automaton for each automaton modeling a component. A timer automaton periodically enters a waiting state to synchronize (over channel timer_Ch[id], where id ∈ {1, 3, 4}) with its corresponding component every loop rate time units. This synchronization event takes the components back to their idle states. So in our model, we have the three timer automaton: ClientOneTimer for component C1 (loop rate: X), ServerOneTimer for component S1 (loop rate: Y) and ServerTwoTimer for component S2 (loop rate: Z). The invariant and guards of the idle states of components S1 and S2 ensures at most one time unit may pass in that state after which it is forced to
**Architectural Model:** Components interacting over the PS pattern

**Analysis.** For this network of timed automata, we show how safety and performance properties specified as timed temporal logic formulas can be verified using UPPAAL.

- **Absence of deadlocks:** This is a sanity check for models in UPPAAL, expressed by the formula: $A[] \neg \text{deadlock}$. The model was found to have no deadlocks.

- **Buffer utilization:** The objective here is to ensure that none of the channels $CH1$ and $CH2$ are full in any of the reachable states. In UPPAAL this property can be expressed as: $E() \ CH1.\text{Full} \ or \ CH2.\text{Full}$. We verify the following conditions:

  If we set the loop rates as $X = 8, Y = 3, Z = 3$ then it turns out that the channels overflow. However, setting the loop rates to $X = 10, Y = 3, Z = 3$ ensures that the channels never become full. Note that this is one of the several possible combination that achieve this behavior.
Change propagation time: The objective here is to compute the bounds on the delay incurred from the moment of publishing a message by a component, say $C1$ till its consumption by the components $S1$ and $S2$.

For this, we add two global clock variables named $transfer[1]$ and $transfer[2]$. Whenever a message is sent over the channels $CH1$ and $CH2$ by the component $C1$, the clocks $transfer[1]$ and $transfer[2]$ are reset, respectively.

We must also ensure that in all reachable states, the network channels are never full and the component $C1$ does not reset the clocks $transfer[1]$ and $transfer[2]$ before the server has received these messages and returned back to idle. This can be ensured by first identifying the right combination of loop rates $X$, $Y$ and $Z$ such that none of the buffers are full and then reassigning the value $X = \max\{Y, Z\} \times K$, where $K \geq 2$.

Next, we inspect the bounds of the clock variables $transfer[1]$ and $transfer[2]$ in the state $received$ of components $S1$ and $S2$, respectively. Consider component $S1$. The bounds of the clock variable $transfer[1]$ in the state $received$ of component $S1$ can be computed by identifying the two values $L \in \mathbb{N}$ and $U \in \mathbb{N}$ such that (a) $L \leq U$, and (b) $L$ is the first highest value that does not satisfy the property $E() S1.received \land transfer[1] \leq L$ (i.e. there exists a path such that eventually $S1$ is in the state $received$ and $transfer[1] \leq L$), and (c) $U$ is the first lowest value that does not satisfy the property $E() S1.received \land transfer[1] \geq U$. So we have our bounds $[L + 1, U - 1]$, which is the time bound for a message published by component $C1$ to be received by component $S1$. In our example, we find the bound to be in the interval $[1, 10]$.

Cycle time: The cycle time of a component is the time interval specifying how long it takes for a component to return back to its idle state by synchronizing with its timer automaton. Suppose we want to compute the cycle time of $S1$. For this we must add a clock variable $clk$, which is reset everytime the component synchronizes with its timer automaton. By the invariant of the idle state and guards on outgoing edges, the lower bound $L$ of the cycle time is 1, (i.e. this is the earliest time that automaton $S1$ can reach its final state). The upper bound $U$ of the cycle time can be verified by computing the first lowest value $U$ that $E() \ l o o p r a t e[4] \geq U$. The value of $U$ turns out to be 4. So the cycle time of component $S1$ is in the interval $[1, 4]$. This is clearly not desirable because then it means that component $S1$ is not able to maintain its loop rate of 3 time units. One solution could be to increase the value of loop rate corresponding the timer automaton of component $S1$. However, if this update is carried out then all preceding properties must be verified for this setting.
RPC Pattern

In the Fig. 7.19, we present an architectural diagram and its corresponding network of timed automata consisting of two root components (C1 and C2), each containing one client of an RPC pattern connected to one server component of an RPC pattern S1.

The RPC clients embedded in components C1 and C2, communicate with their server component over the network channels REQUEST (REQ) and RESPONSE (RES). Each client sends a request by synchronizing with the network channel REQUEST (REQ) and then waits for a response from the server over the network channel RESPONSE (RES). The two network channels compete for one network resource NR. The network resource induces a delay between 1 and 2 time units between a lock and an unlock event. When the server receives a message over the network channel REQUEST (REQ), it goes into a busy state, where it performs a computation, modeled by a delay between 2 and 3 time units, after which it sends a response back to the client over the network channel RESPONSE (RES) and goes back to its idle state. When the client receives the response message over the network channel RESPONSE (RES), it can return back to its idle state only when a synchronizing with its timer automaton is available. In the idle state of the com-
ponent $S_1$, at most one time unit can elapse before it must change its state to done. In this state it waits for a synchronization event from its timer automaton to become available to handle requests from other clients.

**Analysis.** For this model, we verify the absence of deadlocks, check the buffer utilization and compute the bounds on the cycle time in a similar way as for the PS pattern. However, instead of computing the bounds on change propagation of request and response messages, we will compute the bounds on the delay incurred by a client of an RPC pattern embedded in the components $C_1$ and $C_2$.

Consider the case where the loop rate of $C_1$ is 40 time units, $C_2$ is 60 time units and $S_1$ is 10 time units. Note that clock variable $client[1]$ of component $C_1$ is reset when it synchronizes with its timer automaton leading to the state idle. The bounds on the time that it takes for the RPC client embedded in the component $C_1$ to reach its final state can be verified by checking the value of the clock variable $client$ in the state waiting and final in the following way:

Identify the two values $L \in \mathbb{N}$ and $U \in \mathbb{N}$ such that (a) $L \leq U$, and (b) $L$ is the highest value that does not satisfy the property $E[\langle \rangle C_1.waiting \text{ and } client[1] \leq L]$, and (c) $U$ is the lowest value that does not satisfy the property $E[\langle \rangle C_1.final \text{ and } client[1] \geq U]$. So we have our bounds $[L + 1, U - 1]$, which in this case turns out to be in the interval $[4, 24]$.

**GFR Pattern**

To model the GFR pattern, we must extend network channels to support the semantics of inhibitor and reset arcs (for the places feedback, cancel and result). The former can be modeled as a receive communication event (see $\text{checkEin}[5]$? in the Fig. 7.20) in the idle state of the network channel. The latter can be modeled by adding edges corresponding receive communication events from the states not empty and full back to the idle state (see $\text{resetIn}[5]$? in the Fig. 7.20).

With this model of a network channel, we give an example of two components interacting over a GFR pattern (root component $C_1$ containing a GFR client and its GFR server component $S_1$) and its corresponding network of timed automata, in the Fig. 7.21. There are seven network channels corresponding the goal, accepted, rejected, feedback, cancel, result and done. As before all network channels share one network resource which induces a delay between 1 and 2 time units between each lock and an unlock event. The GFR client initiates the communication by sending a goal over the goal network channel. When the server receives this message, it either sends back an accepted or rejected message over the corresponding network channel. In case of the latter, the client reaches its final state and the server goes back to its idle state. In case of the former, the client goes to state busy and the server to state active.

In the active state of the server (urgent location), it is either able to go to the state busy, succeeded or aborted. In the busy state the server incurs a delay between 1 and 2 time units, after which it sends a feedback to the client and
returns to the active state. To limit the number of iterations of this loop, we use a data variable $e$ and ensure in the guard that this loop is taken at most three times. If we want the number of iterations to be random within an interval then this is also possible. If the client has sent a cancel message, then the server goes to the state canceled. From the states canceled, succeeded or aborted the server sends a result message back to the client and waits for a done message.

In the busy state of the client, it is either able to accept feedback messages from the server or send a cancel message. In the latter case, the state canceled is reached, where it waits for a result message from the server. Once the result message has arrived from the server, the client sends the done message and then resets the feedback channel. If in the busy state of the client, a result message was received, then it goes to the state gotResult. In this state, the feedback channel is forced to empty due to the inhibitor synchronization on the outgoing edge from the state gotResult to the state prepDone. When the client reaches the state prepDone, a done message is sent back to the server and the client reaches its final state.

When the server has received the done message, it resets the cancel channel and waits for its timer automaton to synchronize, leading to its idle state. The client in the component $C1$ also waits for a synchronization with its timer automaton, taking it back to its initial state.
Figure 7.21: GFR Pattern as a Timed Automata
**Analysis.** We will now verify a few properties of this model. Suppose the loop rate of components $C_1$ and $S_1$ are both 41 time units. The analysis of the absence of deadlocks, buffer utilizations and cycle time are similar to the previous approaches. It turns out that there are no deadlocks and none of the channels can become full, except for the feedback channel. This is acceptable because the loss of messages over the feedback channel does not affect the weak termination property since a reset of this channel, eventually occurs.

Next, we check if the feedback and cancel channels of the GFR pattern are empty when the component $C_1$ is in its initial state and component $S_1$ is in its idle state. We specify the two properties as: $E(\langle\rangle C_1.\text{initial and } S_1.\text{idle and } \text{(feedback.NotEmpty or feedback.Full)})$ and $E(\langle\rangle C_1.\text{initial and } S_1.\text{idle and } \text{(cancel.NotEmpty or cancel.Full)})$. As both these properties evaluate to false, our model has the desired behavior.

The propagation time for feedback messages to reach the GFR server can be computed by using the clock variable $FClk$ such that (a) it is reset by the server whenever it sends a feedback, and (b) by modifying the guard on variable $e$ to ensure that the feedback loop is taken at most one time. The latter ensures that a new feedback is not produced before the client has consumed it.

The bounds on the delay incurred by the GFR client to reach its final state can be computed using the clock $CClk$ in a similar manner as for the RPC pattern. It turns out to be in the interval $[2, 21]$.

**Simple Model of a Navigation System**

In the Fig. 7.22, we present a simplified architectural model of a navigation system containing one PS pattern, one RPC pattern and one GFR pattern, and its network of timed automata. The GFR server component labeled $navigation$ contains one client of an RPC pattern and one client of a PS pattern. The human operator makes an indication on the user interface about a goal. The user interface in turn sends this goal to the $navigation$ component. If the navigation component accepts this goal and becomes active then it publishes feedback to the user interface and commands to the platform controller component. Within the same feedback loop, the GFR component also requests the current location of the platform by invoking the RPC server of the platform controller component.

The analysis techniques developed in the previous sections are used to guide the design of the system. i.e. the objective is to identify the right operational values for the loop rate of components, network delays and quantity of network resources such that at least all the following properties hold:

- There are no deadlocks in the system
- None of the buffers are full.
- None of the upper bound of the cycle time of a component exceeds its loop rate, i.e. all components are able to go back to their idle state within the specified loop rate.
Figure 7.22: Architectural Model as a Timed Automata
The bounds on the change propagation time are within the desired limits. This property allows us to answer questions like for instance how long does it take for a user to stop the platform using a cancel message?.

As systems grow larger, their state space grows exponentially and as a consequence, their verification becomes costly and time consuming. One In the next section, we propose some structural reduction rules for DTPN, to combat state space explosion and thereby increase the effectiveness of model checking.

### 7.4.3 Structural Reduction Rules for DTPN

In this section, we present four reduction rules for DTPN, namely transition fusion, priority elimination, place fusion and subworkflow net reduction. These reduction techniques either fuse together transitions or places, or eliminate a subnet of the system.

**Transition Fusion**

We explain the *transition fusion* reduction rule using the Fig. 7.23. This rule may be applied, if two transitions share exactly one common place $R$ such that (a) it is a post-place for one transition $T_1$ and a pre-place for the other transition $T_2$; (b) has non zero delay on incoming arc equal to $X$ and on outgoing arc equal to $[A, B]$; (c) the place $R$ is not connected to any other transition; and (d) the place $R$ is not marked in the initial marking: The application of this rule fuses $T_1$ and $T_2$ into one transition $T$ while preserving the pre-places of $T_1$ and post-places of $T_2$ such that (a) zero delay on incoming arc to transition $T$, and (b) delay interval $[A + X, B + X]$ on outgoing arc from transition $T$. 

![Figure 7.23: Reduction Rule: Transition Fusion](image)
By considering $T1$ as a silent $\tau$-step, it is easy to check that transition fusion reduction preserves weak bisimulation.

**Priority Elimination**

This reduction rule eliminates transitions that will never be enabled. Consider the set $T$ of transitions, all having the same set of pre-places $P$ such that $|T| > 1$ and $|P| > 0$.

The *priority elimination rule* maybe applied, if two or more transitions have exactly the same set of pre-places and there exists a transition (which we call an *active* transition) such that the delay on the arc from each place in its preset is strictly less than the delay from that place to every other transition. Formally, if there exists a transition $T' \in T$ such that $\forall T'' \in T : T'' \neq T' \Rightarrow \forall P' \in P : \delta(P',T') < \delta(P',T'')$, then priority elimination maybe applied.

If there exists an active transition in the set $T$, then all other transitions are called *dead*. The application of the priority elimination rule eliminates dead transitions and all its successors. In the Fig. 7.24, we give two examples of priority reduction.

It is easy to verify that priority reduction preserves strong bisimulation.

**Place Fusion**

We will use the Fig. 7.25, to explain the *place fusion* reduction rule.

The *place fusion* rule maybe applied if (a) two transitions $T1$ and $T2$ exist such that the postset of transition $T1$ (i.e. $\{R,S\}$) is equal to the preset of
transition $T2$ (i.e. $\{R, S\}$); (b) the places belonging to this set (i.e. $\{R, S\}$), are not connected to any other transition.

The application of the place fusion rule fuses the places $R$ and $S$ into one place $RS$ while preserving the preset and postset of these places. Furthermore, for each of the places $R$ and $S$, the input delay associated with this place is added to the lower bound and upper bound of the time interval associated with this place. The delay on the outgoing edge of transition $T1$ to $RS$ is the interval $[L, U]$, where $L$ is the minimum of all the lower bounds of interval delays associated with $T1$ in the original net, and $U$ is the maximum of all upper bounds of interval delays associated with $T1$ in the original net.

**Workflow net Reduction**

This reduction method replaces a subnet of the system that is a workflow net (and a DTPN), by another workflow net (also a DTPN) having a single transition and whose outgoing arc has the time interval corresponding the lower bound and upper bound of executing this workflow net in isolation. By executing, we mean the firing of all possible executable firing sequences from its initial marking to its final marking.

Note that the subnet that is a workflow net has an additional requirement, i.e. all nodes of this subnet except the initial and final places, are not connected to any node of the rest of the system. We call this subnet as a subworkflow net. The time interval of executing a subworkflow net can be determined by constructing its rDTPN (see Chapter 4). For the interval reduction step of the reduction method of DTPN, note that the grid distance of the subworkflow net must be equal to the
grid distance of the system and not that of the subworkflow net. In the Fig. 7.26, we illustrate this strategy.

7.5 Conclusions

In this chapter, we have applied our component-based architectural framework and its construction method (presented in the Ch. 3) to the ITER RH plant and Robot ROSE. In the former, we have used informal descriptions of the ITER RH system to identify components and interaction patterns of the system, and represent their dependencies as an architecture diagram. The Petri net model underlying this diagram is easily derived using the construction method. The architectural diagram of ITER RH adds value by capturing both the structural and behavioral aspects of the system. Furthermore, the construction method is simple for use by software architects. In the latter, we start with a set of use cases and use both bottom-up construction and top-down refinement, to derive the Petri net model of the software system of robot ROSE. For safety and performance analysis, the Petri net model is extended to a DTPN, specifying task execution times and message time-outs. For automatic verification by tools such as UPPAAL, the DTPN model was transformed into a network of communicating timed automata. For a few examples, the verification of timed behavioral properties like the absence of deadlocks, buffer utilization, bounds on the change propagation time and cycle time of components were carried out. Lastly, the four behavior preserving reduction rules for DTPN reduce the computational effort of model checking large systems.
In this thesis, we have made several contributions to the design and analysis of component-based software systems of robots. We focussed on the control flow aspect and used the Petri net formalism to define a component-based architectural framework. The Petri net formalism was chosen due to its nice structural properties and an intuitive graphical notation.

To support structured system design, an incremental, refinement-based construction method was prescribed such that the resulting system was guaranteed to weakly terminate. This way of constructing systems is appealing due to its simplicity and therefore it can be easily understood, even by non-experts.

For performance and reliability analysis, the framework was extended with a notion of time (Discrete Timed Petri nets) and its stochastic variant (Discrete Stochastic Petri nets). The former is based on the timestamp approach and encompasses several well known models of time. For its analysis, structural reduction techniques and a novel state space reduction method were proposed. The latter is based on discrete probabilities for choices and for its analysis, we have shown how Markov techniques and structural techniques can be exploited. Both these formalisms serve as an intuitive means of specifying timed behavior of distributed message-passing systems.

Furthermore, the relationship between Simulink models and our architectural framework was also studied. For this a formal semantics was given to Simulink using the Colored Petri net formalism (subclass of DTPN, discarding color). As a result, several interesting timed properties of Simulink specifications can be verified by model checking.

In the remainder of this section, we summarize our contributions (see Fig. 8.1).
We developed an architectural framework for specifying the behavior of component-based software systems of robots. The framework is based on the two concepts: components and interaction patterns. Petri nets have been used to model components as strongly connected state machines and the four interaction patterns as multi workflow nets. Three of these interaction patterns, namely Remote Procedure call (RPC), Goal Feedback Result (GFR) and Publish Subscribe (PS) are recurring in the software systems of robots. For these patterns, their weak termination property was shown. The fourth interaction pattern, namely the Mirrored Port (MP) pattern is a class of mirrored communicating state machines. For this class, the weak termination property was proven and a construction method to derive arbitrary MP patterns was prescribed. The prescribed construction method can be seen as a best practice in communication protocol design. Furthermore, the class of MP pattern defines a client-server interaction model. So the prescribed method may also be used for extending the ROS communication model with new interaction patterns. (Research Question \textbf{R1}, Section 1.5).

To specify the system in a concise manner, an architectural diagram notation was introduced, specifying the relationship between components and their interaction patterns. For the weak termination property it was shown that the relationship between components must be acyclic. To facilitate the incremental construction of systems, refinement techniques were developed to insert an interaction pattern into a component and elaborate component behavior by a class of weakly terminating workflow nets. These refinement techniques were shown to preserve the weak termination property. The prescribed construction method relies on these techniques to derive a Petri net model of the system, starting from an architecture diagram. The construction method is incremental and is a combination of bottom-up construction and top-down elaboration techniques. As a consequence, the roles of a system integrator and component developer are clearly separated and the component developer can focus on realizing the functionality of each component without having to worry about how the component will behave in a composition. It is not necessary to derive a system in this way, but if it is possible to do so, then the resulting system is guaranteed to satisfy the weak termination property. (Research Question \textbf{R1}, Section 1.5).

For performance and reliability analysis, the Petri net model of the architecture framework was extended with a notion of time. For this, we developed a framework for time in Petri nets called Discrete Timed Petri net (DTPN) and its stochastic variant called Discrete Stochastic Petri net (DSPN). Unlike its subclass Interval Timed Colored Petri net, the class of DTPN is able to specify fixed delays on arcs from places to transitions. This is useful for modeling priorities between events and message time-outs. For the DTPN model, we presented a state space reduction method to transform its infinite state space into a finite one, while preserving simulation equivalence.
The novelty of this approach lies in the transformation of closed rational intervals into a finite one. As a consequence, verification by model checking is possible for DTPNs. Since ITPN is a subclass, the reduction method is directly applicable. Note that existing reduction methods for an ITPN is preserving only trace equivalence, which is a weaker notion than simulation equivalence. For the DSPN model, we have shown how Markov techniques can be used to answer questions like probability of reaching a marking, expected time to leave a marking and expected sojourn times in equilibrium. For a subclass, we show how structural analysis techniques can be used to reduce computational effort. (Research Question R2, Section 1.5).

To position our DTPN model against existing models of time in literature like timed automata and timed extensions of Petri nets like time Petri nets and timed Petri nets, we have studied these formalisms and shown how they can be unified by transformations from one to the other. As a result, the advantages of each formalism can be exploited. We identified two subclasses of DTPN that are trace equivalent and branching bisimilar to a timed automata. We have also shown that for every timed Petri net there exists a branching bisimilar DTPN, and for every time Petri net there exists a trace
equivalent DTPN. Furthermore, by modeling inhibitor arcs with subclasses of DTPN, the Turing completeness property for DTPN was proven. From a modeling perspective, we also considered the notion of modeling comfort using subclasses of DTPN. (Research Question R2, Section 1.5).

- It is common practice by control engineers to model time driven systems using the Simulink toolset. As Simulink lacks a formal semantics and has only operational semantics in the form of an implementation, we gave a formal semantics to Simulink using DTPN. For Simulink models expressed a DTPN several interesting properties were verified by state space exploration. By relating the notion of time progression in our architectural framework to that of Simulink systems, we have shown how the latter can be expressed in the former by refinement techniques. Furthermore, for completeness, we have also shown how the behavior of a Petri net can be simulated using the Simulink toolset. (Research Question R3, Section 1.5).

- We have demonstrated the applicability of the above results using two real life examples: service robot ROSE and the ITER Remote Handling Plant. For both these cases, we have shown how architectural models can be derived in a structured way starting from user requirements. For verification of timed properties by model checking, we have shown how commonly available tools like CPN Tools and UPPAAL can be used. (Research Question R4, Section 1.5).
In this thesis, we have focussed on the control flow aspect of systems and its logical correctness properties like weak termination, as well as performance and reliability in the timed and stochastic settings, respectively.

Although weak termination is an important property, it is only a sanity check and a starting point for system verification. Data plays an important role in the correct behavior of such systems. As data can be used as preconditions for transition enabling, data manipulation by transitions can destroy this property. Therefore, it is interesting to consider extensions of the weak termination property with local checks on data manipulation in components. By considering such extensions, domain knowledge can be transferred to architectural models in order to carry out more detailed analysis, like for instance, the behavior of the system in the presence of faulty sensors, etc.

Out of the four considered interaction patterns in our architectural framework, the GFR pattern is a very useful and recurring pattern in the software systems of robots. The other interaction patterns also occur in other domains such as in networked robot systems, supervisory control systems etc. So our architecture framework can be applied to them as well for analysis of performance and reliability. However, in order to make our framework more specific to robots, it would be interesting to identify and model more such domain specific patterns.

For performance evaluation of architectural models, we rely on verification by model checking. As models become larger, this technique does not scale very well due to a large state space. To some extent, the structural reduction techniques for analyzing a DTPN helped address this problem. Therefore, it would be very interesting to consider more advanced techniques that exploit the structure of our architectural models, i.e. the manner in which interaction patterns are refined into the system using the construction method, by first carrying out timed analysis.
of these patterns in isolation and then combining these results to predict the performance of the system.

The automation of the results presented in this thesis, requires an extension of the various available tools. For modeling systems using Petri nets, Yasper is one of the commonly used tools. By extending this tool to support our architecture diagram notation as a top level view, while carrying out under the hood transformations to a Petri net model (as per the construction method) would tremendously aid software designers. For timed analysis of Petri nets using the timestamp approach, the use of CPN Tools is quite prevalent. However, only a subclass of DTPN can be modeled and analysis based on timed reachability is still immature. Lastly, for analysis of DSPN, no tool support is yet available.
Appendices
There is an impressive collection of work that has been carried out in applications of formal methods to the modeling and analysis of component-based systems. In this chapter, we will present a few relevant and interesting contributions arising from the theory of automata and Petri nets.

A.1 Automata and Petri nets

Automata. Automata are a well-known formalism for specifying a system. The notion of finite automata was introduced by [231] in the mid-forties and since then the area of finite automata has been developed further by [234, 247] in the fifties. Automata are systems consisting of states, some of them designated as initial or final, and (usually) labeled transitions representing events or actions. The basic model of an automata are not suitable for specifying all aspects of real-time distributed systems. As a result, many extensions have been proposed based on synchronous communication paradigm. Some well-known extensions include Communicating Automata [73] (Lossy Channels [124, 9]), Asynchronous Automata [366] and Asynchronous Cellular Automata [98]. For performance and reliability analysis, the formalism of Timed Automata [20] is very popular. For the specification of component based systems, extensions such as I/O automata [218, 217], its extension Team Automata [32] and its modification Interface Automata [106] have been proposed. Another interesting automata-theoretic approach for component-based modeling is interaction systems [143, 142, 141, 297, 144], where
the focus is on the clear separation between interactions and local behavior of components.

In the eighties, Statecharts [159] grew in popularity for specifying designs of reactive systems. The Statecharts formalism offers facilities for hierarchical structuring of states and modularity, making it easier to describe complex systems at a high level description and promoting stepwise development. Many features like synchronous product and synchronous broadcast (see [37, 241, 240]) were already present in Statecharts, but determinism was not ensured and suffered many semantic problems, as pointed out in [340]. A derivative of Statecharts, Argos [223], is a graphical language for describing hierarchical finite state machines. Almost at the same time, many high level abstractions in the form of synchronous languages (like Esterel [44], Lustre [155], Signal [37], etc.) were developed, with an underlying automata-based semantics. These languages describe systems with a global clock, as a set of concurrently-executing synchronized modules that communicate by means of signals. A signal represents an event, that is either present or absent in each clock tick. The modules of the system are reactive in that they only perform computation and produce outputs events at instants where at least one input event has occurred. The drawback of such languages lie in their inability to capture causal dependencies. This stems from the lack of unique and deterministic semantics. The problem is synonymous to the combinatorial loops problem in synchronous circuits [220, 183, 312]. As a consequence, model transformation becomes non-trivial. The weakening of the synchronous communication assumption as a way to get around these problems has been proposed by several authors [68, 69, 62].

**Petri nets.** Petri nets are a generalization of an automata. They were introduced in 1962 by Carl Adam Petri in his doctoral dissertation titled *Kommunikation mit Automaten* (Communication with Automata) [271]. Petri nets are a powerful modeling formalism that combines a well-defined mathematical theory with a graphical representation that captures the dynamic behavior of systems. The theoretical aspect of Petri nets allows for precise modeling and analysis of system behavior, while the graphical representation of Petri nets provides an intuitive visualization of the state evolutions of a modeled system. This combination aids understanding and makes it a natural way to convey concepts of system behavior. Petri nets models are purely logical, describing the order of occurrence of events, without associating time to them. The basic constructs such as causality, choice, and concurrency provide an expressive means to specify event ordering. Unlike other discrete event models, such as automata, the state of a Petri net, also known as its marking, is represented as a vector of non-negative integers. Then the state space can be represented as an algebraic structure. As a result, integer programming methods can be applied for analysis purposes [299].

In the Petri net semantics, both states and events are defined on the same footing. So both interleaving and true concurrency [74] semantics can be defined in Petri nets [53]. Furthermore, the descriptive power of Petri nets [179] lies
between that of finite automata and Turing machines [270]. However, adding inhibitor arcs to a general Petri net leads to Turing completeness [308], which in turn makes most interesting properties undecidable.

As general Petri nets are too expressive, several sub-classes of Petri nets (characterized by structural constraints) have been identified and studied: free choice nets, persistent nets, marked graphs, state machines and workflow nets. These sub-classes represent an interesting class of systems and their restricted modeling power leads to simplified analysis. Furthermore, several Petri net extensions allow the description of logical models (Place-Transition nets, Colored PN (hierarchy and data) [177]), timed models (Time/Timed Petri nets [280, 235]), component models (Open Petri nets) and performance models (Generalized Stochastic Petri nets [13]) have been proposed. Many applications of these models to specify various kinds of discrete event dynamic systems can be seen [346] such as computer networks [225], communication systems [235, 345], manufacturing plants [336, 364, 112], command and control systems [22], real-time computing systems [222, 319], logistic networks [335], workflows [330] and many more.

The properties of a Petri net are classified as either behavioral or structural [250]. The properties of the former type depend on the initial state or marking of a Petri net. Some behavioral properties include boundedness, reachability, reversibility, coverability, persistence, liveness, synchonic distance and fairness. For the analysis of these properties many algorithms based on mathematical and practical foundations have been developed [16, 105, 105, 299] The analysis methods for Petri nets may be classified as reachability [230, 191, 206, 284], matrix-equation approach [249], refinement and reduction techniques [251, 47, 45]. The techniques for refinement and reduction facilitate the analysis of complex systems by means of behavior preserving transformations. The former refines an abstract model into a more detailed model in a hierarchial manner, while the latter reduces a system model into a simpler one, while preserving behavioral properties like liveness and boundedness. So the former is a synthesis technique while the latter is applied to ease system analysis.

In contrast to behavioral properties, structural properties [250] (like structural boundedness, conservative, repetitive, consistent, controllable and structural liveness) do not depend on the initial marking of a Petri net, rather they depend on the topology, or net structure, of a Petri net. The structural analysis of Petri nets rely on the concepts of place/transition invariants, siphons and traps. Furthermore, the structural properties of many sub-classes of Petri nets like state machine, marked graphs and free choice nets, have been extensively studied [110]. For these sub-classes, behavioral properties such as freedom of deadlocks and live-locks (known as soundness, weak termination etc.) can be guaranteed by their structure. These results form the basis for many correctness by construction design methods, which we discuss in a later section.

The notion of Time. The notion of time has been extensively studied in the context of real time systems using both these formalisms resulting in many
well-established model checking techniques and good tool support.

On the one hand, timed automata are a widely used formalism for modeling and verification of real-time systems. A timed automata is a finite state automata extended with a set of non-negative clock variables and set of clock constraints. As time is non-decreasing, the state space of a timed automaton is infinitely large. For this many equivalent and decidable finite reduction methods based on concepts of region graphs and zone graphs were proposed. Since their introduction almost twenty years ago, around eighty variants of timed automata (parametric, probabilistic etc.) have been proposed to address many aspects of real time systems (see [341] for an extensive survey) supported by a huge number of tools for analysis, verification, controller synthesis and code synthesis. Tools such as UPPAAL [33], RED [342] and VerICS [261, 182] are the most mature.

On the other hand the interest for time in Petri nets is quite old starting with timed Petri nets [280], which associated a fixed time to transitions and its variant [296], which associated time to places. From a modeling perspective, the former is interpreted as the duration of an activity while the latter is interpreted as the duration for which a condition holds in a place. The duration of an activity specified as a fixed deterministic delay is very restrictive. For practical applications, it is more natural to use delay intervals specifying the upper and lower bounds of the duration of an activity.

The following work [256, 244] proposes the class of stochastic Petri nets where transitions are associated with a duration modeled as a probability distribution. If these delays have a negative exponential distribution then they can be translated into a continuous markov chain which can be used to analyze interesting mean valued properties and also general behavioral properties. However, such analyses do not provide absolute guarantees, as a negative exponential distribution means that a transition may fire with a delay between zero and infinity.

The time Petri nets formalism is a more general model that overcomes these drawbacks by associating an interval specifying an upper and lower bound for the firing delay associated with transitions.

Other approaches have proposed different ways to integrate time intervals in Petri nets like Place/Transition-nets with timed arcs [157], high-level timed Petri nets [123] and interval timed colored Petri nets [2]. The latter goes a step further and associates data (color) and timestamps with tokens which are assigned when they are produced and these values are used to determine transition enabling. The notion of time in tokens has been adopted from [162]. Some commonly used tools for modeling and verification of timed extensions of Petri nets include GSPN, TINA [51], ROMEO [130], CPN Tools [347] etc.

A.2 Model Checking versus Correctness by Construction

As both automata and Petri nets give their semantics over transition systems verification of behavioral properties by model checking techniques can be carried
Model checking [92, 276] is an algorithmic method for exhaustively examining if the state space of a model expressed as a state transition system satisfies a correctness specification. The correctness properties of reactive systems are generally specified in temporal logic [273]. It is a modal logic with operators to describe safety, liveness and precedence properties over a Kripke structure [78]. Intuitively, safety properties express that something bad will never happen, while liveness properties express that something good will eventually happen. On the other hand, precedence properties express an ordering relation between events. Many different types of semantics can be given to temporal operators depending on whether time is linear or branching, time is quantified, time is implicit or explicit, local or global etc. If time is quantified then it is called real-time temporal logic. The first model checking algorithms for branching time logics (CTL) were proposed in the early eighties by [92] and independently by [276]. A few years later, the first linear time algorithm for CTL was proposed in [93].

Unfortunately, model checking techniques suffer from the well-known state space explosion problem [108], which is further aggravated by concurrency. Many strategies try to address this problem to some extent by (a) efficient representation of states [94, 138, 311, 320, 209, 322, 45, 153] based on explicit state methods, partial order reductions, symbolic model checking techniques [233] etc. (b) state space abstraction techniques like predicate abstraction [149]. For the latter, various techniques for increased efficiency have been proposed like path slicing, loop detection, statically computed invariants and proof based refinements. In particular, the Petri net reduction techniques provide an added advantage of an even greater reduction in size of state space.

For large distributed systems, such techniques are yet inefficient due to high computation and memory overheads. In such cases, compositional verification techniques [42, 58, 95, 184, 198] are widely adopted. The underlying strategy involves the verification of a component while making assumptions that summarize the properties of other parts of the system they interact with. However, such assumptions lead to an over-approximation of unknown parts of the system and only safety properties can be verified. As a result, they are prone to generate false-positive counter examples.

The most desirable way to overcome the state space explosion problem is by constructing systems from their constituent parts while guaranteeing certain generic behavioral correctness properties like freedom of deadlock and/or livelock. Such techniques are commonly known as correctness by construction. These techniques prescribe rules for reasoning about the structure or topology of the system such that the desired correctness properties hold globally under certain assumptions about its constituents parts. As a consequence, they provide significant guidance during design process and at the same time avoid the need for posteriori model checking.
A.3 Formal Component-Based Approaches

In automata theory, I/O automata, its extension Team Automata and its modification Interface Automata provide a structured way to specify components and their compositions based on the compositional Assume-Guarantee reasoning (for an overview, see [91]). As stated in [142], these approaches are useful, for instance, in the verification of safety properties, provided that they can be easily decomposed into a conjunction of component properties.

Another interesting automata-theoretic approach for component-based modeling is interaction systems. These systems clearly separate interactions from the internal behavior of components. The work in [141] presents a construction methodology for constructing deadlock-free systems from components. Here the notion of a component is general with well-defined interfaces and behavior. However, to guarantee deadlock freedom (interaction safety), components must satisfy specific properties like the existence of non-trivial deadlock-free invariants, computation of guards etc. These checks can only be performed if the components have a finite state space. Furthermore, in the general case, guard computations could lead to state space explosion. As for such systems, the detection of deadlocks and verification of liveness is NP-hard, the work in [57, 29, 142] address these issues by establishing conditions that could be tested in polynomial time. All extensions of automata discussed in this paragraph support as their communication style variants of the strong and weak (non-blocking send and blocking receive) synchronization mechanism.

In the realm of Petri nets, composability and compositionality of nets have been extensively studied in the context of component-based reactive systems supporting both synchronous and asynchronous compositions. In Petri nets this is achieved by either fusing common transitions (synchronous composition) or places (asynchronous composition) of communicating components.

For synchronous compositions, the process algebra approach has been investigated in a number of works starting with [236, 359, 259] (that do not distinguish interface and internal places), to Petri Box Calculus [52, 193, 192] (distinguished interface places), leading to Petri net algebra [54]. However, in a distributed reactive system, synchronous communication may lead to inefficiencies and tight coupling between components as it requires the participation of both components. Furthermore, synchronous communication between processes cannot be naturally used to analyze state-based properties of asynchronous systems [140, 180, 181].

As a consequence, the asynchronous communication paradigm is widely adopted for the specification of distributed systems. The asynchronous composition of nets are performed over a set of shared places [305, 306]. A place based composition strategy is also adopted in the well-known formalism of hierarchial CPN [177] and supported by its associated tools like CPN Tools and DesignCPN. The survey in [140], presents several compositional approaches for high-level Petri nets.

The general model of a Petri net does not have a modularity concept and therefore lacks the capability to distinguish an interaction from an internal activity.
Many authors have addressed this issue. The first Petri nets with the notion of an interface occurs in [260, 274]. Here an interface was partitioned into an input part consisting of places and an output part consisting of transitions, with a composition operation connecting inputs to outputs. In [25], open nets were introduced. They were partly inspired by the notion of open graph transformation systems [161], an extension of graph transformation for specifying reactive systems. An open net is more general, as it endows a composition operation that is suitable for modeling both asynchronous interactions via open places and synchronization through common transitions. The work in [186], proposed the idea of a module concept for Petri nets each having one or more distinguished interface(s) (set(s) of places) for asynchronous communication and advocated the use of temporal logic to reason about their behavior. As a consequence of an interaction between modules over interfaces, tokens can appear and disappear from these places. This is a natural way to model the message passing paradigm and such nets are commonly referred to as open Petri nets (OPN). The class of OPN is widely used in different contexts, most notably in service oriented system design [310, 228, 331, 215] and business process modeling [61, 327, 328, 362].

Of particular interest are developments in the area of workflow nets [326, 323, 324, 325] and its extension open workflow nets [228, 331, 215] and the notion of soundness/weak termination [334, 326]. A workflow net models a procedure having a clear start and a finish. These nets have proven useful for modeling and analyzing business processes. An open workflow net is a generalization of a workflow net by introducing an interface for asynchronous message passing. The weak termination property [5] states that in each reachable state of the system, the system always has the possibility to reach the final state. Note that it is not necessary that the system terminates but the fact that there is always a possibility to do so. This property guarantees the absence of a deadlocks and livelocks. Furthermore, the weak termination property generalizes the soundness property, which in addition, requires the satisfaction of the proper completion property [334, 326, 145].

Although soundness property for workflow nets can be verified (in polynomial time for free choice nets [329]) using standard techniques like coverability analysis, it has been shown [163] that for three sub-classes of Petri nets, namely state machine nets, marked graphs and ST-nets, soundness is implied by their structure. These classes of nets are able to model non-deterministic choices and concurrency, so they are ideal for describing procedures in a component. Note that the class of free-choice Petri nets have been studied extensively [110] as they provide a good compromise between expressiveness and analyzability that is supported by strong theoretical results and efficient analysis techniques.

In the context of Service Oriented Architectures (SOA) [228, 329, 18], both static and dynamic composition of open Petri nets (components/services) have been extensively studied. According to the SOA paradigm, a systems is a (possibly open) network of asynchronously communicating components, where components provide a service (service provider) and may in turn consume services (service
consumer) from other components. In [216] an automata theoretic approach based on a so-called finite state service automata [229] is proposed, where the focus is on computing operating guidelines. However, deciding correct interaction using an operating guideline has the same complexity as model checking deadlock freedom in the composition itself. Other versions of automata based service models can be found in [170, 41]. A compositional approach based on Petri nets was proposed in [5], where a sufficient condition is presented to pairwise verify weak termination for tree structured compositions. As it is not necessary to know beforehand the whole structure of the service tree, the notion of dynamic binding is supported, which is an interesting feature of SOA. In [164], for a subclass of compositions of pairs of components, called ATIS-nets, weak termination is implied by their structure. ATIS-nets are constructed from pairs of acyclic marked graphs and isomorphic state machines, and the simultaneous refinement of pairs of places [334] with multi-workflow nets (generalization of workflow nets for specifying multiple client-server based communication protocols).
APPENDIX B

PROOFS OF LEMMAS AND THEOREMS

B.1 Proofs of Chapter 3

We present here the proof of Lemma 3.3.7

**Lemma B.1.1.** Consider the system $MP(B, \{A\}) = (N, M_0, M_f)$ with clients $B = \{B_1, \ldots, B_k\}$, where $k \in \mathbb{N}$, and initial marking $m_0 \in M_0$. Then for any reachable marking $m \in R(N, i_N)$ and an executable firing sequence $\sigma \in T_N^*: i_N \xrightarrow{\sigma} m$, there exist the firing sequences:

- $\sigma_A \in T_A^*: [i_A] \xrightarrow{\sigma_A}$ and $\sigma_j \in T_{B_j}^*: [i_{B_j}] \xrightarrow{\sigma_j}$, with $j \in \{1 \ldots k\}$, such that

  - $\sigma_A = \sigma |_A$ and for all $j \in \{1 \ldots k\}$, $\sigma_j = \sigma |_{T_{B_j}}$;

  - $\exists l \in \{1 \ldots k\} : \forall j \in \{1 \ldots k\} : j \neq l \land \forall t \in \sigma_j : \lambda(t) = \text{send}$, i.e. all firing sequences of clients, except for the chosen client $B_l$, contains only send transitions;

and one of the following conditions hold:

- $\sigma_A \cong \sigma_l$, i.e. firing sequences of the server and chosen client are isomorphic,

- $\exists \sigma' \in T_l^* : \sigma_A \cong \sigma_l \circ \sigma'$ where, $\forall t \in \sigma' : \lambda(t) = \text{receive}$, i.e. the firing sequence of the chosen client can be extended such that it is isomorphic with the firing sequence of the server,
Proof. We prove the lemma by induction on the length of the firing sequence \( \sigma \). The statements hold for the base case \(|\sigma| = 0\) and \(m = i_N\).

Suppose that it holds for \(|\sigma| = n\). Now consider \(\sigma'\) with \(|\sigma'| = n + 1\) and \(\sigma' = \sigma \circ t\) for some transition \(t \in T_N\). So for the firing sequence \(\sigma\) the induction hypothesis holds. There are now four cases to consider:

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>(</td>
</tr>
<tr>
<td>Case 2</td>
<td>(</td>
</tr>
<tr>
<td>Case 3</td>
<td>(</td>
</tr>
<tr>
<td>Case 4</td>
<td>(</td>
</tr>
</tbody>
</table>

**Figure B.1: The Case:** 1

**Case 1.** First, consider the case \(|\sigma_A| < |\sigma_l|\), i.e. the server \(A\) has fired less transitions than the chosen client, and the next enabled transition belongs to the server, i.e. \(t \in T_A\). Let \(\sigma'_A = \sigma_A \circ \langle t \rangle\).

By the induction hypothesis, there exists \(\sigma'' \in T_A^*\) and \(t' \in T_A\) such that \(\sigma_l \cong \sigma_A \circ \langle t \rangle \circ \sigma''\). If transitions \(t\) and \(t'\) are the same then we have the induction hypothesis. Suppose \(t \neq t'\). This means both transitions \(t\) and \(t'\) are enabled at the server. This is possible only if \(\bullet t \cap \bullet t' \neq \emptyset\) because the skeleton is safe and satisfies the state machine property. By the induction hypothesis and \(|\sigma_A| < |\sigma_l|\), we know that the isomorphic partner transition \(\rho(t') \in \sigma_l\) has already fired. Hence \(\lambda(t') = \text{receive}\). By the choice and leg property, \(\lambda(t) = \text{receive}\). Since transition \(t\) is also enabled, there exists a client \(B\), for some \(j \in \{1 \ldots k\}\) such that transition \(\rho(t)\) has fired in its firing sequence \(\sigma_j\). There are two cases to consider:
Figure B.2: Case 1: \( |\sigma_A| < |\sigma_l| \) and \( t \in T_A \)

- Suppose the transitions \( t' \) and \( t \) have been enabled at the server by the firing sequences \( \sigma_l \) of the chosen client and \( \sigma_j \) of one of the non-chosen client, respectively. We consider two cases:

  - Suppose the firing sequence of clients have only send transitions. Suppose the first \( |\sigma_A| \) transitions of \( \sigma_l \) and \( \sigma_j \) are not the same but both these subsequences end up marking the place \( \rho(t \cap t') \). This means \( \sigma_l \) and \( \sigma_j \) contain different legs. But \( \sigma_j \) has only send transitions, so it cannot contain a leg. Hence one of the firing sequences \( \sigma_l \) or \( \sigma_j \) are such that one is the prefix of the other. So we may swap the values of \( l \) and \( j \) and then we have the induction hypothesis.

  - Suppose the firing sequence of the chosen client \( \sigma_l \) has a receive transition. Then the firing sequence \( \sigma_l \) has a leg. This means that the firing sequence \( \sigma_j \) could not have taken this leg and enabled transition \( t \).

- Suppose both the transitions \( t' \) and \( t \) have been enabled by the same firing sequence \( \sigma_l \) of the chosen client, i.e. \( \sigma_l \) has fired both \( \rho(t) \) and \( \rho(t') \) and the interface places of these transitions have one token each. This can only happen if \( \sigma_l \) has a loop starting at the place in \( \rho(t) \cap \rho(t') \) and \( \sigma_l \) has
taken this loop (i.e. a leg) at least once more than $\sigma_A$. But this cannot be possible since the choice and leg property requires at least two transitions with different communication directions.

\[\text{CASE: } |\sigma_A| = |\sigma_l|\]

\[\sigma_A \sigma_l \sigma_j \sigma_A \sigma_l \sigma_j\]

\[\text{Figure B.3: The Case: 1}\]

Consider now $|\sigma_A| = |\sigma_l|$, i.e. the firing sequences of the server and the chosen clients are of equal length, and the next enabled transition belongs to the server, i.e. $t \in T_A$. Let $\sigma'_A = \sigma_A \circ \langle t \rangle$.

Suppose $\lambda(t) = \text{receive}$. This means that transition $\rho(t) \in \bigcup_{j=1}^{k} T_{B_j}$ has fired more times than $t$, by one of the clients. We consider two cases.

- Suppose transition $\rho(t)$ has fired at least one more time in $\sigma_l$, i.e. in the chosen client. This is not possible since $|\sigma_l| = |\sigma_A|$ and by induction hypothesis, we have $\sigma_A \cong \sigma_l$. This means all tokens produced by $\sigma_l$ have already been consumed by $\sigma_A$.

   Therefore $\lambda(t) = \text{send}$ and transition $\rho(t)$ is enabled by the firing sequence $\sigma_l$, i.e. $\bullet \rho(t) \cap \sigma_l \bullet \neq \emptyset$. Hence $\sigma_A \circ \langle t \rangle \cong \sigma_l \circ \langle \rho(t) \rangle$.

- Suppose transition $t$ has been enabled by some $\rho(t) \in \sigma_j$, $j \neq l$, i.e. one of the non-chosen clients, and then one of the following is possible:

  (a) $\sigma_l$ and $\sigma_j$ contain different legs and they both mark places $\rho(\bullet t \cap P_A)$ in their respective ports. But $\sigma_j$ contains only send transitions, so this is impossible by the leg property.

  (b) $\sigma_l$ is a prefix of $\sigma_j$. This means both $\sigma_l$ and $\sigma_j$ contain only send transitions. In such a case we may swap the value of $l$ and $j$ and then we are in the case where $|\sigma_l| > |\sigma_A|$.\n
\textbf{Case 2.} We are in the case where $|\sigma_A| \leq |\sigma_l|$, i.e. the server $A$ has fired less transitions than the chosen client, and the next enabled transition belongs to one of the clients, i.e. $t \in \bigcup_{j=1}^{k} T_{B_j}$. This leads us to two cases.
- Suppose transition \( t \in T_{B_l} \), i.e. the chosen client. Let \( \sigma'_l = \sigma_l \circ \{t\} \), i.e. \( \sigma'_l \cap \cdot t \neq \emptyset \). It must be the case that \( \lambda(t) = \text{send} \) because if \( \lambda(t) = \text{receive} \) it means \( \rho(t) \in T_A \) (server) has fired more times than transition \( t \). But this is not possible since \( |\sigma_l| \geq |\sigma_A| \) and by the induction hypothesis there exists \( \sigma'' \in T_A \) such that \( \sigma_l \equiv \sigma_A \circ \sigma'' \). Therefore \( \lambda(t) = \text{send} \). So the isomorphic partner transition \( \rho(t) \in T_A \) (server) must be enabled by the firing sequence \( \sigma_A \circ \sigma'' \). Hence \( \sigma_l \circ \{t\} \equiv \sigma_A \circ \sigma'' \circ \{\rho(t)\} \).

- Suppose \( t \in T_{B_j} \) and \( t \notin T_{B_l} \), i.e. \( t \) belongs to a non-chosen client. Let \( \sigma'_j = \sigma_j \circ \{t\} \) i.e. \( \sigma'_j \cap \cdot t \neq \emptyset \). It is not possible that \( \lambda(t) = \text{receive} \) because by the induction hypothesis we know that for every token produced by the firing sequence of the server, there exists an isomorphic partner transition in the firing sequence \( \sigma_l \) that has consumed this token. Hence \( \lambda(t) = \text{send} \) and then by the induction hypothesis \( \sigma_l \equiv \sigma_A \circ \sigma'' \) holds.

Case 3. We are in the case where \( |\sigma_A| > |\sigma_l| \), i.e. the server \( A \) has fired more transitions than the chosen client, and the next enabled transition belongs
to the server, i.e. \( t \in T_A \). Let \( \sigma'_A = \sigma_A \circ \langle t \rangle \). By the induction hypothesis
\[ \exists \sigma'' \in T^*_I : \sigma_A \cong \sigma_l \circ \sigma''. \]

There are now two cases to consider

- Suppose \( \lambda(t) = \text{send} \), then there exists \( \rho(t) \in T_{B_l} \) such that \( \lambda(\rho(t)) = \text{receive} \) and \( \rho(t) \) can be enabled by \( \sigma'' \). Therefore \( \sigma_A \circ \langle t \rangle \cong \sigma_l \circ \sigma'' \circ \rho(t) \).

- Suppose \( \lambda(t) = \text{receive} \). Since \( |\sigma_A| > |\sigma_l| \), all tokens produced by transitions in \( \sigma_l \) have already been consumed by the server transitions in \( \sigma_A \), hence \( t \) must have been enabled by one of the \( \rho(t) \in \sigma_j \), where \( j \neq l \) (non-chosen client) and \( \lambda(\rho(t)) = \text{send} \). Since \( \rho(t) \) has already fired either one of the following is possible: (a) \( \sigma_l \circ \sigma'' \) and \( \sigma_j \) contain different legs but they both mark places \( \rho(\bullet t \cap P_A) \) in their portnets. But \( \sigma_j \) contains only send transitions, so this is impossible by the leg property. (b) \( \sigma_l \) is a prefix of \( \sigma_j \). This means both \( \sigma_l \) and \( \sigma_j \) contain only send transitions. In such a case we may swap the values of \( l \) and \( j \) and then the statement holds.

**Figure B.6:** The Case: 4

**Case 4.** We are in the case where \( |\sigma_A| > |\sigma_l| \), i.e. the server \( A \) has fired more transitions than the chosen client, and the next enabled transition belongs to one of the clients, i.e. \( t \in \bigcup_{j=1}^k T_{B_j} \). There are two cases to consider.

- Suppose \( t \in T_{B_l} \) (in the chosen client). Let \( \sigma'_l = \sigma_l \circ \langle t \rangle \) i.e. \( \bullet t \cap \bullet t' \neq \emptyset \). By the induction hypothesis exists \( t' \in T_{B_l} \) and \( \sigma'' \in T^*_{B_l} \) such that \( \sigma_A \cong \sigma_l \circ \langle t' \rangle \circ \sigma'' \). We have to show that \( t' = t \).

Let \( t' \neq t \). This means both \( t \) and \( t' \) are enabled in the chosen client. By the structure of a client (i.e. skeleton is an S-net), this is possible only if \( \bullet t \cap \bullet t' \neq \emptyset \). Now \( \lambda(t') = \text{receive} \) since \( \sigma_A \cong \sigma_l \circ \langle t' \rangle \circ \sigma'' \) and \( |\sigma_l| < |\sigma_A| \) which means \( \rho(t') \in \sigma_A \) has already fired. Hence \( \lambda(t') = \lambda(t) = \text{receive} \), since \( \bullet t \cap \bullet t' \neq \emptyset \). Now \( \sigma_A \) is a firing sequence in the server \( A \), it is possible that both \( \rho(t) \) and \( \rho(t') \) could have fired only if there is a loop starting at \( \bullet \rho(t) \cap \bullet \rho(t') \) and the server with firing sequence \( \sigma_A \) has taken this loop (i.e. a leg) at least once more than \( \sigma_l \). But this is not possible due to the leg property. Therefore \( t' = t \). It follows that \( \sigma_A = \sigma_l \circ \langle t \rangle \circ \sigma'' \).
Lemma B.2.1. Consider a sDTPN with the set of markings \( \mathbb{M} \). Let \( m, m' \in \mathbb{M} : m \xrightarrow{t} m' \). Then \( \exists \tilde{m} \in \mathbb{M}, \tilde{a} \in A(\alpha(m), t) : \alpha(m) \xrightarrow{t} \tilde{m} \) and \( ft(\tilde{m}) = ft(m') - ft(m) \) and \( \alpha(m') = \alpha(\tilde{m}) \).

Proof. For \( \alpha(m) = m \) the lemma holds trivially. Suppose \( \alpha(m) \neq m \).

- We will first prove \( \exists \tilde{m} \in \mathbb{M}, \tilde{a} \in A(\alpha(m), t) : \alpha(m) \xrightarrow{t} \tilde{m} \).

\[
ft(m) = \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \tau(a(i)) + \delta(\pi(a(i)), t) \}
\]

\[
0 = \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \tau(a(i)) - ft(m) + \delta(\pi(a(i)), t) \}
\]

By the Def. of \( \delta^+_i \) the term

\[
-\delta^+_i + \delta(\pi(a(i)), t) \leq 0
\]

Furthermore, for all \( t \in T \) and \( a \in A(m, t) \) the term

\[
\max_{i \in \text{dom}(a)} \{ \tau(a(i)) - ft(m) + \delta(\pi(a(i)), t) \} \geq 0
\]

Hence,

\[
0 = \min_{t \in T} \min_{a \in A(m, t)} \max_{i \in \text{dom}(a)} \{ \max\{ \tau(a(i)) - ft(m), -\delta^+_i \} + \delta(\pi(a(i)), t) \}
\]

By the Def. 4.3.8 and Cor. 4.3.7 we may write

\[
= \min_{t \in T} \min_{a \bar{a} \in A(\alpha(m), t)} \max_{i \in \text{dom}(\bar{a})} \{ \tau(\bar{a}(i)) + \delta(\bar{a}(i), t) \} = ft(\alpha(m))
\]
So there exists an activator \( \tilde{a} \in A(\alpha(m), t) : \text{dom}(\tilde{a}) = \text{dom}(a) \land \pi(a(i)) = \pi(\tilde{a}(i)) \land \tau(\tilde{a}(i)) = \max\{-\delta_i^1, \tau(a(i)) - \text{ft}(m)\} \) and a marking \( \tilde{m} \in M : \alpha(m) \overset{\tilde{a}}{\rightarrow} \tilde{m} \).

Consider the effect of firing transition \( t \) from markings \( m \) and \( \alpha(m) \).

Consumption: Since \( \text{dom}(a) = \text{dom}(\tilde{a}) \), the same tokens are consumed from marking \( m \) and \( \alpha(m) \).

Production: In marking \( m' \), tokens are produced in each \( p \in t^* \) with a timestamp \( x = \text{ft}(m) + \delta(t, p) \). In marking \( \tilde{m} \), tokens are produced in corresponding places but with a timestamp \( \delta(t, p) = \max\{x - \text{ft}(m), -\delta_i^1\} \). Note that the identities of the produced tokens can be chosen the same in both systems.

For the remaining tokens, \( \forall i \in \text{dom}(m) \setminus \text{dom}(a) : \tau(\tilde{m}(i)) = \max\{\tau(m'(i)) - \text{ft}(m), -\delta_i^1\} \), holds. Hence we have proven \( \forall i \in \text{dom}(\tilde{m}) : \tau(\tilde{m}(i)) = \max\{\tau(m'(i)) - \text{ft}(m), -\delta_i^1\} \).

- Next we will prove \( \text{ft}(\tilde{m}) = \text{ft}(m') - \text{ft}(m) \).

\[
\text{ft}(\tilde{m}) = \min_{t \in T} \min_{\tilde{a} \in A(\tilde{m}, t)} \max_{i \in \text{dom}(\tilde{a})} \left\{ \tau(\tilde{a}(i)) + \delta(\pi(\tilde{a}(i)), t) \right\}
\]

\[
= \min_{t \in T} \min_{a' \in A(m', t)} \max_{i \in \text{dom}(a')} \left\{ \max\{\tau(a'(i)) - \text{ft}(m), -\delta_i^1\} + \delta(\pi(a'(i)), t) \right\}
\]

\[
= \min_{t \in T} \min_{a' \in A(m', t)} \max_{i \in \text{dom}(a')} \left\{ \max\{\tau(a'(i)), \text{ft}(m) - \delta_i^1\} + \delta(\pi(a'(i)), t) \right\} - \text{ft}(m)
\]

\[
\max_{i \in \text{dom}(a')} \{\text{ft}(m) - \delta_i^1 + \delta(\pi(a'(i)), t)\} - \text{ft}(m)
\]

Since \( \delta_i^1 \geq \delta(\pi(a'(i)), t) \)

\[
\max_{i \in \text{dom}(a')} \text{ft}(m) - \delta_i^1 + \delta(\pi(a'(i)), t) \leq \text{ft}(m)
\]

Since \( \text{ft}(m') \geq \text{ft}(m) \), we have for all \( t \in T \) and \( a' \in A(m', t) \)

\[
\max_{i \in \text{dom}(a')} \tau(a'(i)) + \delta(\pi(a'(i)), t) \geq \text{ft}(m)
\]

Hence \( \text{ft}(\tilde{m}) = \text{ft}(m') - \text{ft}(m) \)

- Finally, we prove \( \alpha(\tilde{m}) = \alpha(m') \). By the Def. 4.3.8, \( \text{dom}(\alpha(\tilde{m})) = \text{dom}(\alpha(m')) \) and \( \forall i \in \text{dom}(\alpha(\tilde{m})) : \tau(\alpha(\tilde{m})(i)) = \max\{\tau(m'(i)) - \text{ft}(m'), -\delta_i^1\} \), and

\[
\forall i \in \text{dom}(\alpha(m')) : \tau(\alpha(m')(i)) = \max\{\tau(m'(i)) - \text{ft}(m'), -\delta_i^1\}
\]

Since \( \forall i \in \text{dom}(\tilde{m}) : \tau(\tilde{m}(i)) = \max\{\tau(m'(i)) - \text{ft}(m), -\delta_i^1\} \),
We may write
\[ \forall i \in \text{dom}(\alpha(\tilde{m})) : \tau(\alpha(\tilde{m})(i)) = \max\{\max\{\tau(m'(i)) - \text{ft}(m), -\delta_i^-\} - \text{ft}(\tilde{m}), -\delta_i^-\} \]

Since we know already that \( \text{ft}(\tilde{m}) = \text{ft}(m') - \text{ft}(m) \), we may write
\[ \forall i \in \text{dom}(\alpha(\tilde{m})) : \tau(\alpha(\tilde{m})(i)) = \max\{\max\{\tau(m'(i)) - \text{ft}(m'), -\delta_i^-\} - \text{ft}(m') + \text{ft}(m), -\delta_i^-\} \]

Since \( \text{ft}(m') \geq \text{ft}(m) \), we have \( -\delta_i^- - \text{ft}(m') + \text{ft}(m) \leq -\delta_i^- \). Hence \( \forall i \in \text{dom}(\alpha(\tilde{m})) : \tau(\alpha(\tilde{m})(i)) = \max\{\tau(m'(i)) - \text{ft}(m'), -\delta_i^-\} \). So we have proven \( \alpha(\tilde{m}) = \alpha(m') \).
In the following sections, we show how the computational aspects of components can be modeled as a control flow in the architectural model of ROSE using a few representative components.

C.1 The Platform Controller, Switch and Joystick Driver

First, note that both the platform controller and joystick driver components are root components, while the platform switch component is an RPC server. In the Fig. C.1, we show the Petri net model corresponding these components, as they occur in the architectural diagram. Such a component model describes fully its interactions with other components, but does not yet describe the logical structure of its functional behavior. We model this behavior as a set of state machine workflow nets having a clear start and a finish, i.e. as procedures. Once this behavior has been sufficiently defined, it is embedded into the component model using well known place/transition refinement techniques. Note that that such refinement techniques preserve the weak termination property.

C.1.1 Functional Design of the Joystick Driver

The joystick is a 3-DOF device. It produces periodic outputs corresponding the current position of the joystick, measured along its three axes. These outputs are read by the joystick driver and assigned to variables \( J_x, J_y, J_z \), where \( J_x \) is the measure of forward/backward movement, \( J_y \) is the measure of left/right
Figure C.1: The Platform Controller, Switch and Joystick Driver Components
movement and $J_z$ is the measure of the twist. We will call the assignment event as \textit{read joystick}.

The joystick driver is able to detect joystick movements by comparing the current reading with the previous reading. Whenever a joystick movement is detected, the current position of the joystick is transformed into a goal message for the platform controller component. We will call the detection event as \textit{detect movement} and the transformation event as \textit{transform goal}. The goal message has three attributes, namely linear velocity $V_x$, radius of turn $r$ and strafe velocity $V_y$, whose values are computed in the following way:

$$V_x = J_x, \quad r = -\log(J_y), \quad V_y = J_z$$

In the Fig. C.2, we show the Petri net model of Joystick Driver Component, refined by a state machine workflow net that maps events: read joystick, detect movement and transform goal to its corresponding transitions.

\subsection*{C.1.2 Functional Design of the Platform Switch}

The platform switch component has two modes of operation: joystick and navigation. In the former mode, the contents of the last received goal from the joystick driver is chosen. In the latter mode, the contents of the last received goal from the
planner component is chosen. In the Fig. C.3, we show how this can be modeled as a choice. Furthermore, the procedure of the RPC server provides a facility to toggle between these modes. In the joystick mode, a safety check must also be carried out before publishing the goal message to the platform controller. The safety check interprets the last received laser scan data to predict potential collisions. If a potential collision is detected then the joystick goal is overwritten with values that would cause the platform to stop as soon as possible. We call this computation as interpret scan safety and map it to the corresponding transition.

C.1.3 Functional Design of the Platform Controller

The platform controller subscribes to goal messages published by the platform switch component. The contents of this goal message are copied to local copies of variables $V_x, r, V_y$, corresponding linear velocity, radius of turn and strafe velocity, respectively. Depending on the values of these variables, there are two modes of operation: regular and strafe. The former is chosen if strafe velocity is non-zero, otherwise the regular mode is chosen. In both the modes, the desired wheel angles and their velocities must be computed using the valuation of these variables.

To describe these computations, we must understand the kinematics of the platform. The platform is a four wheel steer and drive base. A schematic of the platform is shown in the Fig. C.4. Starting from the front, $W_1, W_2, W_3, W_4$ represent the four wheels of the platform. Their angles are denoted by $\alpha_i$ and
their velocities by \( v_i \), where \( i \in \{1, \ldots, 4\} \). The radius of turn \( r \) and the velocity vectors \( V_x \) and \( V_y \) are specified relative to the center of the platform. Note that \( a = b \). For all our computations, we will use the notations presented in the figure.

In the regular mode, the following computations are carried out:

- **Compute wheel angles:**
  - \( \alpha_1 = \tan^{-1}(a/(r - t)) \),
  - \( \alpha_2 = \tan^{-1}(a/(r + t)) \),
  - \( \alpha_3 = \tan^{-1}(-b/(r + t)) \),
  - \( \alpha_4 = \tan^{-1}(-b/(r - t)) \).

- **Compute wheel velocities:** To compute the velocities \((V_1, V_2, V_3, V_4)\) of each wheel such that singularities do not appear, a distinction between a turn to the left \((r < 0)\) or a turn to the right \((r > 0)\) must be made.

  Note that as \( a = b \), we have that \( V_1 = V_4 \) and \( V_2 = V_3 \). Furthermore, by Pythagoras theorem, the turning radii of wheels 1 and 2 denoted by \( r_1 \) and \( r_2 \), respectively, can be computed in the following way: \( r_1 = \sqrt{(r - t)^2 + a^2} \) and \( r_2 = \sqrt{(r + t)^2 + a^2} \). Now the wheel velocities can be computed:

  - If \( r > 0 \) then we assign \( V_2 = V_3 = J_x \) and compute \( V_4 = V_1 = V_2 \times \frac{\alpha_1}{\alpha_2} \).
If $r < 0$ then we assign $V_1 = V_4 = J_x$ and compute $V_3 = V_2 = V_1 \times \frac{r_2}{r_1}$.

In the strafe mode, the following computations are carried out:

- **Compute wheel angles:** $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \tan^{-1}\left(\frac{J_y}{J_x}\right)$

- **Compute wheel velocities:** $V_1 = V_2 = V_3 = V_4 = \sqrt{J_y^2 + J_x^2}$

Once the wheel angles and velocities have been determined, they become one of the inputs for the eight control algorithms (four velocity and four steer). The other input is obtained from current wheel angles and velocities being published by the platform driver. From these inputs, each control algorithm generates a command, which is then published to the platform driver. For the control algorithm, we use the PID scheme and model it as a single transition. In the Fig. C.5, we show the elaboration of the platform controller component with a state machine workflow net describing the above computation logic.
C.2 The Platform Driver Component

In the Fig. C.6, we present the Petri net model of the platform driver, as it occurs in the architecture diagram.

The platform driver component interfaces directly with the 8-DOF platform device containing one steer motor and one drive motor, for each of its wheels. For each of these motors, an encoder (sensor) is attached to measure the state of each motor. The device periodically produces as its output the current angle and the number of accumulated rotations of each wheel. The platform driver reads these values and tries to estimate the current position and orientation of the platform relative to its start position and orientation. This estimation method is commonly referred to as **odometry**.

To sketch how odometry is computed, we first present a schematic in the Fig. C.7. In this figure, let $C$ and $C'$ represent the first and second poses of the platform (relative to its center) between two encoder readings. The radius of the turn relative to the center of platform is denoted by $r$. The distances covered by the inner and outer wheels of the platform are denoted by $d_1$ and $d_2$, respectively. Furthermore, the radius of turn of the inner and outer wheels of the platform are denoted by $r_1$ and $r_2$, respectively. Recall that the angles of the wheels are denoted by $\alpha_i$, where $i \in \{1, \ldots, 4\}$. The angles of the inner wheels $\alpha_1$ and $\alpha_3$ are equal and the angles of the outer wheels $\alpha_2$ and $\alpha_4$ are equal as well. For this reason, it is sufficient to consider wheel angles $\alpha_1$ and $\alpha_2$ for our computations.
So we have the initial pose \( C: (X, Y, \theta) \). From the encoder readings, we can deduce (a) the arc lengths \( d_1 \) and \( d_2 \), and (b) the two front wheel angles \( \alpha_1 \) and \( \alpha_2 \). Now we must compute the new pose estimate of the platform \( C': (X', Y', \theta') \). The computations are as follows.

First, we must determine the value of the radius of turn \( r \) and the change in orientation of the platform \( \phi \). From the Fig. C.7, we get the following relations:

\[
\begin{align*}
    d_1 &= \phi r_1 \\
    d_2 &= \phi r_2 \\
    r_2 &= \frac{r + t}{\cos(\alpha_2)} \\
    r_1 &= \frac{r - t}{\cos(\alpha_1)}
\end{align*}
\]

Rearranging and substituting we get \( d_2 \cdot \cos(\alpha_2) - d_1 \cdot \cos(\alpha_1) = \phi (r + t) - \phi (r - t) = 2 \phi t \). From this relation, we are able to compute \( \phi = \frac{d_2 \cdot \cos(\alpha_2) - d_1 \cdot \cos(\alpha_1)}{2t} \) and then \( r = r_2 \cdot \cos(\alpha_2) - t \). From this we can compute the distance covered by the center of the platform \( d \) as \( d = (d_1 + d_2)/2 \), since \( r = (r_1 + r_2)/2 \). We now have all the parameters required to derive the new configuration of the platform.
Figure C.8: The Platform Driver Component

$C' = (X', Y', \theta')$. We distinguish between two cases depending on the value of the radius $r$. If this value is infinity then it means the platform is moving in a straight line, otherwise it is changing its orientation.

- If $r \neq \infty$ then $X' = X_p + rcos(\phi + \theta - \pi/2)$, $Y' = Y_p + rsin(\phi + \theta - \pi/2)$ and $\theta' = \theta + \phi$.

- If $r = \infty$ (i.e. all wheel angles are equal and all wheels cover the same distance) then $X' = X + d_1cos(\theta - \beta)$, $Y' = Y + d_1sin(\theta - \beta)$, where $\beta = (\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4)/4$.

We map the reading of encoder values to transition labeled read device and odometry computations to the transition labeled compute odometry. Furthermore, the platform driver also subscribes to command messages being published by the platform controller. These messages contain control values that must be written to the platform. We map this to the transition write device. In the Fig. C.8, we show the elaboration of the Petri net model of the platform driver component by two state machine workflow nets describing the above computation logic.
C.3 The Planner Component

In the Fig. C.9, we present the Petri net model of the planner component, as it occurs in the architectural diagram. The planner component defines a navigation algorithm that is capable of generating a sequence of goals (linear and angular velocities) that command the platform to move from its current location to the desired destination while avoiding both static and dynamic obstacles along its path. The desired destination is specified by the goal arriving at the GFR server. The planner component is a GFR server having two RPC clients and one publisher. One RPC client is used to fetch a copy of the map from the map server and the other RPC client is used to look up the current location of the robot. The navigation algorithm is defined as a procedure that elaborates the busy place of the GFR server. The algorithm relies on its two planners: global and local.

The global planner is a grid based planner that uses the map to generate a global plan from the current location to the desired destination. The grid-based global planner implements the well known Dijkstra’s shortest path algorithm. Given the current location of the platform on a map and a desired destination to which the platform must navigate, the global planner generates a trajectory of 2D points starting from the current location to the desired destination such that no map obstacles are traversed. However, a grid based planner implies that the generated trajectory is not smooth and does not take into consideration the initial configuration and kinematic limitations of the platform.

The local planner uses the global plan and a copy of the costmap (generated by the costmap manager) overlayed on the map, to generate the next command for the platform. The local planner operates by sliding a window over the trajectory computed by a global planner and generates the next control command (containing linear and angular velocities) for the platform, while taking into consideration its current configuration. The local planner is able to react to dynamic obstacles in the environment by using the costmap to check for potential collisions. The check for potential collisions is carried out by forward simulating the current state of the platform in time, given its current state and a computed command. If a potential collision is detected for the generated command, then a set of recovery behaviors are carried out. If none of the recovery behaviors are able to find a good solution, then the planner aborts. If the platform is within a specified threshold of the desired goal, then the planner succeeds.

Note that an operator, receives feedback about the progress of the planner and may cancel the current goal at any time.

In the following sub-sections, we discuss and model the strategies underlying the navigation algorithm and its recovery behaviors.

Functional Design of the Planner Component. When the planner component becomes active, it first fetches the latest map from the map server (RPC). When this map is received, the navigation algorithm is triggered. The execution of this algorithm progresses periodically till either the goal has succeeded, aborted
or been canceled. The algorithm has three parts: Generation of Local Plan and Trajectories, Planner State Machine and Generation of Recovery Behaviors.

The first part of the algorithm generates a global plan (sequence of 2D points) and uses a sliding window technique over a subset of this plan to generate a local plan (linear and angular velocity). The second part of the algorithm tracks the current execution state of the algorithm and uses the generated local plan to compute a command for the platform driver. The third part of the algorithm checks the safety of the computed command and if necessary, attempts a hierarchy of recovery behaviors.

We model the three parts of the algorithm as three state machine workflow nets, WF1, WF2 and WF3 (see Fig. C.10). In the following sub-sections, we
explain their construction.

**Generating Global and Local Trajectories**

The workflow net WF1 models the logic underlying the generation of a global plan, the derivation of a local plan (subset of the global plan trajectory) and the generation of local trajectories.

First, the local copy of the costmap is updated with the latest copy maintained by the costmap manager. This is modeled by the transition \textit{update CM}.

Then a global plan (sequence of 2D waypoints) is generated by the global planner, if necessary. We model this behavior as a choice construct having two transitions, one \textit{computes the global plan} and the other gives the possibility to \textit{skip} this computation.

Depending on the current pose of the robot, the closest point on the global plan, that has not yet been passed, is identified. The strategy is to project the points of the global plan, on the arc made by the current heading of the platform. We model this identification step by the transition \textit{compute closest point}.

Next, a local plan is created as a new sequence containing the robot’s current pose followed by a subsequence of the global plan starting with the closest point, that was identified in the previous step. The length of the subsequence represents the width of the sliding window. In our model, we create two more local plans by varying the width of the sliding window. We model the creation of the local plan as the transition labeled \textit{update local plan from global plan}.

If the size of the smallest local plan is above some specified threshold, then we compute the radius of turn that the platform must now follow in order to track the waypoints in the local plan. More precisely stated, \textit{given the current pose of the robot and a local plan, compute the radius of the arc starting from the current pose of the robot such that the distance to all the points in the local plan is minimized?} We show how this can be computed in the Appendix A. In our model, we compute this for the three local plans, modeled as the transitions \textit{compute trajectory radius one/two/three}. If the size of the smallest local plan is equal to or below the specified threshold, the above computation is skipped.

**Planner State Machine**

The workflow net WF2 models the state-based behavior of the navigation algorithm. The navigation algorithm has six states: orienting, controlling, finishing, finalize orientation, done, recovery and fail.

The state \textit{orienting} is the initial state and the states done, recovery and fail are the final states. In the state orienting, the angle made by the current orientation of the platform relative to the first point of the local plan is computed. If this angle is not within a specified threshold then a control command specifying a rotation (zero linear velocity and non zero angular velocity) of the platform is
Figure C.10: Modeling the Functional Behavior of the Planner Component
generated. If the computed angle is within the specified threshold then the state is changed to controlling.

In the state controlling, a control command specifying a goal (linear and angular velocity) for the platform is computed for three circular trajectories (computed by considering subsets of the local plan). If the size of the local plan is below a specified threshold, then the state is changed to finishing.

In the state finishing, the last computed goal is held constant until the platform has arrived within some specified distance from the desired destination. Once within the specified distance, the state is changed to finalize orientation.

The state finalize orientation is similar to the state orienting, except that the current orientation of the platform is compared against the desired goal orientation. The control command specifying a rotation for the platform is held constant until the orientation of the robot is within some specified threshold and then the state is set to done.

The states fail and recovery are set outside the net WF2. There are two transitions that set the current state to fail. One of them fires whenever a global plan could not be found (WF1). The other fires when all recovery behaviors have failed (WF3). The state is set to recovery, when collision checks for the generated control command have failed.

**Generation of Recovery Behaviors**

The workflow net WF3 performs a weighted collision check (through forward simulation) for trajectories defined by the three radius of turns, computed in the previous step. The strategy is to use a costmap and a 2D projection of the robot called its footprint. The footprint is in essence a 2D polygon that is super-imposed onto a costmap, which is then used to compute a score of how close the robot is to objects in its immediate environment, using a scoring function. Furthermore, for each computed radius of turn, from the robots current configuration, discrete points on its local trajectory are generated for a fixed time duration (simulation time) and then checked for collisions. The best scoring collision free trajectory is chosen. If all the three trajectories are not collision free, then the state is set to recovery and recovery behaviors are attempted in some order. We define the three recovery behaviors in the following order:

- **Dynamic local window technique**: This strategy takes advantage of the fact that varying the size of a sliding window results in alternative local trajectories that can be followed. Hence, if the initial sliding window resulted in a trajectory that will result in a collision, we vary the window size until a local trajectory is found that does not result in a collision. If all attempts results in a collision, then we try the orientation technique.

- **Orientation technique**: The strategy is to check if a point-turn by the platform results in a trajectory that does not result in any collision. One way
to check is by computing the angle between the platform’s current pose and the first waypoint in the local plan.

- Strafing technique: The last attempt made by the recovery procedure takes advantage of the holonomic capabilities of the platform. A platform is said to be holonomic if it is able to change its position while maintaining a constant orientation. Such movements are commonly referred to as strafing and are in fact trajectories with infinite radius. The strategy is to check if there exists a collision free trajectory (corresponding non-zero $X$ and $Y$ components of the velocity vector and zero angular velocity) that is intersecting the first waypoint of the local plan.

If all the recovery behaviors do not lead to a safe trajectory then the state is set to failed and the workflow terminates.

We embed the three workflow nets into the planner component presented in the Fig. C.9, using the place refinement technique. The places labeled $Y$ and $Z$ in the Fig. C.9, are refined in the following way: First, the net WF1 refines the place labeled $Y$. Then the net WF2 refines the final place of WF1. Lastly, the refinement of place $Z$ by the net WF3.

### C.4 The Macro Component

In the section 4.3.4, we have seen how the orchestration net of the macro component was defined and indicated the places of this net that must be refined with clients of other interaction patterns. In the Fig. C.11, we present a snapshot of the orchestration net of the macro component after all the clients of RPC and GFR patterns have been added. The orchestration net along with clients of RPC and GFR patterns defines the behavior of the macro component, when a goal is accepted. Note that the orchestration net refines the busy place of the GFR server, so it is triggered by a token in its initial place one or more times. When a goal is accepted by the GFR server, it is first transformed into the coordinate frames of the two arms and platform. If this step fails, then only the server of the platform switch is invoked to request a change to manual mode and the server is aborted, otherwise, depending on the choice of use case, the mode of the platform switch component is set, accordingly. The macro component has five main states (manual/map, fail, navigation, arm-navigation, arm) and three sub-states (left, right, left-right). The order of invocation of the remaining clients of the GFR server (i.e. RPC Client-Mapping, RPC Client-Localization, GFR Client-Navigation, GFR Client Arm Manager(L/R)) depends upon the state of the macro component. If state is fail, then the orchestration net terminates. If state is manual or map, then the two RPC clients are invoked sequentially. If the state is navigation then the GFR client-Navigation is triggered. If the state is arm then depending on the sub-state either one or both the GFR client Arm Managers
Figure C.11: The Orchestration net of the Macro Component after Construction
Figure C.12: The Use Case Workflow: Macro Component
is/are triggered. If the state is arm-navigation then both GFR client-Navigation and one or both the GFR client Arm Managers are triggered.

However, to realize the eleven use cases, we require a procedure to compute the state of the macro component and construct goals for GFR clients. We model this procedure as a state machine workflow net (weakly terminating), as shown in the Fig. C.12, and refine the place labeled UC in the Fig. C.11, with this net. If the workflow net is triggered in a fail state, then the net terminates, otherwise, one of three defined procedures are triggered. The first procedure defines use cases 3 and 4. For the former, only the state is set to map, while for the latter a navigation goal is constructed and then the state is set to navigation. The second procedure defines use case 8 by distinguishing three cases. In the first case, only a navigation goal is constructed. In the second case, a goal for one of the arm managers is constructed. In the third case, a navigation goal and a goal for one of the arm managers is constructed. The third procedure defines all the remaining use cases by distinguishing three cases. In the first and second cases are similar to that of use case 8. In the third case, two goals, one for each arm manager is constructed.


[10] Parosh Aziz Abdulla and Aletta Nylén. Timed Petri nets and BQOs. In José Manuel Colom and Maciej Koutny, editors, Application and Theory of Petri


40] Béatrice Béard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier H Roux.


[72] Marc Boyer and François Vernadat. Language and bisimulation relations between


Didier Dubois and Kathryn E Stecke. Using petri nets to represent production pro-


[147] V. Graefe. Visual recognition of traffic situations by a robot car driver. pages


[225] M. Ajmone Marsan, Gianfranco Balbo, and Gianni Conte. Performance models of
multiprocessor systems. 1986.


[244] M.K. Molloy. On the integration of delay and throughput measures in distributed

[245] Luis Montano, Francisco José García, and José Luis Villarroel. Using the time
petri net formalism for specification, validation, and code generation in robot-
control applications. The International Journal of Robotics Research, 19(1):59–76,
2000.


[248] Gabriel A Moreno and Connie U Smith. Performance analysis of real-time com-
ponent architectures: An enhanced model interchange approach. Performance

[249] Tadao Murata. State equation, controllability, and maximal matchings of petri


[251] Tadao Murata and J. Koh. Reduction and expansion of live and safe marked

[252] Tomohiro Murata, Norihisa Komoda, Kuniaki Matsumoto, and Koichi Haruna. A
petri net-based controller for flexible and maintainable sequence control and its
applications in factory automation. Industrial Electronics, IEEE Transactions on,
1:1–8, 1986.

[253] Robin R. Murphy. Rescue robotics for homeland security. Commun. ACM,

intelligent real-time control architecture. IEEE Transactions on Systems, Man,

[255] Y. Narahari and N. Viswanadham. A petri net approach to the modelling and ana-
lysis of flexible manufacturing systems. Annals of operations research, 3(8):449–


[258] Issa A. D. Nesnas, Richard Volpe, Tara Estlin, Hari Das, Richard Petras, and
Darren Mutz. Toward developing reusable software components for robotic ap-
plications. In IEEE/RSJ International Conference on Intelligent Robots and

[259] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures


[278] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy


[326] W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory...*


ACKNOWLEDGEMENTS

The journey to this chapter of my thesis would not have been possible without the support and guidance of Kees van Hee and Henk Nijmeijer.

I was first a student of Kees during my Masters thesis on a software architectural design problem for Capgemini Nederland, seven years ago. This interesting exercise turned out to be very addictive for me. A few months into a career in the software development industry, I decided to write to Kees about the possibilities of returning to an academic environment. A fortunate turn of events ensued and I was once more a student of Kees, this time to pursue a PhD in the exciting area of formal software architectural methods for robotic systems. The five year journey since then has been one of the most challenging and at the same time, the most satisfying and enjoyable of experiences, having that nice mix of theory and practice. Many phases of this journey have seen me through steep curves with moments of difficult and slow progress. I am very indebted to Kees for being patient with me during these phases and critically mentoring my thought processes. Your enthusiasm has always been infectious and I have thoroughly enjoyed our discussions filled with new ideas, learning and discovery. I have learned a lot over an inspirational journey. It was a privilege to be your student.

Towards the end of my second year, I had the fortunate opportunity to carry out the last phases of my research in the lively and enthusiastic group of Henk Nijmeijer. I am very thankful to Henk for making my stay here, an invaluable experience in gaining a control systems perspective on the development of robotic systems. I have thoroughly enjoyed the intriguing presentations and discussions within your group as well as collaborations with colleagues from different research backgrounds under the H-Haptics project. Your valuable insights and comments about my work have been crucial in identifying and addressing interesting challenges as well as positioning the results in this thesis.

I am deeply indebted to Kees and Henk for having looked out for me through the many complex administrative hurdles in keeping me employed, always ensuring that I had enough room to complete my thesis in a comfortable manner.

The journey to this chapter also required the enthusiasm, support and cooper-
ACKNOWLEDGEMENTS

ation of many people along the way.

Some of the most notable and memorable experiences in the early years came from my participation in the TSR project. First, I am very grateful to Kees for sharing his vision and giving me the opportunity to work on the software development of service robot ROSE. It has been the most uniquely challenging and satisfying experiences in product development, with a profound impact on my future aspirations. None of this would of course have been possible without the support and cooperation of Michiel van Osch and Henk Zeegers. Thank you for the memorable experiences filled with celebration and endurance through the ups and downs of developing and demonstrating robot ROSE. It would be impossible to mention all the people I cooperated with, but I would like to thank everybody involved in this project.

My first two years at the Information Systems group of Wil van der Aalst and Kees van Hee was a memorable one, particularly the Rostock Petri net summer school experience! For valuable guidance with my work here, I would like to thank Jan Martijn van der Werf for sharing his insights and experiences about writing good scientific articles. It was a great pleasure and a nice learning experience for me during our collaborations. I would also like to thank Eric Verbeek, Christian Stahl and Dirk Fahland for their helpful comments about various chapters of my thesis, while publishing their contents as Computer Science Reports. For all the valuable assistance with CPN Tools through the years, I would like to thank Michael Westergaard.

I am grateful to STW for sponsoring my PhD position within the H-Haptics project. I would like to thank Frans van der Helm and David Abbink for their valuable feedback during the very interesting H-Haptics symposiums and autumn schools. I would also like to thank the user committee of the H-Haptics project, in particular Cock Heemskerk for sharing our enthusiasm and keen interest towards the goal of our research from the early days of the project. Your valuable inputs and suggestions have been very important in making the first steps towards the architecture of ITER RH. I would also like to thank Dennis Heck for the lively and interesting discussions we have had, while sharing the progress of our work and thinking about ways in which we could collaborate and present our results in a cohesive manner. In many ways this has led to a nice result in this thesis. For the very interesting cooperation during the H-Haptics project and particularly during the autumn school last year at TU Delft, I would also like to thank Jeroen Wildenbeest, Arvid Keemink and Femke van Beek.

I would also like to take this opportunity to make a special note of thanks to Tonny Wildvank, Jos Baeten and Manohara Pai. In many ways, the journey to this point was started over an interview on a rainy afternoon in Manipal, India.

I would definitely have been at a loss of direction without our secretaries Riet van Buul and Ine van der Ligt at the information systems group and Geertje Janssen–Dols at the dynamics and control group and Petra Aspers at the control systems technology group. Thank you so much for all the help through this journey!
I am thankful to my friends Manju, Mehaal, Ernest, Harsh, Ram, Shreya, Saba, Mac and Poojith for the memorable times we have shared and for giving me a feeling of home away from home.

I thank my parents and brother for their support and encouragement as well as for shielding me from distractions throughout this period. I also happened to meet my now loving wife during this journey. I am grateful to you for believing in my endeavours and standing by me through thick and thin.

Debjyoti Bera
Eindhoven, 2014
Debjyoti Bera was born on 25th March 1982 in Sarenga, India. After completing his higher secondary education in the year 2000, he started his engineering studies at Manipal Institute of Technology, India. In the year 2005, he obtained a bachelors degree in Computer Science and Engineering, after which he spent six months working as a software design engineer on defense project contracts at Sunlux Technologies Ltd., Bangalore, India. During this period, he also qualified the national exam for post graduate admissions in Computer Science, GATE 2006 (Graduate Aptitude Test in Engineering) with a score of 95 percentile.

In July 2006, he was admitted to the first batch of eight students for the dual degree masters program by Manipal Institute Technology, India and Technische Universiteit Eindhoven, The Netherlands. The two programs included an M.Tech degree in Software Engineering and an MSc. degree in Computer Science. The program was sponsored by three Dutch companies: Philips, CapGemini and Cordys. In August 2008, he graduated on the masters thesis An Adaptive Architecture for the Polis Administration System. This thesis proposed a software design strategy for the Polis Administration System at UWV, The Netherlands.

After his graduation, he worked for one year as a consultant and coach at Capgemini, Utrecht. Here he had the opportunity to work on a variety of challenging projects for organizations like Kadaster and Rijkswaterstaat, and conduct technical workshops for personnel at Belasting Dienst. In October 2009, he returned to the Information Systems group at Technische Universiteit Eindhoven for his PhD studies under the supervision of Prof. Dr. Kees van Hee. For the first two years, he worked on the two projects: Remote Robotics and Tele-Operated Service Robot (TSR). The former concerned the research and development of theoretical aspects of system design, while the latter concerned the design and development of a TSR named ROSE (Remotely Operated Service Robot). In 2012, he continued his PhD research at the Dynamics and Control group at Technische Universiteit Eindhoven under the supervision of Prof. Dr. Henk Nijmeijer and Prof. Dr. Kees van Hee. Here, he worked on the H-Haptics project (funded by STW), where existing research were applied to the nuclear fusion domain and design techniques
for analysis of timed systems were further developed. The research carried out in the period 2010-2014 led to a number of technical reports and publications in international journals, conferences and workshops.

His main research interests include Model Driven Engineering, Component-Based Software Architectural Frameworks, Modeling and Verification of Timed Systems, Petri nets and its (un-)timed Extensions, Timed Automata, Correctness by Construction, Model Transformation, Design and Analysis of Algorithms, Design and Development of Service Robots.