Master’s Thesis

Configurable *Declare*

by

Dennis Schunselaar

Supervisors

dr. Natalia Sidorova
dr. Fabrizio M. Maggi
dr. George Fletcher

Eindhoven, August 19, 2011
Abstract

In literature, there is an increasing interest in configurable process models [Got09]. Configurable process models are process models which can be (slightly) altered, for instance, by adding or removing tasks from the model. This allows companies to adjust the process models to their processes, instead of adjusting their processes to the process models.

Unfortunately, the existing configurable process models are all based on imperative languages, which makes them rigid. Everything which is not stated to be allowed is forbidden. When we move to environments with high variability, e.g. a hospital, one has to choose between maintainability of the system, versus the possibility to cope with the variability. We can best compare the consideration between maintainability and variability with a road network. It is easier and cheaper to maintain a small road network instead of a large road network (maintainability). However, a large road network is more flexible and it is able to cope with, for instance, traffic jams, road reconstruction, disasters, etc. (variability).

Since imperative languages are very rigid, i.e. you can only drive on the roads, we move to declarative languages, and Declare in particular. Within Declare everything is allowed unless stated otherwise, we can best compare Declare with the sea where we are allowed to sail everywhere except where the beacons disallow. This removes the rigidity from process models. However, companies still want to be able to adjust the models to their processes. More specifically, they want to have flexibility at configuration-time, i.e. between design-time and run-time, whilst maintaining the flexibility at run-time. Therefore, we develop Configurable Declare, a true extension of Declare.

Configurable Declare allows the removal of tasks and constraints from models as well as the substitution of a constraint by a different constraint. We define rules to maintain implicit constraints, i.e. constraints that are not explicitly encoded in the model but deducible from other constraints, e.g. if task Opening the patient for surgery is always followed by Inserting pacemaker and Inserting pacemaker is always followed by Closing the patient after surgery then implicitly task Opening the patient for surgery will always be followed by Closing the patient after surgery. Using these rules, we are able to remove activities from the model whilst maintaining the implicit constraints. Furthermore, we use these rules to combine a set of Declare models into a single minimal Configurable Declare model, i.e. we use the rules to remove constraints already encoded implicitly. After that, we define configurations which, when applied to the Configurable Declare model, yield the original set of Declare models.

We have implemented the combining of the different Declare models into a single Configurable Declare model, as well as transforming a configured Configurable Declare model into a Declare model. Furthermore, we show with a case study that we can apply our approach on real life models.

Keywords: Declare, Configurable Declare, Configurable Models, Configuration, Patterns
Contents

1 Introduction ........................................... 6

2 Preliminaries ......................................... 8
   2.1 Process flexibility .................................... 8
   2.2 Declare .............................................. 8
   2.3 Linear Temporal Logic ................................. 14

3 Known Configuration Patterns ......................... 15

4 Configurable Declare ................................... 19
   4.1 Patterns supported in Declare ......................... 19
   4.2 Extending Declare to Configurable Declare ............ 21
   4.3 Configuring Configurable Declare ..................... 21
   4.4 Patterns supported in Configurable Declare ........... 22
   4.5 Running example ..................................... 35

5 Combining models ...................................... 40
   5.1 Combining the models ................................ 40
   5.2 Removing redundant constraints ...................... 41
   5.3 Running example .................................... 42

6 Implementation ........................................ 46
   6.1 Configurable Declare to Declare....................... 46
   6.2 Combining Declare models ............................. 47
   6.3 Rule engine ........................................... 50

7 Case Study ............................................ 54
   7.1 Combining the models ................................ 54
   7.2 Configuring the models ............................... 54
   7.3 Running example .................................... 55
   7.4 Comparing the models ................................. 56

8 Future work ........................................... 58

9 Conclusion ............................................. 64

Appendices .............................................. 67

A Meta-models ........................................... 67

B Input models .......................................... 69

C Configuration models .................................. 72
1 Introduction

Process models are everywhere and each branch of industry uses its own set of process models. However, within the same branch, the process models are often very similar due to legislation and standardisation, e.g. registering a birth or extending a driving license. The maintainability of this set of (very similar) process models is very low. For instance, when legislation changes, all process models have to be updated to support the new legislation. Furthermore, a one-size-fits-all approach is usually incorporated, i.e. although different companies use slightly different processes, they are all forced to use the same process model.

Within the industry, [Got09] (amongst others) partly solves the aforementioned problems by introducing configurable process models. Such process models allow the user to change or remove parts of the model so that the model adheres more closely to the user’s processes. This solves the one-size-fits-all problem and improves maintainability because it is possible to describe several slightly different models by a single configurable model. When the configurable model is changed, all process models will be updated automatically. Examples of configurable process modelling languages are: C-SAP WebFlow, C-BPEL, and C-YAWL [Got09].

However, the aforementioned configurable process modelling languages are all imperative and thus inherently rigid, i.e. if something is not specified as allowed, it is forbidden. If such languages would be used in an environment with high variability, e.g. a hospital, one has to choose between the maintainability of the system versus the possibility to cope with the high variability. Imperative models become very complex when they have to cope with variability, i.e. every possible execution path has to be encoded in the model, and complex models are harder to maintain than simple models. For instance, for a hospital one has to choose between the maintainability of the system versus the freedom of the medical staff to perform their tasks.

To address this problem, we use a declarative approach and the declarative language Declare. Within Declare [PvdA06, PSvdA07], everything is allowed unless stated otherwise (differently from imperative languages in which everything is forbidden unless stated otherwise). For instance, when applied to a hospital, we only need to prohibit executions which are against legislation or hospital policies and Declare allows the medical staff to perform their task the way they want.

Declare partly solves our problems, since it removes the rigidity from the models, but is does not allow us to solve the one-size-fits-all problem neither present a solution to increase maintainability. To solve the one-size-fits-all and maintainability problems, we create Configurable Declare, a true extension of Declare. Configurable Declare is new and it has not been defined earlier, we consider this thesis as a starting point for Configurable Declare. Due to time limitations, we were not able to address all issues identified in this thesis. However, in the future work section (Section 8), we show that most issues (which still exist) can be easily addressed.

Imperative configurable process models have to support a number of operations, e.g. the removal of a task from the process model. Some patterns have been identified to support these operations [DRvdA+06, SLS10, BDKK04, Got09]. We want that Configurable Declare supports the identified patterns which entail the model itself, i.e. which reason about the control-flow instead
of, for instance, the different views on the model.

Our goal in this thesis is to define Configurable Declare so that it supports similar configuration possibilities as in imperative configurable process models. Furthermore, it should support the patterns identified in literature w.r.t. the control-flow. Configurable Declare should maintain the implicit constraints after the removal of a task. Finally, Configurable Declare should be applicable to real life models.

Almost every Declare model has implicit constraints. Implicit constraints are constraints which are derived from other constraints, e.g. if task Opening the patient for surgery is always followed by Inserting pacemaker and Inserting pacemaker is always followed by Closing the patient after surgery then implicitly task Opening the patient for surgery will always be followed by Closing the patient after surgery.

We present a conversion from a Configurable Declare model and a configuration to a Declare model whilst maintaining the aforementioned implicit constraints. These implicit constraints are encoded as a set of rules where, in case there is a constraint between A and B, and between B and C, we specify the implicit constraint between A and C. Each of these rules reason on a semantic level w.r.t. the constraints.

We also present a conversion from a set of Declare models into a single Configurable Declare model. This allows organisations to use Configurable Declare on their current set of Declare models without the need to create the Configurable Declare model from scratch. This Configurable Declare model can be manually configured to obtain the original set of Declare models. We use the aforementioned rules to minimise the Configurable Declare model by removing constraints deducible from other constraints present in the Configurable Declare model.

We present an implementation of the conversion in both directions. We will elaborate on the encoding of the application of the rules. We use this implementation for a case study involving a set of Declare models describing business processes of different Dutch municipalities. These models represent the production of an excerpt from the civil registration and all models are combined into a single Configurable Declare model. On this Configurable Declare model, we manually define configurations that, when applied to the Configurable Declare model, yield the set of input models. This case study represents a real application of the proposed approach.

This document is structured as follows: Section 2 gives an extensive explanation of Declare. In Section 3, an overview of related work with respect to configurable models is given as well as the configuration patterns found in literature. Configurable Declare is explained in Section 4 along with a further explanation of the configuration patterns listed in Section 3. In Section 5, we explain how to combine a set of Declare models into a single Configurable Declare model. The implementation of the conversions is described in Section 6. Section 7 presents the case study where we apply our approach. Finally, future work and the conclusion are presented in Section 8 and 9 respectively.
2 Preliminaries

In this section, we explain the notion of process flexibility, Declare, and Linear Temporal Logic. All three notions are used throughout this thesis in different sections.

2.1 Process flexibility

In [Sof05], a number of definitions are given for flexibility. The main idea of these definitions is: flexibility is the ability to react to (unexpected) changes. A number of patterns associated with flexibility can be found in [MvdAR]. One of the patterns mentioned in [MvdAR] is the Flexibility by design pattern in which the flexibility is encoded in the process model, i.e. instead of allowing only one path through the model, multiple paths have been defined through the model to offer some flexibility.

We use two types of flexibility: (1) flexibility of the process, i.e. the amount of paths through the model, and (2) flexibility of the process model, i.e. the amount of paths which can be altered or added to the model.

The amount of paths through a model is closely related to the notion of an imperative modelling language. In an imperative modelling language, everything has to be modelled which is allowed. When a certain order of activities is allowed, we have to model this explicitly. To allow flexibility by design, one has to encode this in the model using constructs like parallelism, choice, iteration, etc. This means that the model becomes less structured and harder to maintain.

When we move to a declarative approach, we only specify what it is allowed instead of what is allowed, the opposite of imperative modelling. This yields greater flexibility without the need to add extra information to the model. Within declarative models, we have in general that: the smaller the model the more flexible it becomes. The Flexibility by design is automatically supported by declarative languages because any path is allowed through the model unless stated otherwise.

2.2 Declare

In order to understand Configurable Declare, we first explain Declare. Declare is a constraint-based language proposed in [PvdA06, PsvdA07]. If certain events in Declare are not explicitly allowed nor explicitly forbidden, these events are allowed to take place. Declare consists of a graphical front-end and a formal back-end, expressed in Linear Temporal Logic [Pnu77]. We only focus on the graphical front-end, i.e. we take the different constraints of Declare as regular constructs like the AND-split in process management systems.

We choose to focus on the graphical front-end, since this makes the reasoning better understandable. Most results obtained in this thesis can also be deduced from the formal back-end. However, in some case, we want to rewrite the formula into a non-equivalent formula which captures the semantic meaning. We elaborate further on these cases in the future work section (Section 8).

Within Declare, there are four sets of templates: existence, relation, negative relation, and choice. We go over each of those four sets of templates and present the informal definition and the graphical representation. We, furthermore, give the order of the different templates within a set of templates. This order denotes

8
which template is stronger than the other template, i.e. whenever a template is valid also another template is valid, the first template is considered to be stronger.

Apart from the term template, we also use the term constraint. Constraints are instances of templates, i.e. templates are the different constructs of the language while constraints are the application of the constructs on the events.

**Meta-model**

Figure 1 depicts the meta-model of Declare which is used in this thesis. Here, a Model class consists of a set of Events and Constraints, and a Constraint is associated with at least one Event. The choice constraints have a \( m \) and a \( n \) associated with them to denote that \( m \) out of the \( n \) connected events are performed, and the existence constraints have a \( n \) associated with them to denote the bound.

![Figure 1: Meta-model of Declare](image)

**Existence templates**

The existence templates in Declare are listed in Table 1. In Figure 2, the order between the different existence templates is depicted. We do not have a stronger/weaker relation between the different exactly templates because they are incomparable. The exactly template is the conjunction of the existence and absence, and when the existence template is stronger, the absence template is weaker and vice versa.
**Template name** | **Informal definition** | **Graphical representation**
--- | --- | ---
existence\((n,A)\) | Event \(A\) has to happen at least \(n\) times | ![Diagram](image1.png)
absence\((n+1,A)\) | Event \(A\) can happen at most \(n\) times | ![Diagram](image2.png)
exactly\((n,A)\) | Both existence\((n,A)\) and absence\((n+1,A)\) have to hold | ![Diagram](image3.png)
init\((A)\) | Event \(A\) has to be the first event which is executed | ![Diagram](image4.png)

Table 1: The existence templates

![Diagram](image5.png)

Figure 2: The order between the existence templates in Declare

**Relation templates**

The relation templates are listed in Table 2. In Figure 3, the ordering between the different constraints is shown. Note that co-existence\((A,B)\) is equivalent to co-existence\((B,A)\), therefore, succession\((A,B)\) is stronger than co-existence\((A,B)\) and co-existence\((B,A)\). We have a similar relation between co-existence and responded existence, i.e. co-existence\((A,B)\) is stronger than responded existence\((A,B)\) and responded existence\((B,A)\).
<table>
<thead>
<tr>
<th>Template name</th>
<th>Informal definition</th>
<th>Graphical</th>
</tr>
</thead>
<tbody>
<tr>
<td>responded existence (A, B)</td>
<td>If event A happens event B should also happen</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>co-existence (A, B)</td>
<td>Both responded existence (A, B) and responded existence (B, A) have to hold</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>response (A, B)</td>
<td>If event A happens event B should eventually follow</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>precedence (A, B)</td>
<td>Event B has to be preceded by event A</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>succession (A, B)</td>
<td>Both response (A, B) and precedence (A, B) have to hold</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>alternate response (A, B)</td>
<td>Event A should be followed by event B and we cannot have an A in between</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>alternate precedence (A, B)</td>
<td>Event B has to be preceded by event A and we cannot have a B in between</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>alternate succession (A, B)</td>
<td>Both alternate response (A, B) and alternate precedence (A, B) have to hold</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>chain response (A, B)</td>
<td>Event A should be directly followed by event B</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>chain precedence (A, B)</td>
<td>Event B should be directly preceded by event A</td>
<td>A –&gt; B</td>
</tr>
<tr>
<td>chain succession (A, B)</td>
<td>Both chain response (A, B) and chain precedence (A, B) have to hold</td>
<td>A –&gt; B</td>
</tr>
</tbody>
</table>

Table 2: The relation templates

Figure 3: The order between the relation templates in Declare
Negative relation templates

The different negative relation templates are listed in Table 3. Furthermore, the ordering between the negative relation templates is depicted in Figure 4.

<table>
<thead>
<tr>
<th>Template name</th>
<th>Informal definition</th>
<th>Graphical</th>
</tr>
</thead>
<tbody>
<tr>
<td>not co-existence ((A,B))</td>
<td>Event (A) and event (B) do not occur together in a trace</td>
<td><img src="https://example.com/diagram1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>not succession ((A,B))</td>
<td>Event (B) does not follow event (A)</td>
<td><img src="https://example.com/diagram2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>not chain succession ((A,B))</td>
<td>Event (B) does not follow event (A) directly</td>
<td><img src="https://example.com/diagram3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 3: The negative relation templates

![Diagram](https://example.com/diagram4.png)

Figure 4: The order between the negative relation templates in Declare

Choice templates

In Table 4, the different choice templates are listed. The ordering between the templates is shown in Figure 5. When we have to execute \(n\) of \(n\) events, there is an equivalence between choice and exclusive choice, i.e. choice \((n\) of \(n\)) is equivalent to exclusive choice \((n\) of \(n\)), because we have to execute all events. We do not have a stronger/weaker relation between the different exclusive choice templates because the different templates exclude each other. When exclusive choice \((m\) of \(n\)) is valid, exactly \(m\) events have to be executed, which means that we do not execute exactly \(m - 1\) or exactly \(m + 1\) events.
Template name | Informal definition | Graphical
---|---|---
choice \( (m \text{ of } n) \) at least \( m \) out of the \( n \) events have to occur | ![Graphical representation](image)
exclusive choice \( (m \text{ of } n) \) exactly \( m \) out of the \( n \) events have to occur | ![Graphical representation](image)

Table 4: The choice templates

![Figure 5: The order between the choice templates in Declare](image)

**Branched Declare**

For each of the aforementioned templates, except the choice templates, we also have a branched variant in Branched Declare. A branched variant means that, instead of a single event, we have a multitude of events. Consider, for instance, the response template, in Branched Declare, we have instead of \( \text{response}(A, B) \) two sets of events: \( \text{response}([A_1, \ldots, A_n], [B_1, \ldots, B_m]) \) denoting that if either event \( A_1 \) or \( A_2 \) etc. happens, eventually event \( B_1 \) or \( B_2 \) etc. should happen. In the tables for the different templates, one needs to read \( A_1 \) or \( A_2 \) etc. instead of \( A \) and \( B_1 \) or \( B_2 \) etc. instead of \( B \).

Apart from a different semantic, Branched Declare also has a different graphical representation. Instead of a single arrow between two events, we now have an arrow with multiple heads and tails between the different events. In Figure 6, an example of a branched response constraint is shown.
2.3 Linear Temporal Logic

Linear Temporal Logic [Pnu77], or LTL for short, is a logic which reasons about paths through some model, e.g. an automaton. For instance, if we have a set of traces, we can view each trace as a path through the process model that generated those traces. Using LTL, we can validate whether the traces, and to some extent also the process model, adheres to certain constraints on the paths through the process model. For instance, if we never want to see a \( B \) directly following an \( A \), we can express this with: \( \square (A \Rightarrow \neg \diamond (B)) \). Using a model checker, we can validate this formula on our set of traces.

It goes beyond the scope of this thesis to give a full explanation of LTL. We would like to refer the interested reader to [Pnu77] for a more extensive explanation on LTL. We give a formal definition of the different constructs of LTL. In Table 5, the main LTL constructs are listed which are used in this thesis.

We use the following variables: let \( A \) be the set of actions, \( A = \{b, c, d, e, \cdots \} \), \( w \) a sequence of actions, i.e. \( w = a_1, a_2, a_3, \cdots \) and let \( w_i \) denote the \( i \)th action in the sequence, i.e. \( w_i = a_i \). With \( w^j \), we denote the suffix of \( w \), i.e. \( w^j = a_j, a_j+1, \cdots \). The satisfaction relation \( \models \) between a sequence of actions and a LTL formula is now defined for the different operators of LTL in Table 5, note that \( p \in A \) is a single action, where \( \psi \) and \( \phi \) are arbitrary formulas.

<table>
<thead>
<tr>
<th>LTL construct</th>
<th>Formal meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w \models p )</td>
<td>if ( p \in w_1 )</td>
</tr>
<tr>
<td>( w \models \neg \psi )</td>
<td>if ( w \not\models \psi )</td>
</tr>
<tr>
<td>( w \models \psi \lor \phi )</td>
<td>if ( w \models \psi ) or ( w \models \phi )</td>
</tr>
<tr>
<td>( w \models \psi \land \phi )</td>
<td>if ( w \models \psi ) and ( w \models \phi )</td>
</tr>
<tr>
<td>( w \models \psi \Rightarrow \phi )</td>
<td>if ( w \models \neg \psi \lor \phi )</td>
</tr>
<tr>
<td>( w \models \psi \Leftrightarrow \phi )</td>
<td>if ( w \models \psi \Rightarrow \phi ) and ( w \models \phi \Rightarrow \psi )</td>
</tr>
<tr>
<td>( w \models \text{true} )</td>
<td>if ( w \models p \lor \neg p )</td>
</tr>
</tbody>
</table>
| \( w \models 
\odot \psi \) | if \( w^2 \models \psi \) |
| \( w \models \psi \U \phi \) | if there exists \( i > 0 \) such that \( w^i \models \phi \) and for all \( 0 < k < i \), \( w^k \models \psi \) |
| \( w \models \diamond (\psi) \) | \( w \models \text{true} \U \psi \) |
| \( w \models \square (\psi) \) | \( w \models \neg \diamond (\neg \psi) \) |
| \( w \models \psi \W \phi \) | \( w \models (\psi \U \phi) \lor \square (\psi) \) |

Table 5: LTL constructs
3 Known Configuration Patterns

Some configuration patterns have been identified in literature. We list the papers from which we have used configuration patterns as a starting point for Configurable Declare. Each paper we consider is accompanied by a small introduction and a table listing the different patterns considered in those papers.

As mentioned in the introduction, we want that Configurable Declare supports a number of patterns. These patterns, or configuration patterns, are a collection of constructs or operations which have to be supported by the process model, i.e. they can be used in the process model. We have two types of configuration patterns: patterns related to the views on the process model, and patterns related to the process model itself. Note that we use the terms workflow and process interchangeable to stay consistent with the papers.

In [DRvdA+06] (Table 6), a process model is designed but it can later on be (slightly) altered at build-time, i.e. certain parts of the model can be substituted or removed. E.g. the user has the option to change the relation between different events, so instead of both events occurring the user can choose that exactly one of the events occurs. Similarly, in Configurable Declare, we want that after the creation of the configurable process model the user has the option to (slightly) alter parts of the model.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Effects</th>
<th>Will be considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optionality (Sect. 4.4)</td>
<td>Choose to execute a task or not (build-time)</td>
<td>Yes, we want to have the flexibility to remove events from the configurable model prior to executing the model</td>
</tr>
<tr>
<td>Inter-relationship</td>
<td>Execution of a task depends on the execution of another task</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. co-existence or not co-existence</td>
</tr>
<tr>
<td>Interleaved parallel routing (Sect. 4.1)</td>
<td>Choose in which order a set of tasks is executed</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. precedence</td>
</tr>
<tr>
<td>Parallel split (OR) (Sect. 4.1)</td>
<td>Choose a subset of the tasks</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. choice in combination with the optionality pattern</td>
</tr>
</tbody>
</table>

Table 6 continues on the next page
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Effects</th>
<th>Will be considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive choice</td>
<td>Choose exclusively a task</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. exclusive choice</td>
</tr>
<tr>
<td>(Sect. 4.1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi choice</td>
<td>XOR, OR, and AND split</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. choice and exclusive choice in combination with the optionality pattern</td>
</tr>
<tr>
<td>(Sect. 4.1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Configuration patterns [DRvdA+06]

[Got09] (Table 7) gives an overview of adding configurations to different workflow management systems, and workflow languages. [Got09], furthermore, gives an approach to configure a process model by hiding or blocking inflow and outflow ports, i.e. the incoming and outgoing arcs. These patterns are similar to some patterns in [DRvdA+06].

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Effects</th>
<th>Will be considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>(optional) Blocking</td>
<td>Choose to block (not execute) or not to block (execute) an event</td>
<td>No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. absence in combination with the optionality pattern</td>
</tr>
<tr>
<td>(optional) Hiding</td>
<td>Choose to hide or not to hide an event</td>
<td>Yes, similar to the optionality pattern</td>
</tr>
</tbody>
</table>

Table 7: Configuration patterns [Got09]

In [SLS10] (Table 8), the authors assume a static model in which different users want different views on a workflow model, e.g. a sales manager needs different information than a programmer. One has to be able to support a number of actions with respect to different views, e.g. within the abstraction patterns one can choose from which parts of the model to abstract in a certain view. We only support the patterns by [SLS10] which can be transformed in patterns for the process model itself, e.g. abstraction, since Declare reasons about process models and not about the views on the process model.
Pattern Name  | Effects                                      | Will be considered |
-------------|---------------------------------------------|--------------------|
Omission     | Same as *optionality* pattern               | Yes, we want to have the flexibility to remove events from the configurable model |
Abstraction  | Do not state explicitly what is happening, i.e. we know something is happening but we do not know exactly what | No, since it is part of the view on the model and thus outside the scope of this thesis |
Insertion    | Insert an event                             | No, gives room for too diverse models which are harder to maintain, and a designer of a configurable model can already include a multitude of events to suit the processes of the different users in combination with the *optionality* pattern |
Aggregation  | Combine events into one event               | No, we will not consider this as a pattern since the language constructs already allow such behaviour when specified. E.g. *chain succession* |
Alteration   | Change properties of the model             | Yes, but only the constraints between the events, and only based on rules specified by the designer of the configurable model |
Preservation | States whether a pattern can be applied     | Yes, implicit in the model, i.e. which parts can be changed |
Presentation Patterns | Change the representation of constructs     | No, outside the scope of the thesis |

Table 8: Configuration patterns [SLS10]

Within [BDKK04] (Table 9), a number of patterns are described which allow the user to change her view on the process model. These patterns entail that the user can select sets of “similar” process elements and hide these, each pattern defines a different kind of similarity between process elements. E.g. the user can select all objects which have a certain value for a certain attribute. We only support a slightly altered *Relationship type selection* pattern.

Pattern Name  | Effects                                      | Will be considered |
-------------|---------------------------------------------|--------------------|
Object Type Selection | Selecting all events of a particular type | No, can be done manually |

Table 9 continues on the next page
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Effects</th>
<th>Will be considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>Change the representation of constructs</td>
<td>No, outside the scope of the thesis</td>
</tr>
<tr>
<td>Variations</td>
<td></td>
<td>No, can be done manually</td>
</tr>
<tr>
<td>Attribute-based</td>
<td>Selecting or hiding events with a certain value</td>
<td>No, can be done manually</td>
</tr>
<tr>
<td>object selection</td>
<td>for a certain attribute</td>
<td></td>
</tr>
<tr>
<td>Term-based object</td>
<td>Selecting or hiding events based on an</td>
<td>No, can be done manually</td>
</tr>
<tr>
<td>selection</td>
<td>expression with respect to some identifiers</td>
<td></td>
</tr>
<tr>
<td>Relationship type</td>
<td>Choose the type of the relationship which one</td>
<td>Yes, can be done by switching constraints on or off</td>
</tr>
<tr>
<td>selection (Sect. 4.4)</td>
<td>wants to hide or wants to show</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Configuration patterns [BDKK04]

There are no configuration patterns for *Declare* or for *Configurable Declare*, therefore, we use a number of patterns mentioned in the papers listed here as a starting point for patterns which need to be supported by *Configurable Declare*. Note that all patterns listed in these papers reason about process models specified in an imperative language. Therefore, we cannot apply the patterns directly to *Configurable Declare*. Within each table, we have presented some intuition on how to support the different patterns if they can already be encoded in *Declare*. Each pattern, which we want to support or is already supported, is considered in more detail in the next section.
4 Configurable Declare

Configurable process models have been defined for some of the imperative modelling languages. However, as mentioned in the introduction, we do not yet have configurable process models for declarative languages. Therefore, we define Configurable Declare which is explained in this section.

Prior to defining Configurable Declare, we list the patterns identified in Section 3 which are already supported by Declare (Section 4.1). In general, these patterns are supported by the flexibility which a Declare model already offers. For example, with the inter-relationship pattern, we want to introduce correlation between the execution of activities A and B which is supported by either a co-existence or a not co-existence between A and B.

After defining the supported patterns, we extend Declare to Configurable Declare. We, furthermore, list the different types of configuration patterns we allow. Then, we list how to configure a Configurable Declare model. Finally, we list how we support the patterns identified in Section 3 but which are not supported by Declare.

4.1 Patterns supported in Declare

The following patterns are already present as a construct in Declare. We list how the different patterns can be supported using the constructs.

Inter-relationship

Inter-relationship means that the execution of a task A depends on the execution of a task B and vice versa. According to [DRvdA+06], we have two types of inter-relationship, mutually dependent and mutually exclusive. With mutually dependent, we have that either A and B occur together or they do not occur at all. This can be modelled with a co-existence between A and B. When we have a mutually exclusive relation between A and B, this means that A and B cannot occur together. We can model this by having a not co-existence between A and B.

Interleaved parallel routing

We want to choose in which order a set of tasks is executed with the interleaved parallel routing pattern. Let the set of tasks be the tasks $A_1, \ldots, A_n$. By using the precedence constraint between a number of events, the user can choose which events should occur before which other events by simply switching the precedence constraint on or off. If we want to support all possible orders of events, we need to add the precedence constraint between all pairs of events. Assume we want that the tasks are execute based on their indices, i.e. $A_i$ is executed before $A_{i+1}$, then we add a precedence constraint between two consecutive tasks. This precedence constraint enforces that we can only execute the tasks in a certain order.

The aforementioned approach only works the first time the tasks are executed, i.e. after $A_1$ has been executed we can do any number of $A_{i+1}$ independent of $A_i$. When we strengthen the precedence constraints, i.e. substitute them by alternate precedence constraints, we can enforce that the order of execution is
always maintained after the first execution of the events. With the \textit{alternate precedence} constraint, \(A_{i+1}\) cannot happen twice in a row without the execution of \(A_i\).

When we want to strengthen the previous model even further, by stating that the first task cannot execute another time before the last task has been executed, we have to add \textit{alternate succession} between all pairs of tasks. This means we have \textit{alternate succession}\((A_i, A_j)\) for all pairs \(i\) and \(j\) for which \(i < j\).

**Parallel split**

Within this pattern, we want to choose a subset of the set of tasks consisting of \(A_1, \cdots, A_n\). We can construct this pattern by using the \textit{choice} \((1 \text{ of } n)\) constraint between the tasks, this constraint states that we need to do at least one task and we can do at most \(n\) tasks, i.e. all of them. If we combine this constraint with the \textit{optionality} pattern for greater flexibility, i.e. in some cases one does not want to consider some tasks, we can express the parallel split pattern as described in [DRvdA+06].

**Exclusive choice**

Within \textit{Declare}, we have the \textit{exclusive choice} \((1 \text{ of } n)\) which expresses that we want to have exclusively 1 task out of the \(n\) tasks. Using the \textit{optionality} pattern in a similar way as for the \textit{parallel split}, we can support the \textit{exclusive choice} pattern.

**Multi choice**

The \textit{multi choice} can be translated in a \textit{parallel split}, an \textit{exclusive choice} and an \textit{and-split}. Both the \textit{parallel split} and the \textit{exclusive choice} can be made in the same way as described before. The \textit{and-split} can be made by using the \textit{choice} \((n \text{ of } n)\) constraint which states that all \(n\) tasks should be executed. Again, we can use the \textit{optionality} pattern to remove some tasks.

**Aggregation**

With the \textit{aggregation} pattern, one wants to combine two events into a single event. Assume these two events are event \(A\) and event \(B\), we can aggregate them by placing a \textit{chain succession} between \(A\) and \(B\), i.e. every \(A\) is directly followed by a \(B\) and every \(B\) is directly preceded by an \(A\). Using this construction, one can see \(A\) and \(B\) as a single event, i.e. no other event can occur between \(A\) and \(B\).

**Preservation**

The \textit{preservation} pattern denotes which patterns can be applied on a certain part of the model. Within \textit{Configurable Declare}, each event and constraint is augmented with a boolean stating whether it can be removed, and each constraint has a list associated with it denoting in which other constraints it can be altered. In this sense, the \textit{preservation} pattern is already encoded in the model.
(optional) Blocking
Optional blocking gives the user the option to prevent the execution of a particular branch in an imperative model. When we block an activity in Declare, i.e. prevent it from executing, we only block activities dependant on the execution of this blocked activity, e.g. via response or precedence constraints in which, if we want to execute the non-blocked activity, we also need to execute the blocked activity. In this sense, we do not block a particular branch from executing but a set of activities dependant on the execution this activity. We can use the absence to encode that a particular activity is blocked, i.e. it does not occur once. If we also make this absence constraint optional, we can encode the (optional) blocking.

4.2 Extending Declare to Configurable Declare
Configurable Declare is a true extension to Declare in the sense that it only annotates the events and constraints in the model with extra information. We can express a multitude of models in Configurable Declare with the use of these annotations. However, a Configurable Declare model is not executable and operates on a level higher than Declare.

We introduce three types of configuration patterns: (1) the removal of an event, (2) the removal of a constraint, and (3) the alteration of a constraint into a different constraint. In the meta-model for Configurable Declare, depicted in Figure 60 (Appendix A), the events and constraints have been substituted by configurable events and configurable constraints, identifiable by the prefix C. Each configurable event and configurable constraint has an attribute omit which denotes whether this event or constraint can be omitted, i.e. removed from the model. Furthermore, each configurable constraint has a, possibly empty, list associated with it containing all the constraints to which it can be changed to.

4.3 Configuring Configurable Declare
After the design phase of the configurable model, we transform the configurable model in a configuration model. A configuration model is a configurable model which can be annotated, or configured, by the user to model her processes. I.e. in the configuration model the user sets which events and constraints she wants to keep in the model and to which constraints the configurable constraints are changed to. The meta-model for the configuration model is depicted in Figure 61 (Appendix A).

When the user has finished configuring the configuration model, i.e. changed the configuration model in such a way that it resembles, as much as possible, the users processes, it will be transformed into a Declare model. These transformations are listed in Section 4.4, and the implementation of these transformations is summarised in Section 6. The different steps are depicted in Figure 7.

In order to clarify the difference between configurable, configuration and a Declare model, we use the analogy with a multiple choice test. The configurable model can be viewed as a multiple choice test in which the teacher specifies which options can be chosen for certain questions, events and constraints in our case. The student receives the multiple choice test with the possibility to select which answers she thinks are correct, the configuration model. After
Figure 7: The different transformation when transforming a Configurable Declare model into a Declare model

completing the multiple choice test, i.e. setting all of the configuration options in the configuration model, the teacher transforms the filled in test to a grade, or a Declare model in our case, which takes the selected options into account.

4.4 Patterns supported in Configurable Declare

In the remainder of this section, we elaborate on the patterns we were not able to support within Declare but which we can support in Configurable Declare.

Optionality pattern

The optionality pattern allows the user to configure whether one does want to include an event or one does not want to include an event in the model. We have to define some conversions from the old set of constraints to a new set of constraints in order to remove an event.

We have to define these conversions such that the implicit constraints between events are preserved, e.g. if we have that $B$ and $C$ do not occur together and $A$ and $B$ always occur together, we implicitly have that also $A$ and $C$ do not occur together. In the configured model, we would like to have those implicit constraints since in this way we have a new model which stays, with respect to the allowed behaviour between $A$ and $C$, close to the configurable model.

We first look at the different existence templates and their implication on the other events when we remove an event. Afterwards, we explain how we deal with the relation templates such that the implicit constraints still hold between the remaining events. We then show how the different negative relation templates are handled, and finally, the conversions of the choice templates are mentioned.

At the end of this section, we show how to solve the implicit constraints for models which contain branched constraints. We, furthermore, present problems which have not been solved in this thesis but we present solutions to those problems in the future work section (Section 8).

Shorthand notations    Within this section, we use shorthand notation to denote certain sets of templates, e.g. we write $\bowtie B \Rightarrow \bowtie A$ for the set of templates which denote that if we have a $B$ in a trace we also have an $A$ in that trace. In Table 10, a full list is presented with the shorthand notations used in this section. Furthermore, the meaning of the shorthand notation, and the list of templates which are associated with that shorthand notation are listed in that table. Note that if we do not denote the order of the arguments of a template, we assume it is $(A, B)$, e.g. when we write chain succession we mean chain succession($A, B$). We use some shorthand notations identified by [DAC99].
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>( #A \leq #B )</td>
<td>Means that the total amount of times A is executed does not exceed the total amount of times B is executed</td>
<td>alternate response, alternate precedence(B, A), alternate succession, chain response, chain precedence(B, A), chain succession</td>
</tr>
<tr>
<td>( #A \geq #B )</td>
<td>Means that the total amount of times A is executed is at least the total amount of times B is executed</td>
<td>alternate response(B, A), alternate precedence, alternate succession, chain response(B, A), chain precedence, chain succession</td>
</tr>
<tr>
<td>( #A = #B )</td>
<td>Means that the total amount of times A is executed is equal to the total amount of times B is executed</td>
<td>alternate succession, chain succession</td>
</tr>
<tr>
<td>( \Diamond A \Rightarrow \Diamond B )</td>
<td>Means that if there is an A in the trace, there is also a B in the trace</td>
<td>responded existence, co-existence, response, precedence(B, A), succession, alternate response, alternate precedence, alternate succession, chain response, chain precedence, chain succession</td>
</tr>
<tr>
<td>( \Diamond A \Leftrightarrow \Diamond B )</td>
<td>Means that if there is an A in the trace, there is also a B in the trace and vice versa</td>
<td>co-existence, succession, alternate succession, chain succession</td>
</tr>
<tr>
<td>( \neg BWA )</td>
<td>B can only happen after an A has happened</td>
<td>precedence, succession, alternate precedence, alternate succession, chain precedence, chain succession</td>
</tr>
<tr>
<td>( \Box (A \Rightarrow \Diamond (\neg A \cup B)) )</td>
<td>Means that if there is an A in the trace, it is eventually succeeded by a B and there is no other A between these A and B</td>
<td>alternate response, alternate succession, chain response, chain succession</td>
</tr>
</tbody>
</table>

Table 10 continues on the next page
Abbreviation | Meaning | Templates
---|---|---
¬B ∧ W A ⇒ □(¬B W A)) | Every B is preceded by an A and there is no B between these A and B | alternate precedence, alternate succession, chain precedence, chain succession
□(A ⇒ ♦ B) | Means that if there is an A in the trace, it will eventually be followed by a B | response, succession, alternate response, alternate succession, chain response, chain succession
□(A ⇒ ○ B) | Means that every A is directly followed by B | chain response, chain succession
□(○ A ⇒ B) | Means that every A is directly preceded by B | chain precedence, chain succession

Table 10: Shorthand notation for sets of constraints

In the remainder of this subsection, we list how to deal with the different templates. For each template, we have identified the cases where implicit templates occur. Each case consists of a combination of two templates, or classes of templates denoted by the shorthand notation, which are in the model, and, if applicable, a template which should be added after the removal of the event. When we write an LTL like formula, e.g. ♦ A ⇒ ♦ B, we mean the set of templates associated with this formula. We write the name of a template, e.g. not succession, when we want a particular template.

**Existence templates**  Within the existence templates, we consider two events, A and B, and we want to remove one of the events (B). A can then be in two different positions with respect to B: it is either in front or after event B. We consider both cases separately. The position of an event A w.r.t. the position of event B is based on the templates between A and B. If A is in front of B, we mean that either after the execution of A we need to have a B, e.g. response, or prior to executing a B we need to have executed an A, e.g. precedence. A similar reasoning holds when we consider the case in which A is after B.

The relation between A and B can either be a template or a set of templates based on the shorthand notation. If a rule holds for a set of templates, this means that we can apply this rule for each template in that set of templates.

**A in front of B**  We have the case as depicted in Figure 8, note that n₁ and n₂ are conjunctions of non-contradictory existence templates, e.g. n₁ is init and existence₃ but not that n₂ is exactly₅ and absence₄.

![Figure 8: Part of the model if we want to remove event B](image)

In Table 11, all conversions are presented with n₁ consisting of different templates and the relation between A and B. In the last column, the templates
are listed which need to be added to \( n_1 \) after the removal of event \( B \) such that the implicit templates on event \( A \) still hold. Note that we have split the \textit{exactly} into its \textit{existence} and \textit{absence} parts, e.g. instead of considering \( \text{exactly}(5,A) \) we consider the rules for \textit{existence}(5,A) and \textit{absence}(6,A).

<table>
<thead>
<tr>
<th>( n_2 )</th>
<th>Relation ( A \ B )</th>
<th>Add to ( n_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{existence}_m )</td>
<td>( \Diamond B \Rightarrow \Diamond A )</td>
<td>( \text{existence}_1 )</td>
</tr>
<tr>
<td>( \text{existence}_m )</td>
<td>( # A \geq # B )</td>
<td>( \text{existence}_m )</td>
</tr>
<tr>
<td>( \text{existence}_m )</td>
<td>( \Diamond B \Rightarrow \neg \Diamond A )</td>
<td>( \text{absence} )</td>
</tr>
<tr>
<td>( \text{absence}_m )</td>
<td>( # A \leq # B )</td>
<td>( \text{absence}_m )</td>
</tr>
</tbody>
</table>

Table 11: Steps for removing event \( B \)

\( A \ \text{after} \ B \)  We have the case as depicted in Figure 9, note that \( n_1 \) and \( n_2 \) are, again, conjunctions of non-contradictory \textit{existence} templates.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (n1) at (0,0) {$A$};
  \node (n2) at (-2,0) {$B$};
  \draw[->] (n2) -- (n1);
\end{tikzpicture}
\caption{Part of the model if we want to remove event \( B \)}
\end{figure}

In Table 12, the different steps are presented to maintain the implicit templates. Note that we have split the \textit{exactly} into its \textit{existence} and \textit{absence} parts.

<table>
<thead>
<tr>
<th>( n_2 )</th>
<th>Relation ( B \ A )</th>
<th>Add to ( n_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{existence}_m )</td>
<td>( # B \leq # A )</td>
<td>( \text{existence}_m )</td>
</tr>
<tr>
<td>( \text{existence}_m )</td>
<td>( \Diamond B \Rightarrow \Diamond A )</td>
<td>( \text{existence}_1 )</td>
</tr>
<tr>
<td>( \text{existence}_m )</td>
<td>( \Diamond B \Rightarrow \neg \Diamond A )</td>
<td>( \text{absence} )</td>
</tr>
<tr>
<td>( \text{absence}_m )</td>
<td>( # B \geq # A )</td>
<td>( \text{absence}_m )</td>
</tr>
<tr>
<td>( \text{init} )</td>
<td>( \Box (B \Rightarrow \Diamond A) )</td>
<td>( \text{init} )</td>
</tr>
</tbody>
</table>

Table 12: Steps for removing event \( B \)

\textbf{Relational templates}  We have three events: \( A, B, \) and \( C \), and we want to remove event \( B \) from the model. There are three cases: \( A \) and \( C \) in front of \( B \), \( A \) in front of \( B \) and \( C \) after \( B \), and \( A \) and \( C \) after \( B \). Each of these cases is treated in isolation. We use the same notion of before and after, similar to the \textit{existence} templates. Furthermore, whenever we use an LTL-like formula we use it to denote the shorthand notation, when we mean a particular template we denote this by using the template name, e.g. \textit{chain response}.

\( A \ \text{and} \ C \ \text{in front of} \ B \)  We have a part of the model as depicted in Figure 10, note that \( x \) and \( y \) are \textit{relation} templates.

When we have that event \( A \) is always followed by event \( B \) and event \( B \) is always directly preceded by event \( C \), we have the implicit template that event \( A \) is followed by event \( C \). In Table 13, a full list of rules is presented. Note that we have a symmetric case when \( x \) and \( y \) are swapped. We will not list this symmetric case, however, note that the templates will be inverted, i.e. if we have \textit{precedence}(\( C, A \)), swapping \( x \) and \( y \) yields \textit{precedence}(\( A, C \)).
We only need to consider combinations of templates in which \( A \) implies the existence of \( B \) and \( B \) implies the existence of \( C \). In these cases, we have that \( A \) implies the existence of \( C \) by transitivity. This reasoning also needs to be applied in the opposite direction, i.e. \( C \) implies \( B \) and \( B \) implies \( A \) then we have that \( C \) implies \( A \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>Between ( C ) and ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain response</td>
<td>precedence</td>
<td>precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>succession</td>
<td>precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>alternate precedence</td>
<td>alternate precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>alternate succession</td>
<td>alternate succession</td>
</tr>
<tr>
<td>chain precedence</td>
<td>response</td>
<td>response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>succession</td>
<td>response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>alternate response</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>alternate succession</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain succession</td>
<td>response</td>
<td>response</td>
</tr>
<tr>
<td>chain succession</td>
<td>precedence</td>
<td>precedence</td>
</tr>
<tr>
<td>chain succession</td>
<td>succession</td>
<td>succession</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate response</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate precedence</td>
<td>alternate precedence</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate succession</td>
<td>alternate succession</td>
</tr>
<tr>
<td>( \Diamond A \Rightarrow \Diamond B )</td>
<td>( \Diamond B \Rightarrow \Diamond C )</td>
<td>responded co-existence ((A, C))</td>
</tr>
<tr>
<td>( \Diamond A \Leftrightarrow \Diamond B )</td>
<td>( \Diamond B \Leftrightarrow \Diamond C )</td>
<td>co-existence</td>
</tr>
</tbody>
</table>

Table 13: Steps for removing event \( B \)

\( A \) in front and \( C \) after \( B \) We have the situation as depicted in Figure 11, we again have that \( x \) and \( y \) are relation templates.

Figure 11: Part of the model if we want to remove event \( B \)

When \( x \) and \( y \) are both relation templates, the obtained model has a relation \( z \) between \( A \) and \( C \), where \( z = \min(x, y) \), if \( x \) and \( y \) are comparable, i.e. \( x \) is weaker/stronger than \( y \), for instance, when \( x \) is response and \( y \) is alternate response. We do not have a relation, i.e. no template, between \( A \) and \( C \) if \( x \) and \( y \) are incomparable, for instance, \( x \) is chain response and \( y \) is precedence. In Figure 12, the weaker/stronger relation is depicted. When we have an arrow between \( x \) and \( y \), this means \( x \) is stronger than \( y \). To
compute min\((x, y)\), we take the strongest template which is implied by both \(x\) and \(y\), e.g. \(\min(\text{chain response}, \text{alternate succession})\) is \text{alternate response}, \(\min(\text{succession}, \text{succession})\) is \text{succession}, and \(\min(\text{response}, \text{precedence})\) yields \text{no template}.

\[
\begin{array}{ccc}
\text{chain response} & \rightarrow & \text{chain succession} \\
\downarrow & & \downarrow \\
\text{alternate response} & \rightarrow & \text{alternate succession} \\
\downarrow & & \downarrow \\
\text{response} & \rightarrow & \text{succession} \\
\downarrow & & \downarrow \\
\text{responded existence} & \rightarrow & \text{co-existence} \\
\downarrow & & \downarrow \\
\text{no template} & & \text{responded existence}(C, A)
\end{array}
\]

Figure 12: The order between the relation templates in 

When we have the case as depicted in Figure 13 and we remove event \(B\) from the model, we do not have all possible execution paths when we apply the aforementioned approach. In the original model, the execution of \(ABACBC\) is possible while, if we remove event \(B\) from the model, we want to obtain a new model in which \(AACC\) is a valid execution. Unfortunately, when we apply our approach, this allows only executions of \(AC\).

\[
\begin{array}{ccc}
A & \rightarrow & B \\
\rightarrow & & \rightarrow \\
B_1 & \rightarrow & B_{n-1} \\
\rightarrow & & \rightarrow \\
C & & 
\end{array}
\]

Figure 13: A model in which, after the removal of \(B\), our approach only allows a subset of executions

This problem is more fundamental than meets the eye, consider the model in Figure 14. If we omit all \(B_i\)'s from the model, we need to have a model which supports the traces \(A^nC^n\) which is a non-regular language. Our approach only allows the traces \(AC\). We propose to extend \textit{Declare} with a new template to support these reductions, we elaborate on this extension in Section 8.

\[
\begin{array}{ccc}
A & \rightarrow & B_1 \\
\rightarrow & & \rightarrow \\
\cdots & \rightarrow & \cdots \\
\rightarrow & & \rightarrow \\
B_{n-1} & \rightarrow & C
\end{array}
\]

Figure 14: A model in which, after the removal of all \(B_i\)'s, our approach only allows the trace \(AC\)

We only have this problem when we combine two alternate templates, i.e. alternate response, alternate succession, or alternate precedence, into a single
alternate template. We classify these rules as strong rules, i.e. they allow for less behaviour of the new model compared to the behaviour allowed in the original model.

**A and C after B**  We have the partial model as depicted in Figure 15, with \(x\) and \(y\) being relation templates.

![Figure 15: Part of the model if we want to remove event B](image)

For example, when we have that event \(A\) is always preceded by event \(B\) and event \(B\) is always directly succeeded by event \(C\), we implicitly have that event \(A\) is always preceded by event \(C\). In Table 14, a full list of combinations is presented. Note that, similar to the case in which \(A\) and \(C\) are in front of \(B\), we have a symmetric case when we swap \(x\) and \(y\) as explained earlier.

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>Between (A) and (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain response</td>
<td>precedence</td>
<td>precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>succession</td>
<td>precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>alternate precedence</td>
<td>alternate precedence</td>
</tr>
<tr>
<td>chain response</td>
<td>alternate succession</td>
<td>alternate precedence</td>
</tr>
<tr>
<td>chain precedence</td>
<td>response</td>
<td>response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>succession</td>
<td>response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>alternate response</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain precedence</td>
<td>alternate succession</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain succession</td>
<td>response</td>
<td>response</td>
</tr>
<tr>
<td>chain succession</td>
<td>precedence</td>
<td>precedence</td>
</tr>
<tr>
<td>chain succession</td>
<td>succession</td>
<td>succession</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate response</td>
<td>alternate response</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate precedence</td>
<td>alternate precedence</td>
</tr>
<tr>
<td>chain succession</td>
<td>alternate succession</td>
<td>alternate succession</td>
</tr>
<tr>
<td>(\Diamond A \Rightarrow \Diamond B)</td>
<td>(\Diamond B \Rightarrow \Diamond C)</td>
<td>responded existence</td>
</tr>
<tr>
<td>(\Diamond A \Leftrightarrow \Diamond B)</td>
<td>(\Diamond B \Leftrightarrow \Diamond C)</td>
<td>co-existence</td>
</tr>
</tbody>
</table>

Table 14: Steps for removing event \(B\)

**Negative relation templates**  For the negative relation templates, we have the same cases as for the relation templates. We, furthermore, have the same figures as with the relation templates.

**A and C in front of B**  We have the case as depicted in Figure 10, note that \(x\) or \(y\) are now negative relation templates. In Table 15, the conversions are listed, note that we have symmetric cases when we swap \(x\) and \(y\) similar to the symmetric cases for the relation templates.
A and C not co-existence

Not co-existence

Table 15: Steps for removing event B

The combinations listed are inspired by transitivity, for instance, the not succession between A and B. When we also have that every C is eventually followed by B, i.e., succeeded by B, we do not want to have a C after we have seen an A. A similar reasoning hold for the not co-existence between A and B, we now want to consider templates in which C implies the existence of B.

A in front and C after B In Figure 11, this case is depicted. Note that again x or y are negative relation templates. The conversions are listed in Table 16.

Table 16: Steps for removing event B

Similar to the previous case, i.e. A and C in front of B, we are searching for combinations of templates in which A prevents the execution of B and C demands the execution of B. When we have two templates which fulfill the aforementioned conditions, we know we have to create a template between A and C.

A and C after B This case is depicted in Figure 15 with again x or y being negative relation templates. Table 17 contains the conversions. The symmetric case can easily be obtained in a similar way as with the relation templates.

Table 17: Steps for removing event B

The rules identified are very similar to the rules identified when A and C are in front of B. This is mainly because we have a symmetric cases for the first rule, for the second rule we need to have a template between B and C in the opposite direction, i.e. C followed by B versus C preceded by B.

Choice Templates With the choice templates, we encounter a problem when we want straightforward deduction rules. Consider the model depicted in Figure 16 from which we want to remove event B. We can deduce that B will always be executed, hence we need to keep the choice construct, and m – 1 out
of the $n - 1$ events need to be executed. Although, we can deduce that $B$ will always be executed, we need to follow a chain of response templates to deduce it. This is, however, undesirable from the viewpoint of the user configuring the model since she needs to observe a large portion of the model. This approach is also error prone because one might miss a response template.

![Diagram](image)

Figure 16: Example in which we need to consider a large portion of the model

The best way of solving the problem is to assume that either event $B$ is always executed or event $B$ is never executed independent of the model, i.e. in the example of Figure 16 we might still assume that event $B$ is never executed, although this is not the case. The main reason is that we want a deterministic approach for dealing with choices based on the local properties. The user of the configurable model can then adjust the $m$ attribute of the choice template to reflect the semantics of the model without event $B$ while the user does not need to considering the entire model to find out whether event $B$ will always or never be executed.

We assume event $B$ is never executed; this has the advantage that the user of the configurable model can set the $m$ of the choice and it will not be changed, i.e. it denotes the minimal amount of events needed to be executed with respect to the choice after removing the omitted events. If we would assume that event $B$ is always executed, we would have updated the $m$ and the $n$. In this case, the user would have to set the $m$ at such a value that after the removal of the omitted events the $m$ has the right value, which is obviously harder.

**Supporting branched templates** In order to apply the rules defined earlier on branched templates, we need to extend the language. Consider the example in Figure 17 and we want to remove the events $B_1, B_2,$ and $B_3$.

The implicit template which holds after the removal of the $B_i$s is depicted in Figure 18. Although, this kind of template is not part of Declare, Declare can be extended with it. We elaborate more on this extension in the future work section (Section 8).

We can solve this problem when we only compare the same templates, e.g. only response templates with response templates. Consider the model in Figure 19 where, after the removal of $B_2$, we can deduce and express the implicit templates in Declare. These implicit templates are $\text{response}([A], \{B_1, C\})$ and $\text{response}([B_3], \{C\})$.

When we have composed templates, i.e. templates which consist of the conjunction of other templates, for instance, succession and co-existence, we
Figure 17: A model in which we cannot express the implicit templates

Figure 18: The implicit templates after the removal of the $B_i$s in Figure 17

Figure 19: Model in which, after the removal of $B_2$, we have partial implicit templates
only consider the templates in the conjunction. Consider the model in Figure 20. When we compute the implicit templates, we have the model as depicted in Figure 21 in which we do not have a single succession template.

![Figure 20: Model in which after the removal of B2 we do not have implicit succession templates](image)

We can be slightly more permissive than only comparing the same templates. In general, when we have that A places a constraint on B, e.g. response(A, B), and B places a constraint on C, e.g. precedence(C, B), we have implicitly that A places a constraint on C. With branched constraints, we have two sets B1 and B2 instead of a single B, i.e. response(A, B1) and precedence(C, B2), A can only place a constraint on C if all event in B1 are omitted and B1 is a subset of B2. We want that B1 is a subset of B2 such that we know that A only places a constraint on C. If B1 is not a subset of B2, A might also place a constraint on another event such that we cannot conclude anything about its relation with C. Both cases, i.e. B1 ⊈ B2 and B1 ⊆ B2, are depicted in Figure 22. In the left model, we have that A can be followed by B1. When A is followed by B1, we do not have a constraint on C. If A is followed by B2, we do have a constraint on C, this means that A cannot place a constraint on C. In the right model, it does not matter whether we perform B1 or B2 after A because we always know that C will be executed, hence A places a constraint on C.

![Figure 21: Figure 20 with the implicit templates after the removal of B2](image)

We can be even more permissive when A is in front of B and C is after B, we now do not need that B1 is a subset of B2. In this case it is sufficient that the template between A and B1 is not stronger than the template between B2 and C, in the case A places a constraint on C. When C places a constraint on A, we need to have that the template between C and B2 is not stronger than
the template between $B^1$ and $A$. Consider, for instance, Figure 23, if $B_1$ and $B_2$ are omitted, $A$ still posses a constraint on $C_1$, $C_2$, and $C_3$, although, $B^1$ is not a subset of $B^2$.

All unidirectional templates, i.e. one set of events constrain the other set of events and not vice versa, can be split on one of their arguments. For instance, $\text{response}(\{A_1, A_2\}, B)$ is equivalent to $\text{response}(A_1, B)$ in conjunction with $\text{response}(A_2, B)$, and $\text{precedence}(A, \{B_1, B_2\})$ is equivalent to $\text{precedence}(A, B_1)$ in conjunction with $\text{precedence}(A, B_2)$. This allows us to combine templates and split templates. The templates which can be split on their first argument are: $\text{responded existence}$, $\text{response}$, $\text{alternate response}$, $\text{chain response}$, $\text{not succession}$, and $\text{not chain succession}$, the templates which can be split on their second argument are: $\text{precedence}$, $\text{alternate precedence}$, and $\text{chain precedence}$. The $\text{co-existence}$, $\text{succession}$, $\text{alternate succession}$, and $\text{chain succession}$ templates cannot be split straightforward but we can substitute them by their conjuncts, i.e. $\text{succession}$ consists of the conjunction of $\text{response}$ and $\text{precedence}$. Finally, the $\text{not co-existence}$ can be split on either its first argument or its second argument but not on both at the same time.

The aforementioned equivalence can be used to create the subset relation.
Let \( B^1 \) be the set of events which needs to be a subset of \( B^2 \), we can try to decrease the size of \( B^1 \) or increase the size of \( B^2 \) by combining or splitting templates.

A similar problem, w.r.t. implicit templates which cannot be expressed, occurs when we consider the existence templates. In Figure 24, we cannot deduce what implicit templates hold on \( B_1, B_2, \) and \( B_3 \) after the removal of \( A_2 \). If \( A_1 \) and \( A_3 \) only occur 4 times, or less, we know any of the \( B_i \)'s have to occur at least once. Also this template cannot be expressed in Declare, but we have opted for an extension to Declare to support these cases in Section 8.

![Figure 24: We cannot express the implicit templates after the removal of \( A_2 \)](image)

Not all existence templates are problematic. Consider the model in Figure 25. If we remove \( A_1, A_2, \) and \( A_3 \), we know \( B_1, B_2, \) or \( B_3 \) has to occur at least once. This yields for the following decision, w.r.t. application of a rule: if an existence template is between a subset of events of the connected relation template and all events connected to the existence template are omitted, i.e. \( isOmitted \) is set to true, we can apply the rule.

![Figure 25: We can deduce the implicit templates after the removal of \( A_1, A_2, \) and \( A_3 \)](image)

When we have \( \text{existence}(5, \{A_1, A_2, A_3\}) \) and we omit \( A_2 \), we cannot deduce
anything about how many times $A_1$ and $A_3$ have to be executed. With the \textit{absence}(5, \{A_1, A_2, A_3\})}, where we omit $A_2$, we can deduce that $A_1$ and $A_3$ cannot happen more than 5 times. Therefore, we can keep the \textit{absence} template but we cannot keep the \textit{existence} template when one of the events is omitted.

**Removing a constraint** We have seen how to handle the deletion of an event from the model. The deletion of a constraint is straightforward, we simply remove the constraint from the model. We do not need to take anything else into account since the removal of a constraint $c$ also means we want to remove the implicit constraints with respect to $c$. This holds automatically after the removal of $c$.

**Alteration pattern**

We can change constraints into different constraints, or into a similar constraint but with different attribute values, e.g. \textit{existence}$_5$ instead of \textit{existence}$_7$. The alteration is straightforward since we only need to remove the old constraint and add the new constraint connected to the same sets of events.

**4.5 Running example**

We have listed all of the different rules identified to change a \textit{Configurable Declare} model into a \textit{Declare} model. We present an example from the case study (Section 7) in which we demonstrate the application of the different rules. Consider model A (Figure 26) in which filled red crosses denote that we want to remove that event or constraint.

First, we remove the events which do not yield any implicit constraints, e.g. \textit{Send to department in mailbox}, and the constraints which need to be remove, i.e. \textit{exclusive choice} between \textit{Inform customer via telephone or e-mail} and \textit{Produce extract and sign it}. This yields the model in Figure 27.

We remove the events which are inside a blue or green square. The events in a blue square can be removed straightforward by applying the non-branched rules. Removing \textit{Inform customer via telephone or e-mail} entails more steps. First, this constraint is split into a \textit{response} and a \textit{precedence}. We can update the \textit{precedence} constraint by removing the \textit{Inform customer via telephone or e-mail}, however, we cannot remove this event from the \textit{response} constraint. We, therefore, remove the \textit{response} constraint. The newly acquired model is depicted in Figure 29.

We have annotated the previous output model with blue squares denoting which events are removed in this step. All events can be removed straightforward without any extra computations, note that we remove the duplicate \textit{succession}, between \textit{Send to DMS department} and \textit{Produce extract and sign it}, prior to deducing any implicit constraints. The model obtained after the removal of the events in blue is depicted in Figure 30.

Figure 30 is annotated with which events will be removed in the next iteration. In the next iteration, the duplicate \textit{succession}, between \textit{Process payment} and \textit{Produce extract and sign it}, is removed. The obtained model is depicted in Figure 31 which is also annotated with which events will be removed in the next iteration.
The output model (Figure 32) does not contain any duplicate constraints. Since there are no events which need to be removed, i.e. $isOmitted$ is true, we know we are done with applying the rules.

Figure 26: The input model for applying the different rules

Figure 27: The model after removing all omitted constraints and event which do not yield any implicit constraints
Figure 28: The model from Figure 27 annotated with which events will be removed in the following step

Figure 29: The model obtained after the previous step annotated with which events will be removed in the following step
Figure 30: The model obtained after the previous step annotated with which events will be removed in the following step

Figure 31: The model obtained after the previous step annotated with which events will be removed in the following step
Figure 32: The output model
5 Combining models

When an organisation wants to apply Configurable Declare to increase the maintainability and flexibility of their process models, they have to design this Configurable Declare model from scratch. Designing this Configurable Declare model is difficult, i.e. it has to support all the different Declare models, and it is costly. Therefore, we present a conversion in the opposite direction, i.e. instead of going from a Configurable Declare model to a Declare model, we are now given a set of Declare models and we want to create a Configurable Declare from which all of those Declare models can be deduced from (Figure 33).

![Diagram of combining different Declare models into a single Configurable Declare model](image)

Figure 33: The combining of different Declare models into a single Configurable Declare model

We first show the straightforward approach in which we simply include everything which is in the Declare models. This means that although a constraint can be deduced from the current set of constraints in the Configurable Declare model, it will be added if it is present in one of the Declare models. For instance, assume we have that every A is eventually followed by a B and every B is eventually followed by a C, then we also have implicitly that every A is eventually followed by a C. When a Declare model contains the constraint that every A is eventually followed by a C, this constraint will be made explicit in the Configurable Declare model, although, this might already be encoded.

After the straightforward approach, we define heuristics to reduce the amount of constraints in the Configurable Declare model as well as how to deal with choice and exclusive choice templates. Finally, we go step by step through the merger of two models from the case study.

5.1 Combining the models

We have a set of Declare models which we want to merge. For these Declare models, we create a single Configurable Declare model. The different constructs, events and constraints, are dealt with in the following sections.

**Event**

All of the Events are converted to CEvents with the same name. If an Event appears in all Models, i.e. an Event with the same name is present in all Models, we add the corresponding CEvent to the CModel with the CEvent’s attribute omit set to false. If an Event appears only in some Models, we add the corresponding CEvent to the CModel with the CEvent’s attribute omit set to true.

**Constraint**

Every Constraint is converted to a CConstraint with the same name, and the canBeChangedTo attribute is set to the Constraint which we are transforming.
If a Constraint appears in all Models, i.e. every Model has the same constraint between the same set of Events, its omit attribute is set to false. If a Constraint does not appear in all Models, we set the omit to true.

We check whether a Constraint appears in all Models independent of all events which it constrains are present in the models. If we would take this into account, i.e. a Constraint c(A, B) can be omitted if and only if there is a model in which c does not occur and all events in A and B are present in that model, we would not be able to deduce all models. Consider, for instance, the models depicted in Figure 34, since B occurs always with the succession constraints (succession(A, B) and succession(B, C)), we cannot omit the succession constraints. This prevents us from deducing the bottom model since, after the removal of B, we always have a succession between A and C.

![Figure 34: Two models for which it must hold, after combining them, that all constraints can be omitted, although, B does not occur in all models](image)

After initialising the name and the omit attributes, the from and to attributes have to be set. We want to have a mapping which, after applying it, yields the same set of constraints but now on the set of CEvents. Figure 35 depicts the conversion from Constraint to CConstraint w.r.t. the from and to attributes. Here, c is a Constraint, c' is the conversion from c to a CConstraint where the name and omit attributes have been set. When we apply $\chi$ to an Event, we transform it to a CEvent in the aforementioned way.

### 5.2 Removing redundant constraints

When we have combined all models, we might have redundant constraints in our model. We can use the rules defined in Section 4.4 to compute the redundant constraints in our model. The main difference is that we keep all events in the model instead of removing events.

We cannot apply the rules naively, i.e. if we can omit an event we can apply the rules to deduce implicit constraints. Consider the example in Figure 36, if we combine all three models, we have a response between $A$ and $B$, between $B$ and $C$, and between $A$ and $C$. The response between $A$ and $C$ is redundant since we can obtain this constraint by removing $B$ and applying the rules. However, if we remove the response between $A$ and $C$, we cannot deduce model $c$.

Therefore, we need to take into account which events are present in a certain model prior to applying the rules. If an event $B$ and a constraint $c$ are both present in at least one model, we cannot apply rules which state that, after the removal of $B$ from the model, we can deduce constraint $c$. 

41
We also have to take into account that two rules can deduce each other. Consider the model in Figure 37, where we can deduce both precedence constraint by applying deduction rules on the other two constraints, the chain succession and the precedence. We want to have one precedence in the model to be able to deduce the original set of input models, therefore, we need to take into account which constraints can be deduced by which other constraints. This way we can easily deduce when two constraint deduce each other and hence, one needs to remain in the model.

Choice templates

We have seen with the conversion from a Configurable Declare model to a Declare model that we are not able to convert the choice templates straightforward. We have a similar problem when we combine two models. Consider the two models depicted in Figure 38.

When we combine both models, we can have both resulting models as depicted in Figure 39. For the choice templates, we choose the right resulting model. We choose this model since it allows the same traces as the combination of the traces of the input models. E.g. the trace $CD$ is allowed in the input models and also in the right combined model, but not in the left combined model.

5.3 Running example

In order to get some more intuition for the model combiner, we combine two models used for the case study. Assume we have the models depicted in Figures 40 and 41 (models D and F of the case study).
Figure 36: Three models from which, after combining them, we cannot remove any implicit constraints

Figure 37: An example of a model in which both precedence constraints can be deduced from the other precedence and chain succession constraint

We first need to check which events and constraints are present in both the models. We apply the decision making mentioned earlier, i.e. if an event or constraint is present in all models we set omit to false else we set it to true, furthermore, an open red cross means that omit is set to true. We now combine all events and constraints into a single model which is depicted in Figure 42.

When we apply the rules identified in Section 4.4, we obtain the minimal Configurable Declare model depicted in Figure 43.
Figure 38: Two input models

Figure 39: Two possible output models

Figure 40: Input model D

Figure 41: Input model F
Figure 42: The configurable declare model of the models D and F with the *omit* attributes set

Figure 43: The minimal configurable declare model of the models D and F
6 Implementation

We have implemented both directions mentioned in the previous sections, i.e. going from a Configurable Declare model via a Configuration model to a Declare model, and combining a set of Declare models into a single Configurable Declare model. This implementation is used in the case study to show that Configurable Declare can be applied to real life models.

The implementation is done in Java and Figure 44 depicts the architecture and the communication between the different parts of the code. The Main class is the orchestrator of the code, it takes care that the input models, arguments, and rules are read, and that the output models are written. Based on the arguments it either performs one of the four conversions: (1) Configurable Declare → Configuration, (2) Configuration → Declare, (3) Declare → Configurable Declare, and (4) Declare + Configurable Declare → Configurable Declare. Each of these conversion either belongs to Configurable Declare → Declare or to the Model Combiner. We elaborate on the two different directions in the following sections as well as explain the rule engine which is used in both directions. Note that we do not fully support Branched Declare since Declare does not yet allow all constructs necessarily. When the implementation is applied on cases which are not covered, we cannot guarantee anything about the output of the conversions.

![Figure 44: The architecture for the different conversions](image)

6.1 Configurable Declare to Declare

Transforming a Configurable Declare model into a Declare model entails two steps: (1) going from a Configurable Declare model to a Configuration model and (2) going from a configured Configuration model to a Declare model. The conversion from a Configurable Declare model to a Configuration model is straightforward, i.e. we only need to initialise an extra attribute, isOmitted which is
set to the value of the `omit` attribute, and change the types of the different
constructs, e.g. a `CConstraint` is changed into a `ConfConstraint`.

In Table 18, the different value for the different conversions are listed. Note
that $\phi(A_i)$ means the conversion from a `CEvent` to a `ConfEvent`, and $\psi(A_i)$
means the conversion from a `ConfEvent` to an `Event`.

The conversion from a `Configuration` model to a `Declare` model entails more
work. Apart from converting the different constructs, we also have to apply
the rules as defined in Section 4.4. We only mention the conversion from a
`Configuration` model to a `Declare` model, the application of the rules is described
in detail in Section 6.3.

Prior to applying the rules, we need to perform some preprocessing. We
first have to remove all constraints which are omitted, i.e. the `isOmitted`
attribute is set to true. We, furthermore, need to substitute constraints, when the
`isChangedTo` attribute is set, to the intended constraint.

When our model contains composed constraints, we need to split this com-
position on the different constraints. Consider, for instance, a `succession(A, B)`,
this constraint is composed of `response(A, B)` and `precedence(A, B)`, but also `co-
existence(A, B)` is a composed constraint consisting of `responded existence(A, B)`
and `responded existence(B, A)`. We need to split the constraints on their com-
position to support `Branched Declare`. Note that we duplicate the `not co-
existence` constraint, i.e. when we have `not co-existence(A, B)`, we also add
`not co-existence(B, A)` to the model, such that we can split on both arguments.

We now apply the rule engine on the remaining set of constraints to compute
all the implicit constraints. After having computed the implicit constraints, we
need to remove the omitted events and all constraints, implicit and explicit,
connected to those events. Finally, we map weaker constraints to stronger con-
straints, e.g. when we have `response(A, B)` and `alternate response(A, B)`, we
remove the `response` and keep the `alternate response`.

### 6.2 Combining Declare models

Combining a set of `Declare` models is implemented as an iterative approach. We
first transform a `Declare` model into a `Configurable Declare` model, and then,
we add the remaining `Declare` models, one-by-one, to obtain every time a new
`Configurable Declare` model.

Using an iterative approach increases the flexibility but it reduces the mini-
mality of the model, i.e. the model becomes bigger. The increased flexibility is
achieved by only considering a `Configurable Declare` model and a `Declare` model
without considering all Declare models which lead to the Configurable Declare model. In Figure 45, we have an example in which combining the models in a certain order does not yield a minimal model.

![Diagram](image)

Figure 45: Combining three different models in two different ways.

As mentioned before, combining the Declare models consists of two phases: (1) transform a Declare model into a Configurable Declare model (initialisation), and (2) combining a Declare model with a Configurable Declare model (iterative step). We explain both steps in isolation in the following sections.

**Initialisation**

We transform a Declare model (Model) into a Configurable Declare model (CModel) by transforming the different constructs, i.e. constraints and events. We list per construct the steps taken to transform it. Prior to transforming the events and constraints, we first create a new CModel with its name equal to the name of the Model. Afterwards, we transform the Events into CEvents where the name of the CEvent is equal to the name of the Event and the omit attribute is set to false.

The different initialisations are summarised in Table 19, note that we can use the initialisation for both the CEvents and CConstraints, e.g. we keep the name the same for both the CEvent and the CConstraint. When we write $\chi(A_i)$, we mean the transformation from Event $A_i$ to a CEvent. Prior to applying these transformation, we first split composed constraints, similar to the conversion from a Configuration model to a Declare model, we also duplicate the not co-existence.
### Iterative step

In the iterative step, we need to do some more work apart from simply converting a Declare model into a Configurable Declare model and combine this with the original Configurable Declare model. The first step in the iterative approach is setting the omit attributes correct. For each CEvent, we check whether there is an Event in the Declare model with the same name. If we cannot find an Event with the same name, we set the omit attribute of the CEvent to true else we do not change the value.

We apply a similar approach for the CConstraints, of which first the composed constraints are split, and all not co-existence are duplicated. If there is not a semantically equivalent Constraint present in the Model, we set the omit attribute of the CConstraint to true. Semantical equivalence between a CConstraint and a Constraint means that they both represent the same constraint on the same set of events, e.g. Succession\((A, B)\) is semantically equivalent to the CSuccession\((A, B)\).

After having set all of the omit attributes of the events and constraints already in the Configurable Declare model, we can add the events and constraints not in the Configurable Declare model, but which are in the Declare model, to the Configurable Declare model. We convert each event and constraint in the same way as we did in the initialisation step. The main difference with the initialisation step is that the omit attribute is set to true instead of false.

We use the rule engine to deduce all implicit constraints which can be obtained by considering the events not present in both models as being omitted. For each constraint, we store from which other constraints it can be deduced from (children) and which constraint it deduces (parents). Consider, for instance, the model in Figure 46, here, \(c_1\) and \(c_2\) are the children of \(c_3\), i.e. \(c_3\) can be deduce from \(c_1\) and \(c_2\) using the rules and \(c_3\) is a parent of \(c_1\) and a parent of \(c_2\).

![Figure 46: The parent child relation for a model](image)

The output of the rule engine is a directed graph denoting which constraints can be deduced from which other constraints. Each constraint \(c\) can be part of any of the following classes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Declare</th>
<th>Configurable Declare</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>omit</td>
<td>(\times)</td>
<td>false</td>
</tr>
<tr>
<td>from (A_1, \cdots, A_n)</td>
<td>(\chi(A_1), \cdots, \chi(A_n))</td>
<td></td>
</tr>
<tr>
<td>to (B_1, \cdots, B_m)</td>
<td>(\chi(B_1), \cdots, \chi(B_m))</td>
<td></td>
</tr>
<tr>
<td>canBeChangedTo</td>
<td>(\times)</td>
<td></td>
</tr>
</tbody>
</table>

Table 19: The initialisation of the different attributes when transforming a Declare model into a Configurable Declare model
• Explicit: $c$ cannot be deduced or it occurs in all models, i.e. $omit$ is false

• Implicit: $c$ can be deduced and $c$ does not occur in all models

• Deducible: $c$ is either explicit or it has a pair of children and both children are deducible

We want that all constraints are deducible which means that our model covers all constraints. When some constraints are not deducible, this means that some behaviour is allowed which should not be allowed, i.e. some constraints are in the complete and naive model but are not in the reduced model.

We use Algorithm 1 to compute which constraints should remain in our model. Let $S_c$ denote the set of constraints we want to keep, note that we only include constraints which are present in at least one model, this to prevent the algorithm to choose and add implicit, and thus redundant, constraints to the model.

**Algorithm 1:** The pseudo code for choosing which constraints to include in the model

```
chooseConstraints()

(1) $S_c \leftarrow \emptyset$

(2) foreach constraint $c$ which is present in all models

(3) add $c$ to $S_c$

(4) foreach constraint $c$ which cannot be deduced from any constraint

(5) add $c$ to $S_c$

(6) while there are constraints not yet deducible from the constraints in $S_c$

(7) Let constraint $c$ be a constraint not yet deducible by the constraints in $S_c$ and $c$ is present in at least one model

(8) add $c$ to $S_c$
```

### 6.3 Rule engine

We have created a rule engine which is used in both directions, i.e. going from a *Configuration* model to a *Declare* model and going from a *Declare* model and a *Configurable Declare* model to a *Configurable Declare* model. The rule engine consists of four different functionalities: (1) a *rule container* which is a wrapper for the different rules as defined in Section 4.4, (2) a *rule referee* which determines whether we can apply a certain rule in a certain case, (3) the *closure* takes care of the actual application of the rules on the set of constraints, and (4) a *cleaner* maps equivalent constraints on top of each other.

The rule container mainly consists of the following functionality: given two constraints and the position of the to-be-removed event, i.e. the cases we identified in Section 4.4, which constraint holds implicitly between the remaining events. We use the rule referee to query when we may apply a certain rule on a pair of constraints, e.g. when we transform a *Configuration* model to a *Declare* model we only compute an implicit constraint for two constraints if they share an omitted event.
The application of the rules is done similar to Algorithm 2 which computes the closure. We have included the handling of a single rule, $A$ in front and $C$ after $B$, in Algorithm 3. Each of the functions is dealt with in isolation.

**CalculateClosure** Within the calculation of the closure, we keep iterating through the list of constraints until we reach a fixpoint. This fixpoint is defined to be the maximal length a constraint can have, where the length of a constraint is recursively defined as the length of the constraints from which it is deduced. When a constraint is not deduced from any other constraints, we assume the length of this constraint to be one.

The constraints in the $\ell_c$ are traversed and, for every constraint $c_i$, we check whether we can apply a certain set of rules. For instance, if $c_i$ is an existence constraint, we know we cannot apply the rules for the choice templates. If we can apply a certain rule, we pass the constraint to a function to try and apply the rules on this constraint.

After each iteration of the computation of the closure, we clean the set of constraints we have computed so far. This cleaning entail mapping pairs of constraints on top of each other. Consider Figure 47 where $c_{12}$ can be deduced from $c_1$ and $c_2$, and $c_{34}$ can be deduced from $c_3$ and $c_4$. Obviously, we do not want to keep both $c_{12}$ and $c_{34}$ inside of the model. We, therefore, map $c_{34}$ on top of $c_{12}$ which means that we remove $c_{34}$ from the model, add $c_3$ and $c_4$ as a pair of children to $c_{12}$, and make $c_{12}$ the parent of $c_3$ and of $c_4$.

![Diagram](image)

Figure 47: A model in which the cleaner maps $c_{34}$ on top on $c_{12}
Algorithm 2: The pseudo code for computing the closure

\textbf{CALCULATECLOSURE}()

(1) \textbf{while} We have not yet arrived at a fixpoint
(2) \textbf{copy} the constraints to a temporal list \(\ell_c\)
(3) \textbf{foreach} \(c_i \in \ell_c\): we want to consider \(c_i\)
(4) \textbf{if} the rule referee allows us to apply the \textit{existence} set of rules on \(c_i\) \textbf{then}
(5) \hspace{1em} ...\textbf{:}
(6) \textbf{if} the rule referee allows us to apply the \textit{relation} or \textit{negative relation} set of rules on \(c_i\) \textbf{then}
(7) \hspace{1em} \textbf{APPLYRULESABC}(\ell_c, c_i)
(8) \hspace{1em} \textbf{APPLYRULESACB}(\ell_c, c_i)
(9) \hspace{1em} \textbf{APPLYRULESBAC}(\ell_c, c_i)
(10) \textbf{if} the rule referee allows us to apply the \textit{choice} set of rules on \(c_i\) \textbf{then}
(11) \hspace{1em} ...\textbf{:}
(12) \textbf{Clean} the constraints using the cleaner

\textbf{APPLYRULESABC} When we possibly can apply a rule, we try a constraint \((c_i)\) with all possible other constraints \((c_j \in \ell_c)\). If the rule referee allows us to combine two constraints, we apply the rules and obtain a new constraint \(c'\). We set the \textit{from} and \textit{to} relation of \(c'\), note that we assume that \(c_i\) is between \(A\) and \(B\) and \(c_j\) is between \(B\) and \(C\), if this is not the case, we do not allow the application of this rule.

Similar to the examples in Section 4.4 and in particular Figure 23, we either need to combine the \textit{from} or \textit{to} events of the new constraint with the \textit{from} or \textit{to} of \(c_i\) or \(c_j\). If all constraints are \textit{response} type of constraints, i.e. they can be split on their first argument, we need to add \(c_i\text{'}.to\) to the \(c'.to\). We need to add this since the events which are between \(c_i\) and \(c_j\) which are omitted are now substituted by the events in \(c_j\text{'}.to\).

If all constraints are \textit{precedence} type of constraints, i.e. they can be split on their second argument, we know that the events which are between \(c_i\) and \(c_j\), which are omitted, are now substituted by the events in \(c_i\text{'}.from\). But, we still have to maintain the events which were already in \(c_j\text{'}.from\).

Consider, for instance, the model in Figure 48 from which we want to remove \(B_2\). \(c'.from\) is equal to the \(c_i\text{'}.from\), and \(c'.to\) is equal to the union of \(c_j\text{'}.to\) and \(c_i\text{'}.to\). In other words, event \(B_2\) is substituted, in \(c_i\), by \(C_1\) and \(C_2\), and \(c'\) now represents this constraint.
Figure 48: A model in which both the $c_i.to$, and $c_j.to$ have to be added to $c’.to$

Algorithm 3: The pseudo code for applying the ABC type of rules

\textbf{APPLYRULESABC($\ell_c$, $c_i$)}

\textbf{Input:} $\ell_c$ a list of constraints, $c_i$ a constraint

\textbf{Output:}

1. \textbf{foreach} $c_j \in \ell_c$: $c_i \neq c_j$
2. \textbf{if} the rule referee allows us to apply a rule on $c_i$ and $c_j$ \textbf{then}
3. \hspace{1cm} $c’ \leftarrow$ the constraint after applying the rules, on $c_i$ and $c_j$
4. \hspace{1cm} $c’.from \leftarrow c_i.from$
5. \hspace{1cm} $c’.to \leftarrow c_j.to$
6. \hspace{1cm} \textbf{if} $c_i$, $c_j$, and $c’$ are response type of constraints \textbf{then}
7. \hspace{2cm} $c’.to \leftarrow c’.to \cup c_i.to$
8. \textbf{if} $c_i$, $c_j$, and $c’$ are precedence type of constraints \textbf{then}
9. \hspace{2cm} $c’.from \leftarrow c’.from \cup c_j.from$
10. \textbf{add} $c’$ to the ConfigurationModel
7 Case Study

In order to validate the applicability of Configurable Declare to real life models, we have performed a case study. We performed our case study on models obtained by interviewing different municipalities. These interviews yielded imperative models from which we have deduced the Declare models. The different models entail the request for an excerpt from the civil registration. The different municipalities model this process in different, but very similar, ways.

All models include the following three activities: (1) Fill in e-form in which the customer fills in her credentials on an e-form in order to obtain an excerpt. (2) Produce extract and sign it in this step the municipality produces the excerpt and signs it denoting that this is an official excerpt. (3) Send extract denotes that the excerpt is sent to the customer.

We want to show with this case study that we can combine the Declare models from the municipalities into a Configurable Declare model and configure the Configurable Declare model such that we can obtain all original models.

7.1 Combining the models

We have combined the different input models, depicted in Appendix B, in alphabetical order, i.e. first model A with model B, then the combined model AB with model C, etc. The output model is depicted in Figure 49. Combining the models is done in the same way as explained in Section 5.

The output model is minimal in the sense that we do not have any redundant constraints. Using the model combiner, we were able to remove three redundant constraints. The succession between Process payment and Produce extract and sign it, for instance in model A, has been removed when we added model C to the combined model. Between Fill in payment information and Send to DMS department, we have a succession in model F, this constraint was never added since we already had an implicit succession from model D. Finally, we had a succession between Produce extract and sign it and Send extract, for instance, in model B, but this succession is implicit in model D and has been removed.

We can also see why we need to split the succession constraints in response and precedence constraints. Instead of a succession between Print request and Produce extract and sign it, we have a response. The reason we have a response is because the precedence between Print request and Produce extract and sign it can be deduced via Determine whether request is admissible and Send to the department concerned.

7.2 Configuring the models

We use two different crosses to denote the different reasons for removing a constraint. A complete red cross denotes that we have to remove this constraint to obtain a model which allows all desired behaviour. Red crosses filled with green denote that we remove a constraint to remove redundant constraints, note that, when we transform a Configuration model to a Declare model, we do not consider other implicit constraints already in the model.

In order to obtain the input models, we first set the isOmitted attribute of the activities not in the input model to true. When we have an exclusive choice consisting of only one activity, e.g. when we want to remove Inform customer
via telephone or e-mail, we also remove the exclusive choice. If we would not remove this exclusive choice, it would mean that the other activity is always executed. In some cases we need to remove some constraints from the model to prevent unwanted implicit constraints, e.g. to obtain model E we need to remove, amongst other constraints, the succession between Process payment and Send e-mail to central desk, if we would keep this constraint it would need to hold that every Process payment is eventually followed by a Produce extract and sign it while the user might want to choose to execute activity Inform customer via telephone or e-mail.

After all activities and constraints which need to be removed have been removed, we choose to remove some extra constraints from the model. By removing these extra constraints, we prevent the deduction of redundant constraints on the models. Consider, for instance, model C, if we would keep the succession constraint between Process payment and Send to DMS department, we can deduce that we have a succession between Process payment and Produce extract and sign it. Although, this is not incorrect, it is redundant since we already have this succession implicit via the activities: Send e-mail to central desk and Check request in digital desk.

7.3 Running example

We have seen how we combined the models D and F in Section 5. We now elaborate on how to configure the Configurable Declare model such that we obtain the original models again.

Model D, and the configuration for model D, are depicted in Figure 50 and Figure 51 respectively. We first need to set the isOmitted attribute of all events not present in the original model to true.
The exclusive choice between Produce extract and sign it and Inform customer via telephone or e-mail has to be removed. After the removal of Inform customer via telephone or e-mail, exclusive choice demands that we always execute Produce extract and sign it, although, the user might have chosen to execute Stop after the execution of Send payment request such that we cannot execute Produce extract and sign it.

We remove the succession(Process payment, Send e-mail to central desk), succession(Process payment, Print request), and succession(Send to DMS department, Indicate already paid) to prevent the deduction of redundant implicit constraints.

Figure 50: Top: model D, bottom: model F

Figure 50 contains model F and Figure 51 contains the configuration for model F. Similar to model D, we first set the isOmitted attribute for the event not present in model F to true. We, furthermore, remove the exclusive choice between Produce extract and sign it and Inform customer via telephone or e-mail to prevent the model from demanding that Produce extract and sign it is always executed. Finally, similar to the configuration of model D, we have to remove some succession constraints from the model in order to prevent the deduction of redundant implicit constraints.

### 7.4 Comparing the models

We cannot find any difference between the input and output models after combining and configuring the models in the way described above. We wanted to show that we can combine a set of Declare models into a Configurable Declare
model and by configuring this *Configurable Declare* model we wanted to obtain the input models. This is shown by the case study.

Figure 51: Top: configuration for model D, bottom: configuration for model F
8 Future work

Here, we identify a number of direction in which we can extend Configurable Declare. For each direction, we list the benefits when extending Configurable Declare with this extension but also, if applicable, possible problems one might encounter when exploring this direction.

Meta-constraints

We do not guide the user in configuring its Configurable Declare model, i.e. the user can configure a Configurable Declare model in such a way that it cannot be executed. Therefore, we want to extend Configurable Declare with meta-constraints. Meta-constraints are constraints between constraints or between tasks and constraints. These meta-constraints would allow the designer to restrict the configurability of a model such that the user can only model valid Declare models.

Consider, for instance, the model depicted in Figure 52 in which we have a not co-existence between task $B$ and response $(A, C)$ denoting that if task $B$ is present, we do not have response $(A, C)$ in our model.

![Figure 52: Example of a model with a meta-constraint](image)

Automatic rule discovery

The rules defined in Section 4.4 are defined both manually and automatically. Ideally one has an approach to deduce all rules in an automated fashion which is based on some formal notion. This automated approach can then later on be applied when one decides to extend Declare with new constraints.

One can design this automated approach in such a way that it automatically satisfies the soundness, completeness, and confluence properties. Soundness means that all rules are correct, while completeness means that we have discovered all of the rules. Finally, when the confluence property holds, this means that we can apply the rules in any order.

The main problem with automatic rule discovery is that it is far from trivial, consider, for instance, the model depicted in Figure 53. We have the following LTL formula to denote this model (chain response $(A, B)$ and chain response $(B, C)$):

$$\Box(A \Rightarrow \Diamond(B)) \land \Box(B \Rightarrow \Diamond(C))$$

We want to obtain the following formula denoting that whenever we execute an $A$ we immediately execute a $C$ afterwards.

$$\Box(A \Rightarrow \Diamond(C))$$
This means we have to find an approach which can remove the next operator from the LTL formula and obtain a formula no longer equivalent to the original formula, i.e. in the original formula we have the relation that if we execute an A, we first execute something else (B) and then execute C (\(\Box(A \Rightarrow \lozenge(\lozenge(C)))\)).

![Figure 53: Problematic model for automatic rule discovery](image)

**Solving alternate problem**

As mentioned before, we have a non-regular language when we apply the rules with the *alternate* templates, e.g. *alternate succession*. We can solve this problem by introducing a new binary constraint denoting what the maximal difference in executions can be. Assume we have two activities, A and B, and the maximal difference between the amount of executions of A and B is five. Then A can never happen six times in a row without a B in between.

In Figure 54, we would add the constraint that A can have at most a difference of two w.r.t. C. This means that after we have executed A two times we need to execute a C. We can encode this in LTL with the following formula: \(\Box(A \Rightarrow \lozenge((\neg A\lor C) \lor (A \Rightarrow \lozenge(\neg A\lor C))))\). Intuitively, whenever we have an A we either do not perform A until we perform C, or if we perform a second A, we do not perform an A until we perform a C and we keep on doing this until we encounter two Cs without an A in between.

![Figure 54: A model in which A cannot execute three times without the execution of C](image)

This approach only works in a non-branched setting: consider, for instance, the model in Figure 55. Within this model, after the removal of B2, we allow, amongst other, the traces: AAB1C, AACB1, AACC but not the trace AAB1B1. Therefore, we need to create distance equivalence classes, i.e. two events are in the same distance equivalence class if the same number of alternate response constraints are between A and the events. In our example, B1 is at distance one, and C is at distance two.

When an event is in a certain distance equivalence, it can respond to an amount of As proportional to its distance to A without the execution of a new A. B1 can respond to at most one A, while C can respond to two As. Basically, we are counting the amount of times an event from a distance equivalence class can happen to validate this constraint. Before A can happen a next time, we have to make sure that A did not happen more often than the events at the maximal distance to A, in the example two times. Furthermore, if A happened five times and afterwards the events in the distance equivalence class at distance one also happened five times, we still need to observe four times events from different distance equivalence classes.
Figure 55: A model in which, after the removal of $B_2$, we cannot pose the same constraint between $A$ and $B_1$ and between $A$ and $C$

**Full support for branched constraints**

We have seen in Section 4.4 that we have a problem when we have the case as depicted in Figure 56. We could solve it by allowing constraints similar to the constraint in Figure 57.

Figure 56: A model in which we cannot express the implicit constraints

![Diagram](image1.png)

Figure 57: The implicit constraints after the removal of the $B$s in Figure 56

![Diagram](image2.png)

We have the following LTL formulas for the different constraints, where $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ are sets of events: $\text{response}(A, B) = \Box((a_1 \lor \cdots \lor a_n) \Rightarrow \Diamond(b_1 \lor \cdots \lor b_m))$, $\text{alternate response}(A, B) = \Box((a_1 \lor \cdots \lor a_n) \Rightarrow \neg(a_2 \lor \cdots \lor a_n) \lor (\neg(a_1 \lor \cdots \lor a_n) \lor (b_1 \lor \cdots \lor b_m)))$, and $\text{chain response}(A, B) = \Box((a_1 \lor \cdots \lor a_n) \Rightarrow \Box(b_1 \lor \cdots \lor b_m))$. The different formulas only differ on the righthand side of the implication. We can support the new branched constraint by taking the disjunction of the righthand sides, this yields as LTL formula for the new constraint: $\Box((a_1 \lor \cdots \lor a_n) \Rightarrow (\Diamond(b_1 \lor \cdots b_m) \lor \neg(a_1 \lor \cdots \lor a_n) U(b_1 \lor \cdots b_m) \lor \Box(b_1 \lor \cdots b_m)))$.

The second problem we identified was the problem depicted in Figure 58. We needed a constraints between $A_1$, $A_3$ and the $B$s denoting that if $A_1$ or $A_3$ occurred less than 5 times we needed to execute a $B$.

60
We can extend the \textit{existence} templates with a new template which takes four arguments, basically consisting of the amount of times the events in a certain set of events should occur. When we write $\text{existence}'(nr, A, nr', B)$, this means that the events in $A$ should occur at least $nr$ times or the events in $B$ should occur at least $nr'$ times.

In the aforementioned problem, we can encode the implicit constraint as $\text{existence}'(5, \{A_1, A_3\}, 1, \{B_1, B_2, B_3\}) = \Diamond (B_1 \lor B_2 \lor B_3) \lor (\Diamond((A_1 \lor A_3) \land \Diamond(\text{existence}'(4, \{A_1, A_3\}, 1, \{B_1, B_2, B_3\})))).$ In which $\text{existence}'$ is recursively defined similar to the $\text{existence}$ template.

\section*{Support for more configuration patterns}

When we operate in a Branched Declare setting, we also want to extend the possibilities for configuring the models. One may think of removing a branch from a constraint instead of removing the entire constraint. If we extend Declare such that the different branches can have different semantics, one also wants to be able to change the semantics of some branches, similar to substituting a constraint in the non-branched Declare setting.

\section*{Extending the code}

Within the case study, we had to remove constraints not because they had to be removed but because this yielded nicer models. In the future, we will not only compute implicit constraints between omitted activities but also between activities which remain in the model. When we have computed all implicit constraints between activities, we can remove the redundant explicit constraints still in the model.

We currently have a command line program which, using command line arguments, performs the different conversions, e.g. from Configurable Declare to a Configuration. We use the Eclipse Modelling Framework (EMF) [Fra11] to generate the eXtensible Markup Language (XML) documents representing the models. We want to add a Graphical User Interface (GUI) to the program such that the users can use the GUI to design and configure their models similar to the Declare designer[Des11]. Within this GUI, it also should be possible to define new constraints and rules.
Perform multiple case studies

We have used Configurable Declare in a single case study where the models shared a large number of similarities. In order to see the full potential of Configurable Declare, we want to apply this approach to less homogeneous models. To apply Configurable Declare to those less homogeneous models, we first need to solve most of the problems addressed in this section, since less homogeneity means that we want to combine different constraints into a single constraint.

Apart from simply combining different models and then configuring them so that we obtain the original models again, we also want to apply our approach within a real organisation. This will allow us to observe how Configurable Declare would be used by people who are not familiar with Configurable Declare. Then, we can optimise Configurable Declare to better suit their needs and also remove bottlenecks if they exist.

Automatically configuring the model based on a log

[SM11] presents an approach to automatically validate, given a log and a model, which constraints hold non-vacuously. Non-vacuous satisfaction means that the value, i.e. true or false, of some subformulas of a formula does not change the validity of the entire formula. Consider, for instance, response(A,B) which means that every A is eventually followed by a B. In logs where A does not occur, this formula is always valid, i.e. every A is followed by a B.

We can take our Configurable Declare model and a log from an organisation and deduce which constraints hold non-vacuously on the log and should be kept in the model. The approach by [SM11] also takes into account whether certain events are present in the log, hence we can use this to automatically configure the events in the Configurable Declare model.

Automatic configuration to obtain a given model

To obtain the Declare models, used in the “combining models” algorithm, the user has to manually define the configurations on the Configurable Declare model. We want to define these configurations automatically, so given a Configurable Declare model and a Declare model, we want to obtain the configuration (if it exists) of the Configurable Declare model, which yields the given Declare model.

We first need to check whether the tasks in the Declare model are all present in the Configurable Declare model. If this is the case, we need to validate that all tasks not present in the Declare model can be configured to be omitted in the Configurable Declare model. We remove all tasks not in the Declare model from the Configurable Declare model using the rules defined earlier and obtain the Configurable Declare model. After we have validated whether the tasks can be omitted, we need to validate that the constraints can be configured as needed w.r.t. the constraints of the Declare model.

We compute all implicit constraints on the Configurable Declare model and the Declare model. If the (implicit) constraints of the Declare model are not a subset of the (implicit) constraints on the Configurable Declare model, we know some constraints are present in the Declare model but not in the Configurable Declare model. This means that we cannot define a configuration.
If the constraints of the Declare model are a subset of the constraints in the Configurable Declare model, we remove all constraints in the Configurable Declare model not present in the Declare model and which can be removed, i.e. omit is true. After removing these constraints from Configurable Declare, we obtain Configurable Declare.

By removing the constraints from the Configurable Declare model not present in the Declare model, we might also remove constraints necessarily to deduce implicit constraints present in the Declare model. In order to ensure that we did not remove an explicit constraint needed for this deduction, we recompute the implicit constraints on the Configurable Declare model and obtain the Configurable Declare model. Consider, for instance, the Declare and Configurable Declare model in Figure 59. If we remove the constraints not in the Declare model from the Configurable Declare model, we also remove the implicit responded existence between C and B which is explicit in the Declare model.

![Figure 59: Left: Declare model, right: Configurable Declare model](image)

Instead of a subset relation, we want to have an equivalence between the constraints, implicit and explicit, of the Configurable Declare model and the constraints, implicit and explicit, in the Declare model. If we have an equivalence, we also have the configuration, i.e. all tasks which are not in the Declare model have to be removed and all constraints not in the Declare model (either implicit or explicit) have to be removed. In case we do not have an equivalence, we cannot deduce a configuration, i.e. there are constraints in the Declare model which are not in the Configurable Declare model.

**Configuring the model based on a questionnaire**

In [LRLS+07], the authors propose an approach to configure a model by means of a questionnaire. Each question of the questionnaire is linked to a set of decision point and each decision point has a set of questions associated with it. A decision point is a construct of the language which can be altered, i.e. similar to the alteration of constraints.

We want to use such a questionnaire-based approach for Configurable Declare. This would allow the user of Configurable Declare to configure the model without any knowledge of Configurable Declare, and it would allow the designer of the Configurable Declare model to limit possible configurations. By limiting possible configurations, the designer can forbid configurations which lead to non-executable Declare models or which lead to Declare models which invalidate a legislation.
9 Conclusion

We have presented a solution to extend Declare to Configurable Declare. Configurable Declare allows for flexibility within a process model, i.e. everything is allowed unless stated otherwise. Configurable Declare also allows for flexibility of the process model itself, i.e. the process model can be changed to better cope with the environment in which it will be applied. This means that as well as having flexibility at run-time we also have flexibility at configuration-time, i.e. between design-time and run-time. After the process model has been designed but before the process model is used, we can adapt the process model towards company’s needs.

By supporting both notions of flexibility, we developed a process modelling language which is the preferred choice to be applied in environments with high variability. The application of Configurable Declare is not limited to environments with high variability. In every environment where rules or legislation must be obeyed, Configurable Declare is a candidate worth considering.

All the configuration patterns identified in literature, which are useful to consider, are supported by Configurable Declare. Unfortunately, we were not able to address all problems which occur when moving to Branched Declare. In the future work section (Section 8), we show that most of the issues which still exist can be easily addressed. Most of the patterns which are difficult to incorporate in imperative languages are almost trivial to support in Configurable Declare, especially the patterns which entail the flexibility within the process model.

Considering that Configurable Declare takes implicit constraints into account, we can use it to encode legislation. If the legislation is non-configurable, we know that, after transforming this Configurable Declare model into a Declare model, all legislation is still encoded in the Declare model. If the legislation is configurable, we can remove parts of the legislation and we have to be sure that this does not yield incomplete encoded legislation. Therefore, we opt for the creation of a questionnaire to guide the user in configuring the legislation for her organisation. By using this questionnaire, we can guarantee that all legislation is encoded properly in the Declare model.

The theoretical design is supported with both an implementation and a case study. With the implementation, we can perform the conversions, described in the thesis, in combination with the application of the rules. Using this implementation, we showed in the case study that our approach can be applied to real life declarative process models, whilst yielding the expected outcome.
References


Appendices

A Meta-models

Figure 60: Meta-model of Configurable Declare
Figure 61: Meta-model of a Configuration
B Input models

Figure 62: Input model A

Figure 63: Input model B

Figure 64: Input model C
Figure 65: Input model D

Figure 66: Input model E

Figure 67: Input model F
Figure 68: Input model G
C Configuration models
Figure 71: The configuration for model B

Figure 72: The configuration for model C
Figure 73: The configuration for model D

Figure 74: The configuration for model E
Figure 75: The configuration for model F

Figure 76: The configuration for model G