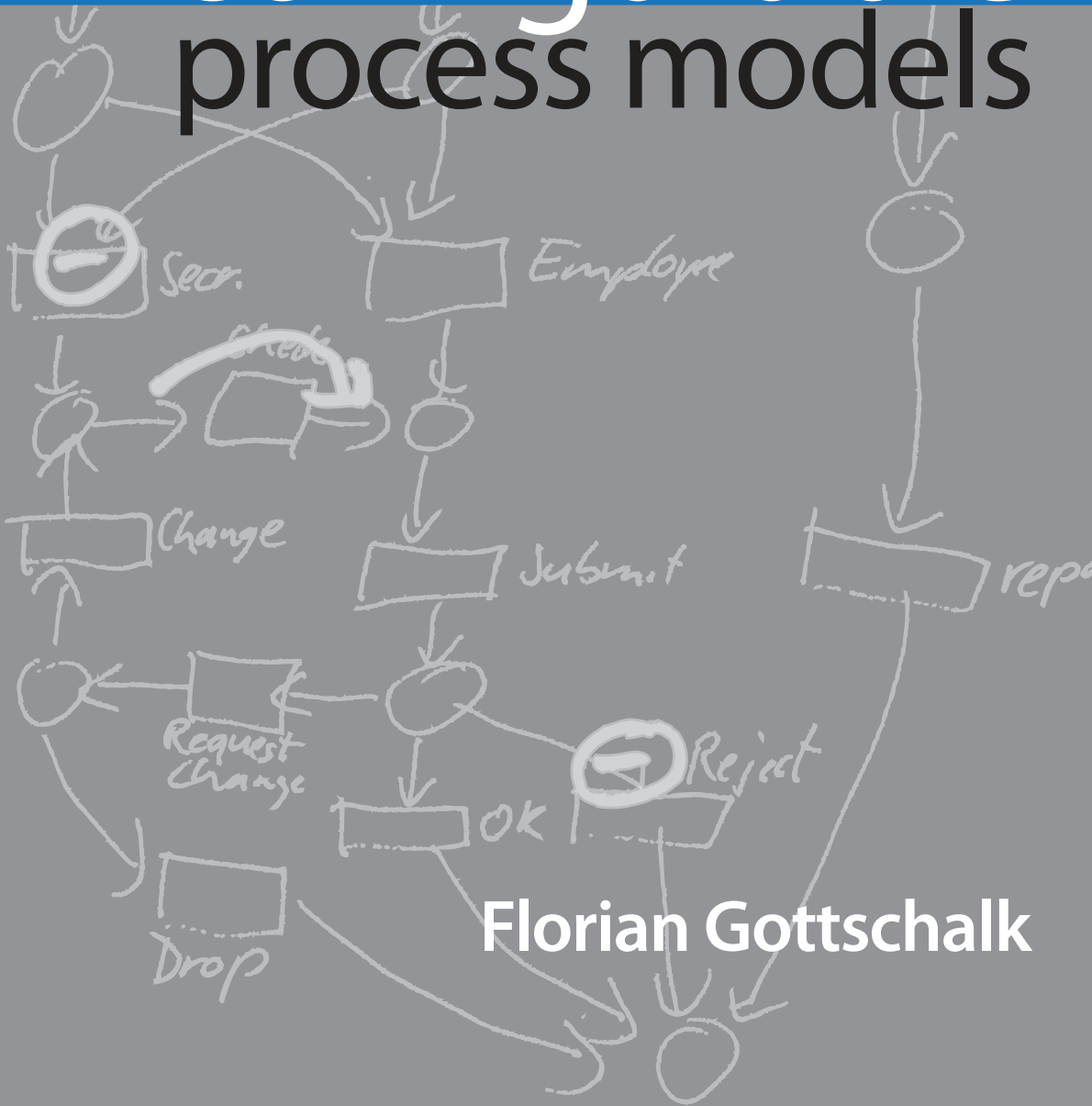


configurable process models



Florian Gottschalk

Configurable Process Models

Copyright © 2009 by Florian Gottschalk. All Rights Reserved.

A catalogue record is available from the Eindhoven University of Technology Library.

Gottschalk, Florian

Configurable Process Models / by Florian Gottschalk.
- Eindhoven: Technische Universiteit Eindhoven, 2009. - Proefschrift. -

ISBN 978-90-386-2085-5
NUR 982

Keywords:
Process Configuration / Reference Models / Workflow / Business Process Management

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

Beta Dissertation Series D124

Printed by University Press Facilities, Eindhoven
Cover design by Christian Gottschalk

Configurable Process Models

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 3 december 2009 om 16.00 uur

door

Florian Gottschalk

geboren te Göttingen, Duitsland

Dit proefschrift is goedgekeurd door de promotor:

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

Copromotor:

dr. M.H. Jansen-Vullers

Contents

1	Introduction	1
1.1	Process Model Reuse	3
1.2	Process Model Adaptation	6
1.3	Research Goal, Methodology, and Contributions	11
1.4	Road Map	12
2	Background Process Modeling	15
2.1	Preliminaries	16
2.2	Languages for Formal Process Definition	19
2.2.1	Labeled Transition Systems	19
2.2.2	Workflow Nets	21
2.3	Workflow Patterns	26
2.4	Business Process Modeling Languages	28
2.4.1	Event-driven Process Chains	28
2.4.2	Protos	32
2.4.3	BPMN	34
2.5	Workflow Languages	35
2.5.1	YAWL	36
2.5.2	SAP WebFlow	40
2.5.3	BPEL	42
2.6	Summary	44
3	Configuring Process Models	45
3.1	Configuration versus Inheritance	45
3.2	Adding Configuration to Process Modeling	52
3.2.1	Configuring Ports of Tasks	52
3.2.2	Restricting Configuration Opportunities	57
3.2.3	Configurable Process Models	59
3.3	Related Work	61
3.3.1	Literature Study on Variability Mechanisms	61
3.3.2	Studying of Adaptation Practices	62
3.3.3	Restricting Choices in Workflow Patterns	62
3.4	Conclusions	63

4	Configurable Workflow Languages	65
4.1	C-SAP WebFlow	66
4.1.1	Identifying Ports	66
4.1.2	Port Configuration	68
4.1.3	Configuration Constraints	70
4.1.4	Process Enactment	71
4.2	C-BPEL	72
4.2.1	Ports and their Configurations	72
4.2.2	Executability of BPEL Configurations	74
4.3	C-YAWL	76
4.3.1	Configurable Elements of EWF-Nets	76
4.3.2	Configuration Requirements and Validity	84
4.3.3	Components of C-EWF-Nets	87
4.3.4	Configurable Workflow Specifications	88
4.3.5	From C-YAWL to YAWL	91
4.3.6	C-YAWL Implementation	98
4.4	Related Work	98
4.4.1	C-EPCs	99
4.4.2	Further Process Configuration Extensions	101
4.5	Conclusions	102
5	Guiding the Configuration Process	105
5.1	Capturing Domain Variability	106
5.2	Capturing Process Variability	109
5.3	Linking Domain and Process Variability	110
5.4	Tool Support	115
5.5	Related Work	120
5.6	Conclusions	121
6	Configurable Process Models for Municipalities	123
6.1	Creating Configurable Process Models	124
6.1.1	Building the Models	124
6.1.2	Observations	133
6.2	Evaluation of the Approach	135
6.2.1	Provider of BPM Solutions	137
6.2.2	Provider of Municipality Software	137
6.2.3	Consultancy Firm	138
6.3	Related Work	139
6.4	Conclusions	140
7	Building the Configurable Process Model	143
7.1	Generating Configurable Process Models from Log Files	145
7.1.1	Pre-processing the Log Files	145
7.1.2	Mining the Basic Process Model	150
7.2	Merging Process Models	157
7.2.1	Function Graphs	159

7.2.2	From EPCs to Function Graphs	161
7.2.3	Merging Function Graphs	165
7.2.4	From Function Graphs to EPCs	169
7.2.5	Tool Support	172
7.3	Deriving Configurations	178
7.4	Case Study Re-visited	183
7.4.1	Mining Models from Log Files	183
7.4.2	Merging Individual Models	188
7.4.3	Identifying Individual Configurations	191
7.5	Related Work	194
7.5.1	Process Mining	194
7.5.2	Model Merging	195
7.5.3	Synthesis	196
7.5.4	Identifying Configurations and Conformance	197
7.6	Conclusions	198
8	Executability of Configurations	201
8.1	Preserving Syntactic Correctness	203
8.2	Preserving Semantic Correctness	211
8.3	Correctness in C-YAWL	214
8.4	Constraints from Resource- and Data-flows	215
8.4.1	Data-flow Correctness	216
8.4.2	Resource Availability	222
8.5	Related Work	223
8.5.1	Soundness from a Control-flow Perspective	223
8.5.2	Soundness from a Data-flow Perspective	224
8.6	Conclusions	225
9	Conclusions	227
9.1	Contributions	227
9.1.1	Process Model Configuration	228
9.1.2	Configurable Process Modeling Languages	228
9.1.3	Guiding Process Configuration	229
9.1.4	Model Merging	230
9.1.5	Soundness of Process Configuration	230
9.2	Limitations and Future Work	231
9.2.1	Adapting Configured Models	232
9.2.2	Configuration Performance	232
9.2.3	Configuration in the Process Life Cycle	233
9.2.4	Process Model Content	234
9.3	Summary	235

A Case Study Process Models	237
A.1 Acknowledging an Unborn Child	239
A.2 Registering a newborn	246
A.3 Marriage	253
A.4 Issuing Death Certificate	260
Bibliography	267
Index	285
Acronyms	309
Symbols	311
Summary	321
Samenvatting	323
Acknowledgements	325
Curriculum Vitae	329

*There is no art that doesn't reuse.
Lawrence Lessig (2001)*

Chapter 1

Introduction

Efficient and reliable processing of data is key in today's information society. Thus, a good organization of the tasks that need to be executed to transform input data into meaningful results is essential. The field that is concerned with the orchestration of individual tasks to executable processes is known as **Business Process Management (BPM)** [9, 12, 190]. To support BPM, process models visualize process behavior [53, 70, 159, 163]. By documenting current or future options available to handle data, by defining the execution order of tasks, as well as by depicting possible outcomes, process models help stakeholders when developing, implementing, executing, or improving business processes. Moreover, process models can also serve as instructions for information systems that support the data processing during the execution of business processes, known as **workflow management systems** [7, 123]. Here, they provide exact, formal specifications of how the various tasks can follow each other and how data is transformed.

Designing high-quality process models is often time-consuming, error-prone, and costly [4, 142]. It first requires identifying all the small steps that need to be taken to get to the required results. The task order as well as whether tasks can be executed in parallel or if they are alternatives to each other must be determined. Then the data and resources needed as input for tasks must be specified and the results produced by each step must be determined. The amount of time and effort needed obviously increases with the level of detail that needs to be specified for a process. While a brief overview of the main tasks may be sufficient for managers, workflow systems require that each single step is formalized in detail in order for it to be understood and automatically controlled by the system. Also, with an increasing level of detail, the risk for errors — especially undetected errors — rises. Hence, these efforts and risks must be in proportion to the efficiency and reliability improvements achievable when modeling a process.

Within the process landscape of corporations we can distinguish two main types of business processes [7], **primary processes** and **support processes**.

Primary processes are core processes which directly drive a company's success by giving it a competitive edge. Examples of such processes can be production processes, customer support, marketing, etc. These processes require innovation and differentiation from other market participants. In contrast, support processes such as verification of invoices, the approval of travel requests, or HR processes like the payments of salaries do not directly contribute to increasing a company's business. Here it is essential that the company can rely on effective and efficient process execution. Furthermore, unless a corporation is itself an outsourcing company specializing in the particular area, innovation in these processes will do little to promote corporate success. And yet failure in these processes can very well put the whole organization in jeopardy. Thus, for these non-core processes, reliability is far more important than innovation, and these processes are therefore good candidates for standardized solutions.

Vendors of so-called enterprise software or **Enterprise Resource Planning (ERP) systems** like IBM, Oracle, or SAP, pick up on this. Their products build up on databases that enable an integrated view on a company's data. By adding workflow support to the products, the provided solutions become capable of automatically handling a process's complete data [52]. For this, the products include both the software as well as documentation on how the various processes can be executed using the particular system. Thus, enterprise software vendors often provide both rather informal process models as documentation as well as technical templates which enable the process execution through their system's workflow engines [48, 139, 162]. Many consultancy firms have specialized in introducing such standard products. For this, they not only rely on the offerings of software vendors, but also provide their own so-called **reference models**. These models utilize industrial standards which the consulting firms derive from various previous implementations, i.e. through their experience in the particular field [167].

Still, while processes like travel approval or invoice verification are executed very similarly among organizations, small variations exist due to specific requirements. For example, issuing a payment authorization might require two or three separate approvals within some companies, while other companies grant officers in charge to issue them directly. Consultants implementing standard solutions in organizations therefore spend huge amounts of time on adapting a standard solution to the individual process requirements of organizations [51].

In this thesis, the aim is to simplify this adaptation process. Current process templates and reference models always depict 'best-practice' solutions to execute a business process. Thus, any deviation from the standard procedures requires a manual adaptation of the process model. Therefore, even small changes require process modeling experience and skills, and hence come with the corresponding risks.

To avoid these difficulties, we will suggest providing an integrated set of all the different process variants, i.e. a single process model covering all the various execution options, even enabling new combinations among them. Such an integrated process model should then enable model users to simply select desired and deselect undesired options. In this way, the process model can be

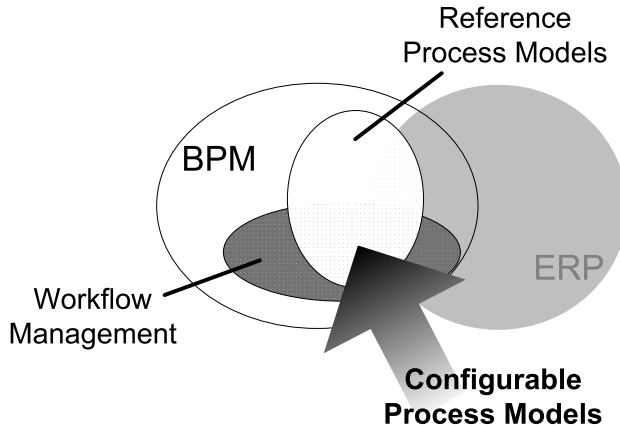


Figure 1.1: Configurable process models within the domains of BPM, Reference Process Modeling, Workflow Management, and ERP.

configured to a process variant containing only those parts of the integrated set of process variants which are required by the particular organization. All other options are eliminated from the model. Furthermore, the goal is to set up **configurable process models** such that they guarantee the correctness of the derived model. This means, a workflow engine should, e.g., be able to steer a business process according to any process variation that a user is able to derive from the configurable process model.

Summarizing, configurable process models enable reusing various existing process definitions by combining them and providing the model user with a choice for or against each individual alternative. Especially when applied in the intersection of the BPM domains of reference process modeling and workflow management (see Figure 1.1) their use promises a significant reduction of manual process modeling efforts for model users.

For that reason, let us continue this introduction to the development of configurable process models by taking a closer look at the reuse of process models (Section 1.1) and the choices and variation opportunities that are provided by process models (Section 1.2). Afterwards, Section 1.3 lists the research goal of this thesis, the research questions connected to this goal, the research method used to answer these questions, and the contributions made by answering each question. Section 1.4 then gives an outlook on the thesis' different chapters, linking them to the research questions posed in Section 1.3. In this way, it provides a road map for reading the thesis.

1.1 Process Model Reuse

The goal when reusing process models is to avoid 'reinventing the wheel', i.e. to avoid designing business process models which have already been well defined

and used by others.

The idea of reusing established artifacts has especially inspired research in the context of software engineering [35, 36, 69, 109]. From the simple idea of reusing software code in the late nineteen-sixties, software reuse has matured over the last forty years. Still, practitioners were quite reluctant to reuse existing software in new projects for a long time. This was mainly attributed to the ‘not invented here’ syndrome, i.e. a supposed unwillingness to use code written by others [177]. However, Favaro [65] discovered in the early nineteen-nineties that practitioners were actually quite willing to reuse existing software but they just found it far too hard because of the lack of sophisticated concepts and tools (or their lack of awareness thereof). With the success of inheritance concepts in object-oriented programming, software libraries, and design patterns, this has now changed significantly. Today, using such concepts, all software development builds on previously existing software, e.g., reusing user interfaces, database access, etc. [29, 68].

With the growing popularity of BPM during the nineteen-nineties, both researchers and practitioners soon discovered that reuse of process models could also significantly reduce process modeling costs [28, 30, 62, 95, 116, 169]. This has led to the development of a number of reference models and template repositories, providing process models that are considered ‘best-practice’ approaches for the particular business processes [66, 67, 161]. The most prominent example is probably the set of reference models provided by SAP which includes more than 600 non-trivial process models (and more than 3,000 models in total) documenting the processes of their enterprise system which has more than 100,000 installations worldwide [48, 155]. Such reference models also aim at being a better starting point for the development of process variants according to individual requirements than starting from scratch. Nonetheless, although there is plenty of literature on the existence of reference process models, it is hard to find documentation that shows how these models have significantly contributed to successful process implementation projects. For example, Daneva [51] claims that requirements engineering for enterprise system implementation projects is all about reuse, and she proposes starting with the process documentation offered by system vendors. However, after analyzing a total of 67 SAP implementation projects, Daneva also realized that adapting the standard solution to the individual needs often required far more work than initially anticipated, and the tools provided by the system vendors do not capture the impact of any changes made.

Besides the process models documenting the system behavior, SAP also delivers hundreds of simple, predefined process templates for the workflow system that is part of any installation of their enterprise system — from process templates for logistics and material management to personal time management, sales and distribution, or compensation management [139]. The templates, which typically can be printed comfortably on one A4 page, can easily be activated in the SAP system. They are then triggered automatically whenever their execution is required, without a process designer ever having spent a significant amount of time on the workflow definition.

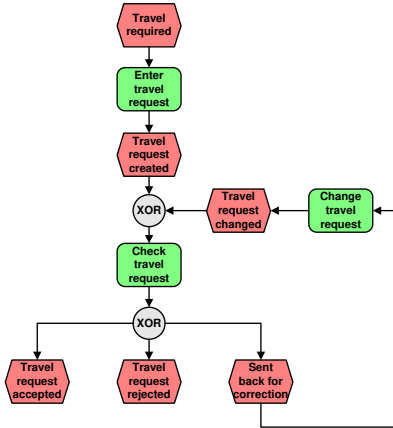


Figure 1.2: Approving travel requests manually (adapted from [156])

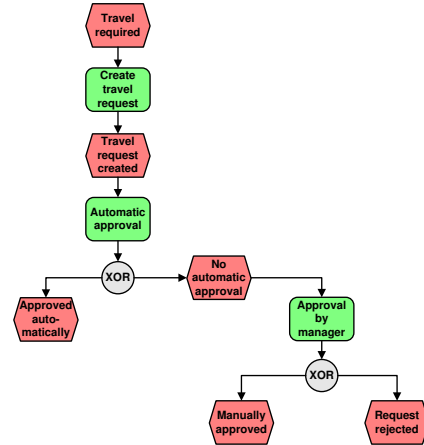


Figure 1.3: Approving travel requests automatically (adapted from [157])

As there are usually different options for how a process can be executed, SAP’s repository often even includes multiple template variants for the same business process, each variant suggesting a different implementation of the particular process. For example, there is a dedicated workflow template not only for the approval of a travel request (as shown in Figure 1.2), but also one for the automatic approval of a travel request (as shown in Figure 1.3). In addition, there are workflow templates for the approval of a travel plan, the approval of a trip, and the automatic approval of a trip.¹

For deciding on which process variant to implement, a process designer has to compare these templates. As all these templates are similarly structured, and as none of the differences between the two templates is highlighted, finding the small differences manually can be a difficult, and time-consuming task. If, as in the example from figures 1.2 and 1.3, there is a certain degree of inconsistency in the documentation of the templates because it is unclear if *Create travel request* and *Enter travel request* actually depict the same task, this comparison requires even further efforts.

It becomes even worse, if the process designer concludes that a combination of two templates would be the optimal solution as each template has its strength at a different point in the model. For example, the process designer might want to combine the automatic approval step from Figure 1.3 with the option to send the travel request back for correction from Figure 1.2. As such a template is not available, she then has to manually adapt one of the templates at its weak point to match the one not selected as closely as possible — an obviously unsatisfying solution for the designer who just wants to select necessary and deselect unnecessary options.

¹The manual travel approval is accessible in the workflow builder of SAP’s enterprise system as workflow template WS20000050, the automatic travel approval is accessible as workflow template WS12500021. All the various templates are documented at http://help.sap.com/saphelp_erp2005vp/helpdata/en/d5/202038541ec006e1000009b38f8cf/frameset.htm.

Similar issues can also be found when looking at other reference process model repositories [58]. Hence, comparable to the lack of reuse concepts and tools in software engineering projects in the early nineteen-nineties, reference modeling practice lacks sophisticated concepts and tools that support the reuse of process models.

1.2 Process Model Adaptation

To be able to improve the support for the adaptation of process models, it is first necessary to find out how a process model can be adapted. For this, Becker et al. [33, 34] developed a framework, categorizing adaptation techniques. They distinguish two main types of process model adaptation: **configuration mechanisms** and **generic adaptation mechanisms**. The essential difference between these two types of mechanisms is that configuration mechanisms are based solely on eliminating content that is already contained in the model while generic adaptation mechanisms also allow for adding content to the model as required when adding new steps or even simply re-directing the process flow, i.e. changing the source or target of corresponding process flow arcs. That means that any adaptation that requires ‘creative freedom’ is classified as a generic adaptation mechanism while if the adapted model is a subset of the original model, it is classified as a configuration mechanism.

To support generic adaptations, there are two approaches which provide dedicated adaptation support by providing partial models which have to be completed during the adaptation, called **aggregation** and **instantiation** [34]. In an aggregation approach, a library of process model building blocks is provided. These building blocks can be combined and nested, i.e. aggregated, to create complex process models (see Figure 1.4). Thus, the approach is similar to the provision of software libraries in software engineering. For example, Blin et al. [37], Han et al. [86], Kilov [106], and Koschmider and Blanchard [108] suggest frameworks for process model reuse utilizing aggregation mechanisms.

Instantiation provides the process designer with the inverse to aggregation. In other words, it provides a framework of process models which contain placeholders that need to be filled when adapting the process models (see Figure 1.5). This is similar to the concepts of abstract classes and interfaces in object-oriented programming, as well as to design patterns, which all deliver frames that have to be filled with details when deriving variants from them. Vom Brocke [38] and Becker et al. [34] outline such approaches further.

Process model configuration combines these two ideas for supporting reuse of process models. It eliminates all manual process modeling efforts while still providing options to adapt process models. To enable process model configuration, the overall framework needs to be defined (as for process instantiation), process model libraries need to be defined (as they are provided for process aggregation), as well as it is necessary to define how the placeholders in the framework can be filled from the process model libraries (see Figure 1.6). Without adding any components, such a process model can be changed in two ways. On the one

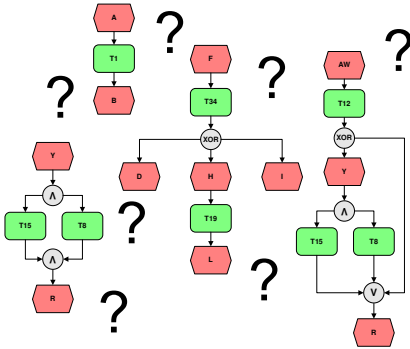


Figure 1.4: Aggregation of process model building blocks

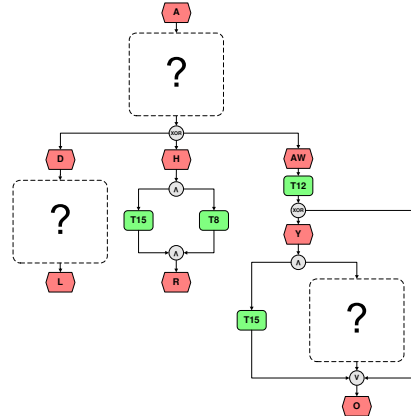


Figure 1.5: Instantiation of a process model

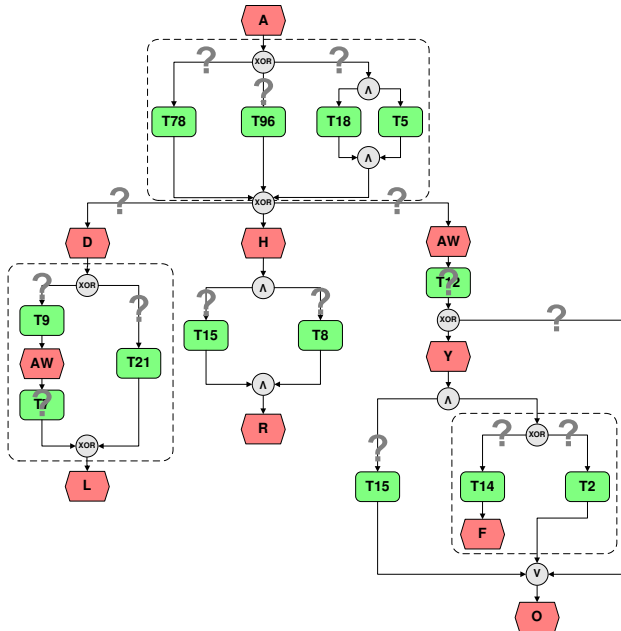


Figure 1.6: Process model configuration: selecting those elements that should be preserved in the model

hand, elements that are already in the model can be made invisible such that reading the model becomes easier, i.e. they remain in the process model such that the process model's behavior is preserved. On the other hand, elements can be eliminated completely from the model. Then, the behavior that is possible according to the model is restricted. If we want to change the possibilities how a process can be executed, we obviously need to restrict the behavior, i.e. do the

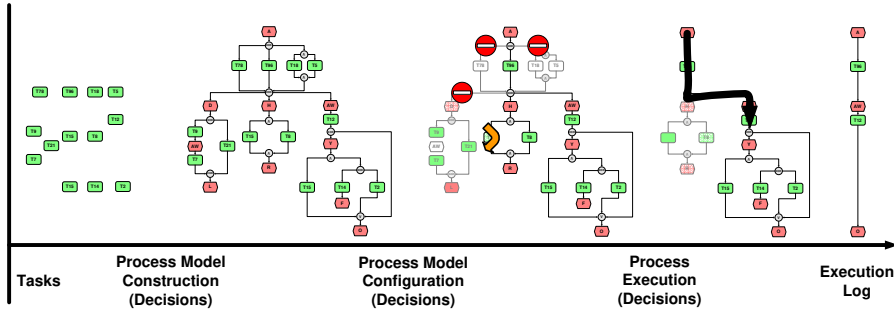


Figure 1.7: Process model construction, configuration, and execution: The possible process behavior is restricted with each decision.

latter. Hence, *process model configuration means restricting the behavior that is possible according to a process model (and thus observable when executing the corresponding process)*.

In this way, process configuration is an intermediate step between the construction of a process model and the execution of the process. To visualize this, let us have a look at Figure 1.7: By explicitly defining a process in a model, performing tasks arbitrarily is prevented. Therefore, constructing a process model means defining an execution order among the tasks as well as defining variation options that can occur when executing the process. After the process model has been constructed and enforced, tasks can no longer be executed freely, i.e. they must be executed according to the process model. The possible process behavior is restricted. Process configuration restricts this behavior further. It eliminates elements from the process model and thus forbids their execution. The decisions that still remain open are only made when executing the process: each decision determines which path in the process model is followed. These run-time decisions thus also eliminate (non-selected) paths from the possible process behavior of the particular process instance.

Hence, process model construction, process configuration, and process execution follow each other to determine the actually executed process behavior which is captured in the execution log. While in fact most decisions are made when constructing the process model, process configuration allows making further decisions on how processes are executed before they are enacted, and the decisions that remain open are so-called **run-time decisions** that are made while executing the process (see Figure 1.8). With each decision, the variation options that remain available during the process execution are reduced (see Figure 1.9).

Of course, a process model per se can also be executed without process configuration. Process configuration is added to the decision making process as an intermediate step to improve the support for reusing process models in related process execution scenarios as follows: The more process behavior is supported by a process model, the more it is applicable in various scenarios, i.e. it fits more applications [144, 145]. For that reason, a process model allowing for a lot of

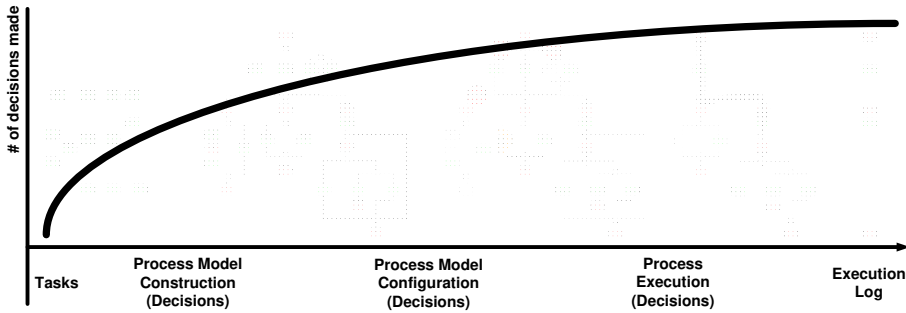


Figure 1.8: Decisions made: Constructing a process model means making plenty of decisions while the last decisions are made only shortly before the execution of a process instance completes.

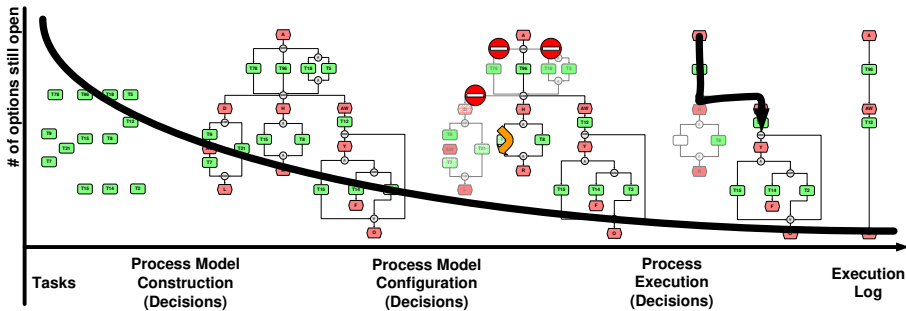


Figure 1.9: Variation opportunities: Before the process model is constructed, it is completely free which tasks can be executed in which order. This freedom decreases with each decision made (while decisions made early in the decision making process usually have the biggest impact on the further variation options).

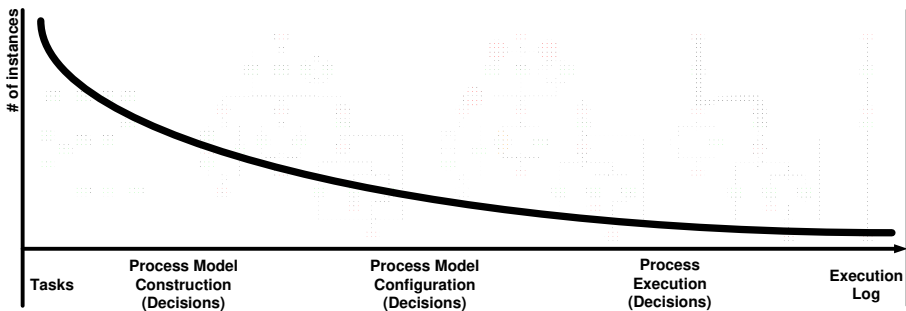


Figure 1.10: Number of process instances that behave according to the model: Obviously, the more the behavior possible according to a process model is restricted, i.e., the less variation opportunities remain in the model (Figure 1.9), the less process instances behave according to the restricted process model. While the constructed process model is applicable to a lot of process instances, configuration decisions reduce the amount of process instances covered by a process model. Finally, each execution log covers the behavior of a single process instance.

behavior enables more reuse. However, much of the behavior that is added to achieve this general applicability, is not really desired behavior from the viewpoint of an individual application [144, 145]. The single, individual application only requires a subset of this behavior. Thus, while winning a lot of reuse potential through fitting to far more applications, adding behavior to a process model makes the model at the same time less appropriate to the individual application, reducing its reuse potential. However, by eliminating undesired behavior through process configuration, the appropriateness of a process model can be increased. *Hence, process configuration aims at increasing the reuse opportunities of process models on the one hand through allowing additional behavior in a process model, and on the other hand by providing a tool to tailor this additional behavior to what is really needed* (see Figure 1.10).

Process model configuration must therefore be clearly distinguished from the issues that occur when a process model has been changed, and process instances that are being executed at the moment of the change should use the new process model from that point in time on. Systems tackling these problems of switching from one ‘configuration’ to another are also often called configurable, re-configurable or adaptive workflow systems (e.g. in [42, 64, 86, 87, 99, 140, 175]). However, these approaches typically neglect the preceding problem of how the process model itself can be easily and safely changed, which is our focus here.

The idea of extending process modeling languages with opportunities for process model configuration has been picked up by several researchers. For example, Rosemann and van der Aalst [143] suggest the use of specific configurable node types which allow including or excluding process behavior during process execution. Both Becker et al. [34] and Czarnecki and Antkiewicz [49] propose to include or exclude model elements based on attributes assigned to these elements, while Puhlmann et al. [132] advocate adding conditions to decision nodes and deciding based on evaluating these conditions if a certain process part can be executed. Dreiling et al. [60] provide a set of general configuration patterns, depicting which options to limit the process behavior exist in certain execution scenarios.

All these approaches identify the particular configuration options by observing how process models are adapted or varied in practice, i.e. the suggested configuration options conform to the observed practical needs. However, this means that completeness of the offered options cannot be proven or assumed, especially given that the practical success of process model reuse achieved up to now is limited.

When suggesting process model configuration, efficiency benefits for the implementation of enterprise systems are often anticipated. Still, none of the approaches listed above shows the applicability of process configuration in this context through using a toolset that derives a configured process model which is executable in a classical workflow system. In fact, none of the approaches demonstrates process model configuration based on a process modeling language that is designed to support the automated process execution through a workflow system.

Therefore, in this thesis we aim on the one hand at providing a sound foundation for process model configuration by analyzing process behavior. And on the other hand we aim at using this knowledge to create configurable process modeling languages which contain everything that is needed to even configure process models that have the power to steer process executions in workflow engines.

1.3 Research Goal, Methodology, and Contributions

The goal of the research presented in this thesis can be summarized as follows:

Improve the support for process model reuse by defining process model configuration such that it enables tailoring process models which are applicable in many contexts to the behavior desired in an individual context. The resulting models should be usable to enforce the individually desired behavior.

To achieve this goal, we need to answer the following, connected research questions:

- What is process configuration and what are configurable process models?
- How can existing process modeling languages be extended with configuration options?
- Is it possible to define configurable process models such that users without process modeling skills can adapt these models to individually desired behavior?
- Are configurable process models practically feasible, i.e., is their complexity manageable and do organizations see benefits in using configurable process models?
- Which challenges arise when constructing configurable process models and how can they be addressed?
- Which challenges arise when executing configured process models and how can they be addressed?

A design science² research approach as, e.g., outlined by Hevner et al. [91], was used to answer these questions. The main contributions made in this way are:

- a sound and complete definition of what process configurations means using the assumption that process configuration is the inverse of adding behavior to a process model (Chapter 3, Section 3.1),

²While natural and behavioral science explains how and why things are like they are, design science attempts to create things that serve human purposes. Thus, while behavioral science focuses on discovering and justifying new things, design science focuses on building and evaluating new things [91, 101, 117].

- a methodology to setup configurable process modeling languages (Chapter 3, Section 3.2),
- the formal definition of a configurable process modeling language which is also of practical use (Chapter 4, Section 4.3), demonstrated by a case study (Chapter 6),
- a framework that enables adapting process models without any process modeling knowledge, i.e. one that is based on domain specific questions expressed in natural language (Chapter 5, Section 5.3),
- an algorithm for merging process models while preserving the individual behaviors (Chapter 7, Section 7.2), and
- the definition of a propositional formula which — as long as it evaluates to true — preserves the soundness of a free-choice process model while eliminating elements from the model (like we do when configuring the model; Chapter 8, Sections 8.1/8.2).

The main results from this thesis have also been published in international journals and key conferences in the field of information systems and BPM [16, 17, 73, 74, 75, 76, 77, 78, 79, 113].

1.4 Road Map

The thesis is divided into nine chapters and an appendix.

Chapter 1 (this chapter) explains the need for configurable process models in the context of reference process modeling, presents the main research questions, and outlines how these questions will be addressed throughout the thesis in a design science research approach.

Chapter 2 provides necessary background information by introducing the formal notations as well as the process modeling languages used throughout the thesis. For this, formal notions like Labeled Transition Systems and Petri nets, as well as business process modeling languages like Event-driven Process Chains (EPCs), Protos, and BPMN, and workflow languages like SAP WebFlow, YAWL, and BPEL are introduced.

Chapter 3 identifies how process configuration can restrict process behavior. To do this, it first looks back on how behavior is added to process models, and defines process configuration afterwards as the inverse. Then, in the second part, Petri nets are used to show how a configurable process modeling language can be built based on a basic process model, i.e. a traditional process model which integrates the process behavior of related process variants, a set of configuration constraints that restrict the configuration space, and a default configuration which defines a starting point for process configuration.

Chapter 4 shows how advanced process modeling notations used by practitioners to enable automatic workflow executions can be extended with configuration options. While in Chapter 3 Petri nets are used to discuss

the theory of process configuration, Chapter 4 informally suggests configuration extensions for SAP WebFlow, and BPEL. For the ‘configurable YAWL’ language even a formal specification is provided, and a transformation algorithm shows how process specifications, which are executable in the YAWL workflow engine, can be derived based on the configuration decisions.

Chapter 5 proposes a framework which allows steering process configuration decisions through a natural language questionnaire. For this, the answers given in the questionnaire must be directly or indirectly mapped onto one or more process configuration decisions. The framework’s implementation is demonstrated through a running example, showing each step from answering the questionnaire to getting to a completely configured and executable YAWL model. In this way, even subject matter experts who lack process modeling knowledge can derive individual process model variants from a configurable process model.

Chapter 6 outlines a case study in which configurable process models were built for four common registration processes of municipalities (getting married, registering being the father of a not-yet-born child, registering a newborn child, issuing a death certificate). The configurable process models, which are built in YAWL, incorporate the process variations occurring among four Dutch municipalities and a reference model. Each variant as well as further variants can be derived by answering a natural language questionnaire using the toolset from Chapter 5 and is then executable in the YAWL workflow engine. The chapter also reports on interviews performed with stakeholders in the application of these models, i.e. a software provider for municipality models, the provider of the software used for business process modeling by most of the Dutch municipalities, as well as by various consultants.

Chapter 7 discusses the integration of various process variants into a single process model, i.e. the construction of a configurable process model’s basic process model. Building the integrated model is far more complex than building a traditional process model because the basic process model contains the behavior of several process variants (which obviously increases complexity). For that reason, Chapter 7 suggests techniques that can help in building such models: On the one hand, existing process mining techniques can build a basic process model if log files of various existing systems executing the process in question are available. On the other hand, an algorithm is presented which is capable of merging process models directly while preserving the behavior of the individual models. In addition, the chapter depicts a way to identify configurations of existing systems in the basic process model by replaying log files. This can, for example, help in finding default configurations or dependencies among configurations. The chapter ends by briefly showing how the depicted techniques could be used in the context of the case study process models from Chapter 6.

Chapter 8 discusses constraints that can be imposed on the configuration of process models to preserve correctness of the process model during process configuration. By restricting the control-flow of process models, process configuration can easily inhibit more process behavior than desired — up to the point that the process model is not a correct process model anymore at all. Besides these control-flow issues, the data-flow of processes can be impaired when eliminating process behavior because, for example, tasks which are eliminated by process configuration create data which is necessary for tasks preserved in the process model. The constraints suggested in Chapter 8 guarantee the absence of the mentioned issues.

Chapter 9 concludes the thesis by summarizing the contributions made for creating configurable process models. As future research directions the identification of preferred configurations, the adaptation of process configurations within the life cycle of process models, and the need for creating good content for configurable process models are suggested.

Appendix A provides all the process models created during the case study outlined in Chapter 6.

Readers familiar with the aforementioned process modeling languages, may prefer to use Chapter 2 solely as a reference guide to the formal definitions used throughout the thesis. In addition to this, each of the subsequent chapters 3–7 provides a section that discusses work related to the issues addressed in the particular chapter.

The core ideas for process model configuration are provided in Chapter 3. Chapters 4–6 focus on using process configuration to enact process model executions in practice. Chapter 7 addresses the issues arising in the interplay with the phase preceding process configuration in the process model life cycle, i.e. the process model construction, while Chapter 8 addresses issues arising in the interplay with the succeeding phase, i.e. the process execution. In this way, chapters 7–8 discuss issues in a more theoretical way than chapters 4–6. Still, readers interested in process configuration practice should not simply skip chapters 7 and 8 as the discussed issues are practically very relevant.

*No thought exists without a sustaining support.
Mel Bochner (1970)*

Chapter 2

Background Process Modeling

Process models are used to define and depict which tasks need to be performed when executing a business process as well as ordering constraints among these tasks. Various languages have been developed to support the modeling of processes in different contexts. In this thesis, we distinguish three categories of process modeling languages, depending on their main application area.

First, there are a number of well-defined languages for the **formal specification of processes**, which are mainly used in academia for depicting and formally proving various assumptions and characteristics of process modeling. These languages provide the foundations for advanced process modeling languages in the two other categories.

Business process modeling languages, as the second category of process modeling languages, provide practitioners with the opportunity to quickly draw the process flow as a basis for discussions, as well as to provide documentation that is easily understandable by stakeholders of the process. Thus, they usually abstract from implementation details.

Workflow models, as the third category of process models discussed here, enable the enactment of the depicted processes through Information Technology (IT). They thus enrich the business process model with additional, precise information required by information systems to execute the process. Similar to the formal languages, workflow notations thus need well-defined execution semantics. Moreover, they need to include information like which resources are authorized to perform certain tasks of the process or on how to handle the input and output of the various activities.

While languages for formal process definitions precisely depict all potential changes of parameters of the overall state of the process, business process modeling notations and workflow languages try to combine commonly jointly occurring changes of state parameters, so-called **workflow patterns**, into ex-

explicit modeling constructs. In this way, the depiction of larger processes becomes clearer.

To formalize and thus unambiguously define the various languages and their application throughout this thesis, this chapter starts with some preliminaries introducing the formal notations and concepts used later on. Afterwards, Section 2.2 introduces and formally defines **Labeled Transition Systems (LTSs)** and **workflow nets** as two languages for formally defining processes. Section 2.3 gives a very brief overview of workflow patterns which are usually addressed through explicit modeling constructs in business process modeling and workflow languages. In Section 2.4, **Event-driven Process Chains (EPCs)**, **Protos**, and the **Business Process Modeling Notation (BPMN)** are introduced as examples for business process modeling languages while Section 2.5 presents three examples for workflow languages, namely **Yet Another Workflow Language (YAWL)**, **SAP WebFlow**, and the **Business Process Execution Language (BPEL)**. EPCs and YAWL are notations with an academic background and we will also provide formal definitions for these languages. The modeling notations of Protos and SAP WebFlow were both developed for commercial tools while BPMN and BPEL are open standards developed for the particular modeling purpose.

2.1 Preliminaries

There are several mathematical notations used for defining concepts throughout the following chapters. This section therefore gives an overview on the notations used.

For reasoning based on the properties of the introduced concepts we will use **propositional logic**. Each statement we make about such properties is called a **proposition** or **propositional formula** and has a **truth value**, i.e. it can be *true* or *false*. For example, a statement could be ‘The car is red’. Usually, we will use a **propositional letter** like p or q for referring to a certain statement. An **atomic formula** is a proposition of only one propositional letter. Using **logical operators**, propositions can be combined to more complex propositions.

Definition 2.1 (Logical Operators) *Let p and q be two propositional statements. Then:*

- \bar{p} is the **negation** of p , i.e. \bar{p} is true iff p is false,
- $p \wedge q$ depicts the **conjunction** of p and q which is true iff both p and q are true,
- $p \vee q$ depicts the **disjunction** of p and q which is true iff either p , or q , or both p and q are true,
- $p \dot{\vee} q$ depicts the **exclusive disjunction** of p and q which is true iff either p , or q is true, but not both of them,
- $p \Rightarrow q$ depicts that p implies q which is false iff p is true while q is false (**implication**),

- $p = q$ depicts that p equals q , i.e. both p and q have the same truth value (**equivalence**).

A **literal** is an atomic formula or its negation. Any propositional formula can be brought into the specific structures of the **conjunctive normal form** and the **disjunctive normal form**.

Definition 2.2 (Conjunctive Normal Form) A propositional formula that consists of a conjunction of clauses where each clause is a disjunction of literals is in conjunctive normal form. No further logical operators are allowed.

Definition 2.3 (Disjunctive Normal Form) A propositional formula that consists of a disjunction of clauses where each clause is a conjunction of literals is in disjunctive normal form. No further logical operators are allowed.

Thus, a proposition $(p_1 \vee p_2) \wedge (\overline{p_3} \vee p_4)$ is in conjunctive normal form while a proposition $(p_1 \wedge \overline{p_2}) \vee (p_3 \wedge \overline{p_4}) \vee (p_5 \wedge p_6)$ is in disjunctive normal form.

To define categories of elements and their relationship, we will use the mathematical concepts of **sets of elements**, **functions**, and **sequences**.

Definition 2.4 (Set) A set is a collection of distinct elements.

- $s \in S$ expresses that an element s is contained in a set S ,
- $S = S_1 \cup S_2$ depicts that the set S is the union of two sets S_1 and S_2 , i.e. S contains all elements of S_1 and S_2 ,
- $S = S_1 \cap S_2$ depicts that the set S is the intersection of two sets S_1 and S_2 , i.e. S contains those elements that are contained in both sets S_1 and S_2 ,
- $S = S_1 \setminus S_2$ expresses that the set S contains those elements that are contained in S_1 but not in S_2 , i.e. all elements that are contained in S_2 are removed from S_1 ,
- $|S|$ represents the number of elements that are contained in S ,
- $S \subseteq S_1$ denotes that S is a subset of S_1 , i.e. if all the elements of S are contained in S_1 ,
- $S \subset S_1$ denotes that S is a proper subset of S_1 , i.e. $S \subseteq S_1 \wedge S \neq S_1$,
- $S = S_1 \times S_2$ is the cartesian product of two sets, i.e. $S = \{(s_1, s_2) | s_1 \in S_1 \wedge s_2 \in S_2\}$,
- $\mathcal{P}(S) = \{S_1 | S_1 \subseteq S\}$ is the powerset of S , i.e. the set of all subsets of S ,
- \emptyset denotes the empty set, i.e. the set without any elements, and we assume that $\emptyset \subseteq S$ holds for all sets S .

Definition 2.5 (Function) Let X and Y be two sets. Then $f : X \rightarrow Y$ is a function that maps the elements of X onto Y , i.e. for all $x \in X$ holds that $f(x) \in Y$, where the application of the function f to the element x is denoted as $f(x)$. For a function $f : X \rightarrow Y$ we call $\text{dom}(f) = X$ the domain of f and $\text{rng}(f) = \{f(x) | x \in \text{dom}(f)\}$ the range of f .

Definition 2.6 (Partial Functions) A partial function $f : X \dashrightarrow Y$ is a function that is only defined for a subset of X , i.e. $\text{dom}(f) \subseteq X$.

By assigning natural numbers to elements of a set, we can create a **multi-set**, i.e. a set that can contain multiple elements of the same type:

Definition 2.7 (Multi-set) A multi-set is a function $Z : S \rightarrow \mathbb{N}$ mapping the elements of S to the natural numbers. A set is a special case of a multi-set where $\forall s \in S : Z(s) = 1$. The sum of two multi-sets $Z_1 = S_1 \rightarrow \mathbb{N}$ and $Z_2 = S_2 \rightarrow \mathbb{N}$ is denoted as $Z_3 = Z_1 \uplus Z_2$ such that $Z_3 : S_1 \cup S_2 \rightarrow \mathbb{N}$ where for all $s \in S_1 \cup S_2$ holds that $Z_3(s) = Z_1(s) + Z_2(s)$, while their difference is denoted as $Z_3 = Z_1 \setminus Z_2$ such that $Z_3 : S' \rightarrow \mathbb{N}$, $S' = \{s \in S_1 \mid Z_1(s) - Z_2(s) > 0\}$, and for all $s \in S'$ holds that $Z_3(s) = Z_1(s) - Z_2(s)$.¹ An element s is part of a multi-set Z , i.e. $s \in Z$, iff $Z(s) > 0$. The size of a multi-set $Z : S \rightarrow \mathbb{N}$ is defined as $|Z| = \sum_{s \in S} Z(s)$. $B(S)$ denotes all multi-sets over S .

While (multi-)sets are not sorted, the elements of a (multi-)set can be arranged in sequences.

Definition 2.8 (Sequence) Let S be a set of elements. A sequence $\sigma \in S^*$ is a sequence of the elements of S , where S^* is the set of all sequences composed of zero or more elements of S . We use $\sigma = \langle s_0, s_1, \dots, s_n \rangle$ such that $\forall_{0 \leq i \leq n} s_i \in S$ to denote a sequence. $\langle \rangle$ denotes an empty sequence, $+$ concatenates sequences, and \triangleleft denotes sub-sequences, i.e. $\sigma \triangleleft \sigma'$ if and only if there exists $\sigma_{pre}, \sigma_{post} \in S^*$, such that $\sigma' = \sigma_{pre} + \sigma + \sigma_{post}$.

To work with sequences, let us furthermore define how functions can be applied to sequences, the length of a sequence, the elements of a sequence, and a filter for sequences that allows eliminating elements from a sequence.

Definition 2.9 (Operations on Sequences) Let S, S' be sets of elements, $f : S \rightarrow S'$ be a function, and $\sigma \in S^*$ be a sequence such that $\sigma = \langle s_0, s_1, \dots, s_n \rangle$. Then $f : S^* \rightarrow S'^*$ is a function such that $f(\sigma) = \langle f(s_0), f(s_1), \dots, f(s_n) \rangle$, i.e. f is applied to all elements of σ ; $|\sigma| = n + 1$ is the length of σ ; and $s \in \sigma$ if and only if $\exists_{0 \leq i < |\sigma|} s_i = s$. Moreover, if $S' \subseteq S$, then we define $\pi_{S'} : S^* \rightarrow S'^*$ as a filter such that $\pi_{S'}(\sigma) = \langle s'_0, s'_1, \dots, s'_m \rangle$ is the sequence σ with $m \leq n$ and without those elements $s \in \sigma$ for which $s \notin S'$.

Throughout this thesis we will use various graph-based process modeling notations. Thus, let us first define the concept of a **graph** which is composed of a set of **nodes** that are connected through a set of **directed edges**. Edges thus depict the directions in which how one can ‘move’ between nodes.

Definition 2.10 (Graph) Let N be a set of nodes and $E \subseteq N \times N$ a set of (directed) edges. We say that $G = (N, E)$ is a graph.

As the nodes of a graph can be connected through edges, we can clearly identify the nodes from which a certain node can be reached through following a single edge, i.e. the nodes that are in the **pre-set** of the node, as well as those nodes

¹ $Z_1(s)$ is assumed to be 0 if $s \notin S_1$ and $Z_2(s) = 0$ if $s \notin S_2$.

that can be reached when following one of the edges originating from the node, i.e. the nodes that are in the **post-set** of the node.

Definition 2.11 (Pre-set, post-set) Let $G = (N, E)$ be a graph and $n \in N$. Then $\overset{G}{\bullet}n = \{m \in N \mid (m, n) \in E\}$ denotes the set of input nodes to n (pre-set), and $n\overset{G}{\bullet} = \{m \in N \mid (n, m) \in E\}$ denotes the set of output nodes of n (post-set) with respect to G . If the context is clear, we simply write $\bullet n$ and $n\bullet$. Furthermore, if $n \notin N$ we say $n\overset{G}{\bullet} = \emptyset$ and $\overset{G}{\bullet}n = \emptyset$.

Moreover, the ‘movement’ from one node to another along a set of edges and via a set of further nodes in between describes a **path** in the graph.

Definition 2.12 (Path) Let $G = (N, E)$ be a graph. Let $a, b \in N$. Then a path from a to b is a sequence of nodes denoted as $\langle n_1, n_2, \dots, n_k \rangle$ with $k \geq 2$ such that $n_1 = a$ and $n_k = b$ and $\forall_{i \in \{1..k-1\}} (n_i, n_{i+1}) \in E$. Moreover, we denote the set of all paths of G as E^* , i.e. $\langle n_1, \dots, n_k \rangle \in E^*$ if and only if $\forall_{1 \leq i < k} (k_i, k_{i+1}) \in E$.

2.2 Languages for Formal Process Definition

Basically, a process model describes the variation among the behavior that occurs during the execution of a business process. Thus, it depicts which tasks can be executed when, as well as the possible outcomes of the task executions. Formal process definition languages depict all the different ways of how the overall state of a process execution can change. Let us in the first part of this section have a look at Labeled Transition Systems (LTSs), a graph notation which depicts state changes in a direct way by using an explicit node for each and every state of the process. In the second part, we look at a variant of Petri nets called workflow nets. Workflow nets do not explicitly depict every state but rather the properties of states and the changes among these properties. Besides the syntax of workflow nets, we will also provide formal semantics for workflow nets and use them to define which workflow nets depict sound behavior and which behavior cannot be considered as sound.

2.2.1 Labeled Transition Systems

The graph notation of **Labeled Transition Systems (LTSs)** provides one of the most simple and direct ways to depict the behavior of a business process. For example, let us have a look at the simple travel approval process shown in Figure 2.1. The process is started when an employee of a company needs to do a business trip, i.e. she needs to travel. In this situation, she can either file a travel request or she can directly book the trip herself. If she has filed a travel request, the administration can either refuse it or it can approve it and book the trip. In the first case, it can be silently dropped, or it can be re-filed and subsequently be evaluated again. In case a trip was booked, it needs to be paid before the process is completed.

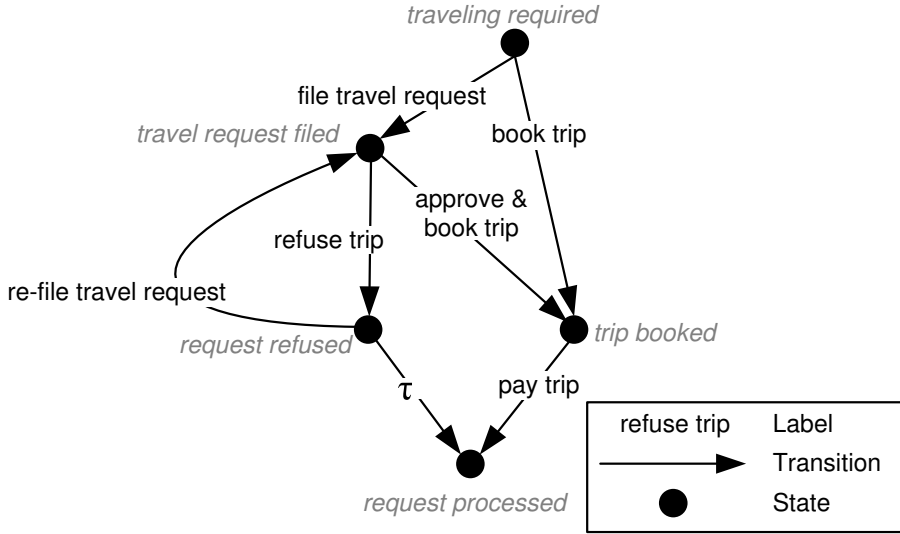


Figure 2.1: A simple travel approval process depicted as Labeled Transition System.

Thus, within an LTS the graph nodes represent **states** like *trip booked* while the edges represent the possible changes of states. A state therefore represents a complete, distinct set of properties which describes the actual situation of an execution of the business process. The edges represent the **transitions** from one state to another. A **transition label** like *refuse trip* is used to describe the cause of the state change. Hence, the actual tasks that are performed during the execution of a business process are represented by one or more transitions, depending on how many different outcomes the execution of the task can have. If more than one transition originates from a state, then there is a choice in which way the process continues. A **silent transition**, labeled τ , is a special transition that transforms a state into another without changing any of the externally visible properties of the state. This means, the state change is not triggered by an execution of a concrete task. Note that in *request refused* the transition *re-file travel request* can still be executed, while in *request processed* no further transitions can be executed. Thus, although τ transitions are not visible they may limit the possible ways a process can continue.

When formally defining LTSs we furthermore distinguish a set of initial states, which depict which states trigger the execution of the process, as well as a set of final states which are those states which depict a successful termination of the process. Hence, if a process deadlocks in a state, i.e. it cannot continue via any further transitions, and this state does not belong to the set of final states, the process execution will be considered as unsuccessful.

Definition 2.13 (Labeled Transition System) A labeled transition system is a five-tuple $LTS = (S, L, T, S_I, S_F)$, where

- S is the set of states,

- L is the set of transition labels,
- $\tau \in L$ is the label reserved for silent transitions,
- $T \subseteq S \times L \times S$ is the set of transitions,
- $S_I \subseteq S$ is the set of initial states, and
- $S_F \subseteq S$ is the set of final states.

LTSs provide a straightforward way to depict simple business processes. LTSs however have the drawback that they require a separate node in the graph for each state the overall process can be in, i.e. for each combination of a process's properties. It is therefore impossible to depict processes with large state spaces in a readable way. For this, we thus require more advanced notations. Nonetheless, it is important to note that any such advanced notation can be mapped onto LTSs [71, 120].

2.2.2 Workflow Nets

Petri nets are a notation allowing for more compact representations of processes. Petri nets are graphs distinguishing two types of nodes: **places** depicted as circles and **transitions** depicted as rectangles. Instead of whole states, a place of a Petri net only represents a property of the process. Thus, Petri nets only require a place for each property and not for each combination of properties like LTSs.

A state change might imply switching several properties. Therefore, in a Petri net the transition from one state to another state cannot be depicted by a simple edge like in an LTS. Instead, transitions of Petri nets are represented by a second node type and depict the changes of properties of the process. To depict which properties must hold before the corresponding transition can execute a state change and which properties will hold afterwards, **arcs** connect places to transitions and transitions to places. Like in LTSs, we use **transition labels** to denote what causes the particular change of properties and use a special label τ for depicting property changes that are not caused by any concrete task execution.

Definition 2.14 (Petri net) *A Petri net is a five-tuple $PN = (P, T, A, L, l)$, such that:*

- P is a finite set of places,
- L is the set of transition labels,
- $\tau \in L$ is the label reserved for silent transitions,
- T is a finite set of transitions ($P \cap T = \emptyset$),
- $l : T \rightarrow L$ assigns labels to transitions,
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).

A Petri net is a graph with $P \cup T$ as the set of nodes and A as the set of edges.

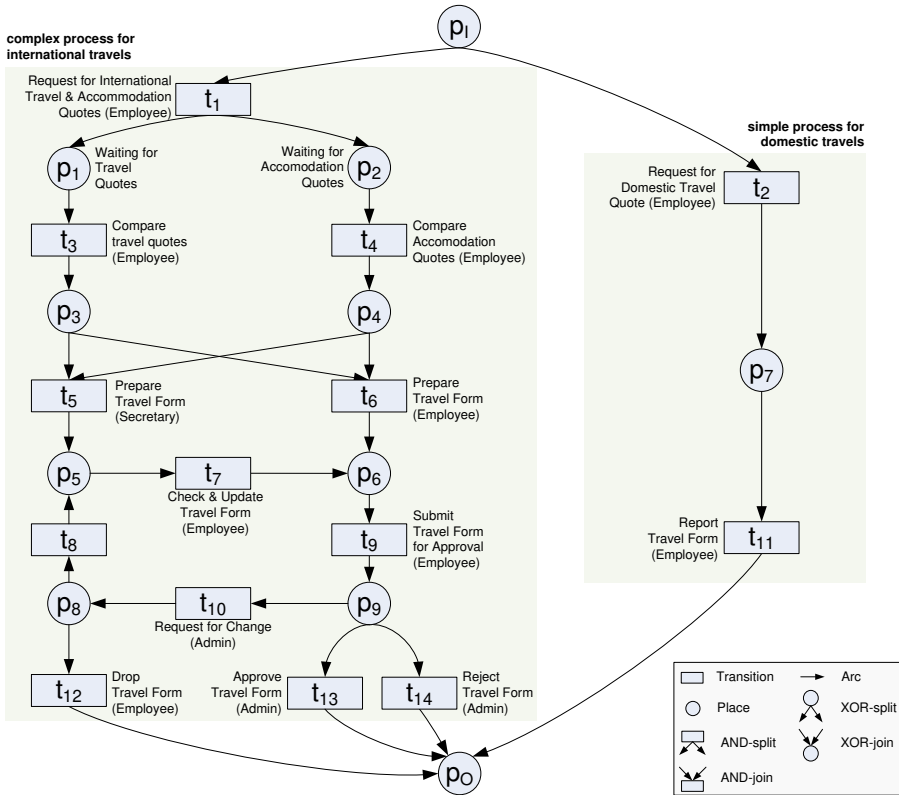


Figure 2.2: A travel approval process distinguishing simple and complex approvals depicted as Petri net.

Figure 2.2 shows a Petri net of a travel approval process which depicts in detail the preparation of a travel request. It incorporates two variants of the travel approval process: a complex one for international travels on the left and a simple one for domestic travels on the right. After requesting quotes for an international travel, the employee has to decide on both the travel quotes as well as the accommodation quotes, and after that either the employee herself or a secretary prepares the travel requisition form. In case the assistant prepares the form, the employee needs to check the form before submitting it for approval. The administrator can then approve or reject the requisition, or make a request for a change. At this point, the employee can update the form according to the administrator's suggestions and re-submit it, or drop the case. In contrast, the application for domestic travel only requires the employee to ask for a quote and to report the travel requisition to the administration.

Such a business process may be executed a number of times to deal with different cases like different travel requests. Each case has a unique identifier and is usually handled in isolation from other cases. We thus also say that a

case is an **instance of the process**.

To deal with instances of business processes in Petri nets, we require that they have a clearly defined starting point and ending point (to mark the completion of the process). To achieve this, we require that a Petri net which represents a business process has a unique **source place** representing the start or input of the process, and a unique **sink place** representing the process's completion, i.e. its output. All transitions and all other places must then be on a directed path between these two places. Otherwise, a property represented by a place or a transition which is not on such a path would either not be reachable from the start of the process and thus not be able to contribute in any way to completing the process, or the place signaling the completion of the process's execution would not be reachable from this place or transition and hence the node would not contribute to completing the process either.

A Petri net representing a business process and satisfying these conditions is known as a **workflow net** [3]. The net in Figure 2.2 is an example of a workflow net.

Definition 2.15 (Workflow net) *Let $PN = (P, T, A, L, l)$ be a Petri net. PN is a workflow net iff:*

- *there exists exactly one $p_I \in P$ such that $\bullet p_I = \emptyset$, and*
- *there exists exactly one $p_O \in P$ such that $p_O \bullet = \emptyset$, and*
- *for all $n \in P \cup T$, $\langle p_I, \dots, n \rangle \in A^*$ and $\langle n, \dots, p_O \rangle \in A^*$.*

Let furthermore Δ be the set of all such workflow nets.

To explicitly depict the state of a process, places of Petri nets can be marked with tokens indicating that the corresponding property holds. Petri net places can be marked with multiple tokens to depict that a property holds multiple times. This is, e.g., useful if a place indicates the number of copies of forms that are filled-in and a subsequent task requires multiple copies for further processing.

The **marking** of places also indicates which transitions are **enabled** in the current situation, i.e. which transitions can be executed. This enabling of a transition depends on the marking of its preceding places with tokens. Only if each of the places preceding the transition is marked with a token, the transition can be executed or **fire** as we say for Petri nets.

When a transition fires, the marking of the Petri net changes as follows. The transition removes one token from each preceding place and puts one token into each succeeding place.

Definition 2.16 (Marking, enabling rule, firing rule) *Let $PN = (P, T, A, L, l)$ be a Petri net:*

- *$M : P \rightarrow \mathbb{N}$ is a marking of PN and $\mathbb{M}(PN)$ is the set of all markings of PN ,*
- *$M(p)$ returns the number of tokens in place p if $p \in P$,*
- *For any two markings $M, M' \in \mathbb{M}(PN)$, $M \geq M'$ iff $\forall_{p \in P} M(p) \geq M'(p)$,*

- For any transition $t \in T$ and any marking $M \in \mathbb{M}(PN)$, t is enabled at M , denoted as $M[t]$, iff $\forall p \in \bullet t \ M(p) \geq 1$. Marking M' is reached from M by firing t and $M' = M - \bullet t + t \bullet$,
- For any two markings $M, M' \in \mathbb{M}(PN)$, M' is reachable from M in PN , denoted as $M' \in PN[M]$, iff there exists a firing sequence $\sigma = \langle t_1, t_2, \dots, t_n \rangle$ leading from M to M' , and we write $M \xrightarrow{\sigma}_{PN} M'$. If $\sigma = \langle t \rangle$, we use the notation $M \xrightarrow{t}_{PN} M'$. $\Phi_{PN} = \{\sigma \mid M \xrightarrow{\sigma}_{PN} M' \wedge M, M' \in \mathbb{M}(PN)\}$ denotes the set of all such sequences. PN can be omitted if sufficiently clear from the context.

To simulate the execution of a process in this way, we require for workflow nets, that initially only the source place p_I is marked while we consider a case to be completed as soon as the final (sink) place p_O is marked with a token.

Definition 2.17 (Initial marking, final marking) Let $WF = (P, T, A, L, l)$ be a workflow net:

- M_I is the initial marking of WF with one token in place p_I , i.e. $M_I(p_I) = 1$ and $\forall p \in P \setminus p_I : M_I(p) = 0$, and
- M_O is the final marking of WF with one token in place p_O , i.e. $M_O(p_O) = 1$ and $\forall p \in P \setminus p_O : M_O(p) = 0$.

Every started execution of a case should at some point in time complete with no further work to do, i.e. with the sink place p_O being marked. Therefore, it must be possible to reach M_O from any marking that is reachable from the initial marking M_I . If a workflow net satisfies this requirement, it is guaranteed that the process will never run into a deadlock, i.e. into a state where no further behavior is possible although the sink place is not yet marked. Moreover, the requirement guarantees the absence of livelocks in the net. That means, the process contains no states from which it can continue to fire transitions arranged in a cycle, but from which it can never reach the final marking. Also, it should only be possible to mark p_O when no other places are marked as it will lead to confusion if the completion of a case is signaled by a marking of p_O while work on the case is still in progress signaled by a marking of any other place .

In the definition of workflow nets we already required that each transition should potentially be able to contribute to the completion of the process and thus required that it should be on a path between p_I and p_O . However, simply being on such a path does not necessarily mean that a transition is able to fire at some point in time, e.g. because two of its preceding places can be marked on their own but never at the same time. Thus, for a transition being really able to potentially contribute to the process, there should be at least one execution sequence from the initial marking to the final marking that includes at least one firing of this transition.

A workflow net fulfilling these requirements is **sound**, i.e. semantically correct [3].

Definition 2.18 (Sound workflow net) Let $WF = (P, T, A, L, l)$ be a workflow net and M_I, M_O be its initial and final markings. WF is sound if and only if:

- *option to complete:* for every marking M reachable from M_I , there exists a firing sequence leading from M to M_O , i.e. $\forall_{M \in WF[M_I]} M_O \in WF[M]$, and
- *proper completion:* the marking M_O is the only marking reachable from M_I with at least one token in place p_O , i.e. $\forall_{M \in WF[M_I]} M \geq M_O \Rightarrow M = M_O$,
- *no dead transitions:* every transition can be reached by the initial marking, i.e. $\forall_{t \in T} \exists_{M \in WF[M_I]} M[t]$.

We indicated earlier in this section that any of the process modeling notations we use throughout this thesis can be mapped onto LTSs. The LTS that corresponds to a workflow net can be generated through analyzing the markings of the workflow net that are reachable from its initial marking through the firing of transitions. These markings correspond to the states of the LTS. Each transition of the LTS corresponds to a firing of a Petri net transition. Formally:

Definition 2.19 (LTS from workflow net) *Let $WF = (P^{WF}, T^{WF}, A^{WF}, L^{WF}, l^{WF})$ be a workflow net, M_I its initial marking, and M_O its final marking. The LTS corresponding to WF is defined as $LTS = (S^{LTS}, L^{LTS}, T^{LTS}, S_I^{LTS}, S_F^{LTS})$, where*

- $S^{LTS} = WF[M_I]$,
- $L^{LTS} = L^{WF}$,
- $T^{LTS} = \{(M, n, M') \in S^{LTS} \times L^{LTS} \times S^{LTS} \mid \exists_{t \in T^{WF}} (M \xrightarrow{t}_{WF} M' \wedge l^{WF}(t) = n)\}$,
- $S_I^{LTS} = \{M_I\}$, and
- $S_F^{LTS} = \{M_O\}$.

Petri nets and workflow nets benefit from a rich body of theoretical results, analysis techniques, and tools. Moreover, Petri nets have been extensively applied for the formal verification of process models. Further details on the research on Petri nets can, e.g., be found in the work of Murata [122], Peterson [127], and Reisig and Rozenberg [138] while van der Aalst [3] and Verbeek et al. [182] provide details on the verification of process models using workflow nets.

Many business process modeling languages and workflow languages use building blocks which can be mapped onto a sub-class of workflow nets called **free-choice** [3, 122]. A Petri net is free-choice if for every couple of places sharing transitions in their post-set, these post-sets coincide.

Definition 2.20 (Free-choice Petri net) *Let $PN = (P, T, A, L, l)$ be a Petri net. PN is free-choice if $\forall_{p_1, p_2 \in P \setminus p_O} [p_1 \bullet \cap p_2 \bullet \neq \emptyset \Rightarrow p_1 \bullet = p_2 \bullet]$.*

As free-choice workflow nets have some desirable properties which have let to the development of efficient analysis techniques for this class of Petri nets, the restriction to free-choice nets provides often a good compromise between expressiveness and verification complexity. Further details on free-choice nets can be found in the work of Desel and Esparza [54].

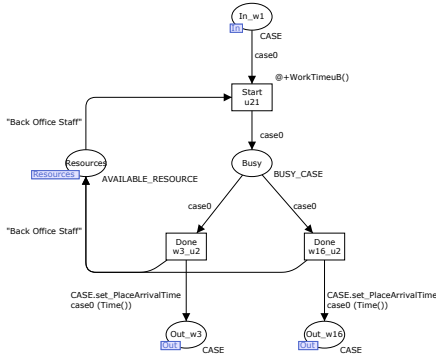


Figure 2.3: XOR-split

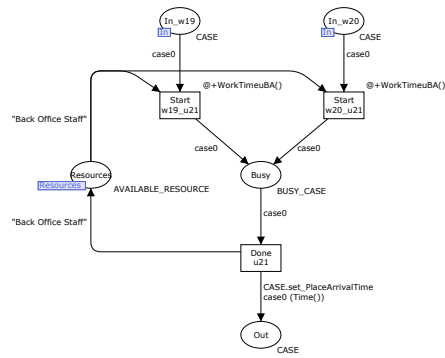


Figure 2.4: XOR-join

2.3 Workflow Patterns

In practice, the execution of a task usually does not refer to a single state change, but at least to two state changes as the execution of a task usually takes time. That means, first the task changes from being enabled to being executed, and later on the state changes from the task being executed to being completed. Furthermore, some tasks can be triggered from varying states or, depending on the outcome of the task, result in different states after different executions. To capture common combinations of such state changes, van der Aalst et al. [11] created a list of common **workflow patterns**.

For example, let us have a look at Figure 2.3. It models the execution of a particular task in terms of a **colored Petri net**. Colored Petri nets extend ordinary Petri nets (as introduced in Section 2.2.2) with data and time. In this way, the tokens of colored Petri nets can carry data values while running through the process. These values can be evaluated and changed by transitions. Furthermore, the consumption of tokens by transitions can be delayed until a later point in time by adding a time-stamp to tokens which determines their earliest possible re-use.

The task in Figure 2.3 is enabled as soon as a case token is put into the task's *In* place. In addition, the task requires someone from the back office to actually process the case. This is depicted through the second incoming arc from the left into the *Start* transition which can then trigger the task's execution. In this way, the staff member from the back office becomes busy with executing the task for a certain period, indicated at the upper right of the *Start* transition by adding a delay of a certain work time to the token ($@+workTimeuB()$). During this time the resource is not available for any other work.

As soon as the execution time has passed, the token in the *Busy* place can, like in a traditional Petri net, be removed by one of the two *Done* transitions. These transitions release the back office staff involved in the task back to the pool of resources, and put the case token into the corresponding *Out* place from which it can be processed further. Hence, there are two possible outcomes of the execution of the task in Figure 2.3, indicated by the two *Out* places. This

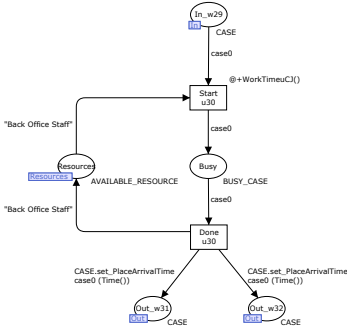


Figure 2.5: AND-split

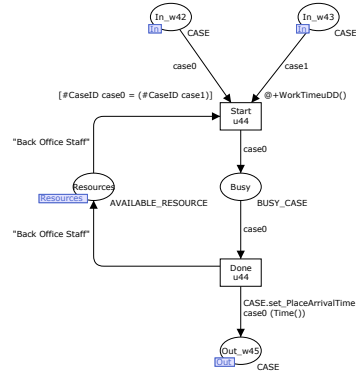


Figure 2.6: AND-join

selection of one out of several possible process continuations is known as an **exclusive choice** or **XOR-split** [11].

Similarly, it is often possible that a task can be started from several different places. For example, in Figure 2.4 several *In* places can be marked with tokens. As each of these places is followed by its own *Start* transition, a case that arrives in one of these *In* places can immediately be processed, i.e. the task can immediately be started, as soon as the necessary resource from the back office is available. This workflow behavior is known as a **simple merge** or **XOR-join** [11] of the incoming process branches as any arriving case token is then processed in an identical way.

The behavior of XOR-splits and XOR-joins can clearly be distinguished from the behavior of transitions in Petri nets, which require all their preceding places to be marked to fire, and which after firing mark all subsequent places. A task that triggers all subsequent paths by marking all the outgoing places is depicted in Figure 2.5. This behavior is also known as **parallel split** or **AND-split** [11], as it allows the parallel execution of all the subsequent paths while the XOR-split triggers only one of them exclusively. Figure 2.6 shows a task that waits until all its *In* places are marked with tokens of the same case before it can be executed, known as **synchronization** or **AND-join** [11] of the incoming process branches. This is ensured by the condition (which is in colored Petri nets called a **guard**) depicted at the upper left of the *Start* transition as $\#caseID\ case0 = (\#caseID\ case1)$. By requiring identical case identifiers, a system is able to distinguish multiple cases which are executed at the same point in time.

XOR-splits, XOR-joins, AND-splits, and AND-joins are four of the most basic **workflow patterns**, describing behavioral blocks commonly required when defining business processes. More advanced patterns are, e.g., the **multi-choice** (also called **OR-split**), the **synchronizing merge** (also called **OR-join**), and **cycles** or **loops** of process behavior [11]. The **multi-choice** depicts how a subset of several process branches can be triggered. The **synchronizing merge** shows how the behavior from various incoming process branches can be synchronized without the requirement that really all preceding branches are executed.

Loops enable repetition of behavior.

Workflow patterns therefore systematically depict common constructs which are required or desired to describe a business process with a process model. Hence, advanced process modeling languages, like those which we will discuss in the remainder of this chapter, often provide dedicated modeling constructs for a number of workflow patterns.

Van der Aalst et al. [11] provide a list of 21 of such workflow patterns. These patterns have been broken down into further, detailed patterns by Russell et al. [151]. All the tasks depicted in figures 2.3–2.6 use just one resource of a specific type and cannot start until this resource is available, while in practice often more advanced selection criteria for resources are applied. Elaborate descriptions of such resource patterns can be found in the work of Russell et al. [149, 150]. In his PhD thesis, Russell [148] specifies all these patterns as colored Petri nets, thus providing a formal foundation of how all the patterns should behave.

Readers interested in more details on modeling colored Petri nets and using them for simulating and analyzing complex systems like workflow management systems should have a look at the work of Jensen et al. [98] and Vinter Ratzter et al. [184].

2.4 Business Process Modeling Languages

Business process modeling languages were developed to enable practitioners to depict the flow of business processes in a consistent manner. In combination with a toolset, they provide a way for easily drawing process models. Usually, the toolsets allow annotating the process models with additional details like resources or data requirements, as well as they provide some basic methods for analyzing the models. We will discuss three particular notations here: Event-driven Process Chains (EPCs), the process modeling language of Protos, and the Business Process Modeling Notation (BPMN). EPCs were developed in an academic environment and are nowadays supported by various popular tools, like Microsoft Visio or ARIS from IDS Scheer. Protos is a commercial business process modeling tool with its own (Petri-net-like) notation, which is very popular in the Netherlands. BPMN aims at being the standard notation for business process modeling and is more and more supported by various process modeling tools.

2.4.1 Event-driven Process Chains

Event-driven Process Chains (EPCs) were developed by Keller et al. [103] in the early 1990s in a collaboration between SAP AG and the University of Saarland. Different from Petri nets, EPCs use three node types to depict the control-flow of a business process: **functions**, **events**, and **connectors**. Functions correspond to the tasks that need to be performed when executing the process. The execution of a function depends on the occurrence of its preceding events. A completed function then causes the occurrence of one or several

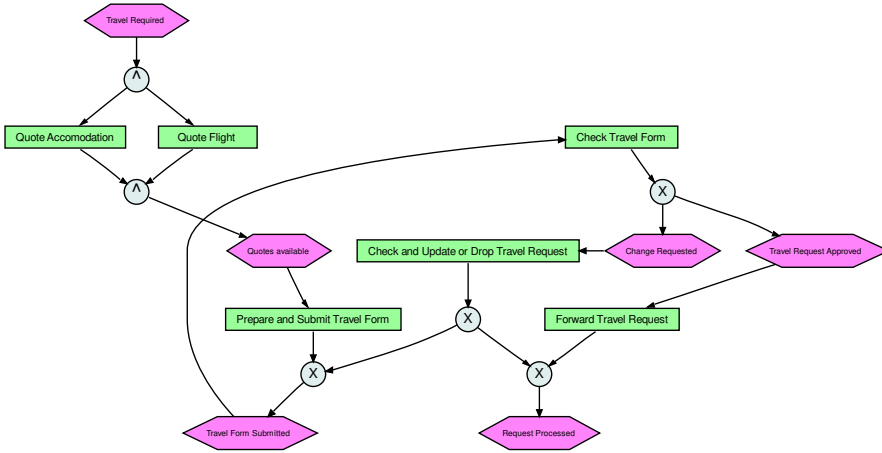


Figure 2.7: An EPC of a travel approval process

succeeding events. If several events precede or succeed a function, these events are not connected directly to the function but via a connector, the third node type of EPCs. The type of a connector then specifies explicitly which events are required to execute a function or which events are triggered after the function’s execution. EPC connectors can be of type \wedge , *XOR*, or \vee . An \wedge connector requires that all its preceding events need to be triggered for the succeeding function to be executed (AND-join) or that all the succeeding events are triggered after its preceding function completes (AND-split). An *XOR* connector depicts that the execution of a succeeding function requires the triggering of one of its preceding events only (XOR-join) or that only one of the succeeding events is triggered after the preceding function completes (XOR-split). The \vee connector specifies that a certain number of events is required or triggered by a function which can also vary from case to case. In this way, the \vee connector, e.g., allows that two out of three succeeding events are triggered.

For an example EPC let us have a look at Figure 2.7 which depicts again a travel approval process similar to the complex one we have seen as a Petri net (Figure 2.2, p. 22). When the need for a travel arises, the process first requires quoting both an accommodation and a flight. Both quotes need to be available before the process can pass the subsequent \wedge connector and can continue with the preparation and submission of the travel form. After the form has been checked, the request can either be accepted or a change can be requested — as indicated by the *XOR* connector subsequent to the *Check Travel Form* function. If a change is requested, a choice exists if the form is either updated or dropped. If it is updated, it re-joins the control-flow as if it would have been submitted as a new travel form. If it is dropped, there is no need for further processing. If the travel request is accepted, it is forwarded to the clearing center and the processing of the request finishes as well.

The following formal EPC definitions are in line with the definitions of Rosemann and van der Aalst [143]:

Definition 2.21 (EPC) *An Event-driven Process Chain is a five-tuple $EPC = (E, F, X, m, A)$:*

- E is a finite (non-empty) set of events,
- F is a finite (non-empty) set of functions,
- X is a finite set of connectors,
- $m \in X \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each connector onto a connector type,
- $A \subseteq (E \times F) \cup (F \times E) \cup (E \times X) \cup (X \times E) \cup (F \times X) \cup (X \times F) \cup (X \times X)$ is a set of arcs.

EPC is a graph with $E \cup F \cup X$ as the set of nodes and A as the set of edges.

Although Definition 2.21 already shows that arcs of an EPC cannot connect two events or two functions directly, the definition still permits to, e.g., connect two events via a connector. However, the idea behind EPCs is that events trigger the execution of functions and the completion of functions triggers again events. This way, functions and events have to alternate on any path of an EPC. Hence, a **well-formed EPC** has to guarantee this. Moreover, Definition 2.21 does not yet ensure that EPCs have to start and end with events or that connectors are the only nodes that are permitted to branch and synchronize the control-flow. We therefore have to define further requirements which need to be satisfied by well-formed EPCs. To formalize them, we introduce some additional notations first.

Definition 2.22 (EPC connector types) *Let $EPC = (E, F, X, m, A)$ be an Event-driven Process Chain.*

- $X_{join} = \{c \in X \mid |\bullet c| \geq 2\}$ is the set of join connectors,
- $X_{split} = \{c \in X \mid |c\bullet| \geq 2\}$ is the set of split connectors,
- $X_{EF} \subseteq X$ such that $c \in X_{EF}$ if and only if there is a path $p = \langle n_1, n_2, \dots, n_{k-1}, n_k \rangle$ such that $n_1 \in E$, $n_2, \dots, n_{k-1} \in X$, $n_k \in F$, and $c \in \{n_2, \dots, n_{k-1}\}$,
- $X_{FE} \subseteq X$ such that $c \in X_{FE}$ if and only if there is a path $p = \langle n_1, n_2, \dots, n_{k-1}, n_k \rangle$ such that $n_1 \in F$, $n_2, \dots, n_{k-1} \in X$, $n_k \in E$, and $c \in \{n_2, \dots, n_{k-1}\}$,
- $X_{EE} \subseteq X$ such that $c \in X_{EE}$ if and only if there is a path $p = \langle n_1, n_2, \dots, n_{k-1}, n_k \rangle$ such that $n_1 \in E$, $n_2, \dots, n_{k-1} \in X$, $n_k \in E$, and $c \in \{n_2, \dots, n_{k-1}\}$,
- $X_{FF} \subseteq X$ such that $c \in X_{FF}$ if and only if there is a path $p = \langle n_1, n_2, \dots, n_{k-1}, n_k \rangle$ such that $n_1 \in F$, $n_2, \dots, n_{k-1} \in X$, $n_k \in F$, and $c \in \{n_2, \dots, n_{k-1}\}$.

We can now restrict the sizes of both the set of input nodes and the set of output nodes to ensure that each event is at maximum preceded by one input node and at maximum succeeded by one output node as well as to ensure that each function has exactly one input node and one output node. Furthermore, we can use these sets to require that each well-formed EPC needs at least one start event that is not preceded by any other node and one end event that is not succeeded by any other node.

Connectors must have at least one input and one output node as they are always located between functions and events or vice versa. They can also have several input nodes or several output nodes, but we require that they do not have both multiple input nodes and multiple output nodes at the same time. We hence require that the sets X_{join} and X_{split} partition the set of connectors X into on the one hand a set of connectors which splits up the control-flow, and on the other hand a set of connectors which joins the control-flow. Finally, we use directed paths to limit the set of routing constructs using connectors. That means, we prevent that paths exist which connect two events or two functions only via connector nodes in between and instead require that connectors are always located on paths from events to functions or from functions to events. Altogether, well-formed EPCs can be formalized as follows:

Definition 2.23 (Well-formed EPC) *A well-formed Event-driven Process Chain (E, F, X, m, A) satisfies the following requirements:*

1. *The sets E , F , and X are pairwise disjoint, i.e. $E \cap F = \emptyset$, $E \cap X = \emptyset$, and $F \cap X = \emptyset$,*
2. *for each $e \in E$: $|\bullet e| \leq 1$ and $|e\bullet| \leq 1$,*
3. *there is at least one event $e_{start} \in E$ such that $\bullet e_{start} = \emptyset$,*
4. *there is at least one event $e_{end} \in E$ such that $e_{end}\bullet = \emptyset$,*
5. *for each $f \in F$: $|\bullet f| = 1$ and $|f\bullet| = 1$,*
6. *for each $c \in X$: $|\bullet c| \geq 1$ and $|c\bullet| \geq 1$,*
7. *X_{join} and X_{split} partition X , i.e. $X_{join} \cap X_{split} = \emptyset$ and $X_{join} \cup X_{split} = X$,*
8. *X_{EE} and X_{FF} are empty, i.e. $X_{EE} = \emptyset$ and $X_{FF} = \emptyset$,*
9. *X_{EF} and X_{FE} partition X , i.e. $X_{EF} \cap X_{FE} = \emptyset$ and $X_{EF} \cup X_{FE} = X$.*

Up to today, the semantics of EPCs are ambiguous as the semantics of its OR-join permits for different interpretations due to its non-locality. Details of this are, e.g., discussed by Kindler [107] and van der Aalst [1]. We will not contribute to this discussion here. Instead, we will always assume the same semantics for EPCs that is used in relevant previous work, i.e. either in previously developed concepts or in previously developed models.

No matter what semantics are used, if there are formal semantics for an EPC, then the EPC implicitly defines an LTS. The particular LTS can be derived by creating the state space of the EPC through re-playing all possible executions of the EPC according to its semantics in the same way as we did for workflow

nets. An algorithm to derive a workflow net from an EPC can be found in the work of van Dongen and van der Aalst [57].

Besides the semantical issues, Sarshar and Loos [158] claim that end-users without much training on the particular notation perceive EPCs as easier understandable than Petri nets. This probably contributed to the success of the ARIS platform as a business process modeling tool which grew while using EPCs as its core notation [160]. It is nowadays used by more than 7.000 major companies worldwide [94]. The ARIS framework as the basis for the ARIS platform allows for the integration of various views on a business process, e.g. between an organizational chart, a database model, and an EPC. For this integration, EPCs have been extended with opportunities to annotate functions with information on how data and resources are involved in the process. Details on this extended EPC (eEPC) notation are, e.g., summarized by Scheer et al. [163] while Scheer [164] provides further insights into the ARIS framework.

2.4.2 Protos

Protos is a business process modeling tool similar to the before-mentioned ARIS platform, but developed by the Dutch company Pallas Athena. Protos is part of Pallas Athena's BPM toolset BPM|one and nowadays used by about 1.500 organizations in more than 20 countries, but especially popular in the Netherlands. Here, Pallas Athena is the market leader for BPM products. For example, more than 250 out of 441 Dutch municipalities have active maintenance contracts for Protos with Pallas Athena and use it for the specification of their in-house business processes [183].

The process modeling notation of Protos looks similar to Petri nets and it is well possible to use it in the same way as Petri nets. Tasks are called activities in Protos and depicted by rectangles, while a status in Protos depicts a property of the process in the same way as a place can do in a Petri net. Thus, it is also drawn as a circle (see Figure 2.8). Different from Petri nets, but similar to the use of connectors in EPCs, Protos has been extended with options that allow users to specify the relation among the incoming and outgoing arcs (called connections in Protos) of an activity in a property dialogue (see Figure 2.8). In this way it can be indicated if all those arcs should be triggered after the completion of an activity in the same way as a transition in a Petri net or an \wedge connector in EPCs (AND-split), if only one of these arcs should be triggered at a time like the *XOR* connector of an EPC implies (XOR-split), or if a certain subset of the outgoing arcs should be triggered like an \vee connector implies for an EPC (OR-split). In the same way, it can be specified if the execution of an activity requires the triggering of all of the incoming arcs (AND-join), if it requires the triggering of one of the incoming arcs only (XOR-join), or if it requires the triggering of a subset of the incoming arcs (OR-join). In the travel approval process of Figure 2.8 this is, e.g., used to depict the decision making as one activity with different outcomes while we had to use different transitions in the Petri net of Figure 2.2 to depict the various possible state changes.

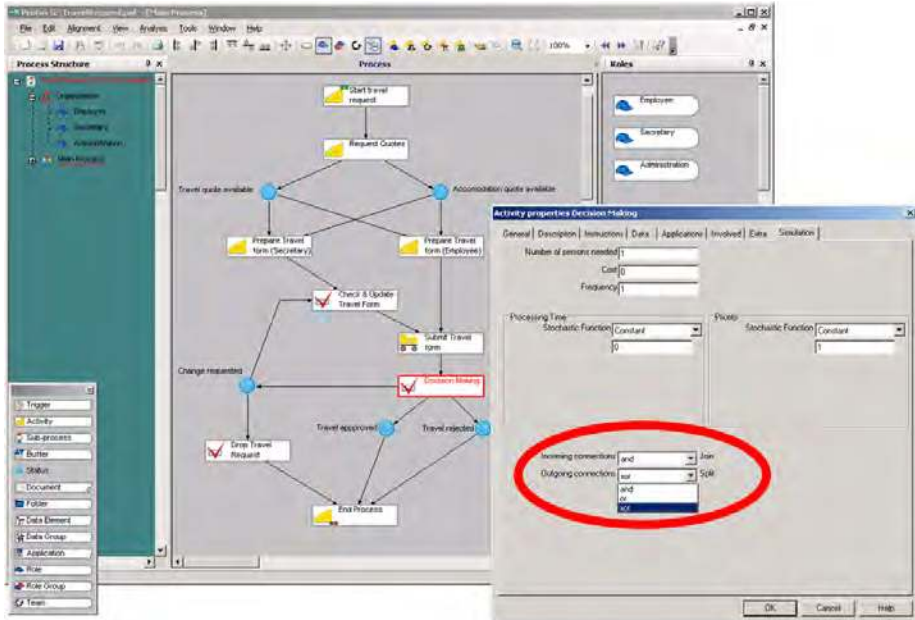


Figure 2.8: The travel approval process for international travels depicted in Protos.

If a Protos model is created in a Petri net way, an LTS can be derived from it in the same way as for Petri nets. If not all statuses between activities are modeled, such implicit statuses can be added by adding the corresponding places. In case *XOR* or \vee connections are assigned to activities, these need to be translated into Petri net constructs by adding an additional place after the activity that represents the choice as well as — if necessary — a set of silent transitions leading to the possible outcomes of the choice.

Besides depicting the process flow, Protos also allows for specifying which resources are involved in the various activities, which data and documents are required or produced by the activities, as well as the potential processing times, priorities, costs, etc. of the various activities. This information can be used to simulate the process. For this, Protos models can be translated into colored Petri nets using **Protos2CPN**² [75]. Protos2CPN generates for each Protos task a colored Petri net like the ones depicted in figures 2.3 – 2.6. According to the Protos model, the input and output places of these individual tasks are then merged through a higher-level colored Petri net. This generates a colored Petri net for the whole Protos process, which can be loaded into CPN Tools³. CPN Tools is a standard simulation engine for colored Petri nets [98, 184, 187]. Through simulating the Protos model in CPN Tools, the behavior of the modeled process can be investigated in detail and it is, for example, possible to derive

²available at <http://www.floriangottschalk.de/protos2cpn>

³see <http://wiki.daimi.au.dk/cpntools>

statistical data, like waiting or overall execution times of the process.

Moreover, Protos models can also be tested on their soundness (see Definition 2.18, p. 24) using an interface to the verification tool **Woflan** [182]. More information on Protos can be found at the website of Pallas Athena⁴ and in the user manual of Protos [125].

2.4.3 BPMN

The **Business Process Modeling Notation (BPMN)** was developed with the goal to form a common standard notation for business process modeling. Similar to EPCs, a BPMN process is always triggered through a **start event** and completes with an **end event**. Within the process flow **intermediate events** can be used to trigger **activities** or to depict the results of activities. For this, BPMN distinguishes different event types, e.g. events can be triggered through or be the result of **messages**, **errors**, or **timers**. Different from EPCs, BPMN does not require the depiction of events between any tasks. Instead events are usually used to depict external influences on or of the process flow. The tasks themselves can either be **atomic** or **composite**, i.e. composed of other processes which can be collapsed into a single task. The routing of cases through the process is determined by **gateways**. Similar to EPC connectors, gateways enable the specification of XOR-splits, AND-splits, OR-splits, XOR-joins, AND-joins, and OR-joins, but it distinguishes between *XOR* gateways which forward cases based on an evaluation of data and *XOR* gateways that wait for events to occur for deciding on how the process will continue.

By dividing a process through swimlanes into **pools**, a process can be split up among the various participants (e.g. roles, organizational units, persons). Each of the participants has then her own process with start and end events while the exchange of messages between the participants coordinates the various processes. In Figure 2.9, we used for example two pools to distinguish the role of the from the role of the in the approval of a travel request. As soon as the employee has organized both quotes (which can happen in parallel) and submits the travel form, the administration is notified through a message and starts the processing of the travel request. During this step, the administration informs the employee whether the request is accepted or declined. Only if the request is approved, it is forwarded to the travel agency; otherwise the administration process terminates immediately. In the meantime the employee's process waits for the approval notification. If the request is indeed approved, this particular instance of the travel approval process immediately terminates while in case the request is declined, the employee has to update and to re-submit the travel form. For the administration this re-starts the approval process.

Further details on the use of additional events and swimlanes and how these elements can be used to organize process models and to depict issues like cancellations, error handling, roll-back, and compensation can be found in the work of White and Miers [191] as well as in the BPMN specification [192]. To give

⁴see <http://www.pallas-athena.com/>

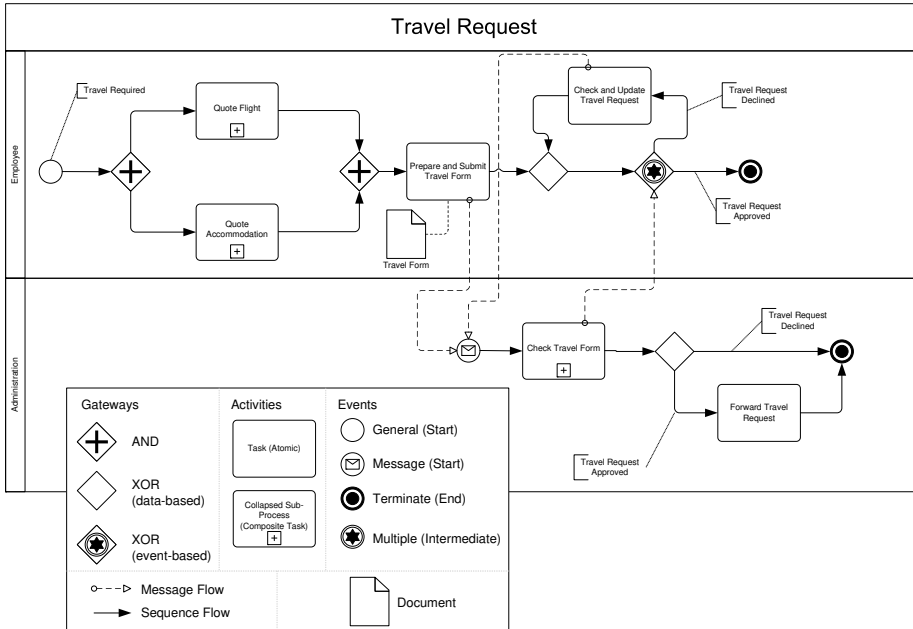


Figure 2.9: The travel approval process for international travels depicted in BPMN.

formal semantics to BPMN, Raedts et al. [134] provide a transformation from a restricted subset of BPMN to Petri nets. Through such a Petri net, we can then also construct LTSs for BPMN.

2.5 Workflow Languages

Workflow management systems are IT systems supporting the execution of business processes like the ones we have depicted in the previous section by automatically distributing the waiting jobs to suitable process participants and providing them with the information necessary to perform these jobs. In the following we will discuss three concrete workflow languages with different backgrounds which enable the specification of business processes such that a system can support their execution in this way. We will start with the academic workflow language **YAWL** (a shorthand for ‘Yet Another Workflow Language’) which was developed with the goal to create a notation that supports all of the desired workflow patterns of van der Aalst et al. [11]. YAWL is both formally defined as well as implemented in a workflow management system. It is therefore also usable in practice. Secondly, we will provide an overview of the commercial workflow language of **SAP WebFlow**. SAP WebFlow is a workflow engine distributed with every installation of SAP’s enterprise system since 1995 [155]. Last, we will give a brief overview on the **Business Process Execution Language (BPEL)**

which is nowadays the standard notation for orchestrating web-services in so-called **service-oriented architectures**.

2.5.1 YAWL

YAWL is a workflow modeling language inspired by Petri nets, but with several important extensions and its own semantics. A YAWL **workflow specification** is composed of a number of so-called **Extended Workflow nets (EWF-nets)**. Each EWF-net consists of **conditions**, which in Petri net terms can be interpreted as places, and **tasks**. Both are connected by arcs to depict the flow of a process. The various EWF-nets form a hierarchy. This hierarchy is created by mapping some tasks of an EWF-net onto other EWF-nets within the workflow specification, i.e., there is a tree-like structure where tasks can be decomposed into EWF-nets. These ‘mapped’ tasks are called **composite tasks**, ‘unmapped’ tasks are called **atomic tasks**.

Figure 2.10 shows an example for such a YAWL model. The example depicts a booking and payment workflow for train travels. After an order has been received, multiple train tickets, a reduction card for train tickets, and/or multiple hotels can be booked. Until a payment method has been selected, the booking can also be canceled. Afterwards the travel has to be paid either in cash or by credit card before the documents can be either send to the customer or collected by her. The tasks *Book hotel* and *Credit card payment* contain further refinements in form of additional EWF-nets.

Each EWF-net has exactly one unique **input condition** and one unique **output condition**. The control-flow determines the flow of tokens through tasks and conditions. AND-joins, OR-joins, and XOR-joins as well as AND-splits, OR-splits, and XOR-splits determine the control-flow behavior before and after each task. AND-joins like the task *Reserve* in Figure 2.10 require tokens in all the conditions preceding the AND-join to enable the execution of the subsequent task, AND-splits like the task *Start search* in Figure 2.10 put tokens into all the post-conditions after the task has completed. Tasks with an XOR-join behavior, as e.g. the *Send documents* task in Figure 2.10, require a token in only one of their pre-conditions to be enabled, tasks with an XOR-split behavior, as the task *Select payment method*, put a single token into one of the post-conditions after the completion of the task. OR-joins, as in task *Select payment method*, allow a synchronizing merge of several process branches by enabling the subsequent task only if there is no chance that any tokens will arrive in unoccupied pre-conditions of the OR-join at any future point in time. OR-splits, as in task *Receive order*, enable a multi-choice, i.e. a selection of several post-conditions.

The specification of a **cancelation region**, as for the task *Cancel booking* in Figure 2.10, allows for the removal of all tokens from the conditions and running tasks within this region during the execution of the task to which the cancelation region is attached to. Independently of the total number of tokens in the conditions, it removes all tokens and therefore supports various cancelation patterns. In addition, tasks can be specified in such a way that they start in

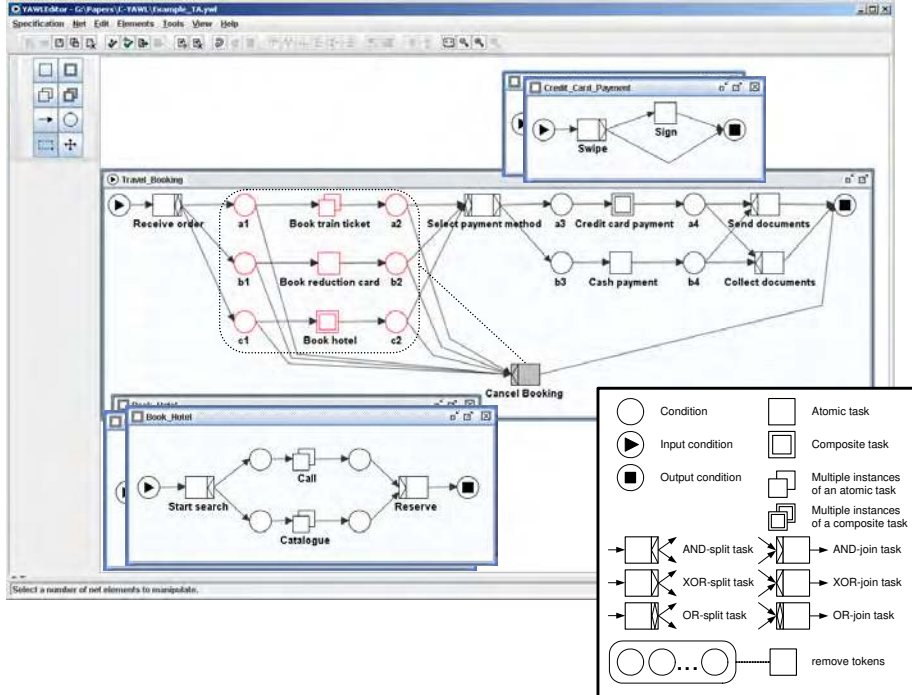


Figure 2.10: An example YAWL model for a travel booking process

multiple instances. Examples are the *Book train ticket* and the *Book hotel* tasks from Figure 2.10 which allow for the booking of multiple tickets or hotels. It is then possible to specify upper and lower bounds for the number of instances of the task that can be started. It can also be specified if instances can only be created at once when the task is started, i.e. **statically**, or if instances can be added **dynamically** while the task is running and the number of started instances is lower than the maximum number. The task's **threshold value** determines the number of instances that have to be completed to complete the task as a whole. As soon as the threshold value is reached, all remaining instances are terminated.

Formally and in line with the definitions of van der Aalst and ter Hofstede [8], an EWF-net can be defined as follows:

Definition 2.24 (EWF-net) An *Extended Workflow net (EWF-net)* is a tuple $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, \text{split}, \text{join}, \text{rem}, \text{nofi})$ such that:

- K is a set of conditions,
- $\mathbf{i} \in K$ is the input condition,
- $\mathbf{o} \in K$ is the output condition,
- T is a set of tasks,
- $F \subseteq (K \setminus \{\mathbf{o}\} \times T) \cup (T \times K \setminus \{\mathbf{i}\}) \cup (T \times T)$ is the flow relation,

- every node in the graph $(K \cup T, F)$ is on a directed path from \mathbf{i} to \mathbf{o} ,
- $split : T \rightarrow \{\wedge, XOR, \vee\}$ specifies the split behavior of each task,
- $join : T \rightarrow \{\wedge, XOR, \vee\}$ specifies the join behavior of each task,
- $rem : T \not\rightarrow IP(T \cup K \setminus \{\mathbf{i}, \mathbf{o}\})$ specifies the cancelation region for a task, and
- $nofi : T \not\rightarrow \mathbb{N} \times \mathbb{N}^\infty \times \mathbb{N}^\infty \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).

The tuple (K, T, F) corresponds to a classical Petri net where K (the set of conditions) corresponds to the set of places, T (the set of tasks) corresponds to the set of transitions, and F (the flow relation) corresponds to the set of arcs (compare Definition 2.14, p. 21). Different to Petri nets, there are the special conditions \mathbf{i} and \mathbf{o} , and tasks can be connected not only via places but also directly to each other by the flow relation. We counteract this ‘unstructuredness’ by defining the extended set of conditions K^{ext} and the extended flow relation F^{ext} for EWF-nets, adding the implicit condition $c_{(t_1, t_2)}$ between two tasks t_1, t_2 if there is a direct connection from t_1 to t_2 .

Definition 2.25 (Implicit conditions) *Let $EFW = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net. Then $K^{ext} = K \cup \{c_{(t_1, t_2)} \mid (t_1, t_2) \in F \cap (T \times T)\}$ is the extended set of conditions and $F^{ext} = (F \setminus (T \times T)) \cup \{(t_1, c_{(t_1, t_2)}) \mid (t_1, t_2) \in F \cap (T \times T)\} \cup \{(c_{(t_1, t_2)}, t_2) \mid (t_1, t_2) \in F \cap (T \times T)\}$ is the extended flow relation.*

The four functions of the EWF-net $split$, $join$, rem , and $nofi$ specify the properties of each task. As the names imply, the first two functions specify the split and join behavior for the tasks. rem specifies from which parts of the net the tokens should be removed. Note that the range of rem includes tasks and conditions, but tokens cannot be removed from input and output conditions. Removing tokens from a task corresponds to aborting the execution of that task. If a task is a composite task, its removal implies the removal of all tokens it contains in its sub-nets. $nofi$ specifies the attributes related to multiple instances. Whenever we introduce an EWF-net EFW in the following we assume $K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem$, and $nofi$ defined as $EFW = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$. For simplicity’s sake, we also assume that $K = K^{ext}$ and $F = F^{ext}$, i.e. we only consider extended nets with implicit conditions. We use $\pi_1(nofi(t))$ to refer to the minimal number of instances initiated, $\pi_2(nofi(t))$ to refer to the maximal number of instances initiated, $\pi_3(nofi(t))$ is the threshold value, and $\pi_4(nofi(t))$ indicates whether it is possible to add instances while handling the other ones.

For convenience, we extend the functions rem and $nofi$ in the following way. If $t \in T \setminus dom(rem)$, then $rem(t) = \emptyset$. If $t \in T \setminus dom(nofi)$, then $\pi_1(nofi(t)) = 1$, $\pi_2(nofi(t)) = 1$, $\pi_3(nofi(t)) = \infty$, $\pi_4(nofi(t)) = static$. This allows us to treat these partial functions as total functions in the remainder.

The mapping of tasks to lower-level EWF-nets which are refining the task (as, e.g., for the tasks *Book hotel* and *Credit card payments* in Figure 2.10) is not part of the higher-level EWF-net, but rather of the workflow specification which organizes the EWF-nets in a tree-like hierarchy. We deviate here from the

original YAWL specification of van der Aalst and ter Hofstede [8] by assigning sets of EWF-nets to composite tasks. The selection which EWF-net from such a set is executed when the composite task is triggered is then performed at run-time. This follows a suggestion by Adams et al. [20] for extending YAWL with such run-time implementation choices for tasks, known as the **worklet service architecture**. Thus, although several EWF-nets are assigned to a composite task, only one of the EWF-nets is executed when the task is triggered. In this way, different implementations of a task can be assigned to the same generic task (e.g., the task *Book hotel* can have an implementing EWF-net for bookings directly with the hotel by phone and another totally different implementation for bookings via a booking portal in the internet).

Definition 2.26 (Workflow specification) *A workflow specification is a tuple $(Q^\diamond, Q, top, T^\diamond, map)$ such that:*

- Q^\diamond is a set of EWF-nets,
- $top \in Q^\diamond$ is the top level workflow,
- $Q \subseteq \mathcal{P}(Q^\diamond \setminus \{top\})$, $(\bigcup_{EWFs \in Q} EWFs) = Q^\diamond \setminus \{top\}$, $\forall_{EWFs_1, EWFs_2 \in Q} (EWFs_1 \cap EWFs_2 \neq \emptyset) \Rightarrow EWFs_1 = EWFs_2$, partitions Q^\diamond into sets of EWF-nets,
- $T^\diamond = \bigcup_{EWF \in Q^\diamond} T_{EWF}$ is the set of all tasks,
- $\forall_{EWF_1, EWF_2 \in Q^\diamond} EWF_1 \neq EWF_2 \Rightarrow (K_{EWF_1} \cup T_{EWF_1}) \cap (K_{EWF_2} \cup T_{EWF_2}) = \emptyset$, i.e., no name clashes,
- $map : T^\diamond \dashv\vdash Q$ is an injective, surjective function which maps each composite task onto a set of EWF-nets, and
- the relation $\{(EWF_1, EWF_2) \in Q^\diamond \times Q^\diamond \mid \exists_{t \in dom(map)} (t \in T_{EWF_1} \wedge EWF_2 \in map(t))\}$ is a tree, i.e. it contains no cycles.







Q^\diamond is a non-empty set of EWF-nets with a special EWF-net top . The tasks in the domain of map are the composite tasks which are mapped onto sets of EWF-nets. This is done in such a way that each EWF-net in Q^\diamond can only be assigned onto one task, but each composite task is mapped onto a set of several EWF-nets that is specified in Q , i.e. a tree-like structure with top as root node is formed. As top is always the root net, it will be part of any workflow execution. This does not necessarily hold for the other EWF-nets in Q^\diamond (i.e. the child elements of top). If there is more than one EWF-net mapped onto a task t , i.e. $|map(t)| \geq 2$, then only one of these nets is selected and becomes part of the workflow execution. The other alternatives and their sub-nets will not be executed. Also note that we always assume that there are no name clashes, i.e., names of conditions differ from names of tasks and there is no overlap in names of conditions and tasks originating from different EWF-nets. If there would be name clashes, tasks and conditions could simply be renamed.

YAWL comes with its own formal semantics which is capable of handling all the different types of task execution by explicitly distinguishing active tasks from individual task instances which can be created, executed, or completed. For composite tasks, the execution phases of individual task instances is halted

as long as the execution of the sub-net lasts. The formalization of the YAWL semantics can be found in the work of van der Aalst and ter Hofstede [8]. It includes the definition of the state space of a YAWL workflow specification by taking these sub-states of tasks into consideration. Van der Aalst and ter Hofstede also include the transition relation which depicts the possible changes among these states. When a label about the various properties that led to the particular state change is assigned to each transition, the transition relation, the state space, and these labels form an LTS.

2.5.2 SAP WebFlow

SAP, as one of the biggest vendors of enterprise systems, delivers the workflow engine WebFlow with any installation of their business suite. It can be used to guide business processes through SAP's enterprise system, but is also capable of incorporating other systems into the workflow. The workflow engine comes with its own workflow modeling notation which for simplification we just call SAP WebFlow in the following. The notation is mainly based on so-called **steps** and **events** which are organized in a block structure⁵. Steps are depicted by boxes with different symbols, representing either routing constructs or functionalities offered by the system, i.e. they refer to tasks. Figure 2.11 depicts a template for a travel approval process that is provided by SAP with their enterprise system. Let us use this example to explain the different modeling elements of SAP WebFlow.

- The most basic step type is the **activity** () which is in the example used for the *Set trip status to approved*, *Change trip*, *Enter and send message*, and *Send mail: Request approved* activities. An activity step is always connected to a task maintained in SAP's enterprise system which is executed when the step is triggered. After completion of the task, the step is completed and the next task is triggered.
- **User decisions** () , such as the *Approve travel request* step shown in Figure 2.11, are providing a list of answers from which the user can choose one. Based on this answer the corresponding subsequent path is selected.
- **Conditions** () , such as the *Travel request approved?* step, evaluate a boolean condition. Based on the outcome of this evaluation the process follows one of the two paths. Similar, **multiple conditions** () , not depicted in the example) contain a set of subsequent paths, of which one is selected based, for example, on the value of a (non-Boolean) data element of the workflow. If no path is connected to the value of the data element, an 'other values' branch is selected.
- **Forks** (, ) are used to specify the parallel processing of paths. All paths leaving the splitting fork are triggered by this step. The joining fork allows the specification of a condition when it is completed. This condition

⁵SAP also provides an EPC translation for workflows specified in SAP WebFlow. However, the usage of EPCs is restricted to constructs realizable in SAP WebFlow. For this reason, we stick to the original SAP workflow notation.

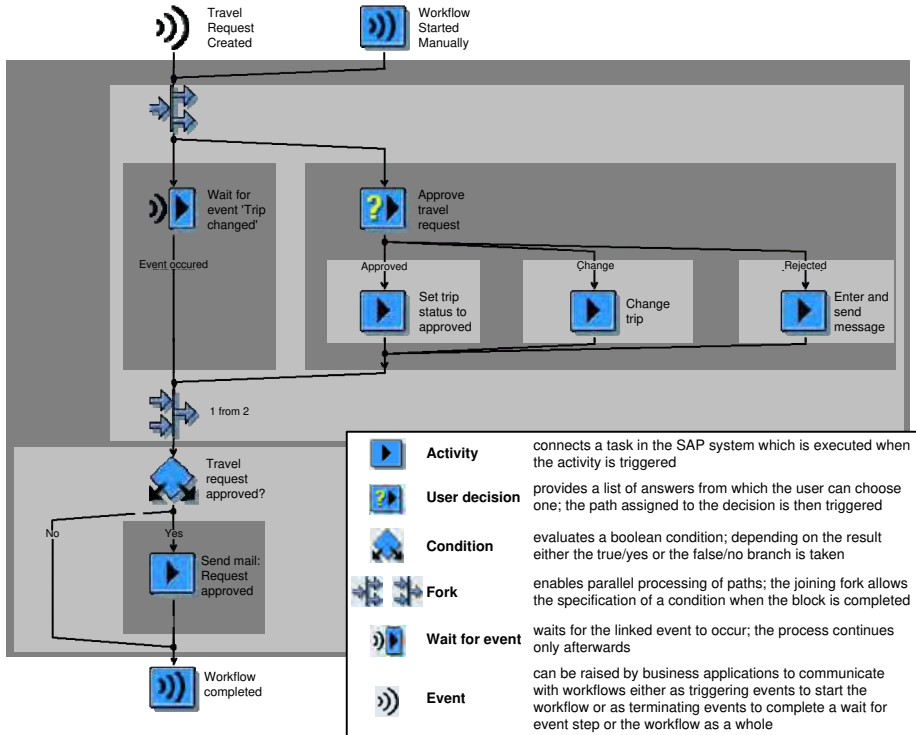


Figure 2.11: A workflow template of a travel approval process provided by SAP for its WebFlow engine (accessible in SAP as workflow WS20000050)

can be the number of preceding paths that have to have completed. In this way, e.g., a two-out-of-three join can be realized, i.e. the process continues as soon as two out of three incoming branches have completed. The condition can also be based on data elements of the workflow. In any case, as soon as the completion condition is fulfilled, any work that is still ongoing in the preceding branches is terminated and deleted.

- As soon as a **Wait for event** step () is triggered, it waits for a linked event to occur and the workflow is only continued after the event has occurred.

Events () can be raised by business applications to communicate with workflows. The trigger to raise an event can be manifold. For example, it can be the creation or the change of a document, a general status message, an exception occurring in an information system, or a business transaction event which occurred in the financial system. Even a workflow can raise events on its own by **Event creator** steps (). Events can be linked to workflows (and tasks which then are handled as if they are workflows on their own) as triggering events to start the workflow (e.g., as the *Travel Request Created* event in the example), or as terminating events to stop a workflow or a *wait for event* step inside a work-

flow. Note that the same event can be linked for different purposes to different workflows at the same time, e.g. to terminate one workflow and instead trigger another workflow.

The LTS corresponding to a model depicted in SAP WebFlow can be constructed in a similar way as the LTS for models depicted in YAWL, i.e., during the construction of the state space all the various intermediate states have to be considered, which occur during the execution of steps and activities. Furthermore, the transition relation has to take not only the obvious process flow into account, but also all possible points in time when events or conditions lead to the cancelation of behavior.

The linkage between steps or events and workflows includes the linkage of the data in the data containers of the step or event and the workflow. This linkage enables the start of a workflow or a step with the right parameters, e.g. to select responsible resources or correct documents. Further details on such mappings and other implementation issues can be found in the book by Rickayzen et al. [139].

2.5.3 BPEL

As a third example notation for specifying workflows, we use the **Business Process Execution Language (BPEL)**, also known as Web Services Business Process Execution Language (WS-BPEL). Nowadays, it is the standard for the composition and orchestration of web services. BPEL uses an XML-based representation to define workflows. The language can be seen as a mixture of graph-based and programming-like constructs. For example, Figure 2.12 (partially) shows the BPEL representation of the travel booking process we used as example for YAWL (see Figure 2.10, p. 37). Similar to SAP WebFlow, activities in BPEL are also block structured. Within this block structure BPEL distinguishes between six structured activities organizing the control-flow of its sub-blocks and three primitive activities that perform the required tasks. Examples for structured activities are the **sequence** which enforces the sequential execution of its sub-blocks (see Figure 2.12), the **flow** which allows for a parallel execution of the sub-blocks, and the **switch** which can be used to define an *XOR* choice among the sub-blocks. The primitive **invoke** activities call a linked operation, e.g. of a web service or another workflow, and wait for its response. For example, in Figure 2.12 the first invoke activity calls the operation *requestTravel* of the booking engine. **Receive** activities are primitive activities which wait for such invoke calls. The results of such requests are then provided through the primitive **reply** activities.

In addition to the structured activities such as a sequence, switch, etc. BPEL allows for expressing control flow relations between activities also through **control links**. Control links establish a control flow from one to another activity possibly breaking BPEL's block structure. For this reason not only the next activity according to the block structure is triggered when an activity completes, but also **outgoing control links** can be activated. In Figure 2.12 three such control links are for example specified for the first invoke activity. An activity

```

<process ...>
  <sequence ...>
    <flow ...>
      <links>
        <linkName="trainTicket"/><linkName="reductionCard"/><linkName="hotel"/>
      </links>
      <invoke partner="BookingEngine" operation="requestTravel"
        inputVariable=... outputVariable="travelNeeds" ...>
        <source linkName="trainTicket"
          transitionCondition="bpws:getVariableData(travelNeeds, trainReq)=true"/>
        <source linkName="reductionCard"
          transitionCondition="bpws:getVariableData(travelNeeds, cardReq)=true"/>
        <source linkName="hotel"
          transitionCondition="bpws:getVariableData(travelNeeds, hotelReq)=true"/>
      </invoke>
      <invoke partner="TicketProvider" operation="getTicket" ...>
        <target linkName="trainTicket"/>
      </invoke>
      <invoke partner="CardProvider" operation="orderCard" ...>
        <target linkName="reductionCard"/>
      </invoke>
      <invoke partner="HotelReservationSystem" operation="reserve" ...>
        <target linkName="hotel"/>
      </invoke>
    </flow>
    <flow ...>
      <links>
        <linkName="creditCardPayment"/><linkName="cashPayment"/>
      </links>
      <invoke partner="PaymentEngine" operation="getPaymentDetails"
        inputVariable=... outputVariable="paymentDetails" ...>
        <source linkName="creditCardPayment"
          transitionCondition="bpws:getVariableData(paymentDetails, card)=true"/>
        <source linkName="cashPayment"
          transitionCondition="bpws:getVariableData(paymentDetails, cash)=true"/>
      </invoke>
      ...
    </flow>
    ...
  </sequence>
</process>

```

Figure 2.12: A travel booking process specified in BPEL

for which a corresponding **incoming control link** is specified can then only be executed if it is triggered through the structure of the workflow and if all the incoming links have been activated. The invoke activity in Figure 2.12 that gets the ticket is thus only activated if on the one hand the enclosing flow activity is activated and on the other hand the link *trainTicket* is activated. In the context of an activity also a **join condition** over the data provided by the links has to be true. Therefore, all incoming links and the control-flow through the block structure synchronize a workflow's behavior in the same way as an AND-join.

For each outgoing control link, a separate condition can be given to specify whether it will be activated. The *trainTicket* link of Figure 2.12 is thus only enabled if the *trainReq* parameter is true. Depending on the conditions, any combination of activating outgoing control links is possible, which means that the links are in an \vee relation.

Translations from BPEL to Petri nets are, e.g., suggested by Hinz et al.

[92], Lohmann [115], Ouyang et al. [124], and Verbeek and van der Aalst [181]. Through such a translation of a BPEL model into a Petri net, also an LTS can be derived from a BPEL process as we have depicted in Section 2.2. As we did for YAWL and SAP WebFlow, we omit further implementation details on BPEL that are irrelevant for the remainder of this thesis here. The interested reader finds these details in the BPEL standard [24].

2.6 Summary

After having defined some formal notations, this chapter introduced three types of process modeling languages, namely notations for formal process specifications, more informal business process modeling languages, and executable workflow languages. Formal definitions were provided for Labeled Transition System (LTS) and workflow nets, which are a the Petri net variant. A brief introduction to common workflow patterns lead to focusing on more practice-oriented languages. Still, we also discussed formal definitions for Event-driven Process Chains (EPCs), and introduced Protos models, and the Business Process Modeling Notation (BPMN) as further examples for business process modeling languages. In the same way, we provided formal definitions for the workflow language YAWL, and also introduced SAP WebFlow, and the Business Process Execution Language (BPEL) as further examples for such languages. Workflow engines capable of automatically executing the particular models exists for all three workflow languages.

Throughout this thesis we will use formal languages for both identifying the essence of process model configuration, as well as for providing the foundations for configuring process models correctly. A process implementation usually starts with drawing a general business process model. We will therefore use business process modeling languages whenever we discuss models which serve as the starting point for building a configurable process definition rather than as its concrete specification. On the other hand, we will use workflow notations whenever we will discuss the execution of processes.

*I saw the angel in the marble and carved until I set him free.
Michelangelo Buonarroti (1475–1564)*

Chapter 3

Configuring Process Models

The goal of configuring a process model is to adapt the model such that it fits the model user’s individual needs better than the original process model. However, ideally, the user should not need to add any content to the process model itself while configuring the model. Thus, configuring a process model means restricting the behavior depicted by an existing process model in such a way that it only allows for the desired behavior of the model, while all the undesired behavior is eliminated from the model.

In this chapter we will analyze what ‘restricting the behavior’ means for process models. For this, we will first have a look how behavior can be added to a process model by introducing concepts based on the inheritance of dynamic behavior. Knowing how behavior can be added, we can do the inverse of it to remove behavior from a process model, i.e. we can configure the process model. In the second section of this chapter, this definition of process configuration is used to define configurable process models. The third section of this chapter on related work compares this approach of defining configurable process models with the work of other authors that have identified configuration opportunities for process models. The chapter ends with summarizing general requirements on process model configuration, providing an outlook on the issues that need to be addressed when developing configurable process modeling notations.

3.1 Configuration versus Inheritance

Obviously, behavior can only be removed from a process model if it has been added beforehand. To get to the essence of process configuration, i.e. how a process model’s behavior can be restricted, we should thus analyze, how this behavior can be added to the model in the first place.

A nowadays very popular approach that deals with adding information and methods to an existing framework in a structured way is called **inheritance**. The inheritance concept is a guiding principle in object-oriented software de-

velopment. In object-oriented programming, classes describe the properties of a certain type of objects as well as the methods that these objects can perform. The basic idea of inheritance is then to provide a mechanism that allows for constructing **subclasses** of such classes, which are inheriting all this behavior and all the properties of the original classes. A subclass can then be extended with additional behavior or properties compared to the original class. Thus, a subclass must always provide at least the same functionality as the class from which it is derived, but can add additional functionality and properties to this.¹ The original classes are then called **superclasses** of the subclasses. To get back to our original question, we can thus say that any class that adds behavior or properties to another class is a subclass of this other class under the inheritance concept.

Basten and van der Aalst [32] applied these inheritance concepts to workflow models and identified how inheritance can be detected between such models of behavior. For this, they compare the behaviors of models, using the equivalence notion of **branching bisimulation** [71].

Basically, branching bisimulation ignores the behavioral impact of silent tasks, i.e. tasks that cannot be observed, but (unlike trace equivalence) takes the moment of choice into account. Two process models are behavioral equivalent under branching bisimulation if all non-silent tasks that can be executed in any particular state of one model can also be executed in the equivalent state of the other model, either directly or after executing a set of silent tasks that leads to a state from which the task can then be executed. For this, branching bisimulation relates states in one process to states in the other process (and vice-versa) such that the ‘possible futures’ of these states match. For example, compare the two LTSs in Figure 3.1a. Here, s_{A1} in process A allows for executing a in the same way as s_{B1} in process B . s_{A2} allows for executing b which requires the execution of a silent transition labeled τ in s_{B2} . In this way s_{B3} is reached which then allows executing b in the same way as s_{A2} . Finally, both processes complete after the execution of b in s_{A3} or s_{B4} . Hence, both processes are equivalent under branching bisimulation and we say they represent identical behavior. This is different in Figure 3.1b: We can easily match s_{C1} to s_{D1} as both states allow executing either a (in case of process D after executing the silent τ transition) or b , as well as s_{C2} to s_{D3} , s_{C3} to s_{D4} , and s_{C4} to s_{D5} . But there is no equivalent state to s_{D2} in process C . In s_{D2} only a can be executed. The only state from which a can be executed in process C is s_{C1} . However, in s_{C1} also b can be executed. Hence, it cannot be considered equivalent to s_{D2} as their ‘possible futures’ are different. Since it is not possible to construct such a relation, C and D cannot be considered as equivalent under branching bisimulation.

For detecting and defining inheritance relations between two workflow models, Basten and van der Aalst [32] employ two mechanisms. Both mechanisms

¹Note that inheritance allows to overwrite the concrete implementation of a particular method. However, for the outside world the functionality remains the same as the identifier as well as the type of the output remains identical. Thus, these parts of the inheritance concept do not add any behavior and can be neglected here.

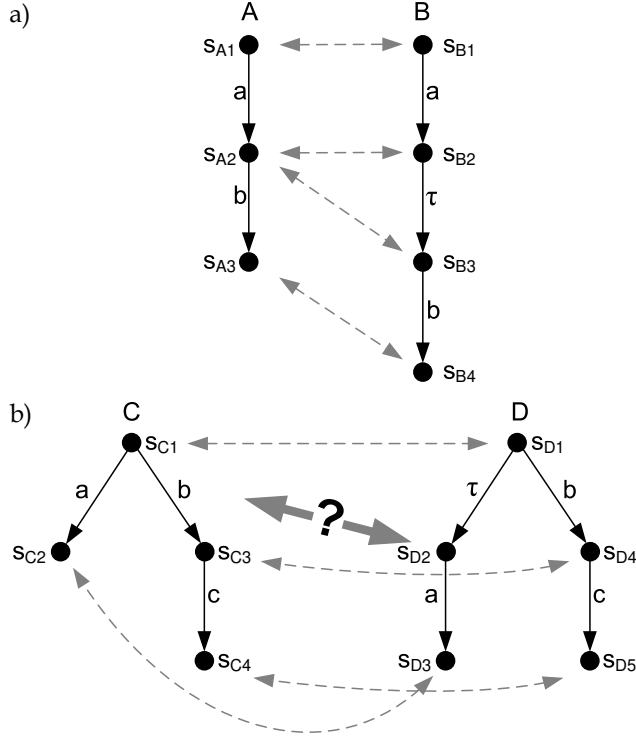


Figure 3.1: The two processes in (a) are branching bisimilar, while the two processes in (b) are not as there is no state in C that matches state s_{D2} in D .

start with identifying the behavior that is depicted in that model of the two models which is assumed to be a subclass of the other one. They then compare this behavior with the subclass's superclass, i.e. the other model, identifying the behavior that is lacking in the superclass compared to the subclass. Thus, while subclasses add behavior to superclasses, the mechanisms to detect inheritance relations are analyzing the process model in the inverse direction by removing behavior from subclasses.

- The first mechanism identifies inheritance relations by **encapsulating**, i.e. inhibiting, the execution of additional functionality. If it is not possible to distinguish the behaviors of a model x and a model y when only transitions of x that are also present in y are executed, then is x a subclass of y . That means, all transitions of the subclass x that are not present in the superclass y are **blocked** from being executed. For example, let us have a look at the LTS *superclass* A and its *subclass* in Figure 3.2. If the subclass's transitions b , j , k , l , m , n , and p are blocked and thus not executed, the behavior of the *subclass* and *superclass* A are identical. Therefore, an inheritance relation exists between the two transition systems.
- The second mechanism **abstracts** from the execution of the transitions

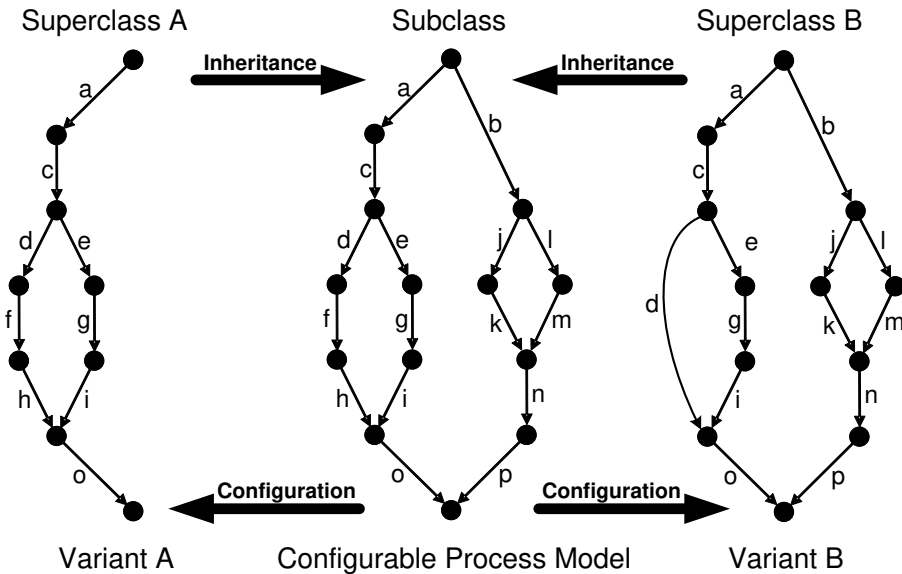


Figure 3.2: Inheritance relations between process models — the inverse of configuration.

themselves, but it analyzes if the two processes vary if only the effects of those transitions of the model x are considered that are also part of the model y . If the behaviors of x and y cannot be distinguished when arbitrary transitions of x are executed, but when only the effects of transitions that are also present in y are considered, then x is a subclass of y . All effects of the subclass x not occurring in y are **hidden** in the superclass y . For example, let us compare the LTS *superclass B* with the *subclass* in Figure 3.2. The transitions f and h exist in the subclass but not in the superclass. However, if we execute the subclass and do not consider that these two transitions are executed in between the execution of transition d and transition o , the *subclass* behaves identical to its *superclass B* where o is executed directly after d .

Figure 3.2 also shows that through multiple inheritance a subclass can be the subclass of multiple superclasses. Such a subclass includes the behavior of all its superclasses, i.e. from the perspective of each individual superclass the subclass is extended with (at least) the behavior of the other superclasses. If such a subclass is minimal, i.e. each extension is motivated by at least one of the given superclasses, we refer to the subclass as the **least common multiple** of the given superclasses [5].

When configuring a process model, the goal is to get rid of undesired behavior in contrast to adding behavior as inheritance concepts imply. Thus, instead of deriving a subclass of a process model, we have to do the inverse and need to find a superclass of the process model which is optimal for the particular purpose

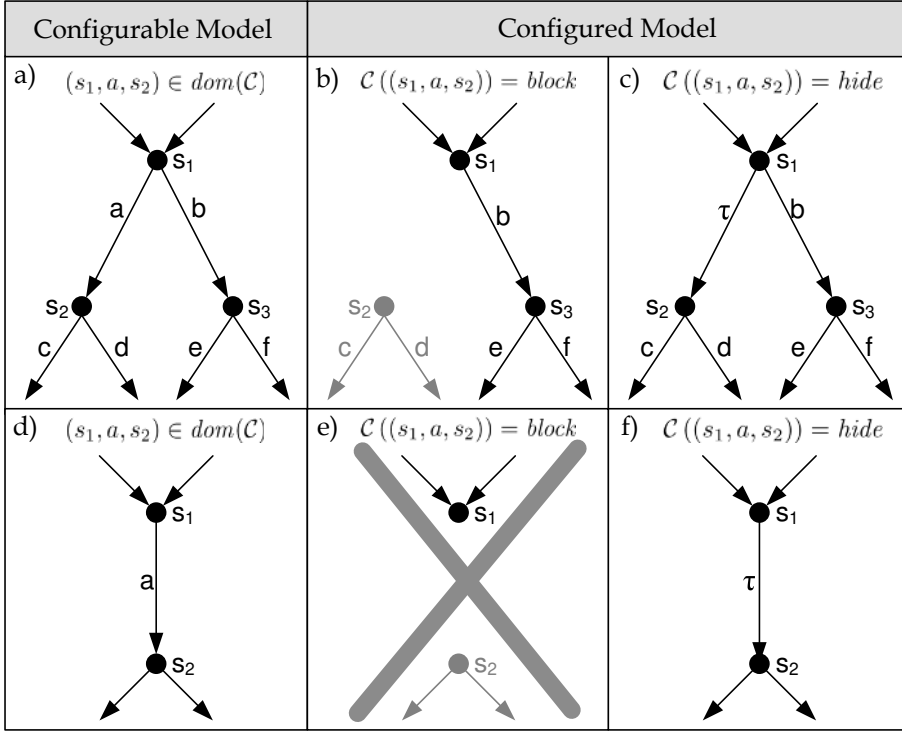


Figure 3.3: Configuration in a labeled transition system

for which the model is configured. As Basten and van der Aalst [32] defined the inheritance relations already by determining if the superclass can be re-produced from the subclass through encapsulating and abstracting from behavior of the subclass, the two mechanisms of blocking and hiding behavior can be used as tools to derive a superclass from the process model, i.e. to restrict the behavior of the process model. For the configuration of LTSs (Definition 2.13, p. 20) we can, e.g., use hiding and blocking as follows :

Definition 3.1 (LTS Configuration) *A configuration of an LTS (S, L, T, S_I, S_F) is a (partial) function $\mathcal{C}_{LTS} \in T \rightarrow \{\text{hide}, \text{block}\}$ where $\text{dom}(\mathcal{C}_{LTS})$ is the set of configured transitions, and for $t \in \text{dom}(\mathcal{C}_{LTS})$:*

- if $\mathcal{C}_{LTS}(t) = \text{hide}$, t is a hidden transition, and
- if $\mathcal{C}_{LTS}(t) = \text{block}$, t is a blocked transition.

To derive a configured model from a configuration, the configuration must be applied to the process model. Figure 3.3 depicts some configuration scenarios using an LTS: the first column depicts the configurable models; the subsequent columns depict the resulting models if transition a is blocked or hidden.

The configuration decision to block a transition implies that the transition will never be executed. That means, the transition should not appear within

the configured model. It must be removed from the model when generating the configured model as depicted in Figure 3.3a/b where the transition labeled a in Figure 3.3a is configured as blocked and thus removed in the configured model in Figure 3.3b. Hence, when reaching state s_1 , the transition can no longer be executed. Instead the process has to execute the alternative transition (s_1, b, s_3) to reach state s_3 . Thus, state s_2 and its subsequent transitions labeled c and d become unreachable, depicted in the figure through greying them out. In fact, as they are not reachable, they could be removed as well, but as they cannot be reached anyway this has from an execution point of view no influence on the process behavior.

If the configuration decision is to hide a transition, the transition's external, i.e. observable, effects will be ignored. However the effects within the model, that means on the execution of subsequent transitions, are kept. Therefore, when generating a configured model, the transition must be transformed into a silent step without output. Thus, the transition is replaced with a silent transition labeled τ (see Figure 3.3a/c). Hence, the choice in state s_1 in Figure 3.3c is preserved in the configured model: it only switches from being between executing (s_1, a, s_2) or (s_1, b, s_3) to being between executing the silent transition (s_1, τ, s_2) without generating any visible output and thus continuing with the subsequent behavior from state s_2 or executing the transition (s_1, b, s_3) .

The definition of hiding given above explicitly says that a task is executed, but the external effect is ignored. However, the desired result when configuring a process model differs slightly. In fact, instead of ignoring the transition's external effects it should not even be executed. Only the non-observable, internal effect of reaching a subsequent state and triggering subsequent transitions has to occur. For that reason, hiding can be considered as **skipping** of the task. As the perceived results are identical, we will continue to call this operation hiding and thus stay consistent with the inheritance concept motivating this configuration operation.

For LTSs the described algorithm of transforming a configuration of a process model into a configured process model can be formalized as follows:

Definition 3.2 (Configured LTS) *Let $LTS = (S, L, T, S_I, S_F)$ be an LTS and $\mathcal{C}_{LTS} \in T \rightarrow \{hide, block\}$ be a configuration of LTS. The configured LTS resulting from this configuration, $LTS^{\mathcal{C}} = (S^{\mathcal{C}}, L^{\mathcal{C}}, T^{\mathcal{C}}, S_I^{\mathcal{C}}, S_F^{\mathcal{C}})$, is defined as follows:*

- $S^{\mathcal{C}} = S$
- $L^{\mathcal{C}} = \{l \in L \mid \exists s, s' \in S (s, l, s') \in T \setminus dom(\mathcal{C}_{LTS})\}$
- $T^{\mathcal{C}} = (T \setminus dom(\mathcal{C}_{LTS})) \cup \{(s, \tau, s') \mid \exists l \in L (s, l, s') \in dom(\mathcal{C}_{LTS}) \wedge \mathcal{C}_{LTS}((s, l, s')) = hide\}$
- $S_I^{\mathcal{C}} = S_I$
- $S_F^{\mathcal{C}} = S_F$

While the states remain the same in the configured process model as in the original process model (they might just become unreachable and thus irrelevant), the transition labels of the configured model are reduced to labels of transitions

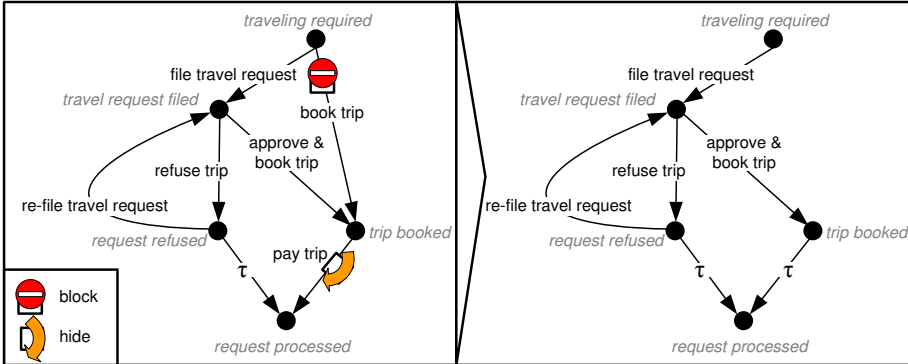


Figure 3.4: Configuring the LTS of Figure 2.1

that are neither blocked nor hidden. The configured process model then contains on the one hand the transitions that were neither hidden nor blocked, and on the other hand a silent transition for each transition of the original process model that was hidden. For example, in Figure 3.4 the LTS from Figure 2.1 (p. 20) is configured such that the transition *book trip* is blocked as employees should not be able to book trips on their own, but should rather file travel requests. In the configured LTS, the corresponding transition is thus removed. Also, the payment for the trip happens independently of the processing of the travel request. Hence, the processing of the request completes as soon as the trip is booked. For this, the *pay trip* transition is configured as hidden, resulting in replacing it with a silent transition labeled τ . In this way, the state *request processed* can be reached from the *trip booked* state without the payment.

Note that we say in Definition 3.2 that the initial states and the final states of the original and the configured net should be identical. This is to preserve the situations which trigger the execution of the process, as well as to preserve any potential behavior outside of the process model which is triggered through reaching the final state. New initial or final states would require manual adaptations to the process's environment. For the process in Figure 3.4 this, e.g., means that we cannot simply stop the process when reaching the state *trip booked* but rather need to ensure that the process really completes with reaching the state *request processed*. Hence, we cannot block the *pay trip* transition but rather need to hide it. This is also shown in the second row of Figure 3.3: Blocking the transition (s_1, a, s_2) would lead to a deadlock in state s_1 which cannot become a final state (Figure 3.3e). Hiding it, however, generates no problems (Figure 3.3f).

Besides such technical reasons, also the domain of the process might inhibit certain configurations. For example, a configuration that prevents the execution of tasks which are essential for the overall business process should be avoided. This means that in Figure 3.4 it would make no sense to, e.g., inhibit all ways to book a trip if a traveling is required. Thus, because of such technical and

domain constraints, only a subset out of all possible configurations is **valid**.

A configurable process model should therefore consist on the one hand of a process model which represents a subclass of all desired configurations, preferably their least common multiple, which we call the **basic process model** of a configurable process model. On the other hand, it should contain a list of **valid configurations** of the basic process model which leads to desired configured models only. For LTSs we therefore say:

Definition 3.3 (Configurable LTS) *A configurable process model is a tuple $CPM = (LTS, CS)$ where:*

- $LTS = (S, L, T, S_I, S_F)$ is a labeled transition system, and
- $CS \subseteq T \rightarrow \{hide, block\}$ is a set of valid configurations.

Configuring the configurable process model then means to select a configuration $C_{LTS} \in CS$ for LTS and deriving the corresponding configured model.

3.2 Adding Configuration to Process Modeling

Advanced process modeling notations abstract from explicitly depicting each system state. Instead they use a single syntactical element to model many transitions as we have seen when introducing workflow patterns in Section 2.3. Also, the overall state of the system is broken down into many individual properties that characterize the overall state by either holding or not holding. To apply the depicted configuration ideas to advanced notations with multiple node types, we thus have to identify in which way these notations depict state changes. This then allows us to apply the techniques of blocking and hiding to these state changes which will all be subject of the first part of this section.

Furthermore, we need to find a practical way to denote and verify the restrictions to configuration decisions, i.e. to ensure valid configurations in the second part of this section. Through identifying the optimal basic process model and combining it with a list of valid configurations, the last part of this section will provide the basis for deriving configurable variants of advanced process modeling notations.

3.2.1 Configuring Ports of Tasks

In process models, property changes and thus state changes are depicted through nodes representing the tasks that are performed. For example, in a Petri net the execution of a transition leads from one marking, i.e. one state of the Petri net, to another marking representing a different state. In the same way, the execution of a function of an EPC results in new events occurring, which mark a new state of the process execution.

In all such notations the triggering of tasks is represented by arcs pointing at the task. However, the meaning of these arcs varies not only among different workflow modeling notations but also within a single modeling language. Different join patterns for input paths of task nodes, i.e. paths that lead to

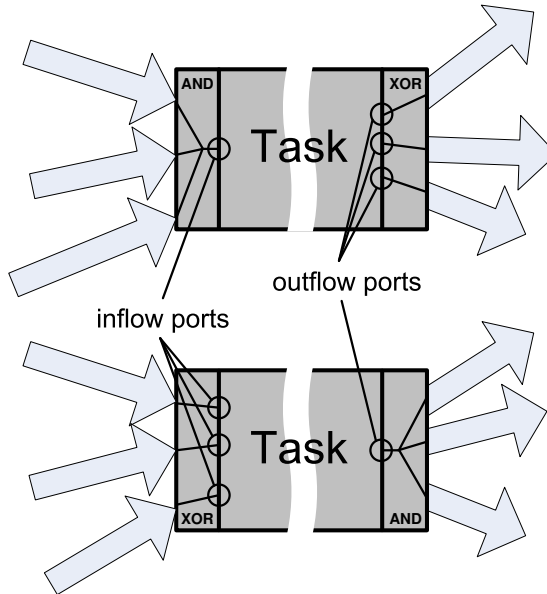


Figure 3.5: The number of a task's ports depends on its joining and splitting behavior: Each way a task can be triggered conforms to an inflow port (left) while each way subsequent paths can be triggered after the task's completion conforms to an outflow port (right)

the execution of the task, define in which circumstances the task will be triggered. For example, some tasks require that all incoming paths are triggered before the task itself can be triggered, i.e. all incoming process branches will be synchronized through an AND-join behavior. Other tasks can immediately be triggered if a single incoming process path is triggered (XOR-join). It might therefore be possible, that a single task can be triggered in multiple ways. To distinguish between the different ways how a task can be triggered, we call each of these combinations of incoming paths through which a task can be triggered an **inflow port** of the task (see the left side of Figure 3.5). Thus, a task with an AND-join behavior for the incoming paths has just a single inflow port, whereas a task with an XOR-join behavior has an inflow port for each incoming path.

After the execution of a task has completed, it releases the particular case by triggering the arcs leaving the task. In this way, the execution of the task leads to the marking of new properties, i.e. to a new state. The number of triggered paths depends again on the semantics specified for the particular task. A task with an AND-split behavior triggers all outgoing paths, whereas a task with an XOR-split behavior only triggers one out of all the subsequent path. Of course, some semantics also allow the triggering of selected paths (OR-split). Aligned with the specification of inflow ports, we say that each case can 'leave' the task only through one distinct **outflow port**, but then triggers all paths connected to this outflow port (see the right side of Figure 3.5).

The state change of a task is thus always represented by combining one

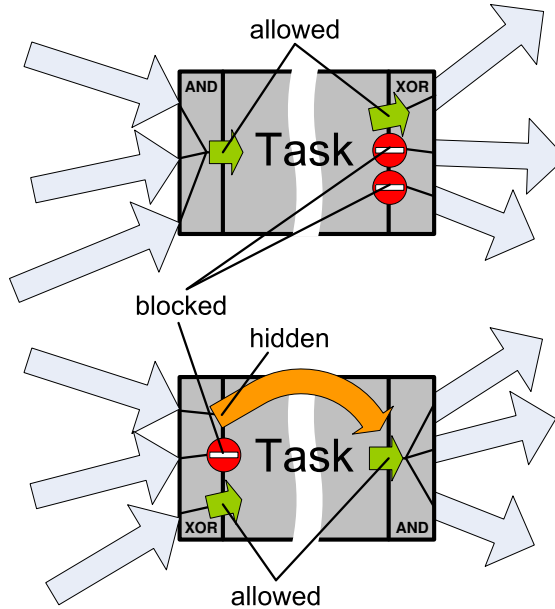


Figure 3.6: Through configuration the flow of cases through ports can be allowed, blocked, and in case of inflow ports also hidden.

inflow port with one outflow port for the flow of a case through this task. As the goal of process configuration is to decide between restricting and allowing state changes, we thus need to allow or restrict the use of inflow and outflow ports through configuration, i.e. *ports are the elements whose use can be allowed, blocked, or hidden*. Hence, they are the configurable elements.

If we allow the use of an inflow port, the tasks can be triggered as implied by the process model. In the same way, if the use of an outflow port is allowed, the subsequent path can be triggered as implied by the model. If a port is, however, blocked the process flow through this port is inhibited, i.e. a task cannot be triggered through a blocked inflow port, and the (combination of) paths connected to a blocked outflow port can then not be triggered after the task's completion (see Figure 3.6). Hence, in case of a blocked inflow port, a case has to find a different path through the process model to continue, while in case of a blocked outflow port, the case has to leave the task through a different (non-blocked) outflow port of this task. Thus, the use of at least one outflow port must remain allowed in any task configuration in which at least one inflow port of the task is not blocked.

If we hide the observable behavior of a state change, the subsequent state is still reached. For a process model this means, everything that is usually happening while the task is executed should not happen. Still, the subsequent paths should be triggered such that the new state is reached. Thus, the triggering of paths leading to hidden inflow ports implies the skipping of all task activities

and a direct forwarding of the case to the outflow ports (see Figure 3.6). *The outflow ports themselves cannot be configured as hidden because outflow ports trigger paths instead of tasks.* A path just forwards the case to the next task without implying any action itself. Thus, a path contains nothing that can be skipped (and any subsequent task should be hidden via its own inflow ports).

To give a first, simple example how a configuration can be defined for a concrete process modeling language, let us have a look at workflow nets (see Definition 2.15, p. 23). Like in LTSs, tasks are called transitions in workflow nets, but are depicted as nodes which can have several incoming and outgoing arcs. They thus have completely different semantics compared to LTSs. The semantics of workflow nets imply that a transition can only fire, i.e. a task can only be executed, if all preceding places are marked with tokens. So, in line with our previous argumentation we can say that an incoming arc of a transition is triggered as long as the arc's source place is marked with a token. Hence, any workflow net transition implements an AND-join pattern. For that reason, it only has a single inflow port. After the firing, i.e. after the completion of the task, the transition marks all subsequent places. Thus, the transition also implements an AND-split behavior. So, we can say that the transition has only a single outflow port. As we showed that the use of at least one outflow port must always remain allowed, this single outflow port of workflow net transitions is not configurable. Any workflow net transition thus has a single configurable port only and we can define a workflow net configuration as follows:

Definition 3.4 (Workflow net configuration) *Let $WF = (P, T, A, L, l)$ be a workflow net. Then $C \in T \rightarrow \{allow, hide, block\}$ is a configuration for WF , and \mathcal{C}_{WF} is the set of all such configurations of WF .*

As workflow net transitions have only a single port, we can here say that the use of the transition is allowed, hidden, or blocked instead of explicitly referring to the port of the transition. Still, we should keep in mind that in fact we configure the port. In Chapter 4 we will see that defining a single port per task is also possible for a number of practical-oriented process modeling languages like SAP WebFlow and BPEL. But, we will also define a configurable YAWL notation which provides more than one port per task.

Figure 3.7 shows a configuration for the workflow net from Figure 2.2. The configuration blocks all transitions in the simple process for domestic travels to make the complex travel approval process compulsory for all travels. However, to not burden the employees with the whole complexity of this process, a comparison of multiple accommodation quotes is not necessary. Thus, this task can be skipped when executing the process which is achieved by configuring the corresponding transition as hidden. Here, it is outside of the secretary's responsibility to prepare travel forms, which is enforced through blocking the preparation of the form by the secretary. In this way, it becomes the employee's responsibility to prepare the travel form when the places p_3 and p_4 are both marked to signal the completion of all quote comparisons.

Moreover, the rejection of requests by the administration is also blocked which in the first moment does not look too bad for employees which have to

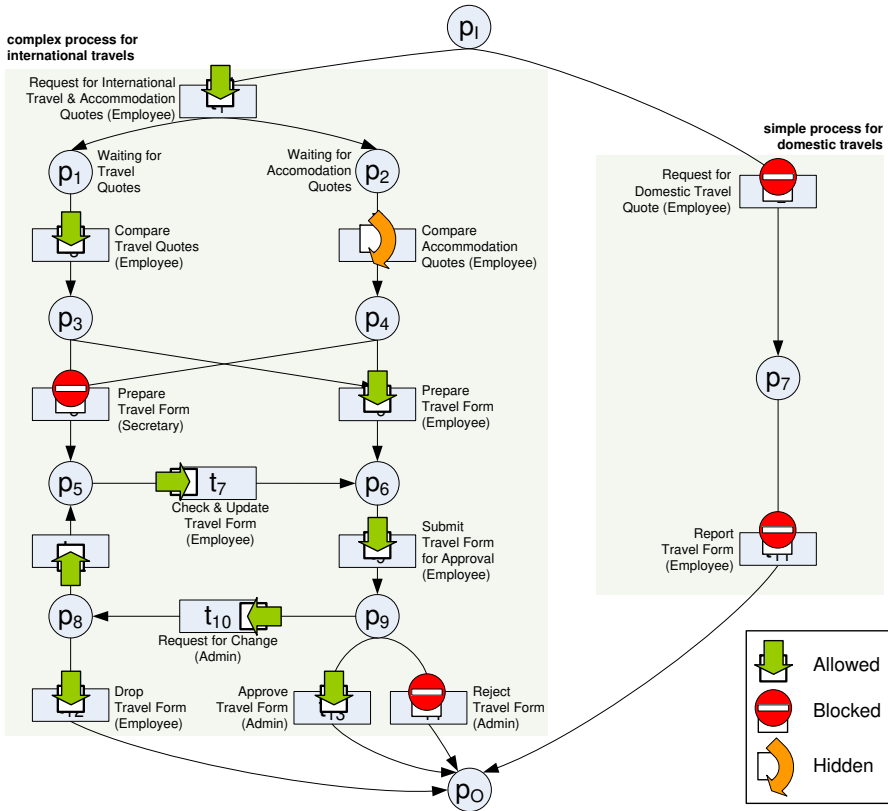


Figure 3.7: Configuring the workflow net of Figure 2.2

travel. Unfortunately, however, the administration just leaves the responsibility for dropping the request to the employee himself by requesting from him a change to the request if it cannot accept the submitted one, i.e. cases reaching p_9 must either be approved, or the administration needs to ask for an update of the request.

The configuration values of transitions can be used to obtain a Petri net representing the behavior of the process model that remains possible according to the configuration. This new Petri net is a restriction of the behavior of the original process model where all the hidden transitions are replaced by silent τ transitions and all the blocked transitions are removed. Also, all the places connected only to blocked transitions and all the flow relations from/to blocked transitions have to be removed. Formally:

Definition 3.5 (Configured workflow net) Let $WF = (P, T, A, L, l)$ be a workflow net and let $\mathcal{C} \in \mathcal{C}_{WF}$ be a configuration of WF . The resulting configured Petri net $WF^{\mathcal{C}} = (P^{\mathcal{C}}, T^{\mathcal{C}}, A^{\mathcal{C}}, L^{\mathcal{C}}, l^{\mathcal{C}})$ is defined as follows:

- $T^{\mathcal{C}} = T \setminus \{t \in T \mid \mathcal{C}(t) = \text{block}\}$,

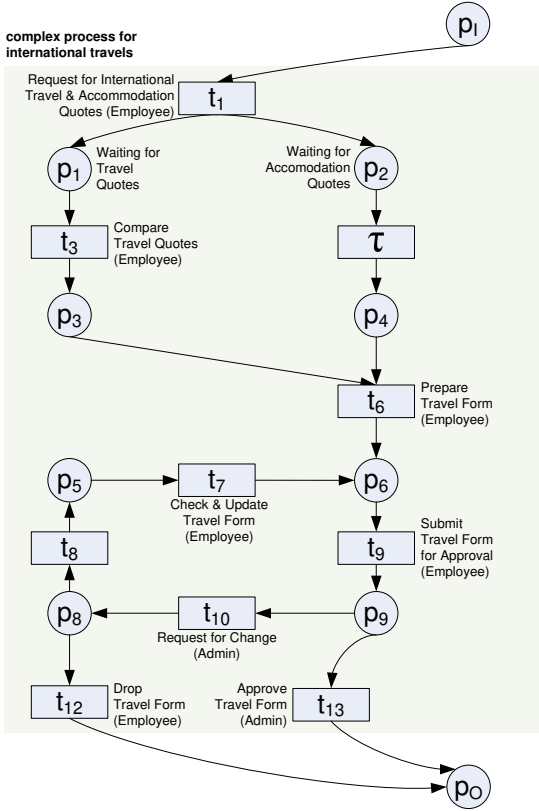


Figure 3.8: The configured workflow net derived from Figure 3.7

- $L^c = \{l(t) \mid t \in T^c \wedge \mathcal{C}(t) = allow\} \cup \{\tau\}$,
- for $t \in T^c$: $l^c(t) = \begin{cases} l(t) & \text{if } \mathcal{C}(t) = allow \\ \tau & \text{otherwise (i.e. } \mathcal{C}(t) = hide) \end{cases}$,
- $A^c = (A \cap ((P \cup T^c) \times (P \cup T^c)))$,
- $P^c = (P \cap \bigcup_{(x,y) \in A^c} \{x, y\}) \cup \{p_I, p_O\}$.

Figure 3.8 shows the configured net derived from the workflow net in Figure 3.7. The *Compare Accommodation Quotes (Employee)* transition has been replaced with a silent transition labeled τ , the transitions *Prepare Travel Form (Secretary)*, *Request for Domestic Travel Quote (Employee)*, *Report Travel Form (Employee)*, and *Report Travel Form (Employee)* have been removed, as well as place p_7 , as it is no longer connected to any of the remaining transitions.

3.2.2 Restricting Configuration Opportunities

As already indicated for LTSs, not all models derived in this way from the configuration of a process model conform to the definition of the used modeling

language or represent desirable behavior. For example, blocking too many ports or a ‘wrong’ port might result in an unconnected net. For many workflow modeling languages this means that the model would become syntactically invalid. In a similar way, hiding of essential tasks, i.e. preventing that important tasks are executed, but continuing with executing the process, can prevent the practicability of the depicted process and it can thus lead to a semantically incorrect model. To avoid the occurrence of such situations, we previously suggested to limit configurations to a pre-defined set of valid configurations. For models with many configurable ports, explicitly listing each valid combination of port configurations is practically infeasible, but also not necessary. The list of valid configurations can be encoded through a **configuration constraint**, restricting the set of permitted combinations of configuration decisions and therefore ensuring both the syntactic and semantic validity of models. Formally:

Definition 3.6 (Configuration constraint) *Let $WF = (P, T, A, L, l)$ be a workflow net and \mathcal{C}_{WF} be the corresponding set of all its configurations. Then $PC : \mathcal{C}_{WF} \rightarrow \{true, false\}$ is a process configuration constraint on WF , and $\mathcal{C}_{WF}^{valid} = \{C \in \mathcal{C}_{WF} | PC(C) = true\}$ is the set of valid configurations of WF .*

An example of a syntactically motivated configuration constraint would be ‘Each task must have at least one port which allows the outflow of cases from that task’; an example of a semantically motivated constraint would be ‘It must always be possible to accept travel requests as well as to reject or drop travel requests’, or better ‘The use of the inflow port of the transition *Approve Travel Form* must always be allowed. Furthermore, if the use of the inflow port of the transition *Reject Travel Form* is blocked, the use of the inflow port of the transition *Drop Travel Form* must be allowed and vice versa’. That means, although this configuration constraint is semantically motivated, it should still be formulated in terms of the model’s port configuration.

For a better understanding, we have presented this example constraint in rather informal natural language. However, the configurable modeling language must be able to test if a configuration of a model satisfies the constraint. Otherwise, the transformation of the model should not be performed. Therefore, a formal specification of configuration constraints is indispensable. We suggest to use propositional logic expressions composed of atomic expressions that test the individual port configurations. In practice, the semantic constraint suggested above would rather look as follows (assuming the process is depicted as a workflow net):

$$\begin{aligned} & (\mathcal{C}(\text{Approve Travel Form}) = allow) \wedge \\ & (\mathcal{C}(\text{Reject Travel Form}) = block \Rightarrow \mathcal{C}(\text{Drop Travel Form}) = allow) \wedge \\ & (\mathcal{C}(\text{Drop Travel Form}) = block \Rightarrow \mathcal{C}(\text{Reject Travel Form}) = allow) \end{aligned}$$

Although solving of such a propositional logic formula is known to be NP-complete, algorithms exist that can efficiently deal with systems of constraints made up of around one million possibilities. This problem is known as the **Boolean Satisfiability Problem (SAT)**. For example, we will later on use a

SAT solver which is based on **Shared Binary Decision Diagrams (SBDDs)** [121]. It allows us to scale with configuration constraints yielding around one million configurations.

How to set up a configuration constraint which preserves the correctness of a configured workflow net will be discussed in more detail in Chapter 8.

3.2.3 Configurable Process Models

By deriving ports from the definition of a process modeling language instead of defining them as elements which have to be added to the process models of the language, each model can serve as the basis for a configurable model without any change. This model represents a subclass of all possible, configured process model variants. It contains the maximal possible behavior which can be achieved by allowing all variants, i.e. configuring the use of all ports as allowed. Thus, it is the basic process model of a configurable process model whose behavior can be restricted by hiding or blocking of selected ports.

A configuration of a basic process model is **valid** if it satisfies all the configuration constraints. The complete basic process model should always conform to the used process modeling notation and thus be syntactically correct. Nonetheless, a configuration allowing the use of all ports is not necessarily valid as it might contain semantically conflicting elements, i.e. the execution of one process part might, e.g. for security reasons, prohibit the execution of another process part. By requiring that every configurable process model should contain at least one valid configuration as a **default configuration**, we ensure the existence of such a valid configuration.

Definition 3.7 (Configurable workflow net) *A configurable workflow net is a tuple (WF, PC, C^{def}) where*

- $WF = (P, T, A, L, l)$ is the basic workflow net and \mathcal{C}_{WF} the corresponding set of all its configurations,
- $PC : \mathcal{C}_{WF} \rightarrow \{true, false\}$ is a process configuration constraint on WF and \mathcal{C}_{WF}^{valid} the corresponding set of valid configurations, and
- $C^{def} \in \mathcal{C}_{WF}^{valid}$ is a valid, default configuration of WF .

Thus, the process models derived from the basic process model through valid configurations are the process variants that can be enacted. This, however, also means that although the basic process model is a correct process model, the complete basic process model is usually not the process model that should be executed. Instead, it is built for being restricted through process configuration, integrating the behavior of various individual process variants available as input. Thus, in fact, we are searching for a common subclass of the available, individual process model variants, optimally a least common multiple of them. In this way, the individual models then represent superclasses, i.e. configurations, of the basic process model (compare Figure 3.2, p. 48).

However, van der Aalst and Basten [5] have shown that a unique least common multiple, and thus an optimal basic process model, often does not exist.

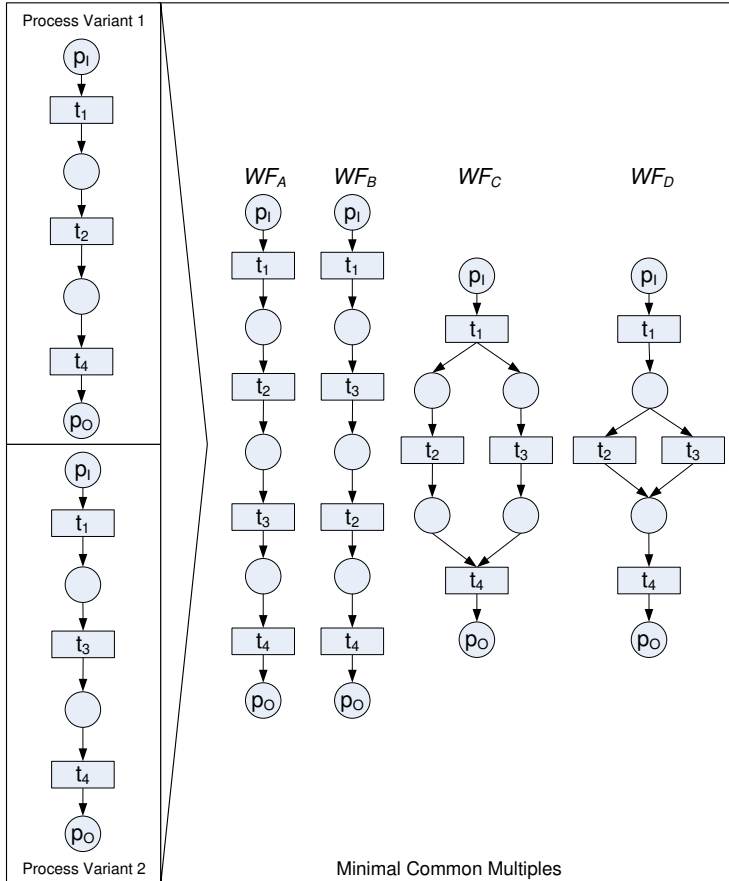


Figure 3.9: There are three different minimal subclasses for the process variants 1 and 2 (adapted from van der Aalst and Basten [5]).

Instead, there is a range of process models which are — from an inheritance perspective — all equally good candidates to form the basic process model. For example, have a look at Figure 3.9. The two process variants which we want to be able to derive from the to-be-formed basic process model both start with an execution of t_1 and complete after firing t_4 . In between, the first variant fires transition t_2 while the second variant fires t_3 . The four processes WF_A , WF_B , WF_C , and WF_D are all subclasses of both processes as they all provide the same behavior as the first process if we hide t_3 in WF_A , WF_B , WF_C , or block t_3 in case of WF_D . Also, they all provide the same behavior as the second process if we hide t_2 in WF_A , WF_B , WF_C , or block t_2 in case of WF_D . Furthermore, all four subclasses contain only the four transitions, i.e. none of them provides any transitions which are not motivated by one of the two process models. Hence, none of them can be classified as the unique least common multiple of the two input processes. Still, we could of course add further transitions to any of them

in such a way that the resulting net would still be a subclass of the two process models. For that reason, the number of transitions that are added to the process model is minimal for all the four subclasses depicted in Figure 3.9. Thus, while none of the classes can be considered as the least common multiple of the two original processes, we can call all of them **minimal common multiples** of the original processes [5].

The decision of which minimal common multiple represents the preferred basic process model depends on both the content and the context of the configurable process model. In Figure 3.9, it seems that t_2 and t_3 are alternatives to each other. Thus, WF_D might be the optimal solution. However, if we consider them as the serving of drinks and the serving of food, it is well possible that both transitions can happen within a single process. For example, the first process might be from a bar which only provides drinks but no food, the second one might be from a place where you buy your drinks and order your meal directly at the counter. As the food is freshly prepared after it has been ordered, it is served to the table as soon as it is ready while the customer takes the drinks directly herself. Still, this does not mean that there are no restaurants where both drinks and food are served. Thus, in this case WF_C might be the best choice for a basic process model since it does not even prescribe a particular order in which the serving has to happen.

The default configuration cannot only serve as a good example configuration, but also as the ‘starting point’ for any individual configuration. In this way, configuring a configurable workflow model to individual requirements just means to modify those port configurations that need to deviate from the default configuration. This can significantly limit the necessary configuration effort, even for models with many configurable ports.

3.3 Related Work

Our goal in this chapter was to discover how the behavior of process models can be configured, i.e. restricted. For this, we took a step backwards and analyzed how this behavior is added to the model in the first place. Other authors took slightly different approaches to identify adaptation options of process models and thus the opportunities to configure process models. In particular, we can compare our results (1) with the results of Puhlmann et al. [132] who searched for adaptation mechanisms described in the literature, (2) with the results of Becker et al. [33, 34] who studied adaptation practices for process models through comparing models developed and adapted in industry, and (3) with the results of Dreiling et al. [59] who analyzed typical control-flow patterns occurring in process models on their configurability.

3.3.1 Literature Study on Variability Mechanisms

Puhlmann et al. [132] performed a literature study on ‘architecturally relevant variability mechanisms’, i.e. variability mechanisms which have a visible impact

on the architecture or design of systems and their models. After identifying these mechanisms, they made suggestions how these mechanisms can be applied to process models, using Unified Modeling Language (UML) activity diagrams, UML state machines, and BPMN as example notations.

The mechanisms identified by them which are relevant for process configuration are **parametrization**, **null classes**, **interface separation**, and **omission of components**. Through parametrization — as suggested by Bachmann and Bass [31], Jacobson et al. [96], and Svahnberg and Bosch [173] — components of a model can be selected or deselected simply by setting or deselecting parameters. Through the re-use of parameters, parametrization thus allows controlling the space of configuration opportunities and restricting it to valid configurations. Nonetheless, if the de-selected behavior is completely blocked or just hidden is unclear. The mechanism of null classes as suggested by Svahnberg et al. [174] allows users to replace optional behavior with empty placeholders which can then be used instead of the behavior. Thus, null classes quasi conform to the τ transitions we introduced earlier when hiding tasks. Through interface separation, also suggested by Svahnberg et al. [174], a number of alternative implementations can be provided of which one is selected by configuration, i.e. its usage is enforced while the others are blocked. The mechanism of omitting components entirely is suggested by Clements and Northrop [43]. However, similar to parametrization it remains unclear, if omitting means that components should just be skipped or that the process flow needs to continue in alternative ways.

3.3.2 Studying of Adaptation Practices

Becker et al. [33, 34] identified adaptation techniques of process models by studying practical models and the changes that have been made to these models. They then distinguish two general types of adaptation techniques: **configuration mechanisms** eliminating irrelevant model elements, and **generic adaptation mechanisms**, subsuming techniques that allow for creative additions to the process model. The mechanisms for restricting the behavior of individual process models out of these adaptation mechanisms is called **element selection**. Similar to the parametrization identified by Puhlmann et al. [132], it allows for the elimination of specific process model elements through setting or de-selecting general configuration parameters or by evaluating logical terms over a number of such parameters. The particular elements are then simply ‘faded out’ [34]. Although it is clear from the description of their approach that this implies a skipping of these tasks, the information how the arcs of the process model connecting these elements should be re-connected after fading out a model element remains fuzzy.

3.3.3 Restricting Choices in Workflow Patterns

For Dreiling et al. [59] process configuration means restricting the choices a user has when executing a business process. For that reason, they examine the list

of control-flow workflow patterns [11] for the choices these patterns incorporate. For eight workflow patterns, Dreiling et al. provide a set of configuration options restricting the behaviors possible according to the original pattern. All the suggested configuration options can be mapped to blocking of process parts, to hiding of process parts, or to creating dependencies between process parts through constraints. In particular, they suggest an optionality pattern basically conforming to the hiding of behavior; they suggest configuration patterns for exclusive choices, multi-choices, simple merges, and interleaved parallel routings which all use the blocking mechanisms; and they suggest a sequence inter-relationship pattern which aims at the restriction of configuration opportunities similar to the configuration constraints we suggested. Dreiling et al. describe the configuration opportunities within their parallel split, synchronization, and synchronizing merge patterns only vaguely, but it seems that these configuration patterns also correspond to the blocking of certain process branches.

3.4 Conclusions

Process configuration means restricting the behavior depicted by a process model. As behavior cannot be inhibited as long as it is not supposed to happen, we analyzed in this chapter how new behavior is added to process models in the first place. For this, we used concepts of inheritance of dynamic behavior. *Configuration of a process model then is the inverse of inheritance of dynamic behavior.*

The two mechanisms used to detect inheritance relations between process models, *blocking* and *hiding*, are already defined in this inverse direction. Hence, these are the tools for process configuration. We can use them to inhibit behavior at inflow ports of tasks, i.e. the moment tasks are usually triggered, and at outflow ports of tasks, i.e. the moment a task completes and indicates the results of its execution.

Nonetheless, not all process model configurations achieved through blocking, hiding, and allowing the use of ports are both syntactically and semantically valid. Therefore, configuration constraints have to limit the configuration space of a process model to those configurations yielding a syntactically and semantically valid process model. The basic process model, used as basis for any process configuration should always be syntactically correct, but might incorporate mutually exclusive behavior. For an easy start of the adaptation process, the configurable process model should come with a default configuration that conforms to the configuration constraints and thus yields to an also semantically valid process model.

Independent of the concrete process modeling notation used, the development of a configurable process model thus always requires the development of a basic process model which integrates all the variants of executing the particular process as well as configuration constraints ensuring that only valid process models, i.e. valid combinations of these process variants, can be derived from the basic process models. How these ideas can be implemented in a sophisticated

and practically used process modeling notation will be subject of the next chapter in which we will develop configuration extensions to the workflow languages of SAP WebFlow, BPEL, and YAWL by applying the concepts suggested in this chapter.

*Every great and deep difficulty bears in itself its own solution.
It forces us to change our thinking in order to find it.
Niels Bohr (1885–1962)*

Chapter 4

Configurable Workflow Languages

Defining configurable workflow nets according to the ideas presented in Chapter 3 was straightforward due to the simple semantics of Petri net transitions. As we have seen in Chapter 2, sophisticated workflow notations like SAP WebFlow, BPEL, or YAWL use various advanced, parameterizable node types to address the different workflow patterns in a simple, i.e. compact, way. Thus, adding configuration opportunities to these more complex notations is not as easy as for workflow nets. In this chapter we will therefore discuss how such advanced workflow notations can be transformed into configurable workflow languages. That means that we will apply the ideas from Chapter 3 (e.g., the usage of configurable ports) to these more advanced notations.

However, while doing so, we will not always ask for identifying each and every port that corresponds to a transition in the underlying LTS and hence for making all these ports configurable. Instead, like the developers of workflow notations, we are looking for a practically manageable, but still sound implementations of configuration opportunities.

For demonstrating how this can be achieved, we will in the following outline how the three workflow notations introduced in Section 2.5 (i.e. SAP WebFlow, BPEL, and YAWL) can be transformed into configurable workflow languages. This chapter starts with sketching how SAP WebFlow can be made configurable. Its block structure enables a very direct application of the configuration techniques of allowing, hiding, and blocking behavior. Next, we will have a look at BPEL to which we can add process configuration in a very similar way as we do for SAP WebFlow, although some additional configurable ports can be identified here. SAP WebFlow and BPEL will already give us a general feeling on how to add configurability to workflow languages.

In a third step, we will then use YAWL to precisely define a configuration extension for a concrete workflow language. We treat YAWL in a more thorough manner because of its extensive workflow pattern support, because both the YAWL notation and its semantics are formally defined, as well as because an open-source workflow system is available for YAWL which allows for an implementation of our ideas. Hence, in the third part of this chapter we will not only sketch, but formally define the configuration extension to YAWL, and we will also provide a transformation algorithm which allows executing configured YAWL models within the YAWL workflow engine.

Independent of the approach suggested in Chapter 3, other researchers have already defined a number of configuration extensions for process modeling languages. In this chapter's related work section we will therefore analyze in which ways our ideas for defining configuration extensions can already be found within these extensions. The chapter ends with drawing some general conclusions on the extendability of process modeling languages with configuration options.

4.1 C-SAP WebFlow

To develop a configuration extension for SAP WebFlow we follow the ideas we introduced in Chapter 3. Thus, we will first identify the ports of the tasks in SAP WebFlow followed by a discussion on how these ports can be configured. Next, we will show how we can depict constraints on these configurations. The introduction of a C-SAP WebFlow (configurable SAP-WebFlow) notation in this section will then end with outlining the changes necessary in SAP's workflow engine to make the configuration extension usable in practice.

4.1.1 Identifying Ports

As discussed in Section 2.5.2 (pp. 40ff), SAP WebFlow models are block structured. In the simplest case a single step, e.g. an activity, represents a block. However, whenever a step causes the branching of the control-flow (as a fork, a condition, a user-decision, or any other step that contains different outcomes) the branching of the control-flow is matched by exactly one corresponding join and all elements until (and including) the matching join belong to the block of the branching step. The elements in each of the branches represent then sub-blocks of the branching block. This is depicted in Figure 4.1. The block of the fork is highlighted in light grey. This block contains two sub-blocks, one for each of the two branches, both highlighted in dark grey. The block of the user-decision step *Approve travel request* branches again in three branches, each containing a block for the particular activity.

Each block can be seen as a task which can contain several sub-tasks. Blocks which contain sub-blocks are tasks on a higher level of abstraction compared to their sub-blocks. The sub-blocks can — as the *Approve travel request* step in Figure 4.1 — again contain sub-blocks which are then on an even lower level of abstraction. In any case, each block contains just one unique input arc and one

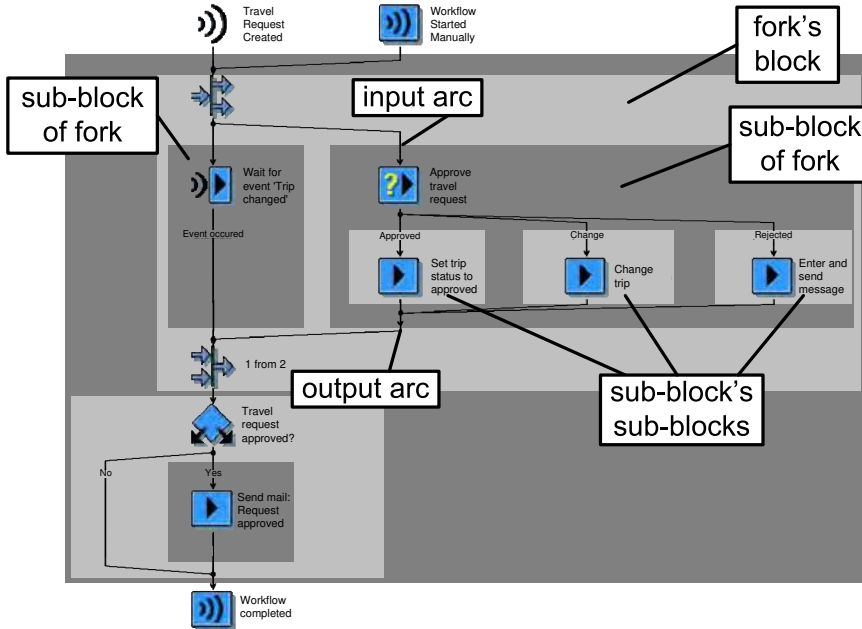


Figure 4.1: SAP WebFlow's block structure.

unique output arc, i.e. there is single entry and a single exit to/from the block. These are thus the ports through which the block, and thus the corresponding task can be triggered or completed. For that reason, let us call the inflow into a block an **input port** and the outflow an **output port**.

The block on the highest level of abstraction is the block of the complete workflow itself. It is the only block which can be triggered in multiple ways as it cannot only be triggered by a manual start of the workflow but also by (various) events which are linked to the workflow block. For example, the workflow from Figure 4.1 cannot only be triggered manually, but also automatically as soon as the event occurs which signalsizes the creation of a travel request. Similarly, events can also be linked to a workflow block to terminate it or to re-start it. Thus, according to our definition of a port from Chapter 3, each of these links connecting events to the workflow block can also be seen as a port to trigger this block. As these ports have some different characteristics from the input and output ports of a block, let us call them **event ports**. In the same way, we also call the linkage between events and *wait for event* steps and between *event creator* steps and events event ports.

In total, we therefore distinguish input ports, output ports, and event ports in a C-SAP WebFlow notation.

4.1.2 Port Configuration

In Chapter 3 we realized that the use of a task's inflow port can be configured as allowed, as hidden or it can be blocked, while the use of outflow ports can only be allowed or blocked. For the input ports of blocks in SAP WebFlow we can apply the possible inflow port configurations in a straightforward manner. If using the input port of a block is allowed, cases of the particular process can normally enter the block, i.e. the corresponding step can be executed. If the input port is configured as hidden, a case entering the block is directly forwarded to the unique output port of the block, bypassing all the content of the block. If a block's unique input port is blocked, the case cannot enter the block at all and needs to continue via alternative branches of the workflow.

For example, let us have a look at Figure 4.2. It depicts a basic process model which combines the process from Figure 4.1 with another travel approval workflow template provided in SAP WebFlow that allows for an automatic approval of travel requests (accessible in SAP WebFlow as workflow WS12500021). In the example, we allow for the entering of cases into most blocks, but we block the input port of the *Change trip* activity (A in Figure 4.2) and we hide the block labeled *Travel request approved?* (B). Thus, by blocking the *Change trip* step the corresponding block is removed from the workflow, whereas the hiding of the *Travel request approved?* step results in skipping this whole block including the sub-block of the *Send mail: Request approved* step. Cases are forwarded directly to the block's output port which means here that the workflow completes immediately after the join condition of the fork is fulfilled.

As the output port of a block is unique, each case entering a block must be able to leave the block via this output port. Thus, a blocking of the output port is not practical as long as cases are able to enter the block. Only if the input port is blocked, the output port can be blocked as well. Such a blocking is however irrelevant because if no cases can arrive at an output port, its configuration has no influence on the process execution anyway. In addition, we already showed in Chapter 3 that hiding of an output port is not really feasible either because the path to the next block does not contain any tasks which could be skipped. If any subsequent block should be hidden, this should rather be done on the subsequent block's input port¹. We can therefore assume that the use of all output ports in an C-SAP WebFlow model is always allowed and hence we consider the configuration of output ports as practically irrelevant.

In SAP WebFlow the triggering of a workflow through an event requires the activation of a link between the event and the workflow. Hence, the SAP WebFlow system already supports the activation and deactivation of such a link. This corresponds to allowing or blocking the particular event port as we require for the configuration of these ports. Although a triggering event port is an inflow port, hiding of such a port is not necessary: it would basically mean skipping the block of the whole workflow without performing any tasks. Thus,

¹To hide a series of blocks, there is an alternative way to using just one configuration port: SAP WebFlow provides a block step into which a sequence of blocks can be encapsulated. In this way, only one input port needs to be hidden.

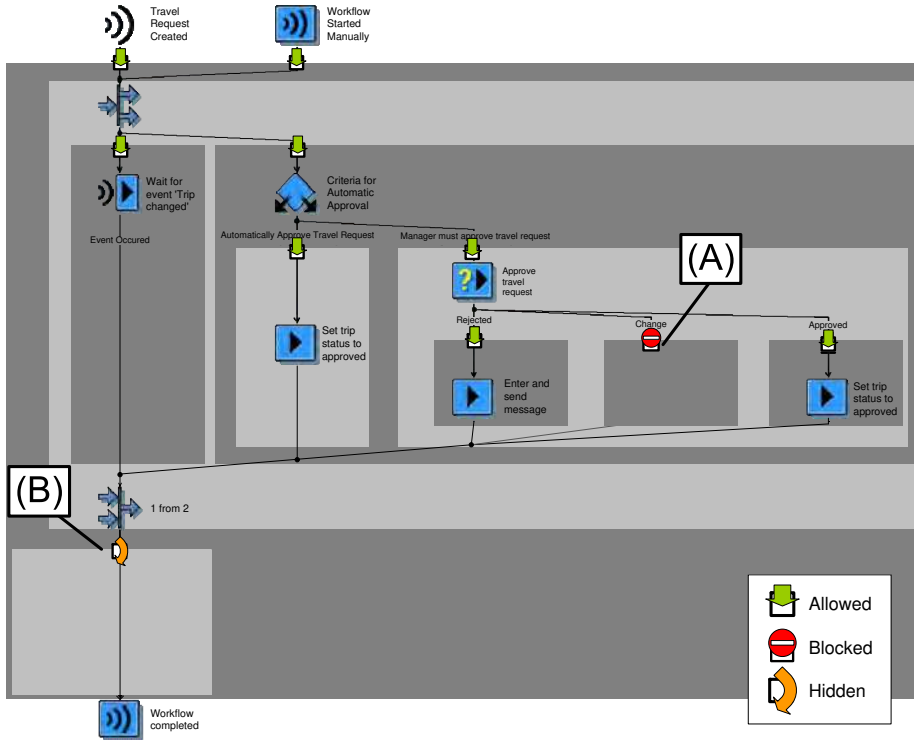


Figure 4.2: The combined workflow template of SAP’s travel approval workflow template (WS20000050) and the automatic approval template (WS12500021), configured as the automatic approval workflow.

the same behavior can be achieved by hiding the corresponding event creator step, i.e. the step in another workflow which triggers the event that activates the particular link. Terminating event ports for wait-for-event steps and the event ports of event creator steps are outflow ports. Even though terminating events are externally triggered, they enforce the termination of the case execution in the particular block. Thus, while SAP WebFlow provides no support for the configuration of input ports, it provides some functionality to allow or block the use of event ports by offering the opportunity to activate or deactivate such linkages.

The configuration depicted in Figure 4.2 corresponds exactly to the SAP WebFlow template WS12500021 for the automatic approval of a travel request which we merged with the template from Figure 4.1. By blocking the sub-block of the *Criteria for Automatic Approval* step’s *Automatically Approve Travel Request* outcome (see (C) in Figure 4.3) and instead enabling the *Change trip* (A) and the *Travel request approved?* (B) blocks, the resulting process is exactly the same approval workflow as the one that was shown in Figure 4.1 (compare Figure 4.1 with Figure 4.3). Of course the workflow in Figure 4.3 still contains

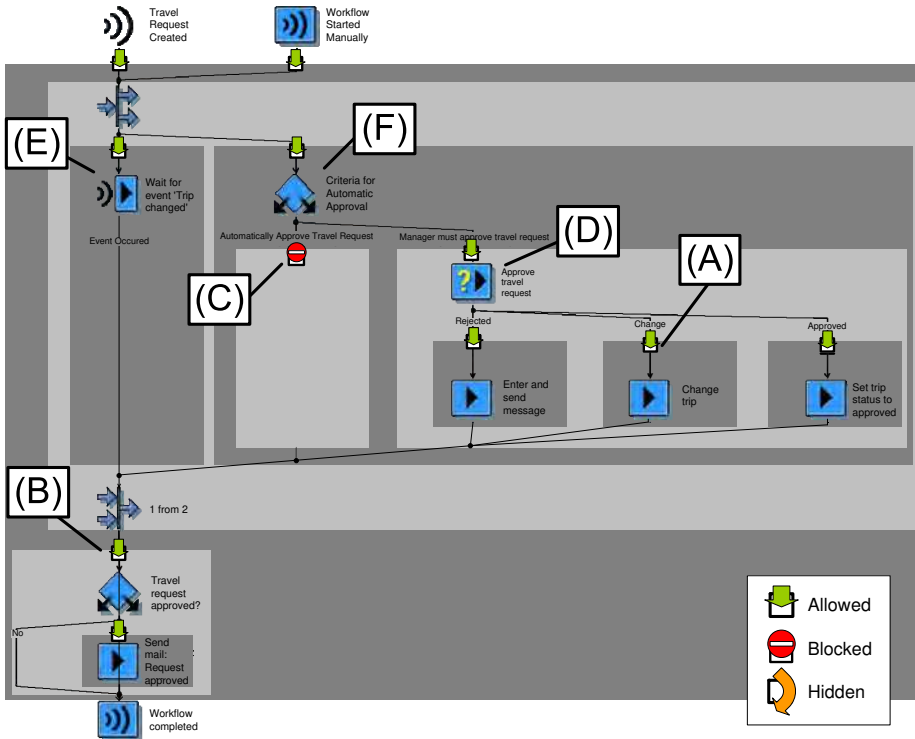


Figure 4.3: The combined workflow template configured such that it corresponds to SAP's travel approval workflow template (WS20000050).

the *Criteria for Automatic Approval* step. However, its only possible outcome is the manual approval sub-block.

4.1.3 Configuration Constraints

As discussed, not all combinations of configuration decisions are feasible in practice. Some combination may lead to non-sound processes, e.g., because they contain deadlocks, or they simply do not make any sense from a content point of view.

The soundness criteria for workflow models as defined in Definition 2.18 (p. 24) require that cases must always have a chance to complete a workflow. That means, they should never get stuck during the execution of a workflow. For that reason, the input port of a block in SAP WebFlow can only be configured as blocked if an alternatively executable process branch exists. In the example from Figure 4.3, this is no problem in case of the (automatic) *Set trip status to approved* step's block (C in Figure 4.3) because alternatively the (manual) *Approve travel request* step's block (D) can be executed. It is however impossible to block the input port of the *Travel request approved?* step's block (B) as the

workflow does not contain any alternative to it which leads to the completion of the workflow. In the case of this particular fork step, it is theoretically possible to block one of the two (dark grey) sub-blocks (E , F), but only because the join requires just one of the two branches to complete. If the condition at the join would have been *2 From 2* a blocking of one of the sub-blocks would have made it impossible to later satisfy this condition and thus caused a deadlock. Therefore, when configuring the sub-blocks of a fork, the condition at the joining fork determines the maximal amount of sub-blocks that can be blocked.

Also from a content perspective the workflow of Figure 4.2 cannot be configured freely. Although, it is, e.g., well possible to block or allow the use of the *Wait for event ‘Trip changed’* step’s block, hiding it would prevent the workflow from working correctly. It would cause a direct forwarding of cases to the joining fork. Since only one of the two branches needs to complete, the join condition would immediately be satisfied. Thus, the other branch would get superfluous and canceled before any decision on the approval can be made. As another example, the *Approved* sub-block of the *Approve travel request* step is mandatory for this particular workflow (there is no point of having a travel approval process if it does not allow for the approval of the process). Hence, the use of this block must be allowed in any configuration.

The setup of such constraints in first-order logic can be done exactly as suggested in Chapter 3. That means, we can for example write $\mathcal{C}(\text{Enter and send short message}) = \text{allow}$ to depict that the use of the particular block must always be configured as allowed or $\mathcal{C}(\text{Wait for event ‘Trip changed’}) \neq \text{hide}$ for the constraint that a block cannot be hidden.² Such atomic statements can then be combined, e.g., to formulate a constraint that if the *Change trip* block is blocked then the block *Travel request approved?* must be hidden ($\mathcal{C}(\text{Change trip}) = \text{block} \Rightarrow \mathcal{C}(\text{Travel request approved?}) = \text{hide}$). To test if a configuration fulfills all such constraints, the constraints are concatenated using logical \wedge operators. Then, a SAT solver can be used to evaluate the constraints [121]. In this way, the SAT solver identifies which blocks need to be bound to a certain configuration value in order to fulfill the constraints. Thus, it identifies the necessary configuration values from the constraint. Those blocks which are not bound to a certain value are the blocks which really are configurable.

4.1.4 Process Enactment

The current SAP WebFlow templates allow for an easy integration of predefined workflow templates into a running SAP system by just assigning the relevant resources to the steps and activating the triggering events. To enable such an easy activation for configurable workflow templates, each template has to have a default configuration. A default configuration can be any configuration satisfying the constraints specified for the workflow. For example, the configuration of Figure 4.2 representing the automatic approval template could be the default

²Note, that in a real implementation scenario we would rather use a unique block identifier than a step’s name, as step names can occur multiple times in a single workflow.

configuration for the combined travel approval workflow template. When activating the triggering event, the workflow corresponding to this configuration would automatically be enabled. However, if it is for example desired that it can be asked for a change to the travel request, it is sufficient to assign the responsible resource to the *Change trip* step and allow the use of the currently blocked port. Without any modeling effort the new configuration of the workflow template can be used.

As configured templates can easily be transformed into executable workflow models according to the current SAP WebFlow notation, it is not necessary to change SAP's WebFlow engine to run workflows derived through the suggested configurable extension of SAP WebFlow. Instead, an implementation of the suggested approach in the context of SAP's enterprise system would solely require (1) an extension of the user interface that depicts the configuration options as well as allows managing configurations, (2) a tool that checks the fulfillment of the configuration constraints to prevent invalid configurations, and (3) an implementation of the transformation algorithm to derive configured workflow models from configurations.

4.2 C-BPEL

As a second workflow language to which we add configuration, let us have a brief look at BPEL. To add configuration options to BPEL, it is again necessary to first identify the ports in the BPEL notation and define how they can be configured before constraints among configuration decisions and their executions can be discussed.

4.2.1 Ports and their Configurations

BPEL uses two ways to express the control-flow of processes. On the one hand, BPEL activities are block structured, similar to steps in SAP WebFlow. On the other hand outgoing and incoming control links can be used to break this block structure (see Section 2.5.3, pp. 42ff).

Within the block structure BPEL activities have the same unique input ports and the same unique output ports as the steps of SAP WebFlow. Thus, we can allow, block, or hide the use of input ports of activity blocks in the same way as we have suggested for SAP WebFlow and we do not need to configure the output ports. For example, Figure 4.4 shows a configuration for the BPEL process from Figure 2.12. In Line 17 the invoke activity *getTicket* is extended with a configuration parameter, showing that the particular block is configured as blocked. In the same way, the invoke activity *orderCard* in Line 20 is configured as hidden.

Control links allow for expressing control-flow relations that break the block structure of BPEL activities like sequence, flow, etc. To overcome the block structure, an activity not only triggers the unique output port when it completes, but it can also activate a number of outgoing control links. Which

```

1 <process ...>
2   <sequence ...>
3     <flow ...>
4       <links>
5         <linkName="trainTicket"/><linkName="reductionCard"/><linkName="hotel"/>
6       </links>
7       <invoke partner="BookingEngine" operation="requestTravel"
8         inputVariable=... outputVariable="travelNeeds" ...>
9         <source linkName="trainTicket"
10          transitionCondition="bpws:getVariableData(travelNeeds, trainReq)=true"
11          CONFIGURATION="blocked" />
12         <source linkName="reductionCard"
13          transitionCondition="bpws:getVariableData(travelNeeds, cardReq)=true"/>
14         <source linkName="hotel"
15          transitionCondition="bpws:getVariableData(travelNeeds, hotelReq)=true"/>
16       </invoke>
17       <invoke partner="TicketProvider" operation="getTicket" CONFIGURATION="blocked" ...>
18         <target linkName="trainTicket"/>
19       </invoke>
20       <invoke partner="CardProvider" operation="orderCard" CONFIGURATION="hidden" ...>
21         <target linkName="reductionCard"/>
22       </invoke>
23       <invoke partner="HotelReservationSystem" operation="reserve" ...>
24         <target linkName="hotel"/>
25       </invoke>
26     </flow>
27     <flow ...>
28       <links>
29         <linkName="creditCardPayment"/><linkName="cashPayment"/>
30       </links>
31       <invoke partner="PaymentEngine" operation="getPaymentDetails"
32         inputVariable=... outputVariable="paymentDetails" ...>
33         <source linkName="creditCardPayment"
34          transitionCondition="bpws:getVariableData(paymentDetails, card)=true"/>
35         <source linkName="cashPayment"
36          transitionCondition="bpws:getVariableData(paymentDetails, cash)=true"/>
37       </invoke>
38       ...
39     </flow>
40     ...
41   </sequence>
42 </process>

```

Figure 4.4: The travel booking process from Figure 2.12 extended with configurations.

control links are activated depends on a condition that can be specified for each outgoing control link separately. For example, the invoke operation *request travel*, specified in Line 7 of Figure 4.4, incorporates three such outgoing control links, named *trainTicket*, *reductionCard*, and *hotel*. Each of them is activated if the corresponding parameter is *true*, i.e. the hotel link is only activated if the *hotelReq* parameter is *true* (Line 14), the reduction card link is activated if the *cardReq* parameter is *true* (Line 12) etc. Thus, outgoing control links are in an OR-split relation and any combination of links can be activated.

According to the underlying LTS this means that each possible combination results in a different state. Hence, in a direct application of the port concept based on the LTS, we would need to introduce a dedicated link port for each combination of outgoing links that can be activated at the same point in time.

However, as we said in the beginning of this chapter, practically such an

enormous amount of configuration ports is often not manageable. For that reason, we suggest addressing only a subset of all possible ports as an alternative here:³ We simply say that each link depicts a dedicated **outgoing link port** whose use can be allowed or blocked. Although, it is in this way, e.g., impossible to say that tickets and reduction cards can only be ordered separately but not together, this solution is quite natural for anyone familiar with BPEL: If the use of such a port is allowed, the activation of the particular link is subject to the corresponding condition. If the use of the port is blocked, the link of the port can never be activated, i.e. a continuation of the process through the link will always be inhibited. This, e.g., applies to the *requestTravel* invoke activity's *trainTicket* link in Figure 4.4, lines 9–11. As it is configured as blocked, it cannot be activated anymore, even if the condition that the variable *trainReq* is *true* would be fulfilled.

Incoming control links ensure that activities can only be executed if they are on the one hand triggered through the structure of the workflow, and if all the incoming links have been activated on the other hand. Therefore, all incoming links and the input port of the activity synchronize the workflow in the same way as an AND-join. For specifying ports this means that, even if we consider incoming links in addition to the input port according to the block structure, an activity only has a single input port which can be configured. Thus, this input port is the same input port as the one based on the block structure mentioned before whose use can be allowed, blocked, or hidden.

In a C-BPEL notation, each activity block can therefore be configured either as allowed, as blocked, or as hidden, and each of the outgoing links of activity blocks can be configured as either allowed or blocked.

4.2.2 Executability of BPEL Configurations

Configuration requirements for C-BPEL can be specified and evaluated using boolean expressions, exactly as we sketched for C-SAP WebFlow. Hence, we skip elaborations on configuration constraints in the context of C-BPEL here. Instead, we focus on how a BPEL specification that behaves according to the specified configurations can be constructed from the configuration decisions. Thus, we have to analyze how the model has to change if an input port is configured as hidden, if an input port is configured as blocked, and if an outgoing link port is blocked.

Dealing with activities with input ports that are configured as hidden is straightforward. As the corresponding activity needs to be skipped during process execution, the activity is simply replaced with a dummy *invoke* activity, which completes without executing any work.

Adapting the process to reflect blocked activities is in principle similarly straightforward. All activities with blocked input ports and their sub-blocks should be completely eliminated from the workflow. However, it is important

³Note that we will discuss the implementation of a configuration extension which addresses all the ports of an OR-split according to the LTS in the context of YAWL in the next section.

```

1 <process ...>
2   <sequence ...>
3     <flow ...>
4       <links>
5         <linkName="trainTicket"/><linkName="reductionCard"/><linkName="hotel"/>
6       </links>
7       <invoke partner="BookingEngine" operation="requestTravel"
8         inputVariable=... outputVariable="travelNeeds" ...>
9         <source linkName="reductionCard"
10          transitionCondition="bpws:getVariableData(travelNeeds, cardReq)=true"/>
11         <source linkName="hotel"
12          transitionCondition="bpws:getVariableData(travelNeeds, hotelReq)=true"/>
13       </invoke>
14       <invoke operation="dummy">
15         <target linkName="reductionCard"/>
16       </invoke>
17       <invoke partner="HotelReservationSystem" operation="reserve" ...>
18         <target linkName="hotel"/>
19       </invoke>
20     </flow>
21     <flow ...>
22       <links>
23         <linkName="creditCardPayment"/><linkName="cashPayment"/>
24       </links>
25       <invoke partner="PaymentEngine" operation="getPaymentDetails"
26         inputVariable=... outputVariable="paymentDetails" ...>
27         <source linkName="creditCardPayment"
28          transitionCondition="bpws:getVariableData(paymentDetails, card)=true"/>
29         <source linkName="cashPayment"
30          transitionCondition="bpws:getVariableData(paymentDetails, cash)=true"/>
31       </invoke>
32       ...
33     </flow>
34     ...
35   </sequence>
36 </process>

```

Figure 4.5: The travel booking process derived from the configuration of Figure 4.4.

to note that this results in semantic problems when sub-blocks of a *sequence* or a *while* loop are blocked. If these sub-blocks are removed, the BPEL semantics imply that the execution continues with the next task in the sequence or that the workflow contains an empty loop. Thus, the activity's execution would not be blocked completely, but rather skipped — as if we would have configured the activity as hidden. An obvious solution to prevent this, would be the introduction of new semantics in line with the configuration idea that could prevent this. However, then the blocking of an activity in a sequence or a loop would result in a deadlock or livelock which is undesired behavior as well. Hence, blocking is generally considered as not possible for the input ports of sub-blocks of these two BPEL activity types. Instead, the whole enclosing sequence or while activity should be blocked. Blocked outgoing link ports are simply removed from the workflow. In this way they cannot be activated anymore and are thus effectively blocked.

Figure 4.5 shows the resulting BPEL process for our example where the link to the *getTicket* activity as well the activity itself are removed, and the *orderCard* activity is replaced by a dummy activity (see Line 14).

The straightforward definitions of ports and transformation rules of C-BPEL,

as well as the minor modification to the BPEL schema necessary for this, allow for an easy implementation of process configuration within BPEL tools. For this, the process definition tools could, e.g., provide a context menu for configurable items which allows for setting the configuration values. Furthermore, the definitions tools should test if the desired configuration decisions fulfill the configuration constraints. If not, the decisions should be automatically prevented. BPEL workflow engines would need to apply the depicted transformation rules when loading C-BPEL models. In this way, the execution engine itself would be provided with a traditional BPEL process. Thus, it would not be required to change the execution engine itself at all.

4.3 C-YAWL

While we rather informally described the C-SAP WebFlow and C-BPEL notations, let us now discuss and define a configuration extension to YAWL, i.e. C-YAWL, in detail. Since YAWL supports more workflow patterns than most languages (including SAP WebFlow and BPEL), it is worthwhile to study this language in more detail. This section is divided in five subsections. In the first three subsections we will analyze how we can define a configuration extension for EWF-nets as the notation for creating process models in YAWL. At first, configuration ports are identified, as well as the concrete configuration options available here. Secondly, a notation to specify configuration constraints over these configuration opportunities is defined, before these two components are combined into a notation for configurable EWF-nets (C-EWF-nets) in the third subsection. In the fourth subsection, the configurability of the hierarchy among the various C-EWF-nets will be analyzed. Last, an algorithm that transforms configured YAWL models into traditional YAWL models is shown, along with an outline of its implementation in the YAWL software system.

4.3.1 Configurable Elements of EWF-Nets

To determine the configurable elements of EWF-nets, the flow of cases through the tasks of EWF-nets needs to be analyzed. Obviously, in EWF-nets the arcs surrounding tasks depict how tokens can flow into and out of the tasks. However, the execution of a task is only enabled if the tokens in the pre-conditions of the task match the task's join behavior. A task with an AND-join behavior (as task *Reserve* in Figure 2.10, p. 37) can only be enabled and executed if tokens are waiting at all conditions preceding the task. That means, in a similar way as the blocks in SAP WebFlow or BPEL, this task contains only a single port through which it can be enabled. However, in YAWL this is different for a task with an XOR-join behavior. Such a task can be enabled via every arc pointing at it. That means that such a task has a dedicated port for each of these arcs. Let us use the EWF-net in Figure 4.6 (which is the main EWF-net from Figure 2.10, p. 37) as an example to visualize this. For example, the task *Send documents* in Figure 4.6 can be triggered either by a token in condition a_4 or by a token

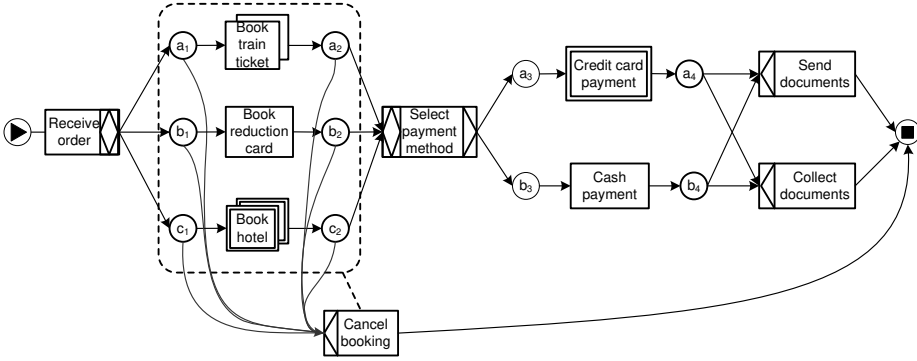


Figure 4.6: The main EWF-net from the booking process depicted in Figure 2.10.

in condition b_4 . Thus, it can be enabled through two different ports. A task with an OR-join behavior (like the task *Select payment method* in Figure 4.6) is only enabled if there is at least one token on one of its input conditions and if it is impossible that further tokens can arrive. Thus, similar to the AND-join, it synchronizes all branches. We therefore assign only a single port to the OR-join even though it can be enabled via any combination of input branches. All ports through which a task in an EWF-net can be enabled are called **input ports** in the following.

Definition 4.1 (Input ports)

Let $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net. Then

- $ports_{in}^{XOR}(EWF) = \{(t, \{c\}) \in T \times IP(K) \mid join(t) = XOR \wedge c \in \bullet t\}$ are the input ports for all tasks with an XOR-join behavior,
- $ports_{in}^{\wedge}(EWF) = \{(t, \bullet) \in T \times IP(K) \mid join(t) = \wedge\}$ are the input ports for all tasks with an AND-join behavior,
- $ports_{in}^{\vee}(EWF) = \{(t, \bullet) \in T \times IP(K) \mid join(t) = \vee\}$ are the input ports for all tasks with an OR-join behavior,
- $ports_{in}(EWF) = ports_{in}^{XOR}(EWF) \cup ports_{in}^{\wedge}(EWF) \cup ports_{in}^{\vee}(EWF)$ are all input ports of EWF, and
- for $t \in T$, $ports_{in}(t) = ports_{in}(EWF) \cap (\{t\} \times IP(K))$ are all input ports of the task t .

Looking at the split behavior of tasks, we find similar semantics. If a task with an XOR-split behavior completes, it can choose between all outgoing arcs through which it will release the case. The task *Select payment method* from Figure 4.6 can, e.g., release a token either to condition a_3 or to condition a_4 . That means, each of the outgoing arcs from the task to the succeeding conditions has its own port. A task with an AND-split behavior like the *Start search* task releases tokens always via all outgoing arcs, i.e. it is only using a single port. Tasks with an OR-split behavior can release tokens to post-conditions via any combination

of outgoing arcs. This means for the task *Receive order* from Figure 4.6 that after its completion it can put tokens either in $a1$, in $b1$, in $c1$, in $a1$ and $b1$, in $a1$ and $c1$, in $b1$ and $c1$, or into all three of these conditions. According to the underlying LTS each of these ways tokens can be released represents a different state change. Hence, a port exists for each of these combinations.

Ports through which tokens are released to post-conditions are called **output ports** in the following.

Definition 4.2 (Output ports)

Let $EWf = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an *EWf-net*. Then

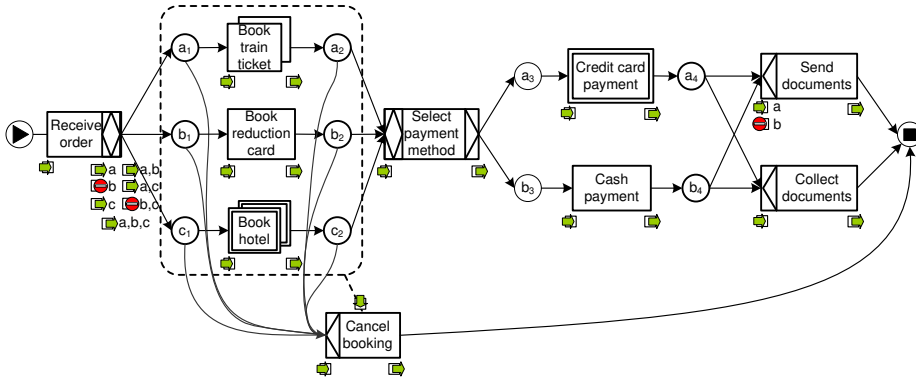
- $ports_{out}^{XOR}(EWf) = \{(t, \{c\}) \in T \times \mathcal{P}(K) \mid split(t) = XOR \wedge c \in t\bullet\}$ are the output ports for all tasks with an *XOR-split* behavior,
- $ports_{out}^{\wedge}(EWf) = \{(t, t\bullet) \in T \times \mathcal{P}(K) \mid split(t) = \wedge\}$ are the output ports for all tasks with an *AND-split* behavior,
- $ports_{out}^{\vee}(EWf) = \{(t, cs) \in T \times \mathcal{P}(K) \mid split(t) = \vee \wedge cs \subseteq t\bullet \wedge cs \neq \emptyset\}$ are the output ports for all tasks with an *OR-split* behavior,
- $ports_{out}(EWf) = ports_{out}^{XOR}(EWf) \cup ports_{out}^{\wedge}(EWf) \cup ports_{out}^{\vee}(EWf)$ are all output ports of *EWf*, and
- for $t \in T$, $ports_{out}(t) = ports_{out}(EWf) \cap (\{t\} \times \mathcal{P}(K))$ are all output ports of the task t .

Each input port can be configured as either **allowed**, as **blocked**, or as **hidden**. This determines if the corresponding task can generally be triggered through the particular port or not. For executing a task at run-time, it is therefore necessary that the use of one of the ports through which the task is enabled is configured as allowed, and that the particular task is enabled through tokens in the corresponding pre-conditions.

The configuration of an output port determines which subsequent (post-) conditions can be marked with tokens after a task has been completed. In line with our definition of outflow ports in Chapter 3 its use can be configured either as allowed or as blocked only.

Figure 4.7 provides two example configurations of the input and output ports for the *EWf-net* from Figure 4.6. The travel agency depicted in Figure 4.7a only sells reduction cards to those clients buying a train ticket at the same time. Train tickets and hotel reservations can also be booked independently. For that reason the output ports of the *Receive order* task which theoretically allow for booking the reduction card without booking a train ticket are blocked (indicated by the ‘Do not enter’-signs labeled b and b, c at the bottom-right corner of the task). The use of the other five output ports, representing all other possible booking combinations, is allowed (indicated by the arrows at the bottom-right corner of the task). Only if the customer pays by credit card, the documents can be sent to him. If the customer pays in cash, the documents cannot be sent. Then he has to collect them. This policy is enforced by blocking the input port b of the *Send documents* task (indicated by a ‘Do not enter’-sign at the bottom-

a)



b)

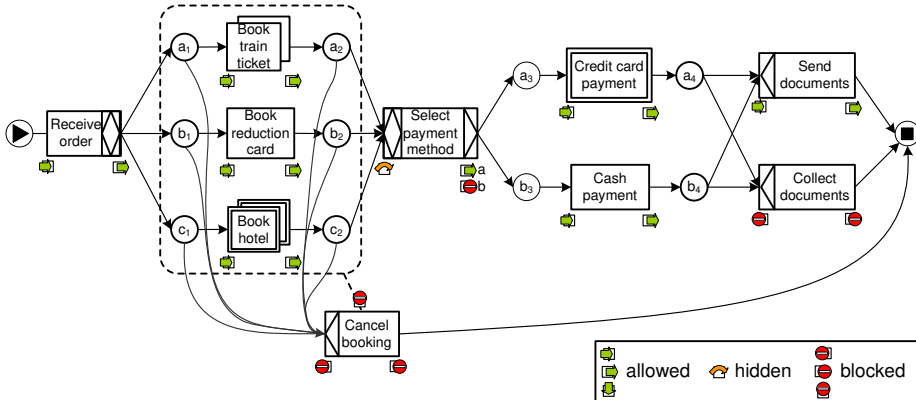


Figure 4.7: The booking process from Figure 4.6, configured to the requirements of (a) a travel agency and (b) an internet shop.

left corner of the task). The use of all other tasks in the model is allowed, i.e. all their ports are configured as allowed.⁴

The process of an internet shop depicted in Figure 4.7b is based on the same basic process model, but uses a different port configuration. It sells reduction cards also without train tickets, payments are only possible by credit card, and documents cannot be collected. In addition, the internet shop does not allow users to cancel their bookings. For that reason, all output ports of the *Receive order* task are configured as allowed, the input port of the *Cancel booking* task is blocked, output port *b* of the *Select payment method* task is blocked, and all input ports of the *Collect documents* task are blocked. In addition, the *Select payment method* task's input port is configured as hidden (indicated by the

⁴Note that whenever all input or output ports of a task are configured to the same value, we just show a single symbol without port names. For example, see the output ports of task *Select payment method*.

‘jumping’ arrow at the bottom-left corner of the task) because the internet shop only offers a single payment method: A selection simply does not need to be made.

In addition to the tokens consumed by a task via the input ports, a task in YAWL can also consume all tokens from a cancelation region. For this reason, we decided to define a **cancelation port** per task in addition to the input ports.

Definition 4.3 (Cancelation ports)

Let $EFW = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EFW -net.

Then $ports_{rem}(EFW) = dom(rem)$ are all the cancelation ports of EFW .

The consumption of tokens from a cancelation region is similar to the consumption of tokens by an OR-join. During the task’s execution always all available tokens are consumed from the cancelation region. However, tokens in the cancelation region cannot trigger a task on their own. The triggering still happens via the input ports, and the enabling of a task does not even require the availability of tokens in the cancelation region. If the cancelation port’s use is configured as allowed, tokens are removed from the cancelation region; if it is blocked, they are not. The decision whether a task is executed or skipped remains determined by the input port configuration. This means, the cancelation port is only a refinement of the input port — the inflow port according to the underlying LTS is a combination of an input port with the cancelation port. This is also the reason why only input ports can be configured as hidden, but cancelation ports cannot.

To depict the configuration of cancelation ports, we use the same pictures of an arrow for cancelation ports which are configured as allowed to be used and a ‘Do not enter’-sign for blocked cancelation ports (see icon on to the arc leaving the top of the task *Cancel booking* in Figure 4.7).

If a task allows for the start of multiple instances, it in fact combines several tasks of the process in a single task. To implement this behavior, we could, e.g., introduce an internal OR-split (see Figure 4.8) that enables the instances of the task. Of course, the output ports of the OR-split can be configured as allowed to be used or as blocked. If some ports of this OR-split are blocked, this might decrease the maximal number of instances that can be started or increase the minimal number of instances that have to be started. For example, the task depicted in Figure 4.8 originally allowed a minimum of a single instance of the task and a maximum of three instances of the task to be started. In the depicted configuration the minimal number of instances that can be started is increased to two. For this, all ports of the task that creates the individual instances and that are connected to a single subsequent condition only are blocked. In this way, at least two instances must be started in any execution of the *Create instances* task. The maximal number of instances that can be started is also reduced to two. For this, that output port of the *Create instances* task that allows starting all instances is also blocked. Therefore, instead of saying that instances are blocked, we will refer to **increasing the minimum number of instances to be started** and **decreasing the maximum number of instances to be started** in the following. If a task allows for the dynamic

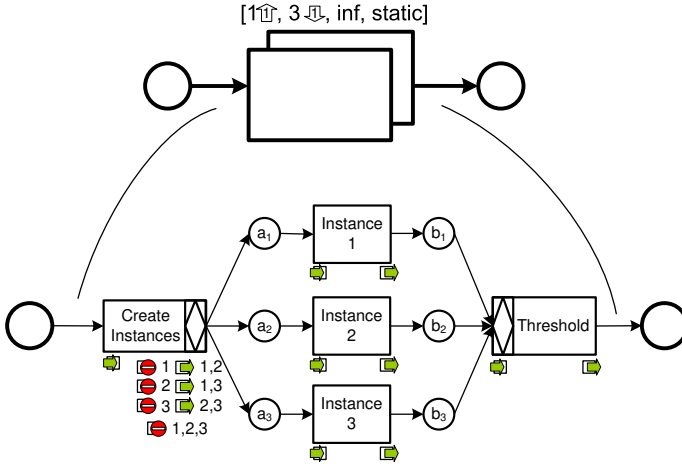


Figure 4.8: A multiple instance task with one to three instances implemented: the configuration $\uparrow\downarrow$ restricts the behavior to the start of exactly two instances.

creation of instances, the task has an additional internal task which creates new instances. By blocking its input port, we **restrict a task with a dynamic creation of task instances to a static creation of task instances**.

When configuring EWF-nets there is also the opportunity to reduce the threshold value of the number of instances that have to be completed to consider the whole task as being completed. This behavior — also known as N-out-of-M-join pattern [11] — can be implemented in an EWF-net by using several AND-joins instead of one OR-join for joining the multiple instances, each connected to the required minimal number of completed instances. On firing, such an AND-join could cancel all remaining instances. Figure 4.9 provides an example for this behavior. Originally, the task required the creation of three instances, but only two instances had to complete to consider the task as completed. By blocking the input ports of those AND-joins that require only two instances to be completed, we increase the threshold value of the task to three. Thus, also the increase of the threshold value of a multiple instance task is possible by means of configuration.

To formalize these four types of configuration opportunities we use four configuration functions, one for each type. The configuration functions assign the described configuration decisions to the ports of EWF-nets. To allow for the configuration of selected parts of an EWF-net, we define the configuration functions as partial functions. Then a configuration does not need to configure every port in the EWF-net.

Definition 4.4 (EWF-net Configuration)

Let $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net, $ports_{in}(EWF)$ be the input ports of EWF, and $ports_{out}(EWF)$ be the output ports of EWF. Then $\mathcal{C}_{EWF} = (\mathcal{C}_{in}, \mathcal{C}_{out}, \mathcal{C}_{rem}, \mathcal{C}_{nofi})$ is a configuration of EWF with

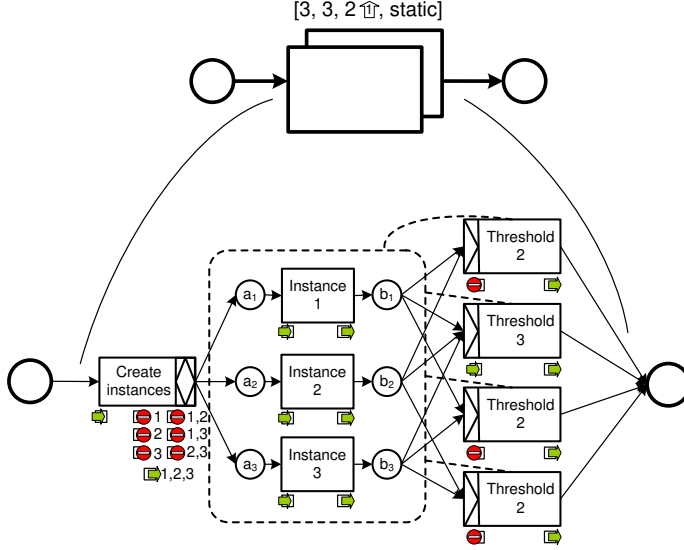


Figure 4.9: Increasing the threshold value from two to three within a task with three instances to be started (example implementation).

- C_{in} defined as a partial function determining configurations for the input ports of tasks:

$$C_{in} : ports_{in}(EWF) \not\rightarrow \{allow, block, hide\}$$

- C_{out} defined as a partial function determining configurations for the output ports of tasks:

$$C_{out} : ports_{out}(EWF) \not\rightarrow \{allow, block\}$$

- C_{rem} defined as a partial function determining configurations for the cancellation regions of tasks:

$$C_{rem} : ports_{rem}(EWF) \not\rightarrow \{allow, block\}$$

- C_{nofi} defined as a partial function determining configurations for the multiplicity of tasks:

$$C_{nofi} : dom(nofi) \not\rightarrow (\mathbb{N}^0 \times \mathbb{N}^{0,\infty} \times \mathbb{N}^{0,\infty} \times \{restrict, keep\})$$

such that

for all $t \in dom(C_{nofi}) : (C_{nofi}(t) = (min, max, thres, dyn)$ and

$$\pi_1(nofi(t)) + min \leq \pi_2(nofi(t)) - max.$$

\mathcal{C}_{EWF} denotes the set of all such configurations of EWF.

Similar as for EWF-nets, we use $\pi_1(\mathcal{C}_{nofi}(t))$ to refer to the increase of the minimal number of instances that have to be created for task t , $\pi_2(\mathcal{C}_{nofi}(t))$ to refer to the decrease of the maximal number of instances that can be created, $\pi_3(\mathcal{C}_{nofi}(t))$ for the increase of the threshold value, and $\pi_4(\mathcal{C}_{nofi}(t))$ to indicate whether the creation of instances for the task t should be restricted to a static creation of instances only.

To be able to transform an EWF-net into a lawful, configured EWF-net, the configuration functions must be defined on every port, i.e. they must be total functions instead of partial functions. If all four configuration functions are total functions, we call the configuration **complete**.

Definition 4.5 (Complete Configuration)

Let $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, \text{split}, \text{join}, \text{rem}, \text{nofi})$ be an EWF-net. Then the configuration \mathcal{C}_{EWF} of EWF is complete if and only if

- $\text{dom}(\mathcal{C}_{in}) = \text{ports}_{in}(EWF)$,
- $\text{dom}(\mathcal{C}_{out}) = \text{ports}_{out}(EWF)$,
- $\text{dom}(\mathcal{C}_{rem}) = \text{ports}_{rem}(EWF)$, and
- $\text{dom}(\mathcal{C}_{nofi}) = \text{dom}(\text{nofi})$.

$\mathcal{C}_{EWF}^{complete}$ denotes the set of all complete configurations of EWF.

When configuring a process, we most of the time do not want to configure all ports explicitly, but rather only those ports that deviate from a standard configuration. Then it is necessary to combine the partial, individual configuration with a default configuration to form a complete configuration. To form complete (or at least ‘more complete’) configurations out of incomplete configurations the domains of the configuration functions from the first configuration have to be extended with the domains of the configuration functions from the second configuration. If, for example, a configuration has only configuration values for the ports of the task *Select payment method* from Figure 4.7 while a second configuration has configuration values for the ports of the tasks *Receive order* and *Book train ticket*, the combined configuration has configuration values for all three tasks. In case both configurations contain a configuration value for a particular port, the value of the first configuration is used. Therefore, the resulting, combined configuration depends on the order of the input configurations which means, combining configurations is not a symmetric operation. But, in this way, combining incomplete configurations with complete configurations always creates complete configurations. The configuration values of the incomplete configuration then overwrite the ones of the complete configuration — as we need it when we want to add individual configuration decisions to a default configuration.

Definition 4.6 (Combining configurations)

Let $\mathcal{C}^X = (\mathcal{C}_{in}^X, \mathcal{C}_{out}^X, \mathcal{C}_{rem}^X, \mathcal{C}_{nofi}^X)$ and $\mathcal{C}^Y = (\mathcal{C}_{in}^Y, \mathcal{C}_{out}^Y, \mathcal{C}_{rem}^Y, \mathcal{C}_{nofi}^Y)$ be two (partial) configurations of EWF-nets. Then \mathcal{C}^X and \mathcal{C}^Y can be combined and generate configuration $\mathcal{C}^X \oplus \mathcal{C}^Y = (\mathcal{C}_{in}^Z, \mathcal{C}_{out}^Z, \mathcal{C}_{rem}^Z, \mathcal{C}_{nofi}^Z)$ where

- $dom(\mathcal{C}_{in}^Z) = dom(\mathcal{C}_{in}^X) \cup dom(\mathcal{C}_{in}^Y)$ and

$$\mathcal{C}_{in}^Z(p) = \begin{cases} \mathcal{C}_{in}^X(p) & \text{if } p \in dom(\mathcal{C}_{in}^X) \\ \mathcal{C}_{in}^Y(p) & \text{else, i.e. } p \in dom(\mathcal{C}_{in}^Y) \setminus dom(\mathcal{C}_{in}^X), \end{cases}$$
- $dom(\mathcal{C}_{out}^Z) = dom(\mathcal{C}_{out}^X) \cup dom(\mathcal{C}_{out}^Y)$ and

$$\mathcal{C}_{out}^Z(p) = \begin{cases} \mathcal{C}_{out}^X(p) & \text{if } p \in dom(\mathcal{C}_{out}^X) \\ \mathcal{C}_{out}^Y(p) & \text{else, i.e. } p \in dom(\mathcal{C}_{out}^Y) \setminus dom(\mathcal{C}_{out}^X), \end{cases}$$
- $dom(\mathcal{C}_{rem}^Z) = dom(\mathcal{C}_{rem}^X) \cup dom(\mathcal{C}_{rem}^Y)$ and

$$\mathcal{C}_{rem}^Z(p) = \begin{cases} \mathcal{C}_{rem}^X(p) & \text{if } p \in dom(\mathcal{C}_{rem}^X) \\ \mathcal{C}_{rem}^Y(p) & \text{else, i.e. } p \in dom(\mathcal{C}_{rem}^Y) \setminus dom(\mathcal{C}_{rem}^X), \end{cases}$$
- $dom(\mathcal{C}_{nofi}^Z) = dom(\mathcal{C}_{nofi}^X) \cup dom(\mathcal{C}_{nofi}^Y)$ and

$$\mathcal{C}_{nofi}^Z(p) = \begin{cases} \mathcal{C}_{nofi}^X(p) & \text{if } p \in dom(\mathcal{C}_{nofi}^X) \\ \mathcal{C}_{nofi}^Y(p) & \text{else, i.e. } p \in dom(\mathcal{C}_{nofi}^Y) \setminus dom(\mathcal{C}_{nofi}^X), \end{cases}$$

4.3.2 Configuration Requirements and Validity

Like for C-SAP WebFlow and for C-BPEL, also for C-YAWL not all configurations make sense. For example, in the workflow from Figure 4.6 (p. 77), it is impossible to book any travel without receiving a corresponding order. Hence, it must be ensured that the *Receive order* task is always performed, i.e. the use of its input port must always be configured as allowed. In the same way, it must be ensured that if the customer has paid the booked travel, the documents are either sent out or collected. That means, for each payment method at least the use of one of the subsequent input port of the tasks *Send documents* and *Collect documents* must be configured as allowed. Similarly, we cannot block the input ports of the tasks *Credit card payment* and *Cash Payment* as long as the customer can choose the corresponding payment method in task *Select payment method*. Otherwise, the case might deadlock in the conditions a_3 or b_3 , i.e. soundness would not be preserved. Hence, in fact, quite restrictive dependencies exist among the configuration decisions for the individual ports.

To formulate such dependencies and restrictions of allowed configurations, let us again use logical expressions. The logical expressions combine constraints on the configuration of single elements of the EWF-net — the so-called **atomic requirements** — by means of common logical operators and quantifiers. The requirement that the use of the input port of the task *Receive order* must be allowed is for example atomic and written as $(in, ('Receive order', \{\mathbf{i}\}), allow)$. The requirement that at least the use of one of the input ports of the task *Send documents* or of the task *Collect documents* must be allowed after a credit card payment has been made (i.e. from condition a_4), is composed of two atomic requirements, connected by a logical operator as

$$(in, ('Send documents', \{a_4\}), allow) \vee \\ (in, ('Collect documents', \{a_4\}), allow).$$

Definition 4.7 provides a list of all atomic requirements that can be imposed on a configuration of an EWF-net.

Definition 4.7 (Atomic configuration requirements)

Let $EFW = (K, \mathbf{i}, \mathbf{o}, T, F, \text{split}, \text{join}, \text{rem}, \text{nofi})$ be an EWF-net, $\text{ports}_{in}(EFW)$ be the input ports of EWF, and $\text{ports}_{out}(EFW)$ be the output ports of EWF. Then

- $\text{req}_{in} = \{\text{in}\} \times \text{ports}_{in}(EFW) \times \{\text{allow}, \text{hide}, \text{block}\}$ is the set of all atomic requirements on the configurations of input ports,
- $\text{req}_{out} = \{\text{out}\} \times \text{ports}_{out}(EFW) \times \{\text{allow}, \text{block}\}$ is the set of all atomic requirements on the configuration of output ports,
- $\text{req}_{rem} = \{\text{rem}\} \times T \times \{\text{allow}, \text{block}\}$ is the set of all atomic requirements on the configuration of cancelation regions,
- $\text{req}_{nofi} = \{\text{nofi}\} \times T \times (\mathbb{N}^0 \times \mathbb{N}^{0,\infty} \times \mathbb{N}^{0,\infty} \times \{\text{restrictable}, \text{non-restrictable}\})$ is the set of all atomic requirements on the configurations of the multiplicity of a task⁵ (maximal increase of the minimum, maximal decrease of the maximum, maximal increase of the threshold for continuation, and if restriction to static creation of instances is possible), and
- $\text{req}_{EFW} = \text{req}_{in} \cup \text{req}_{out} \cup \text{req}_{rem} \cup \text{req}_{nofi}$ is the set of all atomic requirements for EWF.

To combine atomic requirements we allow the use of all the common logical operators ($\neg, \wedge, \vee, \text{XOR}, \Rightarrow, =$ etc.) as well as the use of the quantifiers \forall and \exists over elements of the configurable net, i.e. tasks, conditions, ports, etc. With quantifiers, we enable the specification of requirements which have to hold for the configurations of sets of model elements. In this way it is possible to specify general requirements which are independent of a particular net.

We distinguish **general requirements**, which have to hold for every, or at least a certain group, of EWF-nets, from **specific requirements**, which only have to hold in a specific net. Specific requirements are typically content-driven. Therefore, the example requirements provided above are specific requirements.

General requirements mainly ensure the construction of well-formed nets, i.e. they ensure that the configured EWF-net can be transformed into a syntactically valid EWF-net which also conforms to any applicable modeling guidelines like the soundness of the resulting process model. Typically, general requirements are content-independent and make extensive use of quantifiers. For example, the requirement ‘each task of an EWF-net which can be enabled, i.e. which has at least one allowed or hidden input port, must also have at least one output port whose use is allowed’ would be a general requirement. It is necessary (but not sufficient) to ensure the flow of tokens through the net and thus its soundness. It can formally be specified as follows:

⁵Note that a configuration requirement on the number of instances specifies requirements on all parameters of multiple instance tasks at once. However, when combining multiple of such requirements through logical \wedge connectors, the strongest restriction is applied for each of the parameters individually.

$$\begin{aligned} & \forall t \in T : \\ & (\exists_{p \in \text{ports}_{in}(t)}(in, p, allow) \vee (in, p, hide)) \\ & \Rightarrow (\exists_{p \in \text{ports}_{out}(t)}(out, p, allow)) \end{aligned}$$

If all input ports of a task are blocked, then there will never be any inflow to the task and consequently also no outflow. For that reason, we could formulate the configuration requirement that if all input ports of a task are blocked also all output ports must be blocked (although this is irrelevant from a behavioral point of view):

$$\begin{aligned} & \forall t \in T : \\ & (\forall_{p \in \text{ports}_{in}(t)}(in, p, block)) \\ & \Rightarrow (\forall_{p \in \text{ports}_{out}(t)}(out, p, block)) \end{aligned}$$

Using general requirements, it is also possible to impose requirements on conditions although configuration is defined only on elements of tasks. For example, to ensure the flow of tokens through the net, every token that flows into a condition must also be able to flow out of it (unless it is in the final condition). Therefore, the use of at least one subsequent port must be allowed or hidden for such conditions:

$$\begin{aligned} & \forall c \in (K \setminus \{\mathbf{o}\}) : \\ & (\exists_{(t_1, cs_1) \in \text{ports}_{out}(EWF)} c \in cs_1 \wedge (out, (t_1, cs_1), allow)) \Rightarrow \\ & (\exists_{(t_2, cs_2) \in \text{ports}_{in}(EWF)} c \in cs_2 \wedge \\ & ((in, (t_2, cs_2), allow) \vee (in, (t_2, cs_2), hide))) \end{aligned}$$

A requirement is fulfilled if it evaluates to *true*. To evaluate a requirement, its atomic requirements have to be evaluated first. An atomic requirement is fulfilled if the specific port or task addressed by the requirement is configured accordingly. For example, the requirement $(in, ('Receive\ order', \{\mathbf{i}\}), allow)$ evaluates to *true* if the use of the particular port between the input condition \mathbf{i} and the task *Receive order* is configured as allowed, otherwise it evaluates to *false*. In the same way requirements for hidden or blocked input ports, and for output or cancelation ports whose use is allowed or blocked can be evaluated. A requirement on the configuration of the number of task instances like $(nofi, 'Book\ train\ ticket', (min, max, thres, dyn))$ evaluates to *true* only if

- $\pi_1(\mathcal{C}_{nofi}(t)) \leq min$,
- $\pi_2(\mathcal{C}_{nofi}(t)) \leq max$,
- $\pi_3(\mathcal{C}_{nofi}(t)) \leq thres$, and
- $dyn = non-restrictable \Rightarrow \pi_4(\mathcal{C}_{nofi}(t)) = keep$.

After all the atomic requirements within a composed requirement are evaluated to *true* or *false*, the composed requirement can be evaluated as in propositional logic⁶. Of course, the evaluation of an atomic requirement is only possible if a configuration is defined for the element of the EWF-net that is addressed by the atomic requirement. For that reason, we assume complete configurations here.⁷ We say that a complete configuration is **valid** for an EWF-net, if the configuration fulfills all configuration requirements that are imposed on the EWF-net, i.e. if the concatenation of all requirements can be evaluated to *true*. Thus, this concatenation conforms to the process constraint ensuring the correctness of a workflow net in Definition 3.7 (p. 59).

Definition 4.8 (Valid configuration)

Let $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net and req_{EWF} be the set of all atomic requirements that can be imposed on EWF. Then $PC : \mathfrak{C}_{EWF}^{complete} \rightarrow \{true, false\}$ is a configuration constraint expressed as propositional formula using req_{EWF} as propositional letters. A complete configuration $C_{EWF} \in \mathfrak{C}_{EWF}^{complete}$ is valid, if PC evaluates to true using the values of C_{EWF} to evaluate the propositional letters, i.e. the atomic requirements contained in PC.

4.3.3 Components of C-EWF-Nets

To ensure complete configurations without requiring the user of a C-EWF-net to configure every single element, a C-EWF-net includes a default configuration. This default configuration must be complete and valid for the EWF-net that should be configured. Then, each (incomplete) configuration can be combined with this complete default configuration to form a new complete configuration (as explained in Definition 4.6). The configuration decisions missing in the incomplete configuration are filled up with the default configuration for these elements. The so-created complete configuration can again be tested on its validity. If it is valid, we also consider the incomplete configuration as valid for the particular C-EWF-net.

Summarizing, a C-EWF-net consists of a syntactically correct EWF-net serving as the basic process model, a set of configuration requirements ensuring syntactic and semantic correctness of the configuration, and the default configuration.

Definition 4.9 (C-EWF-net) A configurable extended workflow net (C-EWF-net) is a tuple (EWF, PC, def) where

- EWF is an EWF-net,
- $PC : \mathfrak{C}_{EWF}^{complete} \rightarrow \{true, false\}$ is a configuration constraint on EWF, and

⁶Although quantifiers are used in the expressions, these are always expressed on a finite set of elements (e.g. ‘for all the conditions of the EWF-net’). So this is referable to the conjunction of a set of propositional logic requirements, each of them expressed over an element (e.g. over a condition).

⁷If a configuration is not complete, it should be combined with a complete configuration first. In this way, also incomplete configurations can be tested on their validity.

- $def \in \{\mathcal{C} \in \mathbb{C}_{EWF}^{complete} \mid PC(\mathcal{C}) = true\}$ is the complete and valid default configuration of EWF.

Note that the basic EWF-net might contain semantically conflicting behavior like tasks that exclude each other. Thus, if no explicit configuration decisions are made, the default configuration also ensures that a semantically correct EWF-net can be derived for the basic EWF-net.

4.3.4 Configurable Workflow Specifications

A YAWL workflow specification organizes EWF-nets hierarchically by mapping tasks of EWF-nets onto other EWF-nets (see Definition 2.26). Using C-EWF-nets instead of EWF-nets, this section outlines how a configurable workflow specification can be build-up on top of C-EWF-nets. In this context we will use the expression **(C-)EWF-net** in statements that hold for both EWF-nets and C-EWF-nets.

In a workflow specification each composite task of a (C-)EWF-net is mapped onto a set of (C-)EWF-nets. Whenever the task is triggered, one (C-)EWF-net from the set is chosen as an implementation for the task and initiated. That means, there is a choice between the different nets in the set. The task only completes when the selected (C-)EWF-net signals its completion. This means that the mapping between tasks and (C-)EWF-nets determines the control-flow between the nets. Hence, this mapping offers configuration opportunities in a configurable workflow specification in addition to the configuration opportunities of C-EWF-nets.

Every (C-)EWF-net has a unique input condition through which it can be triggered. Thus, the interface between the upper level, composite task and the input condition of an implementing (C-)EWF-net represents the unique inflow port of the task's implementation in the mapped (C-)EWF-net.

The completion of a (C-)EWF-net is signaled via its unique output condition. The interface from the output condition back to the upper level, composite task therefore represents the unique outflow port of a (C-)EWF-net. However, as the net needs at least one outflow port that can be used to forward the control to subsequent tasks of the superior net, the use of this outflow port must always be allowed, i.e., like in the block structures of SAP WebFlow or BPEL, it cannot be configured.

Hence, on the level of the workflow specification, the only configurable port of an (C-)EWF-net is its inflow port, which we will call **hierarchy port** in the following.

Definition 4.10 (Hierarchy ports) *Let $(Q^\diamond, Q, top, T^\diamond, map)$ be a workflow specification. Then ports_{map} = $\{(t, EWF) \in T^\diamond \times Q^\diamond \mid t \in dom(map) \wedge EWF \in map(t)\}$ are all the hierarchy ports of the workflow specification.*

As hierarchy ports are inflow ports, they can be configured as allowed, hidden, or blocked. If a hierarchy port is configured as blocked, the particular (C-)EWF-net cannot be triggered at run-time, i.e. it cannot be chosen as an implementation

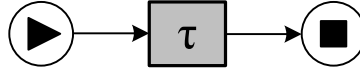


Figure 4.10: The dummy net replacing a hidden net, i.e the τ task corresponds to the skipped behavior of the original net.

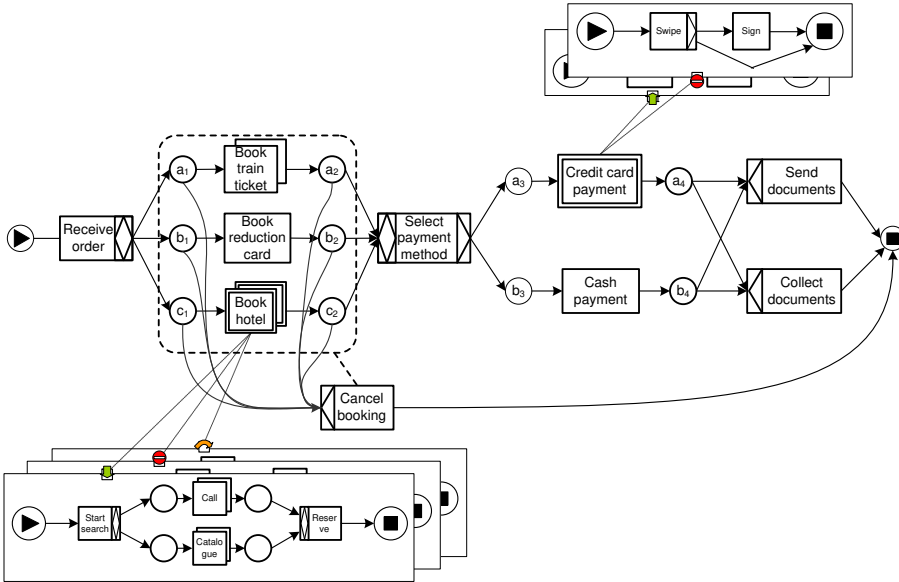


Figure 4.11: The hierarchy within the workflow specification of Figure 4.7 configured.

for the task on the superior workflow definition level. Instead, another (C-)EWF-net from the set of EWF-nets mapped onto the task has to be selected whose hierarchy port is configured either as allowed to be used or as hidden. The nets with allowed or hidden hierarchy ports can be normally selected and triggered. Then, nets with a hierarchy port whose use is allowed are executed in the same way as ordinary (C-)EWF-nets while the behavior of (C-)EWF-nets which have a hidden hierarchy port must be skipped completely. For this reason, we replace such a (C-)EWF-net with the ‘dummy’ EWF-net shown in Figure 4.10 where the τ task corresponds to the skipped behavior of the original net. Figure 4.11 depicts an example configuration for the workflow specification of Figure 4.7.

Besides the hierarchy configuration, a configuration of a workflow specification also requires a configuration for each individual EWF-net. Hence, workflow specification configurations can be defined as on the one hand the configuration of the hierarchy and on the other hand a combination of complete configurations for each of the individual EWF-nets in the workflow specification.

Definition 4.11 (Workflow specification configuration)

Let $SPEC = (Q^\circ, Q, top, T^\circ, map)$ be a workflow specification and let $ports_{map}$ be the hierarchy ports of the workflow specification. Then

- $\mathcal{C}_{map} : ports_{map} \not\rightarrow \{allow, block, hide\}$ is a (partial) configuration of the workflow specification's hierarchy, and \mathfrak{C}_{map} is the set of all such configurations. \mathcal{C}_{map} is complete, if and only if $dom(\mathcal{C}_{map}) = ports_{map}$. The set of all such complete configurations is denoted as $\mathfrak{C}_{map}^{complete}$.
- $\mathfrak{C}_{Q^\diamond}^{complete} = \{\bigoplus_{EWF \in Q^\diamond} \mathcal{C}^{EWF} \mid \forall_{EWF \in Q^\diamond} \mathcal{C}^{EWF} \in \mathfrak{C}_{EWF}^{complete}\}$ is the set of all complete configurations for the EWF-nets contained in Q^\diamond .⁸
- $\mathcal{C}_{SPEC} \in \mathfrak{C}_{map}^{complete} \times \mathfrak{C}_{Q^\diamond}^{complete}$ is a configuration of workflow specification SPEC, and $\mathfrak{C}_{SPEC} = \mathfrak{C}_{map}^{complete} \times \mathfrak{C}_{Q^\diamond}^{complete}$ is the set of all such configurations.

Like for C-EWF-nets, not all combinations of configurations among the different (C-)EWF-nets are feasible. For example, it is not possible to block all the (C-)EWF-nets implementing a task because every composite task must have at least one usable, i.e. either allowed or hidden, implementation. For that reason, atomic requirements must also be impossible on the configuration of mappings between a composite task and a (C-)EWF-net.

Definition 4.12 (Atomic hierarchy configuration requirements)

Let $(Q^\diamond, Q, top, T^\diamond, map)$ be a workflow specification, and $ports_{map}$ be the hierarchy ports of the workflow specification. Then $req_{map} = \{map\} \times ports_{map} \times \{allow, hide, block\}$ is the set of all atomic requirements on the configuration of hierarchy ports.

An atomic requirement that the use of the port between a composite task t and an EWF-net EWF must always be allowed would thus be written as $(map, (t, EWF), allow)$. Such requirements can be combined and evaluated as the requirements on configurations of C-EWF-nets. This also enables us to combine requirements on the higher-level configurable workflow specification with requirements on specific C-EWF-nets within the configurable workflow specification. For example, the possibility to book reduction cards in the workflow of Figure 4.11 might require the use of a certain implementation of the payment tasks that includes an address, age, or student status verification. Some implementations of a task may also depend on the data output of a preceding task while others do not. Thus, this might lead to missing data if an implementation is used for the latter task which needs the data while the data-creating task is not executed because it has been configured as hidden or blocked. Hence, the implementations requiring the data must then be blocked as well.

For that reason, when configuring a configurable workflow specification, not only each of its C-EWF-nets has to fulfill the particular configuration constraint

⁸ $\bigoplus_{EWF \in EWFS} \mathcal{C}^{EWF}$ denotes the combination of all configurations \mathcal{C}^{EWF} such that EWF is contained in the set of EWF-nets $EWFS$. Note that this operator requires that the combination of any two configurations \mathcal{C}^{EWF} for which $EWF \in EWFS$ is associative and commutative. $(\mathcal{C}_{in}^1, \mathcal{C}_{out}^1, \mathcal{C}_{rem}^1, \mathcal{C}_{nofi}^1) \oplus (\mathcal{C}_{in}^2, \mathcal{C}_{out}^2, \mathcal{C}_{rem}^2, \mathcal{C}_{nofi}^2)$ is associative and commutative if the domains of the two configurations' configuration functions are distinct, i.e. $(dom(\mathcal{C}_{in}^1) \cap dom(\mathcal{C}_{in}^2) = \emptyset) \wedge (dom(\mathcal{C}_{out}^1) \cap dom(\mathcal{C}_{out}^2) = \emptyset) \wedge (dom(\mathcal{C}_{rem}^1) \cap dom(\mathcal{C}_{rem}^2) = \emptyset) \wedge (dom(\mathcal{C}_{nofi}^1) \cap dom(\mathcal{C}_{nofi}^2) = \emptyset)$. For the EWF-nets in Q^\diamond this is guaranteed by the definition of workflow specifications (see Definition 2.26, p. 39).

on the net, but the configurations of the hierarchy ports as well as the configurations of the individual EWF-nets also have to fulfill the specification's overall **workflow specification constraint**. Similar to the default configuration for C-EWF-nets, a configurable workflow specification should include a default configuration for hierarchy ports which is complete and which conforms to the workflow specification constraint, i.e. is valid. In this way, any partial configuration of hierarchy ports can be combined with the default configuration to form a complete configuration. Such a configuration can then be tested on its validity according to the workflow specification constraint and thus be used to generate a configured workflow specification based on a partial configuration.

Definition 4.13 (Configurable workflow specification)

Let $SPEC = (Q^\diamond, Q, top, T^\diamond, map)$ be a workflow specification, PC^\diamond be a set of configuration constraints and def^\diamond be a set of default configurations such that

- $PC^\diamond = \{PC_{EWF} \mid EWF \in Q^\diamond\}$,
- $def^\diamond = \{def_{EWF} \mid EWF \in Q^\diamond\}$, and
- any $(EWF, PC_{EWF}, def_{EWF}) \in Q^\diamond \times PC^\diamond \times def^\diamond$ represents a C-EWF-net.

Then the tuple $(SPEC, PC^\diamond, def^\diamond, PC^{SPEC}, def_{map})$ represents a configurable workflow specification with

- $PC^{SPEC} : \mathbb{C}_{SPEC} \rightarrow \{true, false\}$ being the workflow specification constraint, and
- $def_{map} \in \mathbb{C}_{map}^{complete}$ such that $PC^{SPEC}((def_{map}, \bigoplus_{def \in def^\diamond} def)) = true$ being a complete and valid default configuration of the hierarchy ports of $SPEC$.

Altogether, C-YAWL provides two levels for configuring a workflow. Distinct approaches can be handled by different (C-)EWF-nets mapped onto a single task. Different variants of the same approach should be handled within a single C-EWF-net by using its configuration opportunities.

4.3.5 From C-YAWL to YAWL

To demonstrate the applicability of configurable workflow models, let us discuss the implementation of a transformation from C-YAWL to YAWL such that we can derive YAWL models which are executable in the YAWL workflow engine. In this section we focus first on the transformation from C-EWF-nets to EWF-nets before outlining the rather trivial selection mechanism of configurable workflow specifications.

The transformation from C-EWF-nets to EWF-nets is performed in two steps. First, the elements directly affected by configuration decisions are removed; secondly, a cleanup algorithm removes those elements which became obsolete in the first step. The latter step ensures that the created EWF-net conforms to Definition 2.24, which requires that every element is on a path between **i** and **o**. In this way, we can neglect this requirement in the first step.

As input to the transformation, let $CEWF = (EWF, PC, def)$ be a C-EWF-net with $EWF = (K, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$. In addition, let $\mathcal{C}_{EWF} = (\mathcal{C}_{in}, \mathcal{C}_{out}, \mathcal{C}_{rem}, \mathcal{C}_{nofi})$ be a complete, and valid configuration of $CEWF$ that is applied to $CEWF$ to create a configured EWF-net (if \mathcal{C}_{EWF} is not complete, it can easily be transformed into a complete configuration by combining it with def).

As conditions are not configurable, we initially keep all conditions from the C-EWF-net also in the configured EWF-net and remove the superfluous conditions later during the cleanup. The input \mathbf{i} and output conditions \mathbf{o} are the same in the configured net as in the configurable net.

For transforming the tasks' behavior, we will start with the \mathcal{C}_{rem} and \mathcal{C}_{nofi} configurations as these can be applied to tasks in a straightforward manner.

The configuration of the cancelation region \mathcal{C}_{rem} restricts the set of elements returned by the rem function. Whenever the cancelation region is blocked, the function returns an empty list:

$$\bullet \quad rem^{\mathcal{C}}(t) = \begin{cases} rem(t) & \text{if } t \in dom(rem) \wedge \mathcal{C}_{rem}(t) \neq block \\ \emptyset & \text{otherwise.} \end{cases}$$

The $nofi$ function, assigning the number of instances that can be started to each task, must be adapted according to the configuration \mathcal{C}_{nofi} . Here, the configured increase of the minimal number of instances to be started is added to the predefined minimal number of instances, and the configured decrease of the maximal number of instances to be started is subtracted from the predefined value. The configured increase of the threshold value is added to the predefined threshold value. If the predefined task enables the dynamic creation of task instances and the task is configured to keep the current definition, the creation of task instances remains dynamic, otherwise it is set to static.

- $T_{dyn} = \{t \in dom(nofi) \mid \pi_4(\mathcal{C}_{nofi}(t)) = keep \wedge \pi_4(nofi(t)) = dynamic\}$
- For all $t \in dom(nofi)$:
 - $\pi_1(nofi^{\mathcal{C}}(t)) = \pi_1(nofi(t)) + \pi_1(\mathcal{C}_{nofi}(t))$
 - $\pi_2(nofi^{\mathcal{C}}(t)) = \pi_2(nofi(t)) - \pi_2(\mathcal{C}_{nofi}(t))$
 - $\pi_3(nofi^{\mathcal{C}}(t)) = \pi_3(nofi(t)) + \pi_3(\mathcal{C}_{nofi}(t))$
 - $\pi_4(nofi^{\mathcal{C}}(t)) = \begin{cases} dynamic & \text{if } t \in T_{dyn} \\ static & \text{otherwise} \end{cases}$

The configuration of output ports \mathcal{C}_{out} influences the flow relations which originate from a task. If the use of an output port which refers to a particular flow relation is allowed, the flow relation remains part of the configured EWF-net. Otherwise it does not. As tasks with AND-split behavior have just a single port, all flow relations originating from such tasks must be removed if the port is configured as blocked. In case of an XOR-split, each flow relation is addressed by exactly one port. Thus, a flow relation must be removed if the corresponding port is blocked. The output ports of a task with an OR-split semantics can be configured in different ways even if the different ports refer to the same flow relation. Then, as said above, the flow relation must be kept as part of the

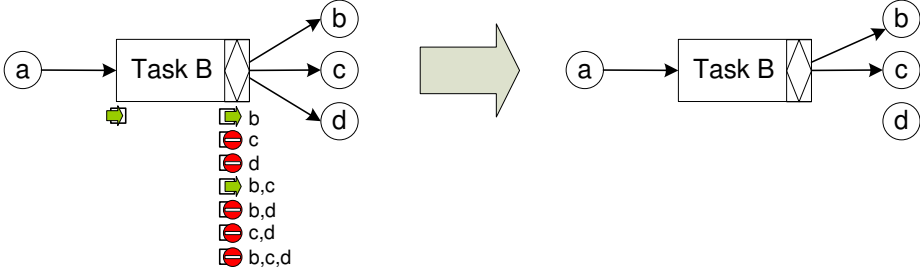


Figure 4.12: Transforming the output port configuration into a YAWL model

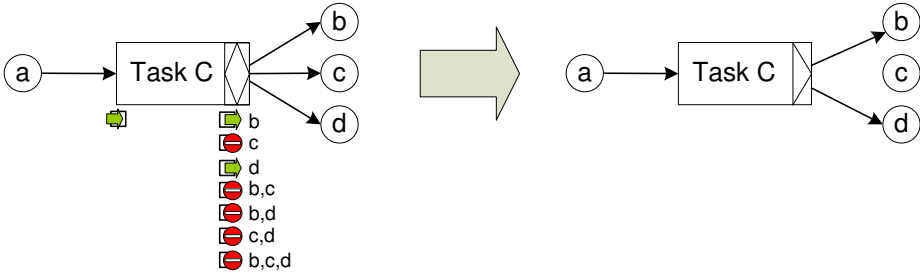


Figure 4.13: Transformation of an OR-split into an XOR-split.

configured EWF-net if the use of any output port referring to the flow relation is allowed. For example, in Figure 4.12 only the two ports connecting *Task B* either with condition *b* or with conditions *b* and *c* are allowed to be used. All other output ports are blocked. Therefore, the flow relation from *Task B* to *d* can be removed, but the flow relations to the conditions *b* and *c* cannot be removed. The blocking of ports referring to these flow relations that must be kept in the net, e.g. the blocking of the output port *c*, is realized by adapting the flows' predicates. Based on process data, predicates determine at run-time if a flow relation is triggered or not. Thus, the new predicate should exclude the activation of *c* in isolation.

- $F_{out}^C = \{(t, c) \in F \mid t \in T \wedge c \in K \wedge \exists (t, cs) \in ports_{out}(EWF) \ c \in cs \wedge C_{out}((t, cs)) \neq block\}$

The split behavior of a task basically corresponds to its behavior in the EWF-net. Only in two special configuration cases of a task with an OR-split behavior the splitting changes. If all the output ports of such a task which refer to more than a single flow are blocked, only ports that refer to a single flow relation remain in the net. This corresponds to the split behavior of an XOR-split. Hence, the task's split behavior is changed accordingly (see Figure 4.13). If all the output ports of a task are blocked except a single port, all flow relations remaining in the net are always triggered through this port. Thus, the split behavior is transformed into an AND-split (see, e.g., Figure 4.14). In all other

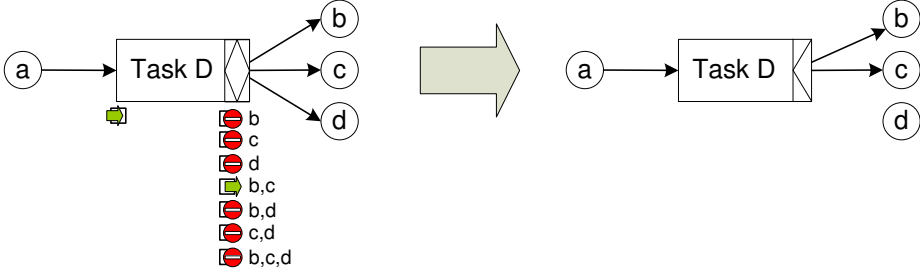


Figure 4.14: Transformation of an OR-split into an AND-split.

cases the split behavior remains the same as in the original net.

- $T_{XOR}^C = \{t \in T \mid \forall (t, cs) \in ports_{out}(t) (|cs| > 1 \Rightarrow \mathcal{C}_{out}((t, cs)) = block)\}$
- $T_{\wedge}^C = \{t \in T \mid \forall (t, cs) \in ports_{out}(t) (|cs| < |\{c \in K \mid (t, c) \in F_{out}^C\}| \Rightarrow \mathcal{C}_{out}((t, cs)) = block)\}$
- $split^C(t) = \begin{cases} XOR & \text{if } t \in T_{XOR}^C \\ \wedge & \text{if } t \in T_{\wedge}^C \setminus T_{XOR}^C \\ split(t) & \text{otherwise} \end{cases}$

Finally, the configuration of the join behavior \mathcal{C}_{in} has to be applied to the EWF-net. If a task has an AND-join or an OR-join behavior, it just has a single input port. If this port is blocked, the task can never be enabled and thus be executed. Hence, all inflows into the task are not part of the configured EWF-net. If the input port of a task t is hidden, this means that the execution behavior of the task must be skipped. For that reason, a bypass to the task has to be introduced that skips all the behavior, i.e. a silent task. The silent task does not include any ‘action’ as, e.g., any execution of work from the original task but has exactly the same join, split, and cancellation behavior as the original task. We thus label this task τ_t .

A task with an XOR-join behavior can have different configurations for different input ports. In this case, it might be required that a net includes both a silent version and an active version of a task. For example, in Figure 4.15 the input ports from the conditions a and b are configured as hidden, the use of the input port from condition c is allowed and the input port from condition d is blocked. Then the conditions a and b should trigger the silent task τ_{TaskA} , whereas condition c should trigger the active task. Therefore, we split up the set of tasks for the configured net into a set of allowed and a set of hidden tasks, i.e. $T^C = T_{allow}^C \cup T_{hide}^C$. The silent task must be introduced whenever a task has at least one hidden input port. The (normal) allowed task remains in the net as long as there is at least one input port whose use is configured as allowed.

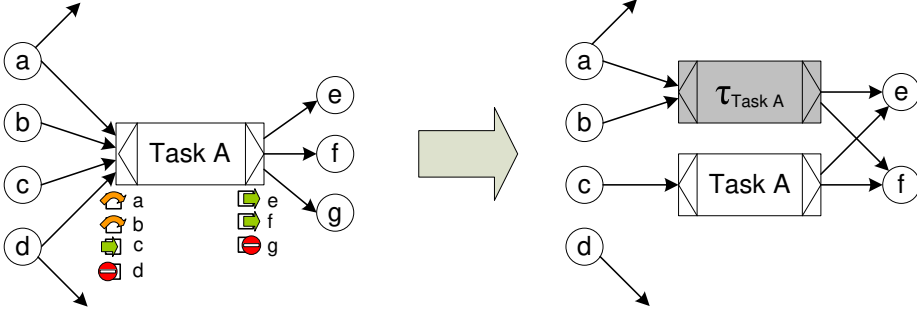


Figure 4.15: Transforming the input port configuration into a YAWL model.

- $T_{allow}^C = \{t \in T \mid \exists (t, cs) \in ports_{in}(t) \mathcal{C}_{in}((t, cs)) = allow\}$
- $T_{hide}^C = \{\tau_t \mid t \in T \wedge \exists (t, cs) \in ports_{in}(t) \mathcal{C}_{in}((t, cs)) = hide\}$
- For all $\tau_t \in T_{hide}^C$:
 - $join^C(\tau_t) = join(t) \wedge$
 - $split^C(\tau_t) = split^C(t) \wedge$
 - $rem^C(\tau_t) = rem^C(t) \wedge$
 - $nofi^C(\tau_t) = nofi^C(t)$

To connect the tasks with conditions, all flow relations connected to an input port whose use is allowed remain in the net. All flow relations connected to a hidden port are reconnected to the hidden (silent) task. Therefore, conditions a and b in the example of Figure 4.15 are connected to the silent task, whereas c remains connected to *Task A*. All flow relations connected to a blocked input port are not part of the configured net. For that reason, the flow relation connecting d with *Task A* is not part of the configured net. The outflow is the same for silent tasks as it is for active tasks. Hence, the configured output flow relation is added to silent tasks in the same way as it is to active tasks.

- $F_{in}^C = \{(c, t) \in F \mid \exists cs \subseteq K (t, cs) \in ports_{in}(EWF) \wedge c \in cs \wedge \mathcal{C}_{in}((t, cs)) = allow\} \cup \{(c, \tau_t) \mid (c, t) \in F \wedge \exists cs \subseteq K (t, cs) \in ports_{in}(EWF) \wedge c \in cs \wedge \mathcal{C}_{in}((t, cs)) = hide\}$
- $F_{out}^{C, \tau} = \{(t, c) \in F_{out}^C \mid \exists cs \subseteq K (t, cs) \in ports_{in}(EWF) \wedge \mathcal{C}_{in}((t, cs)) = allow\} \cup \{(\tau_t, c) \mid (t, c) \in F_{out}^C \wedge \exists cs \subseteq K (t, cs) \in ports_{in}(EWF) \wedge \mathcal{C}_{in}((t, cs)) = hide\}$

The join behavior of all tasks allowed in the configured net is the same as the join behavior in the configurable net.

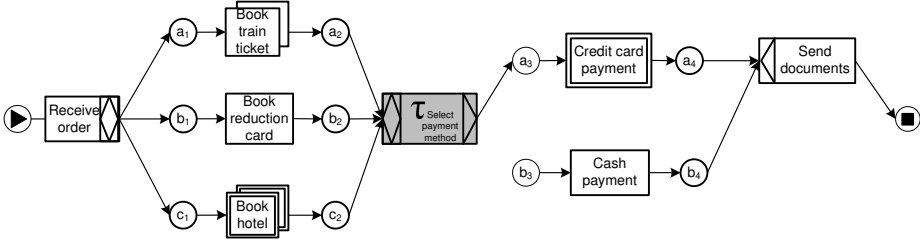


Figure 4.16: The net derived from the internet shop’s configuration, but without the removal of any “dead” parts.

- $\forall t \in T_{allow}^C \quad join^C(t) = join(t)$

Altogether, we transformed the C-EWF-net into the net $EWF^C = (K, \mathbf{i}, \mathbf{o}, T_{allow}^C \cup T_{hide}^C, F_{in}^C \cup F_{out}^C, split^C, join^C, rem^C, nofi^C)$.

However, as mentioned in the beginning of this section, the resulting net does not necessarily conform to the requirements of an EWF-net in which every node in the graph must be on a directed path from \mathbf{i} to \mathbf{o} . Due to the removal of flow relations, some conditions and tasks might not be reachable anymore from \mathbf{i} . For example, have a look at Figure 4.16. It depicts the net resulting from the internet shop’s configuration from Figure 4.7b. The conditions $b3$ and $b4$ as well as the task *Cash payment* are no longer reachable from \mathbf{i} . To create an EWF-net from EWF^C , it is therefore necessary to remove all nodes which are not on a path from \mathbf{i} to \mathbf{o} .

This cleanup step can be performed as a depth-first search, starting with the input condition, looking for all paths to the output condition. If such a path is found, all elements on this path are marked for being kept in the process. In addition, all visited elements are marked, such that elements do not need to be visited multiple times. All tasks and conditions not on such a path are afterwards removed.

However, note that removing of individual flow relations preceding an AND-join would lead to changes in the semantics of the net as we would suddenly say that not all preceding conditions must be marked, but rather only those which can be marked. This is in fact an OR-join behavior instead of an AND-join behavior. Similarly, removing an individual flow relation originating from a task with an AND-split behavior would mean not marking all subsequent condition, but only a subset. Again, this then represents an OR-split behavior instead of conforming to the AND-split semantics. Thus, if such a flow relation must be removed because its originating or target condition is removed, then not only the condition and the flow relation must be removed, but the whole task with the AND-join/AND-split behavior must be removed even if the task is located on a path from \mathbf{i} to \mathbf{o} . In this way we prevent changing the semantics of the task.

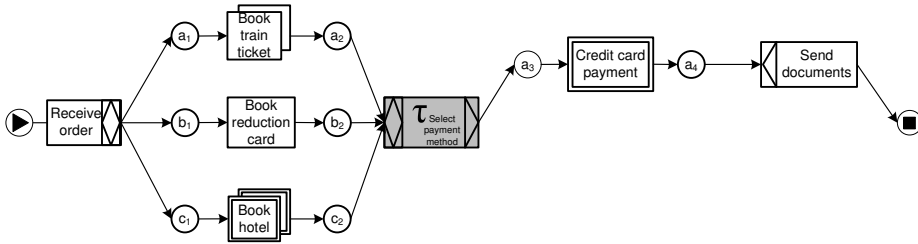


Figure 4.17: The YAWL net derived from the configuration of the internet shop having all “dead” model parts removed.

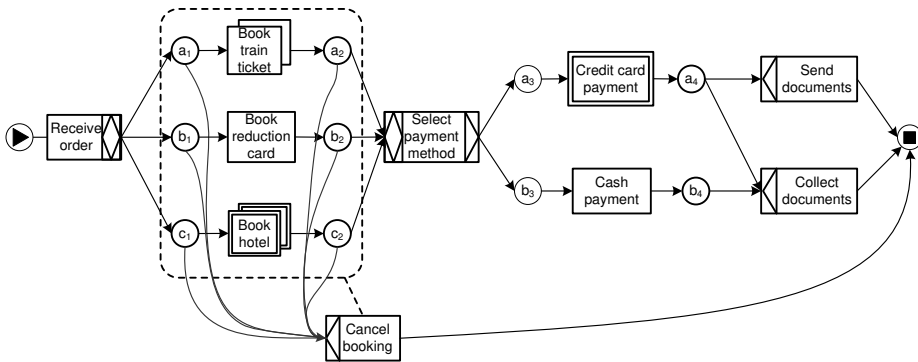


Figure 4.18: The YAWL net derived from the configuration of the travel agency.

But this might make additional elements ‘loosing’ their path from \mathbf{i} to \mathbf{o} . For that reason, we repeat the whole cleanup process either until in one iteration no such tasks are removed or until no path remains between \mathbf{i} and \mathbf{o} at all. If there is no path from \mathbf{i} to \mathbf{o} at all, the configuration is not transformable into a well defined EWF-net and should be forbidden in the requirements on the C-EWF-net.

Figure 4.17 shows the EWF-net derived using this cleanup algorithm from the net in Figure 4.16. Furthermore, Figure 4.18 shows the EWF-net derived from the example configuration for a travel agency as depicted in Figure 4.7a.

Deriving a configured workflow specification from a configuration for a configurable workflow specification is straightforward. On the one hand, for all C-EWF-nets of the workflow specification, configured EWF-nets must be derived from the particular configurations. On the other hand, the mapping function needs to be updated according to its configuration. That means, blocked implementations must be excluded from the specification while hidden implementations must be replaced by the dummy net shown in Figure 4.10.

4.3.6 C-YAWL Implementation

We implemented the transformation from C-YAWL to YAWL in the context of the YAWL system. Using the YAWL editor the basic process model which incorporates the different process variants can be defined as depicted in Figure 2.10 (p. 37). The basic process model can then be loaded directly into the YAWL workflow engine. This enables the engine to execute the workflow while providing choices between all the different variants at the run-time of the process. In practice, however, the basic process model will seldom be loaded completely into the workflow engine. Instead a particular variant of the process is selected. To restrict the workflow to desired variants only, the discussed configuration decisions can be added to the workflow definition. Without any manual modeling effort, the algorithm depicted in this section can then generate a new workflow definition in line with the configuration decisions. As the original definition, the generated definition can directly be used in the workflow engine to execute the desired workflow variant. For example, Figure 4.19 shows a screenshot depicting the worklist of the travel booking workflow configured according to the requirements of the internet shop (as depicted in Figure 4.7b) with several bookings in progress. As already depicted in Figure 4.17, the configured workflow definition can also be imported back into the YAWL editor to inspect or further adapt the resulting workflow.

The technical details on how configuration decisions are added to YAWL's XML schema can be found in [78]. Further details on the implementation of the YAWL system itself can be found in the work of van der Aalst et al. [13] as well as in the technical documentation of YAWL which can be found on the YAWL website⁹. Both C-YAWL and the C-YAWL to YAWL transformation algorithm will be part of future YAWL releases and thus be available to all YAWL users.

4.4 Related Work

Adaptation opportunities for process modeling notations have also been suggested by other authors. Although the ideas of blocking and hiding behavior have not been explicitly taken into account during the various adaptation notions, there are obvious relationships. The configuration extension for EPCs suggested by Rosemann and van der Aalst [143], known as C-EPCs, is very close to the notations we have discussed in this chapter. For that reason, we will elaborate on how the blocking and hiding ideas are implicitly considered in this notation in more detail in the first part of this section. Afterwards, we will point to a range of further suggestions for configuring process models which all use concepts that can be related to the ideas of blocking and hiding.

⁹see <http://www.yawl-system.com/>

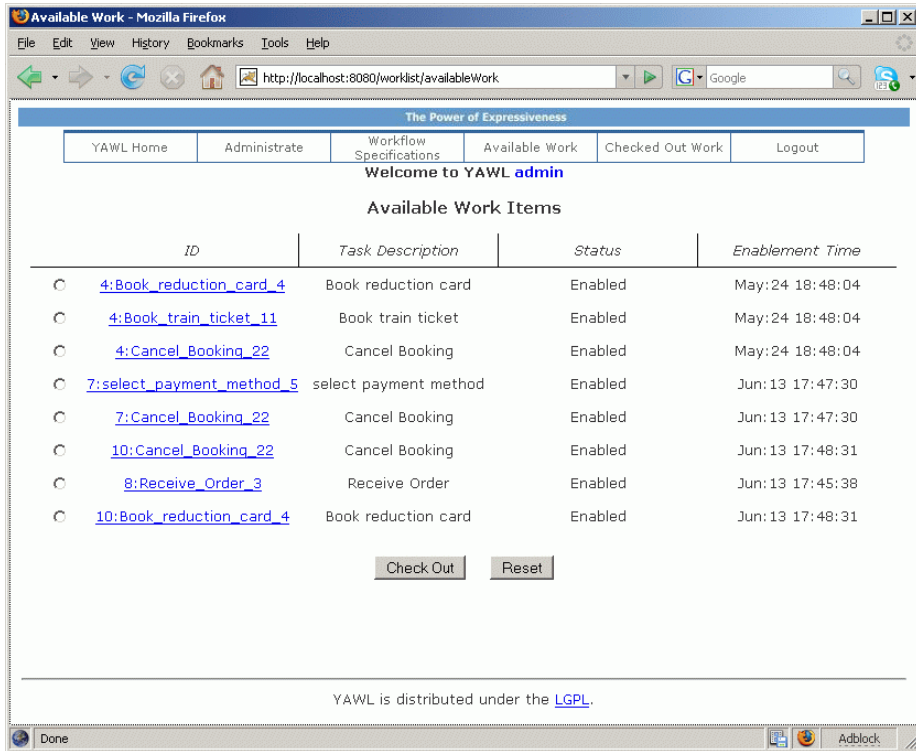


Figure 4.19: The worklist of the internet shop's travel booking workflow with several bookings in progress

4.4.1 C-EPCs

For Rosemann and van der Aalst [143] the configuration of a business process modeling language represents a second level of decision making. Therefore, they require that these decisions must be clearly identifiable and distinguishable from decisions made while the process is executed. To demonstrate this, they developed a concrete language based on EPCs supporting such configuration decisions. Their **C-EPCs** extend EPCs with configuration opportunities by adding process configuration options to functions, i.e. the tasks, and connectors of EPCs. Both configurable functions as well as configurable connectors are highlighted in the traditional EPC using bold borders. Configurable functions can then be configured such that they are included (on), skipped (off) or conditionally skipped (optional) in the configured model. Configurable connectors can be restricted to a subset of their theoretic routing opportunities. A configurable connector of type \vee may be mapped onto an \wedge -connector, an *XOR*-connector, or a sequence, i.e. an elimination of the connector by connecting exactly one of the incoming arcs directly to exactly one of the outgoing arcs. Also, an *XOR*-connector can be mapped onto a sequence if necessary.

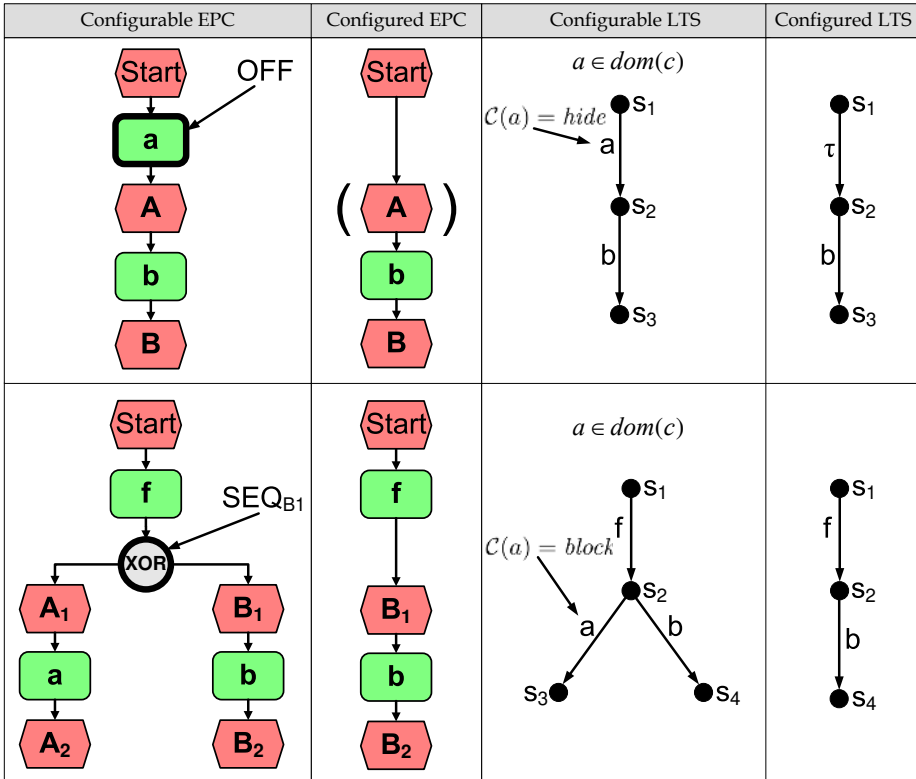


Figure 4.20: Configuring C-EPCs and the corresponding LTS

An \wedge -connector cannot be reduced at all (for further details see Rosemann and van der Aalst [143], Section 5.2.).

Figure 4.20 shows the relation between C-EPCs and the configuration of the underlying LTSs. In the first row the function a within the C-EPC process fragment is switched off. If a function is turned off, the process flow continues after the function without actually executing the function. It is thus replaced in the resulting, configured EPC with a direct arc (to generate a lawful EPC also one of the events surrounding a must be removed, which is indicated by the brackets around event A in Figure 4.20). This corresponds to replacing the transition a in the corresponding LTS with a silent τ transition. Turning off functions of a C-EPC thus corresponds to the notion of hiding tasks.

The process fragment in the second row of Figure 4.20 shows a configurable XOR -connector. It can be configured to remain an XOR -connector or it can be restricted to a sequence. In the example, it is configured to a sequence, enforcing that only the right process branch can be executed. From the corresponding LTS it can be seen that this conforms to blocking the other process path originating from the XOR -connector.

Within C-EPCs, configuration choices like turning a function off may be

limited by configuration requirements in the same way as we have introduced for C-YAWL. Requirements therefore ensure that only valid configurations are generated. In addition to these requirements it is also possible to add guidelines, which are depicted in the same way as configuration requirements, but are only recommendations and thus not enforced.

Altogether, C-EPCs therefore support the three main aspects which we required for configurable process modeling languages: hiding behavior, blocking behavior, and enforcing valid configuration through process constraints, i.e. requirements.

4.4.2 Further Process Configuration Extensions

EPCs are also used by Becker et al. [34] to depict techniques for process model adaptations. For configuring the elements of an EPC they distinguish between the deselection of irrelevant element types on a meta level, and the deselection of specific elements on the modeling level itself. The first mechanism mainly assumes that models integrate different user perspectives and thus include elements for these different types. For the individual user, a lot of this information is often superfluous and thus rather confusing. Hence, the deselection of element types influences the model visualization instead of dealing with changing the behavior depicted by the model.

The second mechanism allows for deselecting specific functions of the EPC. When deselecting functions, this implies according to Becker et al. that the corresponding tasks are skipped. Thus, even though Becker et al. do not provide a formal definition of how the arcs which incorporate the deselected functions into the overall process should be re-connected, this corresponds to hiding the tasks in an underlying LTS.

Reijers et al. [137] introduce a modeling technique called *aggregated EPCs*, which is similar to the deselection of specific functions introduced by Becker et al. Based on parameters attached to EPC functions and events, aggregated EPCs are capable of selecting certain parts of the process. Arcs and connectors are only incorporated into the resulting EPC if the preceding and succeeding nodes are also selected. Thus, the selection mechanism can be compared to the blocking of certain task executions. However, the technique itself only works on the graph level, i.e. it does not consider the EPC semantics. For example, an AND-split implies that all paths originating from a particular node must be triggered at run-time. If a task succeeding the AND-split is not selected, the corresponding arc is removed while all other arcs connected to the AND-split remain in the model. As the deselected function was always triggered in the aggregated EPC, such a behavior was not possible in the original model. Hence, the selection mechanism of aggregated EPCs might add new behavior when determining a relevant subset of the model. To counteract this, Reijers et al. require from aggregated EPC designers that they ensure a structure of the aggregated EPC and its attributes which avoids such situations. As the technique does not provide the opportunity to hide functions, this, e.g., also

includes that in the aggregated EPC a bypass exists for functions which must be skipped when the parameters are set correspondingly.

Czarnecki and Antkiewicz [49] present an approach to include or exclude elements from UML activity diagrams. While Becker et al. and Reijers et al. focus on the nodes of the process model, Czarnecki and Antkiewicz allow for the inclusion or exclusion of both nodes and arcs of the graphs. Thus, as in the approach of Reijers et al., in the approach of Czarnecki and Antkiewicz unnecessary arcs are not re-connected, but simply deleted. This can here even be done without removing any activities from the UML diagram. As the arcs depict the routing of cases through the process model, this technique can thus not only be used to restrict the behavior of process models through eliminating activities but also to restrict the routing possibilities, similar to the configuration of connectors in C-EPCs. Thus, this adaptation is closely related to the blocking of behavior.

However, as the approach of Reijers et al., the approach of Czarnecki and Antkiewicz is applied directly to the graph structure of the process model and ignores any underlying semantics when doing such changes. Thus, after deleting one of the outgoing arcs of an AND-split through process configuration only a subset of these paths can be triggered in the future. As explained for the approach of Reijers et al., this corresponds to creating new behavior which did not exist in the original model. Hence, the approach of Czarnecki and Antkiewicz not only enables a restriction of the run-time behavior like blocking of behavior does, but — as Czarnecki and Antkiewicz do not forbid such constructs — it also allows for state changes which were originally not possible.

As already indicated in the related work section of Chapter 3, Puhlmann et al. [132] also use UML activity diagrams to depict their configuration mechanisms of omitting tasks and parameterizing decision nodes and join nodes. They allow the omission of tasks only for nodes which have exactly one incoming and one outgoing arc. These are directly connected if the task should be omitted. Hence, the omission of tasks corresponds to the hiding of the particular activity. The values of parameters which are added to decision and join nodes determine if a particular control-flow arc can be followed or not. If a certain path cannot be followed this corresponds to blocking the particular behavior. Besides for UML, Puhlmann et al. also demonstrate how parametrization can be used in BPMN.

4.5 Conclusions

Based on the ideas for configuring process models from Chapter 3, this chapter has shown how process configuration can be added to existing process modeling languages. We informally introduced configuration extensions for the workflow notations of SAP WebFlow and BPEL. Also, we formally defined such an extension for YAWL and discussed its application extensively. Furthermore, we discussed how the ideas of blocking and hiding are implicitly used in process modeling notations which can be found in the existing literature like C-EPCs.

When introducing the different configuration extensions, the goal was to develop configurable notations that can be as easily used as the notations to which it is added — even if the number of ports grows significantly. Thus, we did not always provide a configuration option for each and every transition which can be blocked or hidden in the underlying LTS. Instead, we looked for intuitive configuration options, considering the characteristics of the particular process modeling notation while keeping the blocking and hiding techniques in the back of our minds. For example, we used the block structure of SAP WebFlow and BPEL to introduce configuration mechanisms for these notations, and we provided direct configuration options for the multiplicity of tasks and cancelation regions in YAWL. These configuration options were derived from the implicit meaning of these parameters in an underlying LTS and the principles introduced in Chapter 3 (hiding and blocking of ports).

Of course, these generalizations limit the configuration opportunities compared to the configuration opportunities in the corresponding LTS. However, this is done for the same purpose as for which these advanced process modeling notations have been defined: for making the creation of complex business process models easy and intuitive. Any configurable process modeling language should thus be defined such that its complexity is in line with the complexity of the process modeling notation on which it is based.

In general, we can conclude that the idea of configuring process models through hiding or blocking ports can and should be applied to basic control-flow patterns [11] like exclusive choices, simple merges, parallel splits, or synchronization in a straightforward manner. This especially holds if the particular process modeling notation expresses the control-flow similar to Petri nets through tokens that can move along the process model's control-flow arcs. Applying the port concept to advanced branching and synchronization patterns, like multi-choice or synchronizing merge, results in a high number of ports. Thus, depending on the target group of the particular modeling notation, designers of configurable process modeling languages might already consider simplifications in the application of the port concept for constructs representing patterns of this group. While in theory it is also possible to represent patterns involving multiple instances through a direct application of the port concept, the amount of configurable ports explodes for these constructs. Thus, simplifications like we introduced for multiple instance tasks in C-YAWL are usually more appropriate for making constructs configurable that deal with multiple instances of a task.

If advanced process modeling constructs allow to determine the executed state change through several distinct parameters, it often helps to address the distinct parameters through dedicated ports. Examples for such constructs are implementations of cancelation patterns in YAWL or links that break BPEL's main control-flow block structure. In such cases, the combination of the distinct ports represents the inflow port according to the theory of process configuration as it was explained in Chapter 3. In order to develop a configurable language that can be used intuitively, Figure 4.21 provides an overview of which pattern types can be addressed by the ports concept in a direct way, and which patterns should better be addressed in a simplified way.

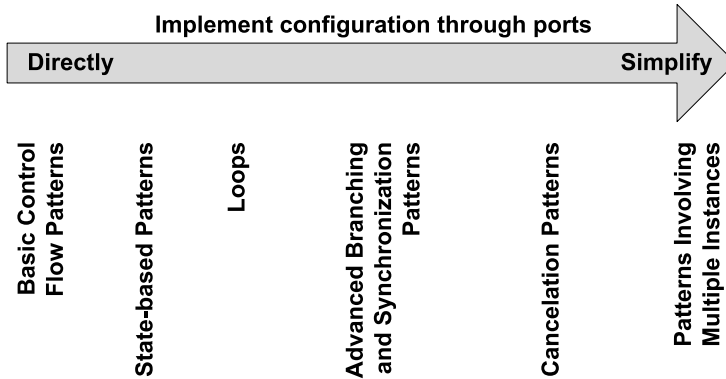


Figure 4.21: How to implement the port concept for the various pattern types [11] occurring in process modeling notations.

To execute the process restricted by process configuration in the corresponding workflow engine, a transformation algorithm can translate a process model's configuration into a new process model without the unwanted behavior. As the definition of executable workflows usually involves defining the resource and data requirements, an adaptation of a workflow is far more cumbersome than adapting a business process model which primarily depicts control-flow aspects of a business process. Hence, a workflow configuration promises even more workload-saving over a manual adaptation than configurable business process models as defined in previous work, like C-EPCs, do.

Moreover, for a successful process execution, the correctness of the derived workflow's syntax, semantics, and contents is essential. Hence, we also showed in this chapter how constraints on the workflow configurations can be formalized for the configurable notations. Still, this only restricts the configuration space without providing any guidance to the model user for finding the next configuration decision that leads to a good, desired configuration. Hence, we will discuss in the next chapter how the constraints defined on the individual process models (as depicted in this chapter) can be used to guide the user to a suitable, executable workflow configuration.

*A lot of times, people don't know what
they want until you show it to them.
Steve Jobs (1998)*

Chapter 5

Guiding the Configuration Process

While the configuration opportunities described so far allow for an easy adaptation of process models to individual needs, the configuration of a particular process still requires a thorough analysis of the basic process model and its various options. Only if users have understood the various possible process executions, they can decide which paths need to be preserved in a configured process variant and which can be eliminated.

These configuration decisions are based on information from the domain to which the model should be applied. If we, e.g., consider the travel booking process from the last chapter (Figure 4.7, p. 79), the configuration decisions to implement the process for a travel agency are different from the decisions made for a travel booking website in the internet. We call the differences in requirements for different organizations the **domain variability**.

The variability of the domain that guides the process model configuration can be captured independently of the process model. For example, it is the decision of a travel agent if payments can be made by credit card or if they must be made in cash. A simple question for the payment methods which can be answered with 'cash' or 'credit card' does not require detailed knowledge of the process model. Still, such an answer provides the information necessary to configure the payment methods that will be allowed in the configured process model. Hence, we suggest in this chapter to first capture domain knowledge and then map the variations in the domain to configuration decisions of the process.

For this, the chapter's first section shows ideas of La Rosa et al. [111, 113] on how the variability of the domain can be captured in structured questionnaires. Afterwards, Section 5.3 depicts how the domain variability can be linked to the variability of process models, i.e. how answers given in a questionnaire can be used to make process configuration decisions while conforming to constraints of

both the domain and the process model. The implementation of these ideas in the context of C-YAWL is described in Section 5.4. The chapter concludes with some links to related approaches capturing domain variability and a summary of the suggested approach.

5.1 Capturing Domain Variability

In line with ideas of La Rosa et al. [111, 113], we suggest to specify the variability of a domain of a business process independently of specific notations or languages, by means of a set of domain facts that form the space of possible answers to a set of questions. A **domain fact** is a boolean variable representing a feature of the domain, e.g. ‘credit card payment’, that can be enabled or disabled. **Questions** group domain facts according to their content such that all the facts of the same question can be set at once by answering the question. For example, the question *Which payment methods are available?* allows users to specify the domain context to choose the payment methods available among fact *Credit card* and *Cash*. A domain fact may even appear in more than one question: in this case it is set the first time and its value is preserved in the subsequent questions. Therefore, if a domain requires multiple payments and if different payment methods are available at these different stages, an individual domain fact must be explicitly referring to each particular payment.

A domain fact always has a **default value** (*true* or *false*). For example, fact *Credit card* is *true* by default as the majority of travel agents allows for credit card payments. Still, a domain fact can also be marked as **mandatory**. Then, it needs to be set explicitly when defining the domain, i.e. the default configuration value must at least be explicitly confirmed through answering a corresponding question when defining a particular domain context. If a non-mandatory fact is left unset, i.e. if no corresponding question is answered, the default value can be used instead. This way, each domain fact will always have a value set — either by giving an answer explicitly, or by using its default value.

To illustrate these concepts, let us have a look at Figure 5.1. It presents a possible structure of questions and domain facts to capture the variability for travel booking processes. All questions and facts are assigned a unique identifier and a description. The questionnaire first determines through question *q1* if the booking of trips is done via the internet or in a shop. This is indicated through facts *f1* and *f2*. By default, *f2* is *true*. Thus, the trips are booked in a shop. Travel bookings can be made for train tickets, reduction cards, or hotel stays, represented by the facts *f3*, *f4*, and *f5*, and grouped through question *q2*. Here, all three facts are *true* by default, i.e. all items can be booked.

Although, all these domain facts have a default value, facts *f1* to *f5* are also mandatory to be confirmed. Therefore, the selection that trips are booked in a shop and that all three types of bookings can be made must be confirmed explicitly.

If *f3* or *f5* are set, then question *q3* can be used to determine how many items can be booked within a single booking process. By default, multiple items can

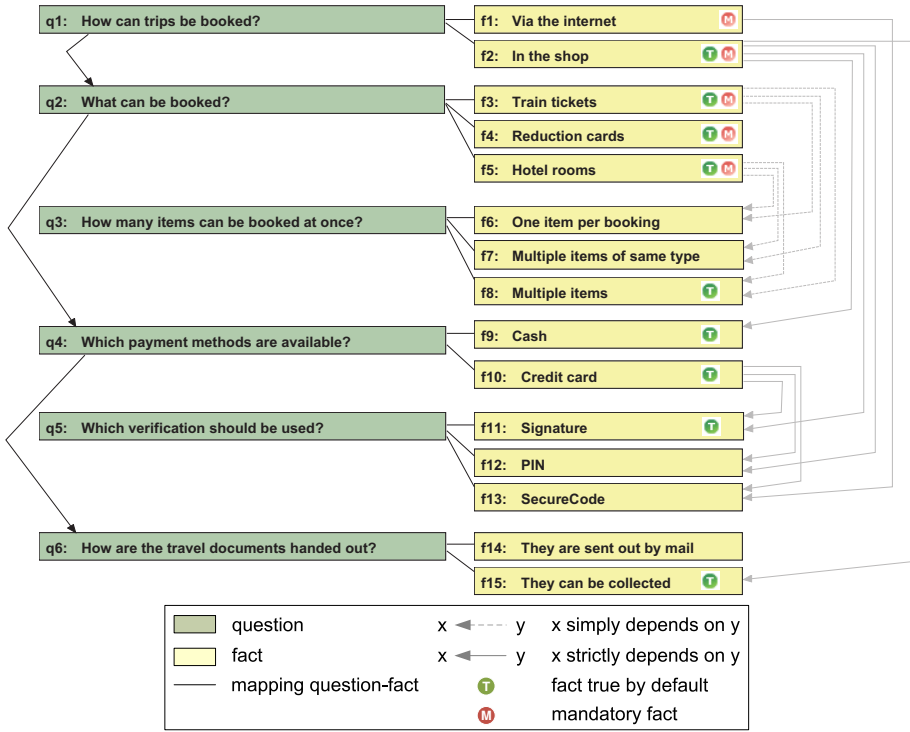


Figure 5.1: A questionnaire for the booking of travels.

be booked within a single booking process ($f8$). However, some systems might only allow for a single item type at a time ($f7$), i.e. within a single booking either train tickets can be bought, a reduction card can be bought, or hotel stays can be booked. Again other systems only allow for one item at all at any time ($f6$), i.e. for booking a single train ticket or a single hotel stay at a time. Facts $f6$, $f7$, and $f8$ **partially depend** on $f3$ and $f5$, but not on $f4$ as there is only one reduction card allowed per (ordering) person. Thus, if the process only allows the booking of reduction cards, there is always only one item per booking.

Interactions like these, which occur among the values of the domain facts, are modeled by a set of **domain constraints** in propositional logic that prune the configuration space. The constraints for the facts of Figure 5.1 are depicted in Figure 5.2.

The constraints therefore ensure that at least one of the two distribution channels must be chosen (DC_1) and that at least one item type can be booked (DC_2). Furthermore, the exclusive choice between $f6$, $f7$, and $f8$ ensures a distinct decision on how many items can be booked within a single booking process (DC_3).¹ If only reduction cards can be booked, but no train tickets or hotel stays, then this is always a single item only, i.e. a single reduction

¹Note that $f1 \dot{\vee} f2$ is true if $f1$ or $f2$ holds but not both.

$$\begin{array}{ll}
DC_1: & f1 \vee f2 \\
DC_3: & f6 \dot{\vee} f7 \dot{\vee} f8 \\
DC_5: & f9 \vee f10 \\
DC_7: & (f1 \wedge \overline{f2}) \Rightarrow (f13 \wedge \overline{f9} \wedge f10) \\
DC_9: & f4 \Rightarrow f14 \\
DC_2: & f3 \vee f4 \vee f5 \\
DC_4: & (f4 \wedge \overline{f3} \vee \overline{f5}) \Rightarrow f6 \\
DC_6: & f10 \Rightarrow (f11 \vee f12) \dot{\vee} f13 \\
DC_8: & f14 \vee f15 \\
DC_{10}: & (f9 \vee f11 \vee f12 \vee f15) \Rightarrow f2
\end{array}$$

Figure 5.2: The domain constraints for the questionnaire in Figure 5.1.

card (DC_4).² DC_5 enforces that at least one payment method is selected. If a credit card payment is possible, i.e. $f10$ is *true*, then a verification method must be selected for credit card payments as well. If the *SecureCode* system is selected, this excludes the use of a *PIN* or a *Signature* as verification mechanism (DC_6). If a booking can only be made via the internet, but not in a shop, the *SecureCode* system must be used in any case (DC_7). The last question $q6$ requires the selection of at least one way how the travel documents are handed over to the client (DC_8). Still, reduction cards are always send out by mail. Hence, if reduction cards can be booked, this must always be possible (DC_9). The existence of a shop is required for the collection of documents as well as for the payment in *Cash* or when using *PIN* or *Signature* as verification method for credit cards (DC_{10}).

A **domain configuration** is a possible valuation over the domain facts that does not violate the domain constraints.

Order dependencies determine the order in which questions are presented to users. The dependencies for determining the order of questions are specified between facts, i.e. between the answering options to questions. If a fact A depends on another fact B , this implies that A can only be set after B has been set. This then means, the question that provides A as an answering option can only be posed after B got a value assigned, i.e. the question to which B is an answering option has been answered.

Facts can also depend on multiple other facts. For this, two types of dependencies must be distinguished: simple dependencies and strict dependencies. A fact that **simply depends** on multiple other facts requires that only one out of the facts it depends on is set before itself can be set, while a **strict dependency** requires that all these other facts must be set before the dependent fact can be set. For example, in Figure 5.1 $f6$ simply depends on $f3$ or $f5$ (captured by a dashed arrow). Hence, the question containing $f6$, i.e. $q3$, can be posed after one of the facts $f3$ or $f5$ has been set, i.e. $q2$ has been answered. Fact $f11$ strictly depends on $f10$ and $f2$ (plain arrow). Therefore, before question $q5$ can be answered, questions $q1$ and $q4$ must be answered to set the values of both these facts.

This way, the most discriminating questions — like $q1$ and $q2$ in Figure 5.1 — can be asked first. By means of domain constraints, this enables to (partly) answer subsequent questions automatically. If, e.g., we answer $q4$ with *Cash* only, the question about the verification method ($q5$) becomes irrelevant. Order

² \overline{f} is *true* if and only if f is *false*.

dependencies between domain facts and questions can be arbitrary, as long as cycles are avoided.

The above concepts form the definition of a **domain configuration model** — a first-class model to capture domain variability. The complete formal definition of domain configuration models can be found in [111]. In the following sections we show how domain configuration models can be applied to support the configuration of process models.

5.2 Capturing Process Variability

While domain configuration models depict the variability of a given domain, the variability of a process is captured by configurable process models as we have defined them in Chapters 3 and 4. Similar to domain facts, variation options in a process model can be identified through so-called **process facts**. Independently of the configurable process modeling notation adopted, a process fact is a boolean variable set to *true* if the variation option it refers to is selected in a given process configuration, and to *false* otherwise. Thus, process facts refer to the variation options of a process model's ports.

Let us, for example, consider an inflow port. It can be configured either as allowed, as blocked, or as hidden. Therefore, there are three variation options for this port, meaning there are three process facts associated with the configurable port, one for each option. Each of the process facts is *true* if the particular configuration option is selected. If it is not selected, the process fact is *false*.

Figure 5.3 shows the various process facts for some of the tasks in the travel booking process from Figure 4.6 (p. 77). To depict process facts, we use a notation very similar to the notation that we use for atomic process requirements (as listed in Definition 4.7, p. 85). In this way, setting the process fact $(in, ('Receive\ Order', \{\mathbf{i}\}), allow)$ to *true* implies that the input port of the *Receive order* task is configured as allowed, while setting $(in, ('Receive\ Order', \{\mathbf{i}\}), block)$ to *true* implies that the port is configured as blocked, and setting $(in, ('Receive\ Order', \{\mathbf{i}\}), hide)$ to *true* implies that it is configured as hidden.

Obviously, a single port cannot be configured in different ways at the same time, i.e. it is not possible to block and hide, to hide and allow, or to block and allow the same port in the same configuration. Thus, in each configuration, only one of the three facts can be set to *true*. Ensuring this is trivial by setting up a process constraint that puts the three process facts into an exclusive disjunction. Thus, for the example port this constraint would be: $(in, ('Receive\ Order', \{\mathbf{i}\}), allow) \dot{\vee} (in, ('Receive\ Order', \{\mathbf{i}\}), block) \dot{\vee} (in, ('Receive\ Order', \{\mathbf{i}\}), hide)$.

The number of process facts related to a task's multiplicity configuration can be unlimited (indicated as '...' in Figure 5.3). Still, it is sufficient to specify only those process facts that might be set to *true*. All other facts represent irrelevant configurations. As only one process fact can be set to *true* at a time, these irrelevant facts are automatically set to *false*.

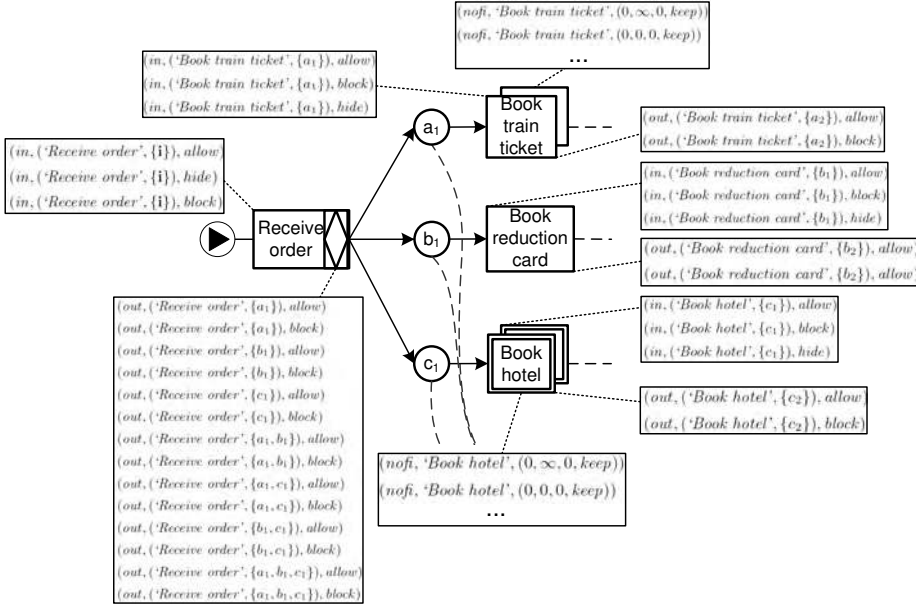


Figure 5.3: Process facts of the travel booking process.

All in all, the valuation of process facts of a configurable model therefore has to satisfy both the constraints guaranteeing an unambiguous configuration for each port, and any configuration constraints imposed on the configurable process model in general (as outlined in Section 4.3.2). Let us therefore in the following assume that the constraint on the process configuration PC enforces such a configuration, i.e. a valuation of process facts that results in a valid, an unambiguous port configuration.

5.3 Linking Domain and Process Variability

In principle, the domain configuration model and the configurable process model are independent models. Both models can be linked by mapping domain facts to process facts. To define this mapping, a two-way impact analysis is required:

- from domain to process: given a domain fact, we need to estimate what are the implications in the process model of setting such a fact to *true* or *false*;
- from process to domain: given a variation point in the process model, we need to consider which domain facts are impacted by configuring such a point to a particular variant.

Figure 5.4 illustrates the mapping. On the one hand, we have a domain configuration model. It consists of a set of domain facts $F_D = \{f_1, \dots, f_n\}$ with DC being a boolean function representing the conjunction of the domain

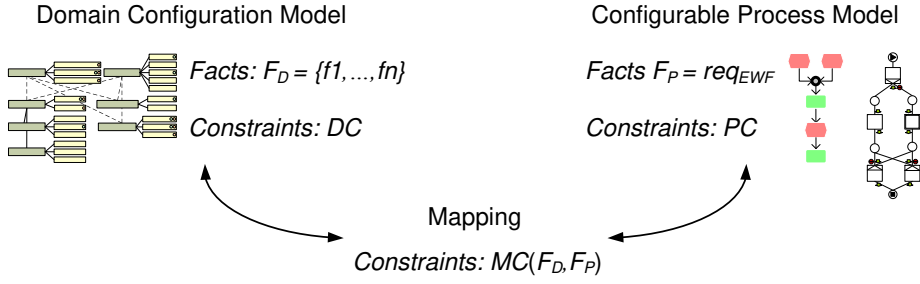


Figure 5.4: Mapping domain configuration and process configuration.

constraints over the domain facts F_D . In other words, $DC = DC_1 \wedge \dots \wedge DC_m$, such that DC holds for every domain configuration. On the other hand, we have a configurable process model which represents the process variability through a set of process facts F_P . For example, we say for EWF-nets $F_P = req_{EWF}$ (Definition 4.7, p. 85). But note again, that process facts can be identified for any configurable process modeling notation, which is indicated through also depicting a C-EPC in Figure 5.4. The process constraint PC is a boolean function guaranteeing the correctness of the process configuration as we have depicted in chapters 3 and 4 as well as in Section 5.2 above. Thus, PC holds for every process configuration.

Domain facts and process facts can be mapped onto each other using a third boolean function $MC(F_D, F_P)$. $MC(F_D, F_P)$ is set up such that each process fact equals a boolean expression over the domain facts. For example, the input ports of the task *Send documents* in Figure 4.6 (p. 77) directly depend on the domain fact *They are sent out by mail* (f_{14}) from Figure 5.1 (p. 107). To send the documents out if f_{14} holds, the use of both input ports of the task must be allowed, i.e. $(in, ('Send\ documents', \{a_4\}), allow) \Leftrightarrow f_{14}$, and $(in, ('Send\ documents', \{b_4\}), allow) \Leftrightarrow f_{14}$. In contrast, the ports must be blocked if the documents should not be sent and therefore f_{14} does not hold, i.e. $(in, ('Send\ documents', \{a_4\}), block) \Leftrightarrow \overline{f_{14}}$, and $(in, ('Send\ documents', \{b_4\}), block) \Leftrightarrow \overline{f_{14}}$. As our simple example always produces some tickets that have to be handed out, neither the sending of documents nor the collection of documents can be skipped. Thus, the input ports of both tasks should never be configured as hidden and we can bind the corresponding process facts to false, i.e. $(in, ('Send\ documents', \{a_4\}), hide)$, $(in, ('Send\ documents', \{b_4\}), block)$, $(in, ('Collect\ documents', \{a_4\}), hide)$, and $(in, ('Collect\ documents', \{b_4\}), block)$. As another example, the task *Select payment method* should be used if both cash and credit card payments are possible, otherwise it should be skipped. Hence, we map $(in, ('Select\ payment\ method', \{a_2, b_2, c_2\}), allow) \Leftrightarrow f_9 \wedge f_{10}$ and $(in, ('Select\ payment\ method', \{a_2, b_2, c_2\}), hide) \Leftrightarrow \overline{f_9 \wedge f_{10}}$. As blocking it would cause a deadlock of the process before this task, we set $(in, ('Select\ payment\ method', \{a_2, b_2, c_2\}), hide)$.

The following list provides the mapping constraints between the domain facts from Figure 5.1 and the process facts of the YAWL model from Figure 4.7. Any process facts that are not contained in the list below are simply set to *false*.

- $MC_1 : (in, ('Receive\ order', \{i\}), allow) \Leftrightarrow true \wedge$
 $MC_2 : (out, ('Receive\ order', \{a_1\}), allow) \Leftrightarrow f3 \wedge$
 $MC_3 : (out, ('Receive\ order', \{a_1\}), block) \Leftrightarrow \overline{f3} \wedge$
 $MC_4 : (out, ('Receive\ order', \{b_1\}), allow) \Leftrightarrow f4 \wedge$
 $MC_5 : (out, ('Receive\ order', \{b_1\}), block) \Leftrightarrow \overline{f4} \wedge$
 $MC_6 : (out, ('Receive\ order', \{c_1\}), allow) \Leftrightarrow f5 \wedge$
 $MC_7 : (out, ('Receive\ order', \{c_1\}), block) \Leftrightarrow \overline{f5} \wedge$
 $MC_8 : (out, ('Receive\ order', \{a_1, b_1\}), allow) \Leftrightarrow f3 \wedge f4 \wedge (f7 \vee f8) \wedge$
 $MC_9 : (out, ('Receive\ order', \{a_1, b_1\}), block) \Leftrightarrow \overline{f3 \wedge f4 \wedge (f7 \vee f8)} \wedge$
 $MC_{10} : (out, ('Receive\ order', \{a_1, c_1\}), allow) \Leftrightarrow f3 \wedge f5 \wedge (f7 \vee f8) \wedge$
 $MC_{11} : (out, ('Receive\ order', \{a_1, c_1\}), block) \Leftrightarrow \overline{f3 \wedge f5 \wedge (f7 \vee f8)} \wedge$
 $MC_{12} : (out, ('Receive\ order', \{b_1, c_1\}), allow) \Leftrightarrow f4 \wedge f5 \wedge (f7 \vee f8) \wedge$
 $MC_{13} : (out, ('Receive\ order', \{b_1, c_1\}), block) \Leftrightarrow \overline{f4 \wedge f5 \wedge (f7 \vee f8)} \wedge$
 $MC_{14} : (out, ('Receive\ order', \{a_1, b_1, c_1\}), allow) \Leftrightarrow f3 \wedge f4 \wedge f5 \wedge (f7 \vee f8) \wedge$
 $MC_{15} : (out, ('Receive\ order', \{a_1, b_1, c_1\}), block) \Leftrightarrow \overline{f3 \wedge f4 \wedge f5 \wedge (f7 \vee f8)} \wedge$
 $MC_{16} : (in, ('Book\ train\ ticket', \{a_1\}), allow) \Leftrightarrow true \wedge$
 $MC_{17} : (out, ('Book\ train\ ticket', \{a_2\}), allow) \Leftrightarrow true \wedge$
 $MC_{18} : (nofi, ('Book\ train\ ticket', (0, \infty, 0, keep)), \Leftrightarrow f6 \wedge$
 $MC_{19} : (nofi, ('Book\ train\ ticket', (0, 0, 0, keep)), \Leftrightarrow f7 \vee f8 \wedge$
 $MC_{20} : (in, ('Book\ reduction\ card', \{b_1\}), allow) \Leftrightarrow true \wedge$
 $MC_{21} : (out, ('Book\ reduction\ card', \{b_2\}), allow) \Leftrightarrow true \wedge$
 $MC_{22} : (in, ('Book\ hotel', \{c_1\}), allow) \Leftrightarrow true \wedge$
 $MC_{23} : (out, ('Book\ hotel', \{c_2\}), allow) \Leftrightarrow true \wedge$
 $MC_{24} : (nofi, ('Book\ hotel', (0, \infty, 0, keep)), \Leftrightarrow f6 \wedge$
 $MC_{25} : (nofi, ('Book\ hotel', (0, 0, 0, keep)), \Leftrightarrow f7 \vee f8 \wedge$
 $MC_{26} : (map, ('Book\ hotel', 'Call\ and\ Catalogue'), allow) \Leftrightarrow true \wedge$
 $MC_{27} : (in, ('Cancel\ booking', \{...\}), allow) \Leftrightarrow true \wedge$
 $MC_{28} : (out, ('Cancel\ booking', \{o\}), allow) \Leftrightarrow true \wedge$
 $MC_{29} : (rem, ('Cancel\ booking', allow) \Leftrightarrow true \wedge$
 $MC_{30} : (in, ('Select\ payment\ method', \{a_2, b_2, c_2\}), allow) \Leftrightarrow f9 \wedge f10 \wedge$
 $MC_{31} : (in, ('Select\ payment\ method', \{a_2, b_2, c_2\}), hide) \Leftrightarrow \overline{f9 \wedge f10} \wedge$
 $MC_{32} : (out, ('Select\ payment\ method', \{a_3\}), allow) \Leftrightarrow f10 \wedge$
 $MC_{33} : (out, ('Select\ payment\ method', \{a_3\}), block) \Leftrightarrow \overline{f10} \wedge$

$$\begin{aligned}
MC_{34} &: (out, ('Select\ payment\ method', \{b_3\}), allow) \Leftrightarrow f_9 \wedge \\
MC_{35} &: (out, ('Select\ payment\ method', \{b_3\}), block) \Leftrightarrow \overline{f_9} \wedge \\
MC_{36} &: (in, ('Credit\ card\ payment', \{a_3\}), allow) \Leftrightarrow true \wedge \\
MC_{37} &: (out, ('Credit\ card\ payment', \{a_4\}), allow) \Leftrightarrow true \wedge \\
MC_{38} &: (map, ('Credit\ card\ payment', 'Signature\ verification'), allow) \Leftrightarrow f_{11} \wedge \\
MC_{39} &: (map, ('Credit\ card\ payment', 'Signature\ verification'), block) \Leftrightarrow \overline{f_{11}} \wedge \\
MC_{40} &: (map, ('Credit\ card\ payment', 'PIN\ verification'), allow) \Leftrightarrow f_{12} \wedge \\
MC_{41} &: (map, ('Credit\ card\ payment', 'PIN\ verification'), block) \Leftrightarrow \overline{f_{12}} \wedge \\
MC_{42} &: (map, ('Credit\ card\ payment', 'SecureCode\ verification'), allow) \Leftrightarrow f_{13} \wedge \\
MC_{43} &: (map, ('Credit\ card\ payment', 'SecureCode\ verification'), block) \Leftrightarrow \overline{f_{13}} \wedge \\
MC_{44} &: (in, ('Cash\ payment', \{b_3\}), allow) \Leftrightarrow true \wedge \\
MC_{45} &: (out, ('Cash\ payment', \{b_4\}), allow) \Leftrightarrow true \wedge \\
MC_{46} &: (in, ('Send\ documents', \{a_4\}), allow) \Leftrightarrow f_{14} \wedge \\
MC_{47} &: (in, ('Send\ documents', \{a_4\}), allow) \Leftrightarrow \overline{f_{14}} \wedge \\
MC_{48} &: (out, ('Send\ documents', \{\mathbf{o}\}), allow) \Leftrightarrow true \wedge \\
MC_{49} &: (in, ('Collect\ documents', \{b_4\}), allow) \Leftrightarrow f_{15} \wedge \\
MC_{50} &: (in, ('Collect\ documents', \{b_4\}), allow) \Leftrightarrow \overline{f_{15}} \wedge \\
MC_{51} &: (out, ('Collect\ documents', \{\mathbf{o}\}), allow) \Leftrightarrow true
\end{aligned}$$

Through such a mapping, answering a question in the questionnaire may affect one or multiple variation points, i.e. process facts. For example, if we answer question $q1$ in such a way that bookings can only be made via the internet, i.e. $f1$ is *true* and $f2$ is *false*, the domain constraints lead to $f13$ and $f10$ becoming *true*, while $f9$ becomes *false* (DC_7 , see Figure 5.2, p. 108). Furthermore, as a result of this, $f11$ and $f12$ become *false* (DC_6). The mapping uses these values to configure the process model: due to MC_{31} , the selection of a payment method is configured as hidden, and MC_{32} and MC_{35} enforce a credit card payment. Moreover, the only possible verification method for the credit card is the *SecureCode* verification, all other implementations of the *Credit card payment* task are blocked (MC_{39} , MC_{41} , MC_{42}).

In addition, it is possible that the configuration of a variation point is determined through more than one question. For example, the configurations of the output ports of task *Receive order* which put tokens into multiple succeeding conditions are determined through the domain facts $f3 - f8$, i.e. through answering questions $q2$ and $q3$ ($MC_8 - MC_{15}$).

A valid process configuration with respect to the domain is given by any domain configuration that leads to a process configuration via a valid mapping. *That means, the conjunction $DC \wedge PC \wedge MC$ must be satisfiable.* The configuration space is obtained by the intersection of the domain configuration space and the process configuration space via the mapping. Thus, like we suggested

for checking process constraints in Chapter 3, SAT solvers [121] can be used to check the satisfiability of the conjunction of process constraints with domain constraints and the mapping.

As the process constraint incorporates the exclusive disjunction of the different configuration values referring to a single port (see Section 5.2), concatenating the mapping with the process constraint automatically enforces that each port can refer only to one process fact that is *true* at a time. This allows us to specify mappings only for those process facts which together guarantee a *true* process fact. For example, MC_{30} configures the input port of the *Select payment method* task as allowed if $f9$ and $f10$ both are true, while MC_{31} hides the port if this is not the case. Hence, any other process fact referring to this port (like the one blocking the port) can never become *true*, even if we do not explicitly specify that it is always *false*. If we explicitly set a process fact to *true* like in MC_1 , MC_{16} , MC_{17} , $MC_{20} - MC_{23}$, etc., it is not necessary to specify a mapping for any other process fact referring to the same port. If the number of port configurations is limited, like it is for input ports, output ports, cancelation ports, and hierarchical mappings of YAWL tasks, we can even ‘underspecify’ the mapping of process facts for a port by one process fact: If the given mapping does not lead to a *true* process fact, the exclusive disjunction among the process facts, automatically sets the non-mapped process fact to *true*. For that reason, we could, e.g., also drop one of the two process fact mappings for output ports from the list above, i.e. either MC_2 or MC_3 , either MC_4 or MC_5 , either MC_6 or MC_7 , etc.

By representing the domain variability in a separate model, we can avoid capturing the interdependencies of the domain in the configurable process model. They are represented by the domain constraints in the domain configuration model and propagated to the process model via the mapping. Constraints over process facts have thus to deal with the preservation of the model correctness only. The correctness of the model with respect to the domain is achieved through the mapping which propagates the domain constraints from the domain configuration model to the process model.

The other way around, as a result of the application of the mapping, the process constraints might also restrict the configuration space of the domain. That means, some answering options in the questionnaire are denied as, based on the mapping, they would lead to non-executable process models. Hence, the process model and its constraints are also measures to check and guarantee the feasibility of what is possible according to the domain model. If the mapping between domain facts and process facts is so restrictive that no correct process model is allowed at all, i.e. $DC \wedge PC \wedge MC$ is not satisfiable, then there is either an obvious discrepancy between the domain model and the process model in the depiction of the process domain, or the mapping is incorrect. Thus, checking this constraint provides a good tool for detecting semantic errors in both models.

Once each variation point has been configured, i.e. each process fact is set to *true* or *false*, these configuration decisions can be used to derive a configured process variant as depicted in Chapters 3 and 4.

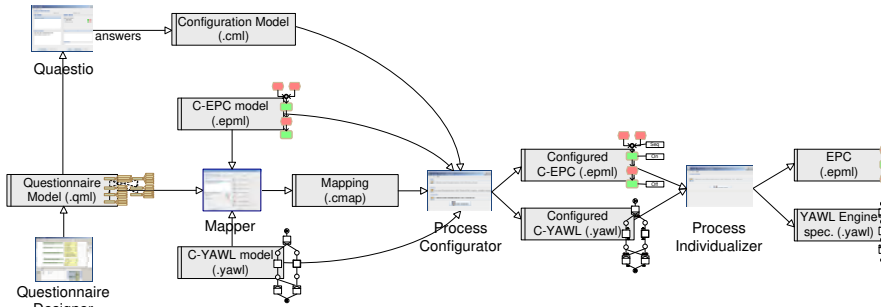


Figure 5.5: The software architecture of the tools implemented.

5.4 Tool Support

Synergia³ is a set of tools supporting process configuration by practically establishing the sketched configuration framework. Each of Synergia’s tools is a stand-alone application responsible for a specific task in the configuration process, from the design of the questionnaire and the collection of its answers, to the release of a configured process model. Figure 5.5 provides an overview of the toolset’s architecture.

In a first step, a domain configuration model like the one from Figure 5.1 and the corresponding domain constraints can be set up using the **Questionnaire Designer**. The Questionnaire Designer saves the domain configuration model as an XML serialization which represents the input to **Quaestio**.

Quaestio presents the domain configuration model as an interactive questionnaire guiding one through the configuration process by posing only the relevant questions in an order consistent with the order dependencies. Questions can be either answered explicitly, or, if they are not set to *mandatory* in the domain configuration model, they can be answered automatically by using the default values given in the domain configuration model. Questions which have already been answered can also be rolled back to re-consider a decision.

Quaestio prevents users from entering conflicting answers among different questions by dynamically checking the domain constraints. For this, it embodies a SAT solver based on SBDDs.⁴ Algorithms based on SBDDs can easily deal with systems made up of around one million of possibilities [121]. As a result, Quaestio can scale with domain configuration models yielding around one million domain configurations.

For example, Figure 5.6 provides a screenshot of Quaestio. Here, the first question *How can trips be booked?* is answered with *Via the internet* only. In the next question on what can be booked (compare Figure 5.1) we furthermore answer that both train tickets and reduction cards can be booked, but no hotels. Figure 5.7 shows that Quaestio is able to automatically determine the

³available via <http://www.processconfiguration.com>

⁴Available at <http://www-verimag.imag.fr/~raymond/tools/bddc-manual>.

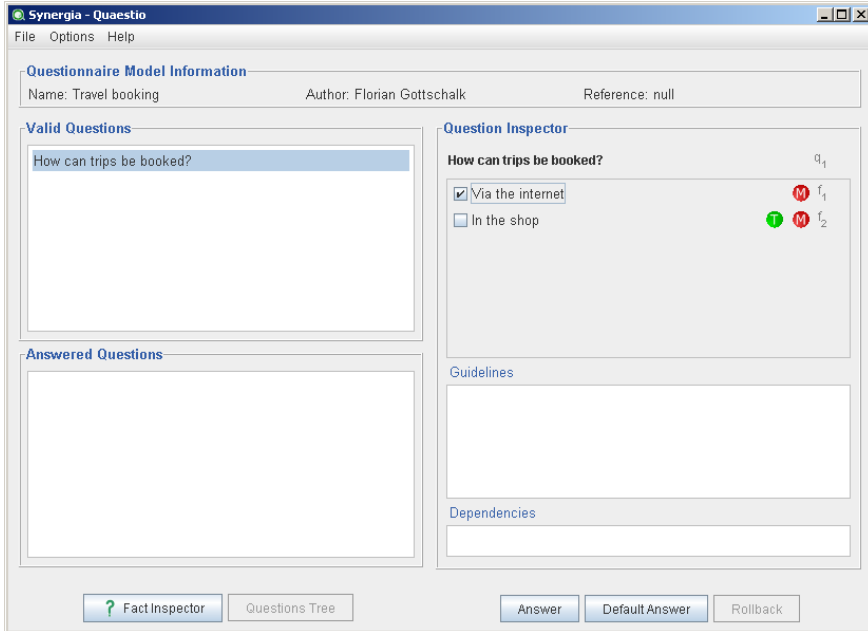


Figure 5.6: Answering the questionnaire from Figure 5.1 (I).

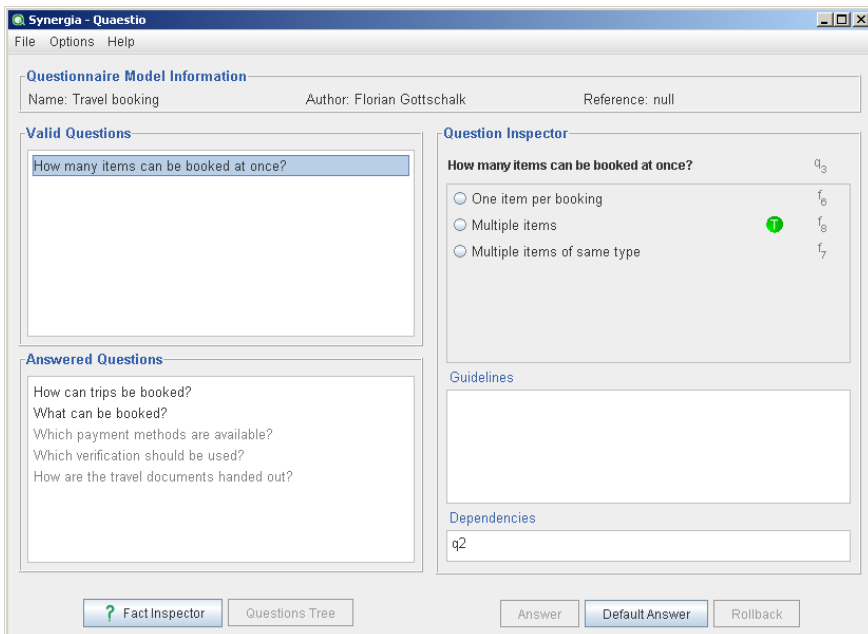


Figure 5.7: Answering the questionnaire from Figure 5.1 (II).

answer for the next questions on which payment methods are available, on which verification should be used, and on how the travel documents are handed out (indicated by their grey font color in the list of answered questions). For this, Quaestio evaluates the domain constraints from Figure 5.2 on $f1 \wedge \overline{f2}$. This sets $f10$ and $f13$ automatically to *true* while $f9$ becomes *false* (Domain constraint DC_7). Thus, all facts connected to the question on the payment method ($q4$) are immediately set. As $f2$ is set to *false*, neither $f9$ nor $f11$, nor $f12$, nor $f15$ can be *true* (DC_{10}). Thus, all facts connected to the question for the verification method ($q5$) are also set. Furthermore, DC_8 requires that at least one of $f14$ or $f15$ must be *true*. As $f15$ is set to *false*, this can only be $f14$. Hence, all facts about handing out travel documents are set automatically as well and question $q6$ is answered.

The last question that remains open is how many items can be booked at once. As this question is not mandatory, Quaestio asks if the question should be answered automatically with the default answer, or if it should be answered manually. Here, we opt for making this decision manually. Figure 5.7 thus shows the available answering options. Quaestio automatically discovers through DC_3 that only one of the three answer can be given. Thus, different from the first question shown in Figure 5.6, it uses now radio buttons instead of check boxes. In this way, the user can really chose only one of the options. However, so far, we have not selected any answer while at least one of the facts must be selected according to the constraints. For that reason, the button to answer the question is currently disabled. The *Answer* button only becomes enabled when we have made our choice. Alternatively, we can simply press the button *Default Answer* in which case the corresponding answer *Multiple items* ($f8$) will be chosen (indicated by the circled *T* next to it). Let us do so here. The given answers are then saved as the domain configuration.

The **Mapper** provides an interface to map process facts from a configurable process model to domain facts from the domain configuration model and create an XML serialization of it. Besides C-YAWL, the framework also supports C-EPCs. In C-EPCs, process facts correspond to the various variants to which a configurable connector can be set, as well as to turning a configurable function on or off. However, note that while a domain configuration is independent of the chosen process modeling notation, a mapping is always created specifically for a particular configurable process modeling language because it has to address the configurable elements which — as we have seen in Chapter 4 — differ among languages. The mapper uses the SAT solver that is also used in Quaestio to check the consistency of the domain constraints and of the process constraints, i.e. if they can be satisfied and if each fact can be freely set or if the mapping already binds certain facts to *true* or *false*. Thus, it also verifies the validity of the generated mapping, checks for redundancies, and shows possible restrictions of the single configuration spaces (domain and process) that may occur from the application of the mapping. The mapping for our C-YAWL example was already shown in Section 5.3.

The domain configuration, the mapping, and the corresponding configurable process model available serve as input for the **Process Configurator** (see Fig-

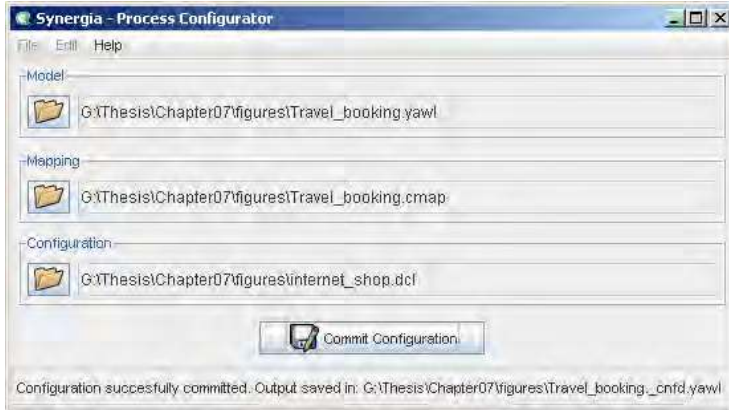


Figure 5.8: The Process Configurator takes the domain configuration, a mapping, and a process model as input to annotate the process model with the configuration decisions.

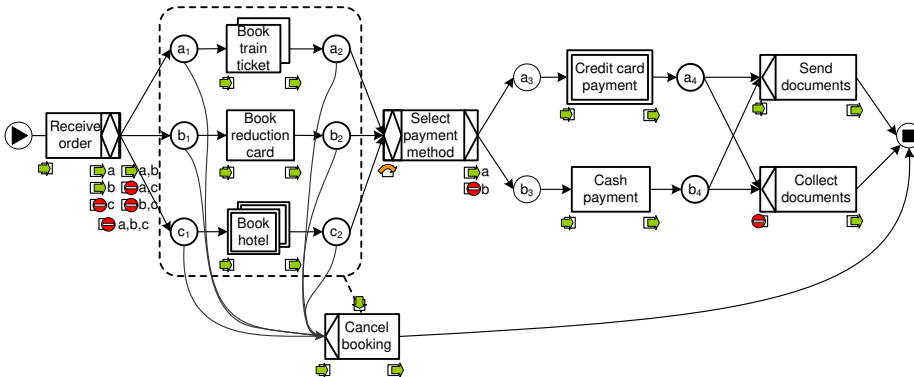


Figure 5.9: The configuration decisions according to the domain configuration decisions given in Quaestio (Figures 5.6/5.7).

ure 5.8). Through the mapping, it generates the process configuration decisions for the configurable process model from the domain configuration and adds these configuration decisions to the configurable process model. That means that the resulting model, which depending on the input is either a C-EPC or a C-YAWL model, contains annotations about the process configuration decisions that correspond to the provided domain configuration. The decisions resulting from our example are shown in Figure 5.9⁵.

In a last step, the **Process Individualizer** uses the annotated, configured process model to generate a configured process model in the original process modeling language which corresponds to the configuration decisions (see Fig-

⁵Note that the result of the Process Configurator is an XML serialization of the depicted model, which can be loaded into the YAWL editor.



Figure 5.10: The Process Individualizer generates a classical process model according to the configuration decisions provided in the model.

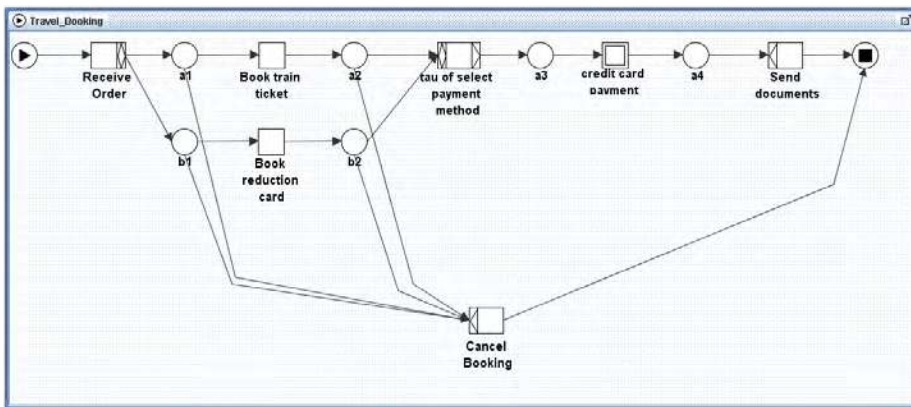


Figure 5.11: The YAWL model produced by the Process Individualizer loaded into the YAWL editor.

ure 5.10). That means, depending on the input, the Process Individualizer is able to generate either an EPC or a YAWL specification. For generating YAWL specifications, the tool uses the algorithm depicted in Section 4.3.5 (pp. 91ff). Thus, the resulting models are directly executable using the YAWL workflow engine or can be edited further using the YAWL editor. For our example, a screenshot of the YAWL editor is shown in Figure 5.11. To generate EPCs that conform to the configuration annotations of C-EPCs, the Process Individualizer uses an algorithm adopted from Rosemann and van der Aalst [143], which can be found in the PhD thesis of La Rosa [110].

5.5 Related Work

The separation of process configuration from the context domain has been investigated, among others, by Becker et al. [33, 34]. In their approach, adaptation parameters and their possible values are linked to model elements to indicate which sections of the model are relevant or not to a specific application scenario. By assigning values to these parameters, a user can configure a process model without looking at the process flow. However, the approach does not yet offer guidance to users when assigning values to the adaptation parameters. Moreover, although it is possible to specify local constraints among parameters, no method is provided to check for model-wide consistency that could, e.g., inhibit deadlocks in the process flow or deny parameter settings that are not feasible from a domain perspective. Hence, to overcome these issues, the approach could be integrated into the configuration framework by mapping the adaptation parameters to domain facts of domain configuration models. Then, Quaestio can be used also to steer the suggested configuration opportunities also in this context.

The use of questions to steer the selection of process alternatives is also advocated by Soffer et al. [170]. Their approach depicts alternatives in the process execution as process specializations. The activation of these specializations is linked to conditions expressed as questions. Thus, the conditions are similar to domain facts. However, constraints over these conditions are not defined and no tool support is offered for this.

More generally, modeling the variability of information systems is a common approach to achieve the reuse of software, known as **Software Product Line Engineering** [129]. Here, models capture how a collection of available options impact the way a software system is built from a set of available software components. Thus, like we integrate several process variants into a single process model and then provide tools to derive an individual process variant from the package, in software product line engineering different software assets are joined into a large package and through selecting the necessary components, a software containing the specifically necessary features can be derived [50].

For example, to capture configuration processes for the Linux kernel the language CML2 was designed [135]. It allows to select or deselect components, like the drivers for specific hardware, that should be compiled into the Linux kernel. For this, CML2 supports the definition of validity constraints based on propositional formulas over so-called symbols and the checking of their consistency. Similar to domain configuration models suggested here, a configuration model in CML2 is composed of questions guiding the configuration process and leading giving values to pre-defined symbols.

A detailed comparison of how domain configuration models address the domain variability with software product line engineering approaches is provided by La Rosa et al. [111]. Especially, two research streams in this field should be mentioned here: **software configuration management** and **feature diagrams**. While software configuration management deals with managing software development projects, feature diagrams describe software product lines

in terms of their particular features. Readers interested in details on software configuration management also in the work of Pressman [130]. For details on feature diagrams, interested readers find details in the work of Kang et al. [100] which introduces a feature oriented domain analysis. Schobbens et al. [166] provide a survey on feature diagram techniques.

5.6 Conclusions

This chapter presented a framework that captures the domain variability independent from the variability of a configurable process modeling language. It does so through a **questionnaire** offering pre-defined answers to the questions. By **mapping** these answers, so-called **domain facts**, to the variability of the configurable process models, so called **process facts**, the process model configuration can be steered through natural language instead of technical constructs likes configurable ports. Once the questionnaire is answered, an individualized model can automatically be derived from the configurable process model.

Through the mapping, also constraints on the possible answers in the questionnaire are linked to configuration constraints of the configurable process model. In this way, a unified set of constraints is obtained which ensures that the answers given always lead to a correct process model configuration, i.e. to a model that is correct both from a context point of view as well as syntactically.

The Synergia toolset implements this approach. It contains tools for defining questionnaires as well as for posing the questions to the user and collecting the corresponding answers. Furthermore, it contains a tool for mapping the domain facts corresponding to these answers to the configurable ports of a process model as well as a tool for deriving the particular process configuration decisions. Finally, the toolset also provides a software for generating the configured process models. Synergia supports both C-EPCs and C-YAWL. It is thus able to generate YAWL specifications and EPCs that conform to the given answers in the natural language questionnaire.

In this way, Synergia allows abstracting from concrete process modeling notations when making configuration decisions. Without confronting subject matter experts with any process models, it guides them through the configuration process and helps them to derive a valid process configuration. Process configuration is reduced to answering a questionnaire.

Obviously, the framework requires to construct both a domain configuration model and a mapping between the domain configuration model and the configurable process model in addition to the configurable process model. Still, as the domain configuration model is based on natural language, constructing a domain configuration model is a far easier task than the construction of a configurable process model. Each domain fact, i.e. each possible answer of the questionnaire, should simply refer directly or indirectly to at least one process variation opportunity identified during the development of the configurable process models. Following this guideline thus helps in defining both relevant questions and relevant process facts.

To test the practical feasibility of the framework (and thus the ease of constructing both the models as well as the mapping between them), we conducted a case study. This case study is described in the next chapter and uses the toolset discussed in this chapter, as well as the ideas on configurable YAWL models developed and discussed in Chapter 4.

*We'll do this as long as it's effective. And feasible.
Devin K. Grayson (2003)*

Chapter 6

Configurable Process Models for Municipalities

Many processes in public administration are driven by legislation [141]. There are laws about registering a child, when a deceased can be buried, what is necessary to get married, etc. For that reason, processes executed in the administration of municipalities are extensively regulated. Still, municipalities have some freedom regarding the concrete implementation of such processes. Thus, they can adapt the particular process executions to local needs and preferences. For example, municipalities with many inhabitants may choose to organize their processes differently from rather small municipalities, or the services provided along with these processes might vary depending on local conditions.

Hence, it is likely that such administration processes are executed similarly among municipalities with small variations. Therefore, these processes are good candidates for being implemented as configurable process models which each municipality can adjust to own requirements. Furthermore, municipalities as public entities have no issues with reporting on their processes, which makes them easily accessible for researchers.

The goal of this chapter is therefore to report on a case study in which configurable process models have been implemented for four registration processes, which are executed on a daily basis in municipalities. The four configurable models, which are created using the configuration extension to YAWL presented in Chapter 4, incorporate all the variations in the execution of these processes among four Dutch municipalities as well as the suggestions of a reference model for these processes. Therefore, a total of $5 \times 4 = 20$ processes were used as input for creating these models. Moreover, a questionnaire was developed for each process that allows steering the configuration of the particular processes through natural language questions as depicted in Chapter 5. Afterwards, the practical usefulness of the resulting models was evaluated through focus group interviews with software providers and consultants. During these interviews the

stakeholders could derive individual process models for these four processes. To test whether the resulting models conform to what was intended by answering the questionnaires, the stakeholders could execute the resulting process definitions using the YAWL system. In this way, end-users could execute example scenarios (in which input screens for the various process steps were provided) to validate the model.

This chapter is organized as follows. First, Section 6.1 depicts how the configurable process models were created and summarizes the practical experiences gained during the model creation phase. Afterwards, Section 6.2 provides details on the interviews performed with the stakeholders and their results. Section 6.3 provides a brief overview on similar case studies before conclusions are drawn in Section 6.4.

6.1 Creating Configurable Process Models

The goal of creating configurable process models for municipality processes is to show the feasibility of building and using, i.e. executing, such models. Thus, it is here sufficient to create configurable process models for a certain selection of the processes executed by municipalities. To build configurable process models, it was therefore necessary to first of all select the processes that should be depicted as configurable process models. Afterwards, it was required to document how these processes are executed in a number of municipalities which forms the basis for the variations in the particular processes. These models then had to be integrated to an executable basic process model, a questionnaire had to be developed covering the variation options, and the questionnaire had to be mapped onto configuration decisions for the basic process model. This then allowed to derive a particular process configuration and execute the corresponding process. The first part of this section explains these steps in more detail. In the second part, we then summarize the challenges that arose during the creation of such models and how they were addressed.

6.1.1 Building the Models

As the purpose of this case study is to determine if configurable process models can improve process model reuse, the processes chosen are deliberately processes for which process model configuration is likely to provide high benefits, i.e. these are processes which are not only highly standardized with small variations, but also executed frequently. Registration processes executed in civil affairs departments are examples of standardized processes of which some are executed relatively frequent. Hence, we chose four of the five most often executed processes in this area:¹

¹The process excluded in this case study is the registration of a couple's divorce. Its main steps are usually a matter of judicature, while the steps taken in the administration of municipalities are trivial.

- *Acknowledging an unborn child:* This process is executed when a man wants to register that he will be the father of a child still to be born while he is not married to his pregnant partner.
- *Registering a newborn:* This process is executed by the municipality to register a newborn child and hand out a birth certificate.
- *Marriage:* This process includes the steps formally necessary before a couple can get married in a Dutch municipality.
- *Issuing a death certificate:* This process is executed when a person deceases to provide the relatives with the documentation necessary to bury the deceased.

The Nederlandse Vereniging Voor Burgerzaken (NVVB²), i.e. the Dutch association for services to the public, offers reference process models for these processes. These reference models provide a single process model for each of the processes which describes the ‘best-practice’ of how the particular process should be executed. They are developed based on the input from the legislative body as well as from municipalities executing these processes.

To detect variations in the process execution in daily practice, the processes of four municipalities in the Netherlands were documented. The selection of these four municipalities was done such that the municipalities vary in the size of their population (between 26.000 and 201.000 inhabitants), as well as such that they use software from different providers to support the process execution. In this way, we can identify a broad spectrum of how these processes can vary.³

Without using the reference model of the NVVB as basis for the discussion, the process owners of the selected municipalities explained how they execute each of the four processes. Based on this input, a separate Protos process model was created for each process in each municipality. Protos was selected as process modeling notation because Protos is very popular among Dutch municipalities for depicting inhouse processes. Thus, the stakeholders of the municipalities were familiar with it. Some of the municipalities even provided us with process models which they had already created to document their processes. Some of these models were clearly adapted from the reference models of the NVVB. If models were provided by the municipalities, the models used in the case study are based on these models. The models were only modified where it became clear from the process descriptions of the particular process owners that a process model did not reflect what was actually happening. To make sure that the processes depict the processes correctly, the final versions of the individual process models were provided to the particular process owners with the request to confirm the validity of the model.

²see <http://www.nvnb.nl>

³Note that besides the broad spectrum of variations, the number of four municipalities is too small to assume that the variations detected are all possible variations, i.e. the input is not sufficient to claim completeness of the variation options.

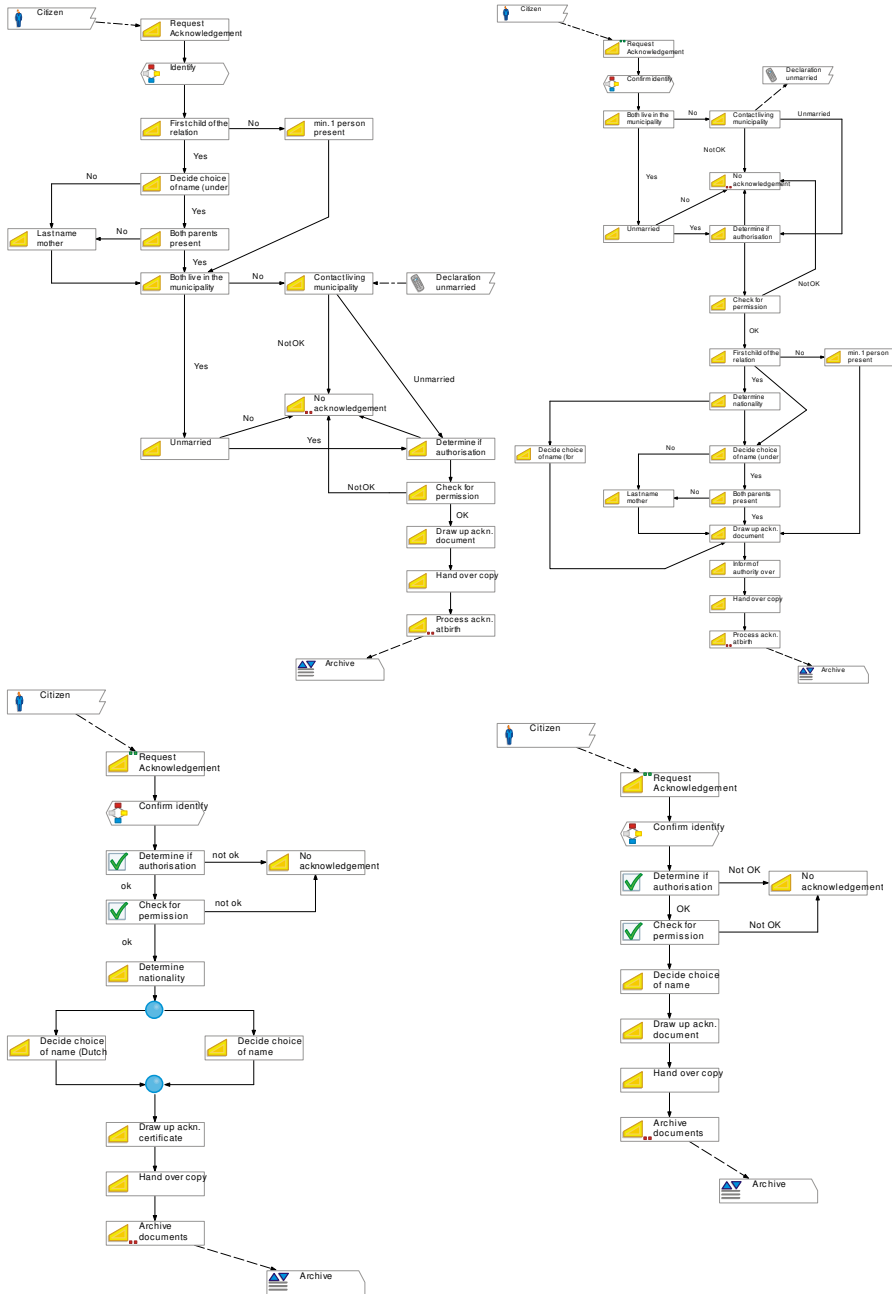


Figure 6.1: The different process variants of how municipalities perform the acknowledgement of an unborn child (enlarged figures can be found in Appendix A).

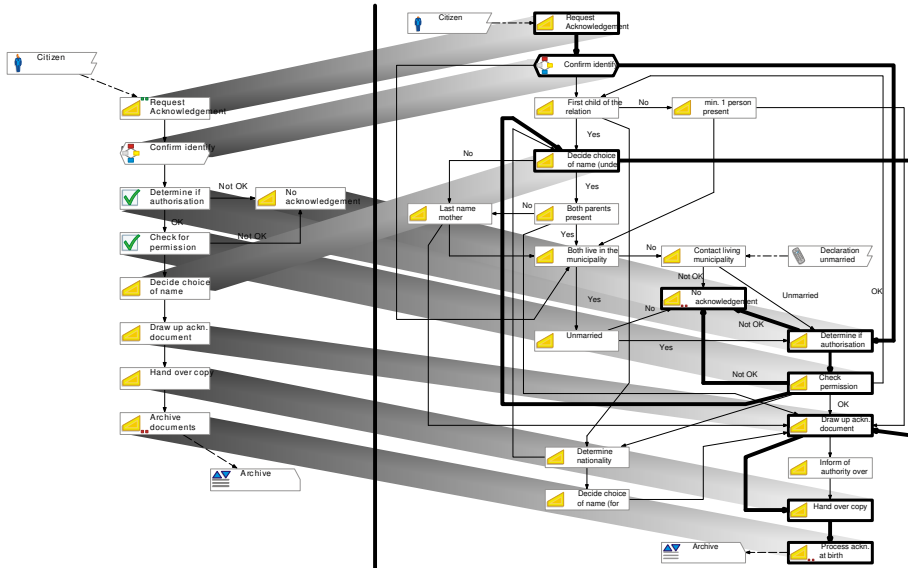


Figure 6.2: Adding a process variant (left) to the integrated process model for acknowledging an unborn child (right).

Figure 6.1 provides an overview of the four Protos models for the process of acknowledging an unborn child, derived from the four municipalities.⁴ While the control-flow of these four processes is similar, the number of steps taken and the concrete order of executing tasks varies among municipalities. This can easily be seen when comparing the four models in Figure 6.1. Thus, the four process models provide sufficient variations to construct a configurable process model.

In addition to these four models, also the NVVB provides their reference models in Protos (and several other notations). Thus, the models of the NVVB even serve as yet another variant of how the described processes can be executed.⁵ Hence, we have $4 + 1 = 5$ process variants per process available as Protos models.

These variants need to be combined into an integrated process model, covering the behavior of the five individual variants. For this, real differences among the process variants had to be distinguished from those differences which refer to the same behavior, i.e. where just different names are used for identical behavior. While naming of identical behavior must be harmonized, the behavior that is really different had to be combined into a single model by introducing choices between the various behaviors of the different models. In this way, we created for each business process a single Protos model that incorporates all the variations from the five input models as ordinary run-time choices. Figure 6.2 shows

⁴The individual models of the other three processes as well as enlarged figures of the four processes depicted in Figure 6.1 are provided in Appendix A.

⁵The reference process models for the four processes are provided in Appendix A.

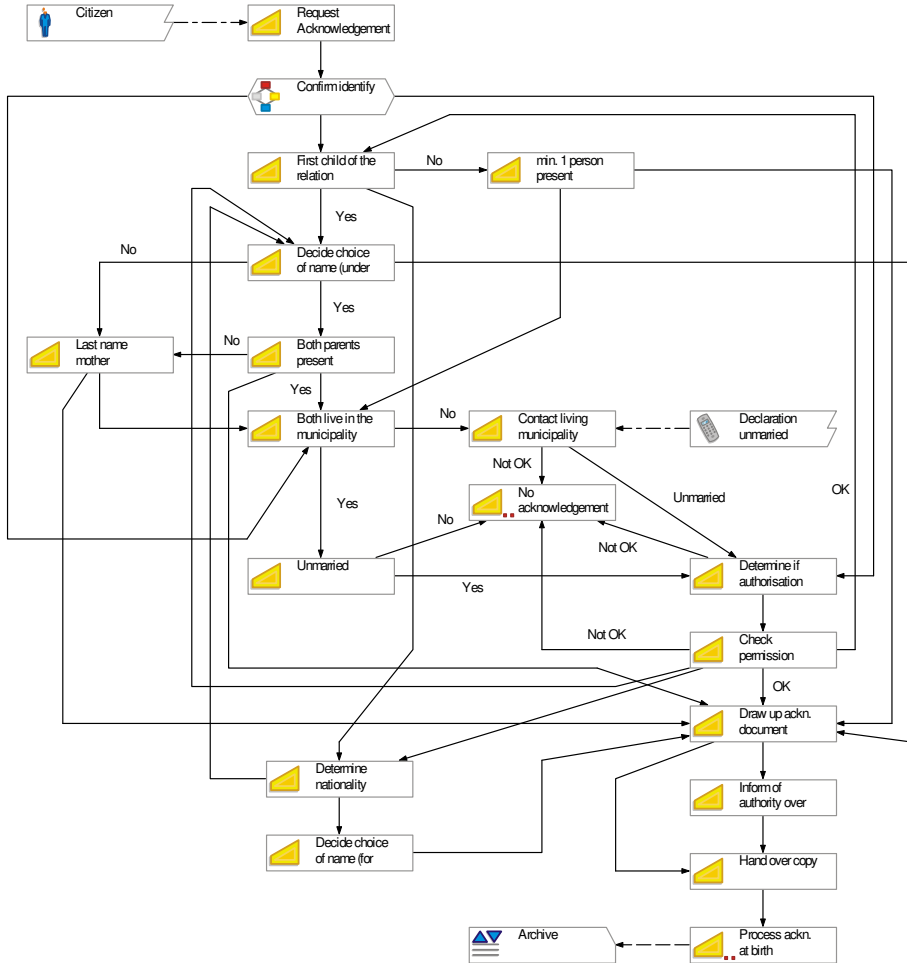


Figure 6.3: All variants for acknowledging an unborn child integrated into a single Protos model.

how the individual model for acknowledging an unborn child on the lower-right of Figure 6.1 was added to the integrated model of the other three models in order to derive the complete basic process model. The complete model is also depicted in Figure 6.3. Note that out of the four business processes in this case study, the process of acknowledging an unborn child is the simplest, i.e. the three other combined process models include both more tasks and more arcs.⁶

To be able to configure and execute the processes, we have to switch to a workflow environment that supports both the configuration and execution of process models. Hence, YAWL is used here such that we can benefit from

⁶Figures A.7 – A.28 in Appendix A provide the integrated Protos models for all processes.

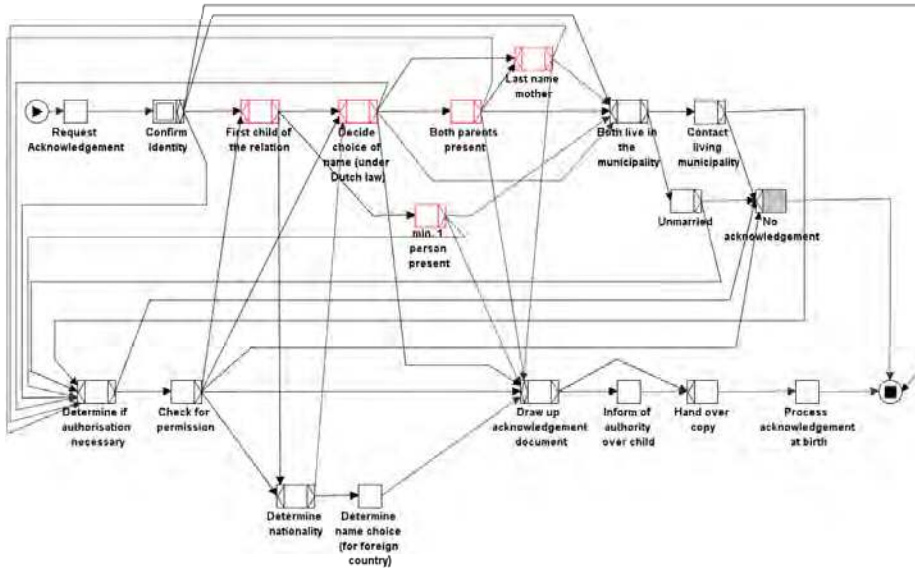


Figure 6.4: The model from Figure 6.3 translated into YAWL

the tools explained in Chapter 5 (Section 5.4, pp. 115ff). The translation from Protos to YAWL was done manually because deriving executable process models not only requires translating the pure control-flow depicted in the figures above, but it also requires implementing the data-flow upon which the process relies. The Protos models provide the data-flow only in a descriptive way, i.e. the definitions cannot be interpreted automatically by a workflow engine. Thus, the resulting YAWL models are fully executable in YAWL's workflow engine. When executing the process, input screens are provided and the entered data is used to determine the necessary run-time decisions. Figure 6.4 shows the YAWL model for the process of acknowledging an unborn child.⁷

When looking at the integrated models of figures 6.3 and 6.4, and comparing these models with the original models in Figure 6.1, it becomes obvious that especially the number of arcs has significantly increased in the combined model. Such YAWL models are far too complex to be used and configured by the stakeholders of the municipalities. To let them configure these basic process models, natural language questionnaires as explained in Chapter 5 are needed. In the questionnaires, each variation opportunity for the process execution is addressed by at least one question. For example, the questionnaire model for the process of acknowledging an unborn child is shown in Figure 6.7. The questions mainly ask if certain tasks should be executed, or in which order these tasks should be executed during the process execution.

The answers to the questions are then mapped to allowing, hiding, or blocking the process flow through various ports. The mapping between the domain

⁷Figures A.8 – A.29 in Appendix A provide the basic YAWL models for all processes.

	Output port (unless otherwise stated)	f1	f2	f3	f4	f5	f6	f6	f7	f8
1	Confirm identity → Both live in the municipality			B						
2	Confirm identity → Determine if authorization necessary			B	B	B				
3	Decide choice of name (under Dutch law) → Draw up acknowledgement Document	B								
4	Last name mother → Draw up acknowledgement Document			B						
5	Both parents present → Draw up acknowledgement document			B						
6	min. 1 person present → Draw up acknowledgement document			B		B				
7	Check for permission → First child of the relationship			B		B				
8	Check for permission → Decide choice of name (under Dutch law)			B		B				
9	Determine nationality → Decide choice of name (under Dutch law)									
10	Confirm identity → First child of the relationship			B	B					
11	Check for permission → Draw up acknowledgement document			B	B	B				
12	Last name mother → Both live in the municipality			B		B				
13	Both parents present → Both live in the municipality			B	B	B				
14	min. 1 person present → Both live in the municipality			B	B	B				
15	Decide choice of name (under Dutch law) → Last name mother				B	B				
16	Decide choice of name (under Dutch law) → Both parents present				B	B				
17	Check for permission → Determine nationality									
18	Draw up acknowledgement document → hand over copy									
19	Draw up acknowledgement document → Inform of authority over child								B	
20	First child of the relationship → Decide choice of name (under Dutch law)									
21	Decide choice of name (under Dutch law) → Both live in the municipality	B	B	B	B	A			B	B
22	Decide choice of name (under Dutch law) → Determine if authorization necessary	B	A	B	B	B	B		B	B
23	Last name mother → Determine if authorization necessary	B	A	B	B	B	B		B	B
24	Both parents present → Determine if authorization necessary	B	A	B	B	B	B		B	B
25	min. 1 person present → Determine if authorization necessary	B	A	B	B	B	B		B	B
26	Contact living municipality → Determine if authorization necessary						B			
27	Unmarried → Determine if authorization necessary						B			
28	Contact living municipality → Draw up acknowledgement document							B		
29	Unmarried → Draw up acknowledgement document							B		
30	All input ports of Both live in the municipality		B							B
31	All input ports of Determine nationality									
32	All input ports of First child of the relationship									

Figure 6.5: Mapping of domain facts to port configurations (I): A=Allowed, B=Blocked, H=Hidden

	f11	f12	f13	f14	f15	f3 ^ f7	f2 ^ f4	f4 ^ f5	f6 ^ f7	f1 ^ f6	f2 ^ f6	f5 ^ f7	f4 ^ f14	f4 ^ f15
1											B			
2			B				A				A			
3														
4										B				
5										B				
6										B				
7											B	B	B	
8											B	B	B	
9														
10														
11														
12														
13														
14														
15														
16														
17						B			B					
18	B													
19		B												
20														
21	B	B	B			B		B						
22	B	B	B				B				B			
23	B	B	B				B				B			
24	B	B	B				B				B			
25	B	B	B				B				B			
26														
27														
28														
29														
30														
31														
32					H									

Figure 6.6: Mapping of domain facts to port configurations (II): A=Allowed, B=Blocked, H=Hidden

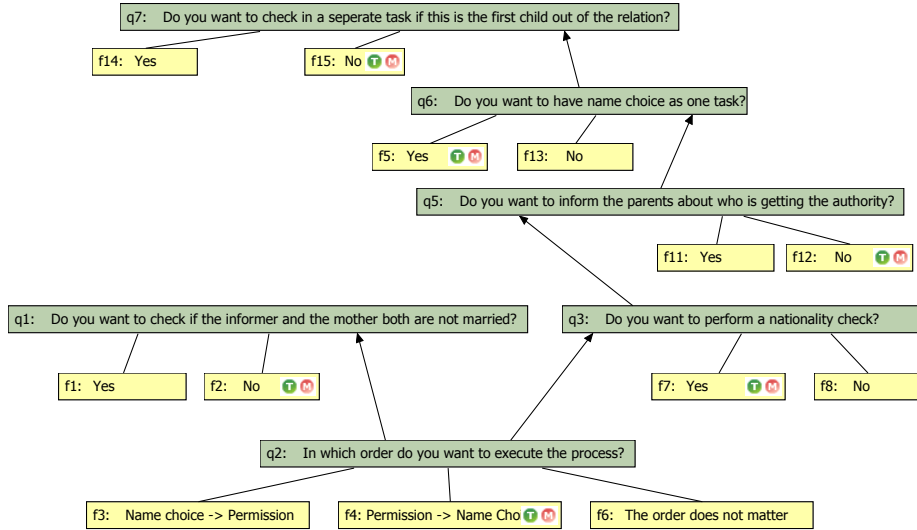


Figure 6.7: The questionnaire model addressing the various options in performing the process of acknowledging an unborn child.

facts of Figure 6.7 and the YAWL model from Figure 6.4 is shown in the table in figures 6.5/6.6. For example, if the question $q1$: *Do you want to check if the informer and mother both are not married* is answered with *Yes*, i.e. $f1$ is *true*, the output port of task *Decide choice of name (under Dutch law)* to *Draw up acknowledgement document* must be blocked (Line 3 of Figure 6.5). This port must also be blocked if $f3$ is *true* or if $f13$ is *true*. Some port configurations also depend on multiple answers. For example, the output port of the task *Last name mother* to *Draw up acknowledgement document* (Line 4 of figures 6.5/6.6) is not only blocked if $f3$ is *true*, but also if $f1$ and $f6$ both are *true*, i.e. if question $q2$: *In which order do you want to execute the process?* is either answered with *Name choice \rightarrow Permission*, or if it is answered with *The order does not matter* and question $q1$: *Do you want to check if the informer and mother both are not married* is at the same time answered with *Yes*.

In this way, the configuration of the process model integrating the five process variants can be done by the stakeholders through simply answering the questionnaire in Quaestio (see Figure 6.8). There is no need for the stakeholders to understand the implications of blocking or hiding certain ports. In fact, they do not even need to be confronted with the integrated process models. After all, the configuration decisions resulting from the answers to the questionnaire can be applied automatically to this model, i.e. a simple specific model is shown rather than a more complicated general model. Using the Process Configurator and the Process Individualizer, for example, the process model shown in Figure 6.9 can be derived from the integrated model in Figure 6.4, using the answers given by one of the involved municipalities. This model can then be loaded into the YAWL workflow engine. Executing the process in the engine, the users of

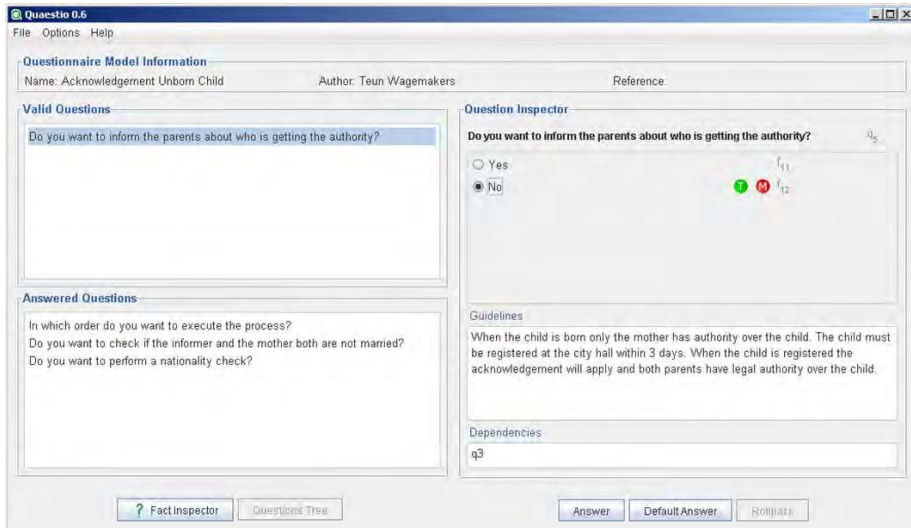


Figure 6.8: Quaestio allows users to answer the questionnaire. Based on the answers the model can be configured automatically.

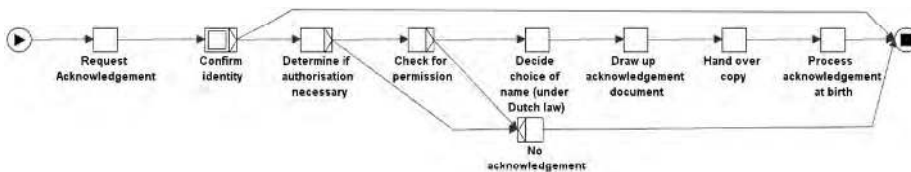


Figure 6.9: The individual process for acknowledging an unborn child.

the model will be filling out input screens, which are generated based on the information of the configured process model. Thus, the users do not see the configured model at all in between answering the questionnaire and executing the process in the engine.

6.1.2 Observations

While performing the case study, several observations were made. Let us thus list which steps of building a configurable process models could be done easily, as well as the biggest issues discovered in this section.

Most important, and thus first of all, it should be noted that *for all four business processes it was possible to create integrated process models and questionnaires that allow users to derive an individual model*. For each process and each municipality it was possible to generate a YAWL model which is equivalent to the original Protos model by answering the questionnaire and applying the resulting configuration to the YAWL model. That means, the four configurable models capture all 20 process variants. For example, the process in Figure 6.9

is equivalent to the process on the lower right of Figure 6.1 (Figure 6.2 highlighted this behavior in the integrated Protos model). This illustrates that it is possible to integrate several process variants of a real business process such that the different desired — and thus realistic — individual variants can be derived from it.

Besides this general satisfaction of the case study's goal, the creation of the configurable models still came with several challenges. While deriving the individual process variants was straightforward, the first challenges arose when integrating the different variants into a single process model: Matching identical tasks among the variants was often only possible after comparing the exact task descriptions. Moreover, during the manual compilation of the integrated model, some paths, i.e. process flows, of the individual models were easily overlooked and thus not incorporated into the integrated model. Only by carefully 're-playing' the processes of the individual models in the combined models [144] these 'forgotten scenarios' were discovered. Also, it was sometimes hard to decide, if tasks should be executed in a certain order or if the order does not matter and these tasks could be executed in parallel.

Due to the extensive support of control-flow patterns in YAWL, translating the control-flow from the Protos models to YAWL models was trivial. More difficult, however, was the implementation of the data-flow for determining the precise run-time routing of cases through the integrated process model. This was especially the case when a choice between various options was introduced in the integrated model while in fact there is no such run-time decision while executing the process in any of the municipalities. The variation is thus a combination of features encountered in the five initial models, while the combination itself was never actually encountered. For example, this applies for the task *Confirm identity* in Figure 6.4 which uses an OR-split to branch into four outgoing paths. The decision, which combination of paths should be triggered after the completion, is partly a run-time decision and partly a configuration decision. During run-time it is decided if the identification was successful or not. If not, the process completes immediately. However, the decision which combinations of the remaining three arcs are triggered in case the identification was successful is already a configuration decision (it might be desired to transform this into a run-time decision, but this was not the case in any of the involved municipalities and is thus no option in the resulting model either). A correct definition of the process flow details in such situations requires the implementation of a 'default' decision as well as a very good anticipation of the implications when this default decision has to change due to a configuration decision.

Questions in the questionnaire abstract from the control-flow of the process and usually address larger process parts. Thus, the interdependencies between the answers that can be given in the questionnaire are not always obvious or immediately derivable from the process's control-flow. Hence, ordering of questions and defining constraints between the answers turned out to be challenging and required a good anticipation of the desired impact of the configuration decisions. This becomes more difficult when the model is complex. Thus, it turned out that defining constraints among the domain facts and defining the mapping

between the domain facts and the configurable ports are in practice not separate phases. They rather have to go hand in hand with each other. That means, ‘bug-fixing’ a domain constraint always had some implication on the mapping between domain facts and process facts and vice versa. Therefore, this was rather an interactive phase between updating the mapping and updating the domain constraints.

In total, the creation of the four configurable process models took approximately six months for a single process designer — from collecting the data of the municipalities to being able to present the stakeholders with the configurable and executable process model. This timeframe also includes the familiarization of the process designer with the process configuration techniques and tools which were new to him at the beginning of the project.

6.2 Evaluation of the Approach

To get insights into the practical applicability of the described models, an additional evaluation was carried out by performing three focus group interviews with one to three employees of the following three organizations, each with a duration of approximately two hours:

- Pallas Athena⁸ as a vendor of BPM products and the supplier of Protos which is actively used by over 250 of the in total 441 Dutch municipalities,
- PinkRoccade Local Government⁹ which provides software to execute municipality processes used by more than half of the Dutch municipalities, and
- a world-wide operating consultancy firm who adapts their own reference process models during process implementations for their clients.

All interview partners were first given a presentation on the techniques used during this project as well as on details of how the four configurable models and their questionnaires were created. Afterwards, the interview partners had the opportunity to derive their own executable process models through answering the questionnaires. The models resulting from the answers given in the questionnaire during the interview were immediately presented to them. Not all the interview partners were domain experts for the given processes. Thus, they were allowed to ask questions on the implications of the various possible configuration decisions in the questionnaire.

Subsequently, we triggered a discussion with the interview partners focussing on potential practical needs for adaptable process models, on the feasibility of creating such configurable models in real-life environments, and on the practical usefulness of applying such configurable models. Sections 6.2.1 – 6.2.3 outline the input received during these interviews. The key results are also summarized in Figure 6.10.

⁸<http://www.pallas-athena.com/>

⁹<http://www.pinkroccadelocalgovernment.nl/>

Interview partner	Potential applications and advantages (+) as well as concerns (-)
Pallas Athena	<p>(+) Configurable Process models would have been useful for the development of a “one point of contact” workflow product for municipalities developed based on a new law that requires municipalities to re-structure the customer interaction of their business processes</p> <p>(+) Potential applications in highly regulated, publicly documented and accessible, or non-core business processes like HR processes.</p> <p>(-) The integrated model must be complete. Is this possible? How can this be derived from existing processes?</p>
PinkRocade Local Government	<p>(+) Questionnaire answers can be linked to other configurable elements, like the configuration of software screens and windows as well as data fields.</p> <p>(+) Configuration through questionnaires enables software providers to create applications that prevent that stakeholders can fail during the process configuration.</p> <p>(+) Stakeholders see in the questionnaire the configuration freedom they have rather than limitations to the configuration space.</p> <p>(+) Clients often ask for software adaptation and modifications for a better support of their desired business processes which is currently expensive due to the need for external consultants. Currently, this often results into workarounds.</p> <p>(-) The configurable models created do not allow for a configuration of the resources that are involved in a process.</p>
Consultancy Firm	<p>(+) Best-practice reference models are often not sufficient: there is no single best-practice.</p> <p>(+) It would have been useful in a world-wide role-out of new business processes where it was a headquarter policy that 80 % of the processes needed to remain conform to the global process while it was allowed to deviate by 20 % to make the process compliant to local regulations.</p> <p>(+) In some industries production processes are so standardized that the technique might even be applicable to core processes.</p> <p>(-) The creation of configurable models seems to require big efforts. Thus, model providers might hesitate to invest in creating such models.</p> <p>(-) The identification of variations between processes is difficult, i.e. tools are necessary for this.</p>

Figure 6.10: The main comments of the interviewed stakeholders

6.2.1 Provider of BPM Solutions

Besides being the software provider offering Protos, Pallas Athena also offers a workflow engine (Flower, like Protos part of BPM|one) and performs BPM projects. After configurable process models were introduced to them, the interview partners immediately saw the applicability of the approach to one of their recent BPM projects in the municipality domain: New laws will require from Dutch municipalities that they provide a ‘single point of contact’ for inhabitants through which these can make any request dealing with issues in the scope of the municipality administration. Hence, many municipalities currently have to restructure their business processes to conform to these new requirements. Therefore, it would have been attractive to Pallas Athena to offer a configurable process models that provides a standard solution. This not only simplifies the adaptation of the processes to individual requirements for the municipalities, but also improves the maintainability of the varying process definitions for Pallas Athena.

The main concerns of the interview partners from Pallas Athena were the completeness of the configurable model. They believe that it is hard to achieve that the model contains all desired options, as well as that the configuration constraints cover all configuration restrictions, i.e. it should be guaranteed that only desired process models can be derived from configurable process models. Hence, they were also very interested in ideas of mining the configurable processes and guaranteeing the correctness of the configured process models as we will discuss in chapters 7 and 8.

The interview partners liked the ideas of steering the process configuration through natural language questionnaires, as it gives process adaptation opportunities to people not familiar with process modeling. Beyond the municipality domain, they also indicated potential opportunities for building configurable process models for non-core, i.e. support processes (e.g. in the human resources domain). The applicability to core business processes, however, is seen as very limited as organizations are usually hesitant to provide insights into their business processes and the corresponding models. Thus, it will be hard to collect the data necessary to identify the process variation options and thus to construct a high-quality configurable process model.

6.2.2 Provider of Municipality Software

PinkRocade Local Government looked at process configuration from the viewpoint of how this could improve their own software provided to municipalities. They, especially liked the fact that through imposing constraints on the configuration options, correct configurations can be guaranteed. Furthermore, they found it very useful that the process execution can be steered through a process model while abstracting from the model is possible when presenting the configuration options to stakeholders in the particular process. They were thus also looking into linking configurations of further elements like a software’s input screens and windows to the domain facts through the same framework as

we used to link process configuration decisions to domain facts. Especially, a linkage to adapting the resource involvement in the process through such a framework would in their opinion provide further benefits.

They see the main benefits of the approach for the model user because questionnaires offer only feasible configuration decisions. All the configuration constraints are hidden, but applied and enforced in the background. Thus, adapting process models is simplified compared to the current situation, where any change of the process requires the involvement of expensive, external consultants. Hence, the interviewed stakeholders expect that this simplification could lead to quicker adaptations of the software to requirements that have changed over time, and thus to less workarounds by process participants that have to use an ‘outdated’ software process.

6.2.3 Consultancy Firm

The consultancy firm interviewed has its own industry-specific reference model which is used as initial template for process implementations. During these implementations, they adapt the reference model to individual needs. These adaptations are, however, not recorded, i.e. the reference model is hardly updated with new insights after projects have been completed successfully. This means that if similar or identical adaptations are required in further projects, these updates need to be performed from scratch. The reason why the reference models are not enriched with knowledge from successful projects is that this update is usually not billable to a customer. Hence, this is also a major issue raised by the consultants interviewed. It will be hard to find a sponsor for the obviously very cumbersome task to create and maintain a configurable process model. According to the consultants, the only way that this issue can be overcome, is by providing a sophisticated tool which builds the configurable process model automatically from the modifications made in various implementation projects.

Still, the consultancy firm could also see benefits of using configurable models. For example, they had done an international process implementation project in which a globally operating organization aimed at standardizing processes worldwide. However, local requirements like local law prevented the implementation of exactly identical processes everywhere. Hence, there was a global policy imposed that stated that local implementations were permitted to deviate at most 20 percent from the globally used process templates. After the presentation of process configuration, the consultancy firm came up with the idea that process configuration could have been a good instrument for measuring these variations during the project.

Furthermore, it is very interesting to see that the consultancy firm had the impression that, based on their experience, core processes in certain industries (like the automotive industry or banks) are organized very similarly. Hence, it would very well be possible to develop configurable process models also for such processes — if the development of the models would not be that costly.

6.3 Related Work

Public administration has already been the subject of several other case studies in the reference modeling domain. For example, Algermissen et al. [23] performed a case study with municipalities to identify best-practice in public administration. Similar to the approach shown here, Algermissen et al. initially visited a number of municipalities to observe and depict their business processes. However, different from our approach, they do not focus on providing a model with various configuration options, but rather aim at deriving a single, ‘ideal’ process model from these variants. Thus, their approach is similar to the one taken by the NVVB, whose best-practice recommendation we incorporated in our models.

Karow et al. [102] provide guidelines specifically for the construction of reference models in public administration. While the goal of the case study presented in this chapter was to test the feasibility of using configurable process models in a reference modeling context and to identify the opportunities provided by such models, we would need to address such guidelines more rigorously if we want to extend our work to providing a complete configurable reference model covering more municipality processes and more process variants in the future.

As mentioned in the introduction, best-practice reference models have also been investigated in several further case studies and projects. For example, Thomas et al. [176] developed a reference model for event management, and Scheer [161] designed a reference model for industrial enterprises. Moreover, commercial process modeling tools often come with standardized libraries of reference process models such as the IT Infrastructure Library (ITIL)¹⁰, the Supply Chain Operations Reference (SCOR) model [171], or IBM’s Rational Process Library¹¹. SAP provides both a reference model [48] depicting a collection of process models corresponding to common business operations supported by SAP’s enterprise system, as well as a repository of workflow templates that can be used to automate processes in the enterprise system (which was already discussed in the introduction of this thesis, as well as in Chapter 4, Section 4.1). Also, consultancy firms often have reference models depicting experiences from their own processes¹². An overview and classification of 30 different reference models is provided by Fettke et al. [67]. However, note that none of all these reference models comes with an explicit tool support to adapt the models to a particular context (like the option for process model configuration we tested in this case study).

A practical case study that resulted in configurable process models was performed by La Rosa et al. [114], building on initial results of Seidel et al. [168]. This case study was conducted in the screen business (post production) and uses the same configuration framework as we use here, but with EPCs a different process modeling notation. Furthermore, instead of applying the techniques

¹⁰see www.itil-officialsite.com

¹¹see <http://www-01.ibm.com/software/awdtools/rmc/library/>

¹²e.g., see <http://www.deloitte.org/dtt/article/0,1002,cid%253D45276%2526pv%253DY,00.html>

to very standardized processes as we do here, they aimed at developing configurable process models in a creative domain.

Further details on the case study presented in this chapter from the viewpoint of the designer of the configurable models can also be found in the work of Wagemakers [185].

6.4 Conclusions

The goal of the case study presented in this chapter was to develop configurable process models for four business processes of municipalities based on information from four different municipalities and a corresponding reference model. Afterwards, the potential use of these models and the underlying process model configuration techniques were evaluated through expert interviews with various stakeholders potentially developing and using such process models.

During the case study, the suggested techniques proved to be suitable for the intended purpose: It was possible to derive all the initial, individual models of the various municipalities as well as further model variants from the integrated models by answering simple questionnaires. Despite that, the creation of the configurable models required significant efforts, modeling experience, and domain knowledge. Moreover, the resulting configurable models do not yet provide a set of variation options for which completeness can be claimed. For this, the amount of five input models is too small.

From interviews with stakeholders in the created models, we can summarize that all interviewees immediately saw a potential value of the technique of configurable process models, which they stressed by mentioning current or past projects where configurable process models could have provided additional benefits. The steering of the actual process configuration is seen as a useful tool to assist end users, but even without this support direct process configuration might prove to be beneficial in various projects where process adaptation is necessary. The interview partners were also concerned about the efforts necessary to create configurable process models, questionnaires, and to establish the links between the potential answers and all the ports.

Thus, the simplified adaptation of process models achieved by using configurable process models is at the expense of an increased amount of efforts required to create configurable models.

Still, the interviewed stakeholders had the impression that many of the time-consuming issues that arose could be improved or even avoided by further tool support, e.g. ensuring consistencies and configuration correctness, or automatically identifying and integrating process variations. Thus, all interview partners were interested in techniques that can help here. Furthermore, the interviewees also made clear that process configuration should not be restricted to the control-flow perspective of business processes, but should also incorporate the resource and data perspectives to provide a strong and universal configuration tool.

We will discuss tools that are beneficial for developing configurable process models in the next chapter, as well as we will develop techniques guaranteeing the applicability of derived configurations in Chapter 8. There, we will also briefly consider the influence of the data-flow and resource involvements on a configuration's correctness.

Combine the extremes, and you will have the true center.
Friedrich von Schlegel (1800)

Chapter 7

Building the Configurable Process Model

A basic process model that integrates the behaviors of different variants of a business process is the basis for any configurable process model. Configuration then enables deriving these and further process variants from the basic process model. The question in this chapter is therefore how such an integration of different process variants into a single basic process model can be achieved.

Usually, the processes represented by configurable process models are not new and do not need to be highly innovative processes. Hence, it should be possible to derive configurable models from **best-practices**, i.e. from well-established implementations which have proven to perform well. That means that when a configurable process model is built, various variants of the process are already operational in different organizations. For example, in the case study from Chapter 6 we started the development of configurable process models based on information on the particular processes given to us by various municipalities. The basic process model thus needs to incorporate the variations among these process variants. The starting point for building a configurable process model is therefore the information available about these best-practice process variants.

Today's IT systems usually generate extensive protocols of what has happened in the form of **log files** (often just called logs for short). Thanks to their use in almost any process execution, such detailed process information is nowadays widely available. Data and process mining techniques have been developed by researchers and practitioners to gain condensed information about the process behavior from such log files. If log files from existing process implementations are available, it thus seems obvious to use these techniques to derive configurable process models from the log files of various systems.

Figure 7.1 explains this approach. At first, the available log files from the different systems must be prepared through filtering of irrelevant content and mapping of different naming conventions among different log files. Afterwards,

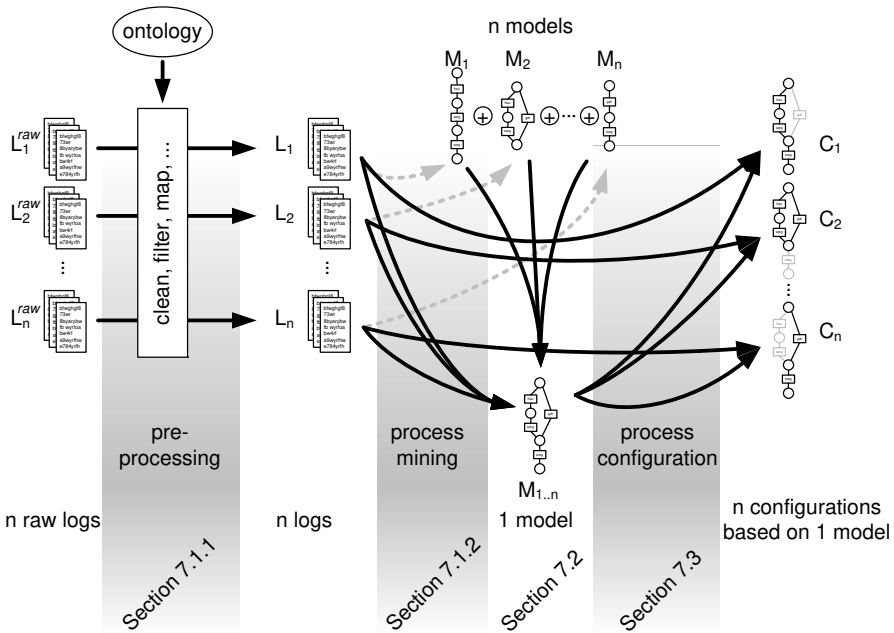


Figure 7.1: Deriving the basic process model and its configurations

process mining techniques can be used not only to create process models for individual systems but also to build process models which are valid for all the systems, i.e. which integrate the various process variants.

Besides the log files, the information available over well-established and well-running processes might already include process models or workflow specifications of these processes. For example, in the case study from Chapter 6, some of the municipalities as well as the NVVB provided us with existing process models documenting the particular processes. In these cases, it can be better to build the basic process model of a configurable model directly from these process models instead of mining such models from log files which might include noise and where the model quality depends on the mining algorithm. Thus, the configurable model is then built by merging the good-quality models (see the vertical arrows from the n individual models to the single integrated process model in Figure 7.1).

In the following we will discuss the details of these two approaches to build the basic process model. In Section 7.1 we will first elaborate on the mining approach before we suggest an algorithm to merge process models directly in Section 7.2.

As the goal of building configurable process models is not the direct execution of these models, but rather the execution of its configured variants, the question that arises when building the integrated process model is how this model can or must be configured. Obviously good configurations are the ones

used for building the basic process model. Thus, the third section of this chapter shows which configurations of the basic process model lead back to the original processes that were used to create the configurable process model.

Having developed the various techniques for improving building configurable process models, Section 7.4 will briefly re-visit the examples from the municipality case study to see to what extent the tools suggested in this chapter help addressing the issues of building configurable process models raised during the case study. Readers interested in further details on the techniques suggested in this chapter as well as on similar approaches, find an overview of related work in Section 7.5 before we conclude this chapter with a brief summary.

7.1 Generating Configurable Process Models from Log Files

In the first approach we assume that the only input we have available for building the configurable process model are log files. Many of today's IT systems constantly write events to log files to record functions that are executed in the system, changes that are made to the system or its data, 'system alive' status updates and so on. For example, most web servers write an entry into a log file for each single requested page including information about the time of access, the IP address of the user, whether the access was successful, and maybe even a user name or submitted data. Due to today's extensive use of information systems in all business areas, such log files containing detailed information about the executed business processes are usually widely available.

Process mining techniques have been developed to discover a model of the behavior behind log files. Our goal here is to use these techniques for *discovering the basic process model of a configurable process model* as we discussed in chapter 3. That means that we want to discover a process model which is valid for all the different variants of a process based on the log files of multiple systems.

Process mining of real-world log files is usually split up into two phases. First the log files are **pre-processed**, i.e. the input log files are adapted such that they form an optimal basis for a process mining algorithm (see the left of Figure 7.2) which secondly generates a process model from the pre-processed log files (see the middle of Figure 7.2). Many techniques have been developed for both these phases. Hence, we will in the following highlight those aspects for each of the two phases that are especially relevant when the mining should lead to a process model that covers multiple systems, i.e. when mining a basic process model.

7.1.1 Pre-processing the Log Files

While almost all IT systems record the executed behavior, there is no common standard for the logging of information. Thus, each application usually uses its own format. To provide a common format focused on the information required by process mining algorithms, the **Mining XML (MXML)** format has been

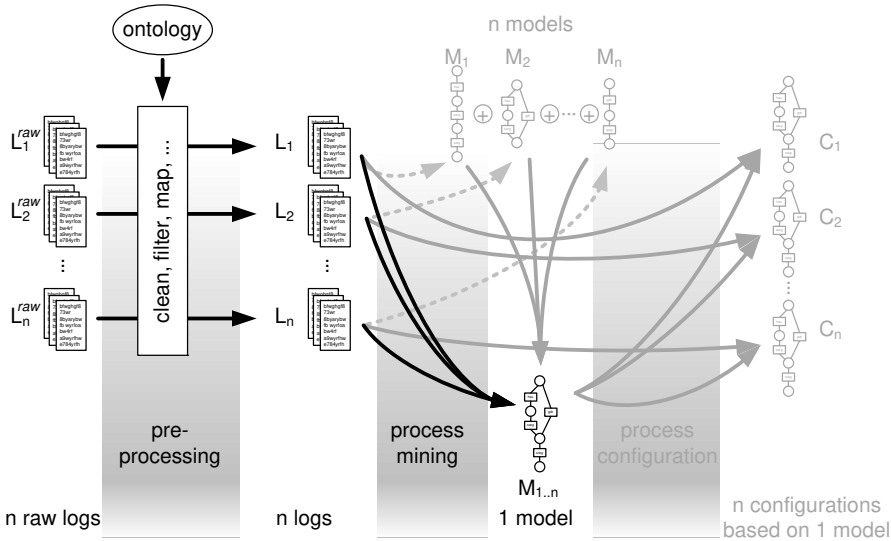


Figure 7.2: After pre-processing the log files, process mining generates the basic process model.

defined [83]. By converting the output files of IT systems into this format, all mining algorithms that use MXML files as input can be applied to these log files. Knowing the information available in the log files, this conversion can be automated. For major process aware information systems this conversion has been implemented in the **ProM import** framework¹. Hence, we assume in the following that the log files we want to use are available in the MXML format.

As an example, Figure 7.3 shows an extract from an MXML log file. It shows a workflow log which consists of logs of the *travel request processing* process we showed when introducing Protos in Figure 2.8 (p. 33). The log of the process contains recorded information on the process instances, i.e. executions, of this process in the form of a number of audit trail entries. Each audit trail entry incorporates the saved information about a particular task execution. This should include a name of the executed task, saved as `<WorkflowModelElement>` and information on what type of event the audit trail entry represents, e.g. if the audit trail entry was saved when the particular task was started or when it was completed. Furthermore, an audit trail entry can include information on when and by whom the particular task was executed, as well as further data that has been relevant to the particular task's execution and thus logged. In this way, we can see in the example in Figure 7.3 that Michael Smith started requesting quotes for a travel he wanted to do on the 27th October 2008 at 13.45h (lines 5–10). Following up on this, the secretary, Tom Brown, started preparing the travel form two days later at 10:07 in the morning (lines 11–16).

¹available via promimport.sf.net

```

1 <WorkflowLog ...>
2   <Process id="TRAVEL" description="travel request processing">
3     ...
4     <ProcessInstance id="154">
5       <AuditTrailEntry>
6         <WorkflowModelElement>Request Quotes</WorkflowModelElement>
7         <EventType >start</EventType>
8         <Timestamp>2008-10-27T13:45:08.000+01:00</Timestamp>
9         <Originator>Michael Smith</Originator>
10      </AuditTrailEntry>
11      <AuditTrailEntry>
12        <WorkflowModelElement>Prepare Travel form - Secretary</WorkflowModelElement>
13        <EventType >start</EventType>
14        <Timestamp>2008-10-29T10:07:50.000+01:00</Timestamp>
15        <Originator>Tom Brown</Originator>
16      </AuditTrailEntry>
17      <AuditTrailEntry>
18        <WorkflowModelElement>Check and Update Travel Form</WorkflowModelElement>
19        <EventType >start</EventType>
20        <Timestamp>2008-10-29T17:51:33.000+01:00</Timestamp>
21        <Originator>Michael Smith</Originator>
22      </AuditTrailEntry>
23      <AuditTrailEntry>
24        <WorkflowModelElement>Submit Travel form</WorkflowModelElement>
25        <EventType >start</EventType>
26        <Timestamp>2008-10-29T17:59:30.000+01:00</Timestamp>
27        <Originator>Michael Smith</Originator>
28      </AuditTrailEntry>
29      ...
30    </ProcessInstance>
31    <ProcessInstance id="155">
32      ...
33      <AuditTrailEntry>
34        <WorkflowModelElement>Submit Travel form</WorkflowModelElement>
35        <EventType >start</EventType>
36        <Timestamp>2008-10-28T16:21:12.000+01:00</Timestamp>
37        <Originator>Tina Williams</Originator>
38      </AuditTrailEntry>
39      <AuditTrailEntry>
40        <WorkflowModelElement>Decision Making</WorkflowModelElement>
41        <EventType >start</EventType>
42        <Timestamp>2008-10-28T17:20:08.000+01:00</Timestamp>
43        <Originator>Martin Thomas</Originator>
44      </AuditTrailEntry>
45      <AuditTrailEntry>
46        <WorkflowModelElement>Check and Update Travel Form</WorkflowModelElement>
47        <EventType >start</EventType>
48        <Timestamp>2008-10-30T09:55:54.000+01:00</Timestamp>
49        <Originator>Tina Williams</Originator>
50      </AuditTrailEntry>
51      <AuditTrailEntry>
52        <WorkflowModelElement>Decision Making</WorkflowModelElement>
53        <EventType >start</EventType>
54        <Timestamp>2008-10-30T13:48:25.000+01:00</Timestamp>
55        <Originator>Martin Thomas</Originator>
56      </AuditTrailEntry>
57      <AuditTrailEntry>
58        <WorkflowModelElement>End Process</WorkflowModelElement>
59        <EventType >start</EventType>
60        <Timestamp>2008-10-30T14:02:40.000+01:00</Timestamp>
61        <Originator>automatic</Originator>
62      </AuditTrailEntry>
63    </ProcessInstance>
64    ...
65  </Process>
66 </WorkflowLog>

```

Figure 7.3: A log file of the processing of travel requests in MXML.

In the afternoon of the same day, Michael checked this travel form (lines 17–22) and submitted it (lines 23–28).

Although such log files are widely available today, the purpose of their creation and their level of details varies. For example, the transaction management of databases requires very explicit and detailed logs for being able to undo all changes completely automatically. However, sometimes log files just serve a programmer to debug a software or system. Then, the log entries are often rather unspecific messages temporarily introduced by the programmer to find errors in the code but never removed. In any case, the log files are rarely created for deriving process models, i.e. to support process mining. Thus, to discover meaningful behavioral patterns in these log files through process mining, the log files must be trimmed to a data basis that promises good process mining results. We call this phase **pre-processing** of the log files.

When aiming at the mining of configurable process models, four aspects are especially relevant in this data collection and pre-processing phase:

- For generating configurable process models, it is very important to gather log files from various systems executing the process in question. The selection of the data sources of course depends on the purpose of the model that should be created. If a model should represent configuration options of a software that is distributed internationally, various sites running successful implementations of the software in different countries would provide a good data basis. If a model should represent good examples for a certain process, various successful implementations of that process which might be supported by different applications can provide a nice fundament for the process mining. All in all, *the source of the used log files should widely cover the targeted scope and all aspects of the model which should be created.*
- At first, *the data in the log files has to be made anonymous.* Log files usually contain a lot of personal data. For example, in Figure 7.3 we can see exactly when Tom Brown was preparing the travel form. This information may be confidential and the usage of such data is in most countries strongly restricted through privacy rights and laws. As configurable process models target at their re-use by others, it is especially important that no personal information is retained in the model. After all, it would be only too interesting for competitors to see on whose processes and process executions the various configurations of the process model were based. Hence, the elimination of such personal information should take place before any data is processed.
- Secondly, *the level of detail of the log files has to be balanced among the different input log files* and adjusted to the level targeted for the resulting model by aggregating related log events. Otherwise, the level of detail in the generated process model will later on be highly inconsistent among different process branches. To reach this balanced level of details an ontology can, for example, be used. Then both single log events as well

as groups of log events can be mapped onto an agreed level of ontology classes.

- As the same ontological concept is hardly called in the same way by different sources, it must also be ensured that *log events from the different source log files are mapped onto each other*. The use of a common ontology for adjusting the level of details might already guarantee this. Otherwise, a direct matching of event names is also possible.

Further details on how to use ontologies in the context of process mining can, e.g., be found in the work of Alves de Medeiros et al. [26]. General information on pre-processing steps in the context of data mining can also be found in the work of Cabena et al. [40], Pyle [133], and Zhang et al. [193]. In their experience, the overall result of automated mining efforts — as also described in the remainder of this chapter — heavily depends on the quality of the pre-processed log files. Therefore, in data mining projects pre-processing comprises 60–80 percent of the whole processing effort. To get meaningful results in the context of process mining, these numbers are probably similar, if not even higher.

We focus here on deriving the control-flow among the different activities executed in a business process from such log files. For our discussion, it is thus sufficient if we consider a log file as being a set of event traces. Thus, we abstract in the following from the details of log files like execution times, data, or the originators of task executions. Each event trace is simply an ordered set of the log event identifiers. Each of these log event identifiers classifies a log event as the execution of the particular activity by using an unambiguous name for the executed activity, but it ignores any of the mentioned particularities of a specific task execution.

Definition 7.1 (Log file) $LOG \in \mathcal{B}(I^*)$ is a log file, i.e. a multi-set of event traces, such that:

- I is a set of log event identifiers,
- I^* is the set of all possible event traces, i.e. $\langle e_1, \dots, e_n \rangle \in I^*$,
- $events : I^* \rightarrow \mathcal{P}(I)$ is a function defined such that $events(\langle e_1, \dots, e_n \rangle) = \{e_1, \dots, e_n\}$ is the set of all log event identifiers in an event trace $\langle e_1, \dots, e_n \rangle$, and
- $\Gamma = \mathcal{B}(I^*)$ is the set of all such log files.

For example, if a set of log event identifiers I would contain the log events a , b , c , and d , i.e. $I = \{a, b, c, d\}$, a log file LOG could consist of several of the event traces $\langle a, b, c, d \rangle$, $\langle a, b, d \rangle$, $\langle a, c, d \rangle$, e.g. $LOG(\langle a, b, c, d \rangle) = 3$, $LOG(\langle a, b, d \rangle) = 12$, and $LOG(\langle a, c, d \rangle) = 5$.

Comparing this formal definition of a log file with an MXML log file, the log event identifiers $e \in I$ conform to `<WorkflowModelElement>` elements, and an event trace $\theta \in I^*$ conforms to a `<ProcessInstance>`.

Let us thus assume that a comprehensive set $\Gamma^{raw} = \{LOG_i^{raw} | 1 \leq i \leq n\}$ of n such raw input log files is available (see the far-left of Figure 7.2). The pre-processing of such a raw log file is then a function $prep : \Gamma \rightarrow \Gamma$ which performs

all mentioned pre-processing steps for a log file, including the re-naming of log events which belong to an identical ontology class. The result of the pre-processing is then a consistent set of log files $\Gamma^{prep} = \{prep(LOG) | LOG \in \Gamma^{raw}\}$ which we can use for the process mining of a configurable process model in the following.

7.1.2 Mining the Basic Process Model

Process mining techniques have been developed to help process analysts with determining the processes executed by organizations — either to document or to improve them. Process mining thus supports gaining objective insights into business processes which are already in place in organizations.

Although configurable process models are derived from the behavior of well-running systems, this does not imply that these processes are documented by models which are correctly describing the executed behavior. For example, by translating models from the SAP reference model [48], i.e. models which are supposed to be of ‘good’ quality, into executable process models, Mendling et al. [119] discovered that quite a number of these models contained errors which made these models non-executable. Hence, these models can hardly depict the behavior which is indeed successfully executed by the SAP system. Thus, process mining can also support the designer of a configurable process model if she has log files from successful process implementations available.

Process mining algorithms search for recurring patterns in the pre-processed log files, i.e. the execution traces, of the systems in question and generalize the overall process behavior as process models. Simplified², a process mining algorithm splits a log file into the event traces of individual cases, i.e. process instances. It then constructs the process model by analyzing and comparing the events in the traces. Each log event is mapped onto a corresponding task in the model. For each event that occurs in the event trace, the algorithm checks in the the so-far derived model if the corresponding task can be reached from the task corresponding to the preceding log event. Is this not the case, a choice is introduced after the task corresponding to the preceding log event, giving the opportunity to choose at run-time between the tasks that already followed the preceding task and this new task that according to the previously derived model did not follow the preceding task. The resulting process model will thus depict that when reaching the particular point of the process, the process flow can either continue as all the previous traces did or it can continue as this deviating event trace did. Figure 7.4 illustrates this. After having processed I_1 and I_2 of the log file, there is a choice between executing $ABCD$ and $AEFD$. In the third process instance, $AEGD$ is executed. Thus, the algorithm needs to add the choice between F and G after E has been executed.

²The description here provides a brief idea of what a process mining algorithm does. In practice, process mining is far more complex as the algorithms, e.g., have to take concurrency, incomplete logs, noise, or invisible tasks into consideration. For further details on these issues the reader is referred to the work of van der Aalst and Günther [6], Alves de Medeiros et al. [27], Günther and van der Aalst [84], and Weijters and van der Aalst [186].

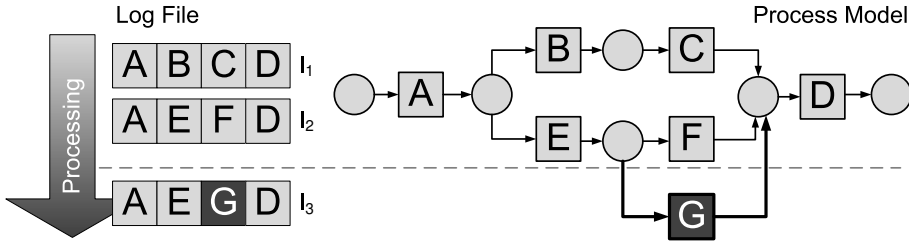


Figure 7.4: Process Mining adds choices according to the log file.

Abstracting from such particularities of the mining algorithms, we say:

Definition 7.2 (Mining algorithm) *A mining algorithm α maps a log file onto a workflow net, i.e.*

$$\alpha : \Gamma \rightarrow \Delta.$$

For each log file $LOG_i \in \Gamma$ that is used for the creation of a reference model, a process mining algorithm can therefore generate a process model $WF_i = \alpha(LOG_i)$ depicting the behavior of the log file's original system (see Figure 7.5a). Thus, α may be any process mining algorithm (e.g., see the work of van der Aalst and Günther [6], van der Aalst et al. [15], Alves de Medeiros et al. [27], Günther and van der Aalst [84], and Weijters and van der Aalst [186] for descriptions of a range of concrete algorithms). Furthermore, we assume that the result of the algorithm fulfills the requirements of a workflow net. This is trivial to achieve for any algorithm that provides a Petri net (or a model that can be transformed into a Petri net) by connecting a unique input place to all its initial elements and a unique output place from all its final elements [57].

Traditionally, a mining algorithm derives a process model for the log file of one system, i.e. if multiple log files are available, it can generate a process model for each of these systems (see Figure 7.5a). However, we aim here on one process model that covers multiple systems, i.e. that is valid for all the log files.

Still, process mining algorithms can also be used to directly generate an integrated model valid for all available log files, i.e. for all the log files in Γ^{prep} . If we concatenate all the log files $LOG_i \in \Gamma^{prep}$ into a single log file $LOG_{1..n} = \bigcup_{i=1..n} LOG_i$, the process mining algorithm α still works in exactly the same way on $LOG_{1..n}$ as it did for each of the individual log files. Due to the alignment of event names in the pre-processing, the algorithm is able to recognize which log events belong to the same class of events and match them. Thus, the algorithm just processes more process instances and creates a process model $WF_{1..n} = \alpha(LOG_{1..n})$ that is valid for all these instances. Figure 7.5b demonstrates this. Assuming all process instances in a first log file are $ABCD$, the process model after completing the processing of the first log file is the simple chain of these log events. The second log file of a different system also contains process instances $ABCD$, but also process instances $AEFD$. When the mining algorithm now discovers these new instances it just adds this new behavior to

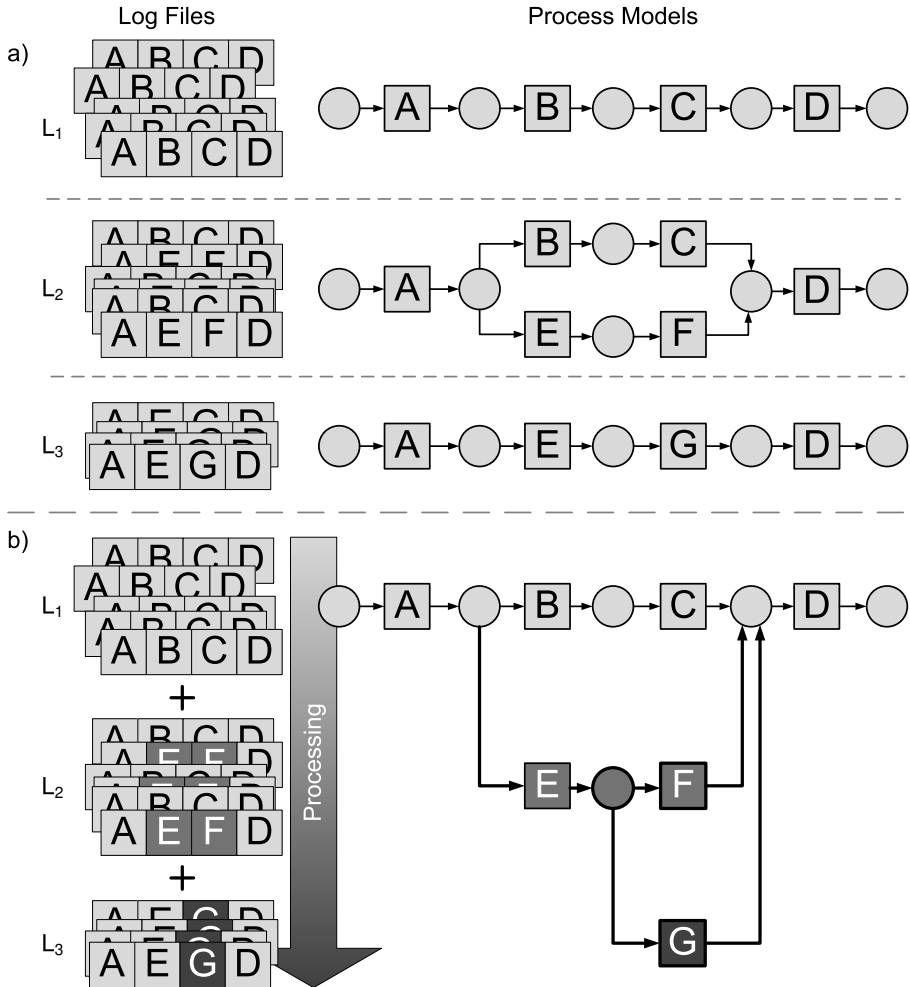


Figure 7.5: A mining algorithm can (a) generate a process model for each system by processing the particular log file. If multiple log files are (b) concatenated, the algorithm processes them in the same way as a single log file (compare figures 7.4).

the model as it did when adding new behavior which was contained in the same log file (compare figures 7.4 and 7.5b). A third log file contains only process instances *AEGD*. Hence, again a choice is added after the execution of *E* to permit the execution of *G* instead of *F*.

Two capabilities are important when selecting a process mining algorithm for building a process model integrating various process variants, namely the introduction of silent tasks and over-approximation:

- Among different systems it is well possible that steps executed in the process of one system are skipped in the other system. In such cases,

the process mining algorithm must be able to introduce a bypass for the skipped tasks in the generated process model, e.g. through adding invisible or silent tasks as an alternative to the skipped tasks. The invisible tasks then allow for state changes without corresponding to any log events and thus without representing any ‘real’ behavior. Thus, the algorithm has to integrate the choice between executing the task or configuring it as hidden as we have discussed in Chapter 3 into the run-time behavior of the model. The process model in Figure 7.6 was, e.g., mined from the log file shown in Figure 7.3 (p. 147) using an algorithm that supports the creation of such bypasses: after the decision making has taken place some process instances caused the execution of the *Drop Travel Request* transition before the process execution ends, while for others the process execution ends directly after the decision making. Thus, the algorithm introduces a silent transition which allows for the skipping of the *Drop Travel Request* transition.

- When a process model is configured later on, it might be desired that the derived process model does not conform exactly to the behavior of one of the systems that was used for the development of the configurable process model. Instead, it might be necessary to combine various aspects of different systems. This requires that the used process mining algorithm over-approximates the behavior of the input systems. Most process mining algorithms achieve this as they analyze choices between events only locally and neglect dependencies between choices that do not directly follow each other. By neglecting such non-local dependencies, the resulting process models permit for example that users can chose in the beginning of the process a process part that only occurred in a subset of the process instances, while at a later stage a choice is made for a process part that was not part of any of these process instances.

Figure 7.8 shows a process model which was mined from a log file that combined the log shown in Figure 7.3 (and which was also used to mine the process model shown in Figure 7.6) with the log file that was used to mine the model shown in Figure 7.7. The algorithm used to mine all these models is the **Multi-phase Miner** as suggested by van Dongen and van der Aalst [56, 57] and implemented in the **ProM** process mining framework³. It neglects non-local dependencies as can be seen when looking at the transitions *Request Quotes*, *Decision Making*, and *Check Travel Form*. Both the transitions *Request Quotes* and *Decision Making* are based on log events occurring in the log file from Figure 7.3, but not in the other log file. Thus, looking at the individual models for each of the log files in figures 7.6 and 7.7, the transition *Request Quotes* is always later on followed by the transition *Decision Making* (Figure 7.6), but it is never followed by the transition *Check Travel Form*. This transition only occurs in Figure 7.7 which on the other hand does not include the transition *Request Quotes*. In the combined model of Figure 7.8 it is then well possible that the transition *Request Quotes* is later on followed by the transition *Check Travel Form*. Thus,

³available via <http://prom.sf.net>, also see van der Aalst et al. [15]

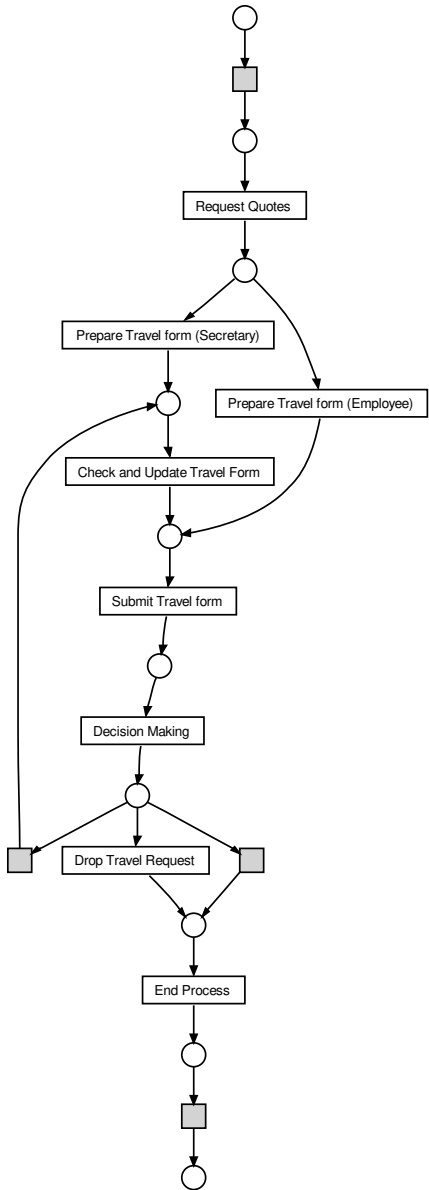


Figure 7.6: The mined model of the log file from Figure 7.3. Note the two silent tasks: one to loop back and the other to skip *Drop Travel Request*.

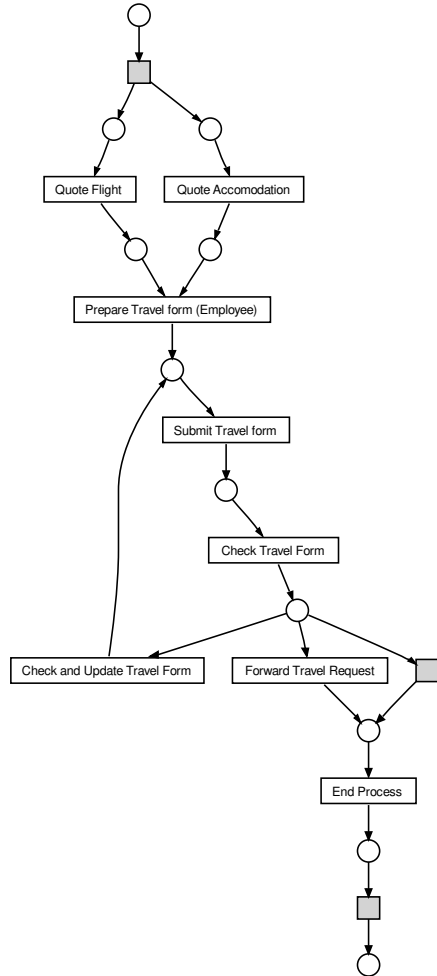


Figure 7.7: The mined model of a log file generated from the BPMN process model shown in Figure 2.9.

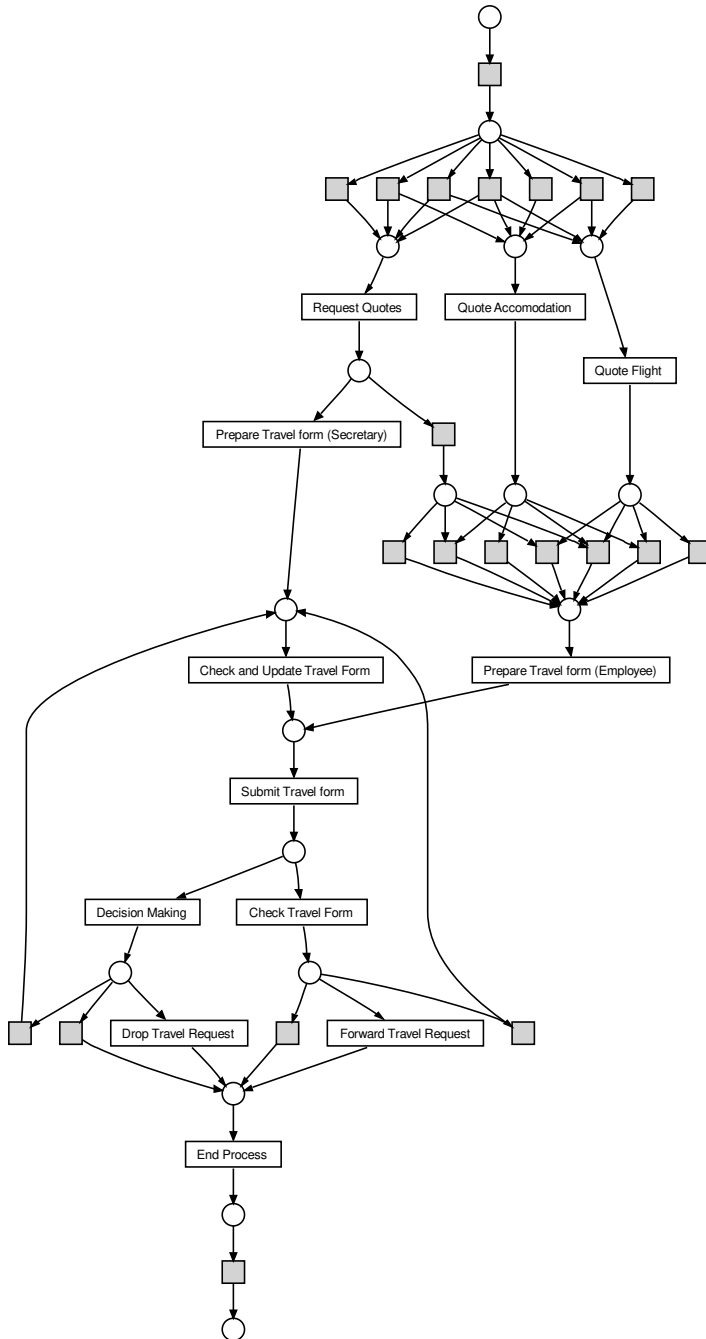


Figure 7.8: Using the multi-phase miner [56, 57], the model was constructed from a combined log file of the one of Figure 7.3 and the one generated from the BPMN process model shown in Figure 2.9.

the non-local choice that the transition *Request Quotes* must be followed by the transition *Decision Making* is ignored and new combinations of process parts from different sources become possible.

Note that the model resulting from mining the log files of multiple systems usually contains more choices than any model M_i that is derived for a log file L_i of a single system. This is because a combined set of event traces normally contains more process variants than a subset of these traces. For example, the model derived from the log file L_1 in Figure 7.5 contains only one choice after executing A while a model derived from L_2 only would contain no choices at all. The combined model then contains two choices, i.e. the one after A plus the one after E . The second choice conforms in fact to the choice between the system of the first log file and the system of the second log file. Thus, when aiming at introducing configuration choices into a process model, we obviously aim at introducing such generalizations, i.e. additional choices that allow for behavior not necessarily seen in one the logs. For realistic models, this effect usually increases, which already comes apparent if we compare the model in Figure 7.8 with the ones mined on a subset of the combined log file and shown in figures 7.6 and 7.7.

For our example, we used the multi-phase miner [56, 57]. In controlled experiences with high-quality input data, it provides good results because it guarantees the fitness of all the event traces to the resulting model. In practice, the choice for a concrete algorithm and the quality of the resulting model very much depends on the input log files [44, 84, 147]. For example, larger, real-world data usually contains a lot of **noise**, i.e. incomplete or wrong information. Even the best pre-processing thus hardly delivers the log file quality required by the multi-phase miner. Hence, in these cases other algorithms that are capable of dealing with noise might perform better. Readers interested in details on the concrete capabilities of various process mining algorithms should have a look at the corresponding publications [6, 15, 27, 56, 57, 84, 186]. An overview and comparison between various techniques can be found in the work of Alves de Medeiros [25] while the ProM framework provides implementations for many of these algorithms.

In any case, process mining usually does not provide a perfect model that can be used without any manual updates. This already becomes apparent when looking at the example of Figure 7.8. Here, the transitions *Check Travel Form* and *Decision Making* might in reality refer to the same task and should have been mapped onto each other already during the pre-processing of the log files. But due to the completely different names of the corresponding log events, this was not obvious as long as we did not see the process models mined from the log files. In addition, this mined model is not sound (see Definition 2.18, p. 24) as the silent transitions before the quoting allow the triggering of the request of quotes along with the quote of an accommodation or a flight while if the secretary prepares the travel form these execution branches are not synchronized later on in the model. These examples thus illustrate that current process mining algorithms are a tool for delivering ideas for the basic process model of a configurable model. However, most of the time they are not yet capable of

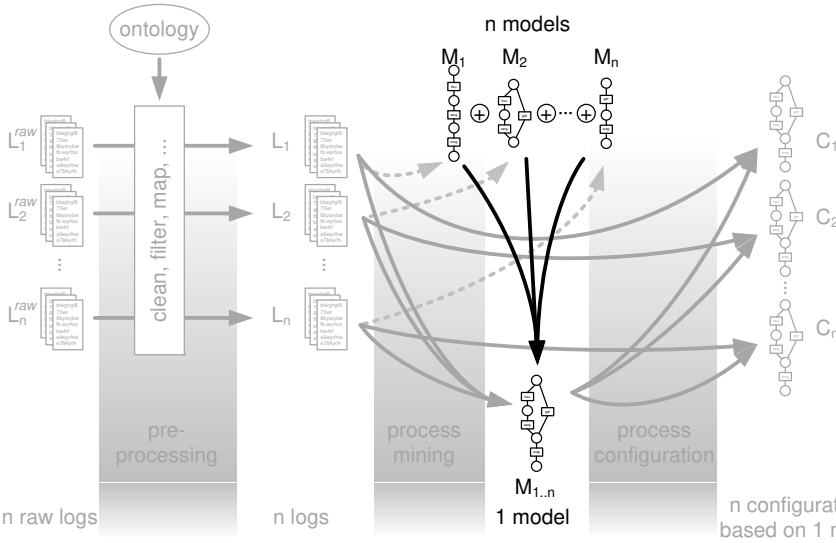


Figure 7.9: After pre-processing the log files, process mining generates the basic process model.

generating ‘perfect’ models. Thus, creating the integrated, basic process models as depicted in Figure 7.2 is still an interactive approach with manual efforts for trimming and improving the quality of both the log files and the generated models.

7.2 Merging Process Models

Often process models are already available for those process implementations that should serve as the basis for the development of a configurable process models. For example, such models were the basis for the case study in Chapter 6. If the models accurately document the process, manually (like in the case study) or even automatically aligning and merging these high-quality models can result in a far better basis for the basic process model than what can be produced by process mining. Have a look at Figure 7.9: during such a merge the various process models M_i with $i = 1..n$ which belong to and depict the behavior of individual systems (indicated by the dashed-arrows from the n individual system log files to the n individual models) are merged into a single process model $M_{1..n}^+ = M_1 \oplus M_2 \oplus \dots \oplus M_n$ representing the behavior of all the individual models, i.e. a basic process model for all these process variants.

While plenty of algorithms are suggested in literature to mine process models, algorithms that can automatically merge business process models are rare. On the one hand, this can be attributed to the lack of consistency among different models and thus the need to a pre-adjustment of process models before they can be merged similar to the pre-processing of log files. This makes a

quick application of such algorithms impossible. On the other hand, a concrete implementation of a merge of models depends on the purpose for which the models are merged. That means, such an implementation depends on which implications depicted by the different models should be preserved and which can be relaxed during the merge.

Here, we aim at a model that can serve as the basic process model within a configurable process model. Thus, our goal is to automatically merge multiple process models into a single process model such that the behavior that is possible according to the new process model is at least the behavior that was possible in each of the original process models. As the behavior of this process model can later on be restricted through process configuration, we do not mind if some additional behavior which was not possible in any of the original process models becomes possible in the new model. Quite the opposite: as explained in the previous section, we even strive for this, especially if this also leads to a more compact and clear model (i.e. less model elements and arcs). Therefore, the resulting model may allow for more behavior than the sum of the parts' behaviors. Hence, the merge algorithm should generalize.

In this section, we will discuss an algorithm explicitly developed for this context. The algorithm is based on a graph notation for business processes called **function graphs** which we will define specifically for merging process models. It is inspired by the multi-phase miner [56, 57] (which we already used to mine the examples of the previous section) as it combines all the behaviors that are possible according to the individual models and introduces choices whenever the behavior varies. To keep the models compact and clear, it relies on the concepts of OR-splits and OR-joins when there is not a clear XOR-split/AND-split or XOR-join/AND-join situation.

As the multi-phase miner which inspired this algorithm uses EPCs, we will depict the approach using EPCs. Any Petri net or Protos model can automatically be translated into a corresponding EPC, and an EPC can be translated back into a Petri net (see [57] for the corresponding algorithm) or a Protos model. Implementations of the corresponding transformation algorithms are, e.g., provided by ProM. Thus, if we want to stay in the Petri net or Protos domains, we can consider EPCs in the same way an intermediate format necessary for the algorithm as we do for function graphs. In principle, the algorithm is applicable to any process modeling notation that directly or indirectly supports the concepts of OR-splits and OR-joins. Thus, besides the direct implementation for EPCs depicted here, and the indirect application for Petri nets and Protos models, it can, e.g., also be applied when merging YAWL models.

The algorithm can be split up into three phases: the conversion of the process models that should be merged into function graphs (see Figure 7.10a to c, and b to d), the merge of the function graphs (see Figure 7.10c and d to e), and the conversion of the resulting function graph back into the used process modeling notation (see Figure 7.10e to f). After giving a definition for function graphs, we will use EPCs in the following to give precise definitions for the algorithm's three phases.

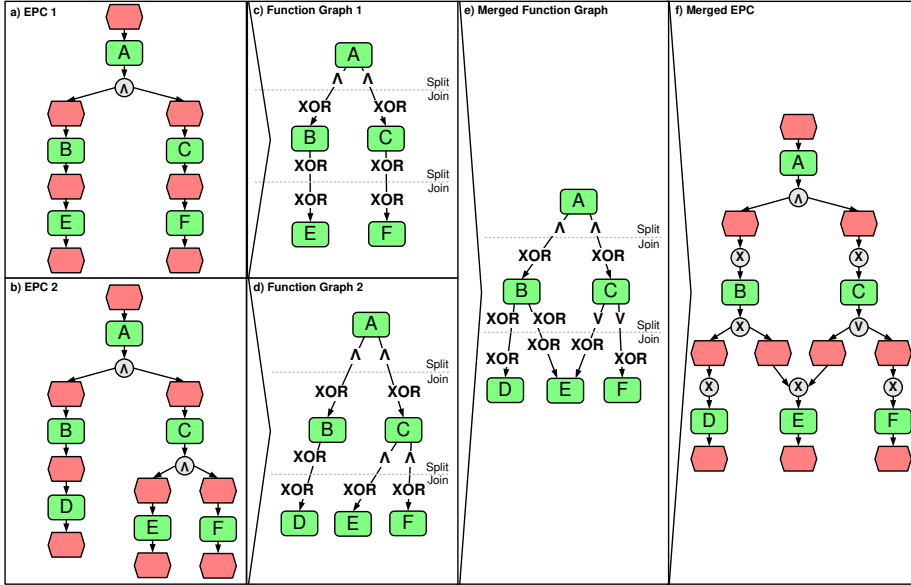


Figure 7.10: Merging two EPCs: The input EPCs (a,b) are transformed into function graphs (c,d) which can be merged. The resulting graph (e) is transformed back to an EPC (f).

7.2.1 Function Graphs

Function graphs reduce process models to tasks as the sole nodes of the model. Thus, all that is depicted in a function graph is the active behavior of the original process model. Like for EPCs, we call these task nodes **functions**. To represent the process behavior, functions are connected through directed arcs. The arcs depict in which order the functions can be executed. The concrete behavior depends on the marking of the arcs with tokens. To determine the marking of arcs during the process execution, a **split type** and a **join type** is assigned to each arc. Both types can have either the value \wedge , XOR , or \vee (see Figure 7.10c,d,e).

Definition 7.3 (Function graph) *A function graph is a four-tuple $FG = (F, A, l_{join}, l_{split})$ such that*

- F is a finite (non-empty) set of functions,
- $A \subseteq (F \times F)$ is a set of arcs,
- (F, A) is a graph⁴,
- $l_{join} \in A \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a join relation type such that $\forall_{f_1, f_2, f_3 \in F} ((f_1, f_2) \in A \wedge (f_3, f_2) \in A \wedge l_{join}(f_1, f_2) = \wedge) \Rightarrow (l_{join}(f_3, f_2) \neq XOR)$, and

⁴Hence, \bullet^x depicts the set of functions preceding a function x in FG and x^{\bullet} depicts the set of its succeeding functions (see Definition 2.11, p. 19).

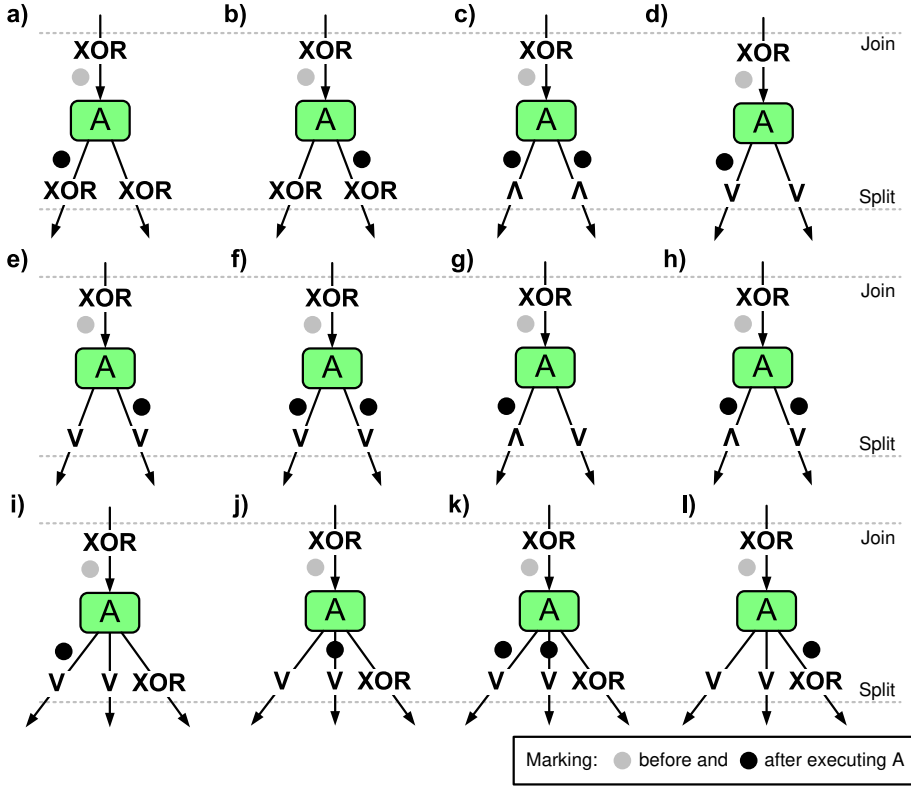


Figure 7.11: The marking of arcs after executing a function in a function graph.

- $l_{split} \in A \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a split relation type such that $\forall_{f_1, f_2, f_3 \in F} ((f_1, f_2) \in A \wedge (f_1, f_3) \in A \wedge l_{join}(f_1, f_2) = \wedge) \Rightarrow (l_{join}(f_1, f_3) \neq XOR)$.

Which arcs are marked after the execution of a function depends on the split type. If an arc has an \wedge split type, a token has to be added to this arc after the execution of its source function (see Figure 7.11c,g,h). In case of an XOR split type, a token can only be added to the arc if none of the other arcs having the same source function receives a token at the same time (see Figure 7.11a,b,i-1). Therefore, a function cannot be the source of arcs with split types XOR and \wedge at the same time. From a logical point of view, such type values would imply that the arc with the XOR type could never be marked as the arc of type \wedge must always be marked with a token. That means, the arcs of type XOR would never be followed and should thus be omitted.

The \vee type depicts that the marking of the arc is optional. However, if a function is the source of a set of arcs, at least one of these arcs must be marked with a new token after any of the function's executions (see Figure 7.11d-l).

For example, have a look at Figure 7.10e: Both arcs leaving function *A* are of split type \wedge . Thus, like in Figure 7.11c, both these arcs are marked after *A* has been executed. Both arcs leaving function *B* are of split type *XOR*. Thus, like in Figure 7.11a/b each of these arcs can only be marked exclusively without the other arc being marked. As one arc must always be marked this means that either the arc to *D* is marked or the arc to *E* is marked after *B* has been executed. Both arcs leaving function *C* are of split type \vee . Thus, at least one of them must be marked after *C* has been executed, but also both these arcs can be marked (like in Figure 7.11d–f).

The execution of a (non-initial) function then depends on the tokens on its incoming arcs. The join types of the arcs determine when their joined target function can be executed. In case an arc is assigned the join type \wedge , the target function needs to consume a token from that arc when it is executed, i.e. for the function's execution this arc must be marked with at least one token (see Figure 7.12c,g,h). An arc of type *XOR* implies that the target function can be executed if this arc is marked with a token, independent of the markings of the other incoming arcs. In this case this token is then also the only token consumed during the function's execution (see Figure 7.12a,b,i–l). Thus, a function can never be the target of an arc of type \wedge and of an arc of type *XOR* at the same time. In such a case, the type values would imply that the function must always consume a token from the arc of type \wedge and thus can never consume a token from the arc of type *XOR* exclusively. As this means that these tokens would never be consumed, it is not necessary to produce them and the arc should be omitted.

An \vee join type means that the consumption of a token from that arc is optional for the execution of its target function (see Figure 7.12d–l). How long the execution of a process waits for a token to arrive on such an optional arc depends on the OR-join semantics of the models that should be merged through the use of function graphs. Independent of the precise semantics, a function with at least one incoming arc can only be executed if it consumes at least one token from one of its incoming arcs.

7.2.2 From EPCs to Function Graphs

Let us now discuss how function graphs can be used to merge two process models, denoted as EPCs. However, note that the same algorithm can also be applied to the core elements of other languages like BPMN, YAWL, UML activity diagrams, etc. and that Petri nets can easily be transformed into corresponding EPCs.

The goal of the merge algorithm is the preservation of all the behavior depicted by the process models that should be merged. Thus, when transforming the input EPCs into function graphs, we have to ensure that the behavior that was possible according to the original EPCs remains possible according to the derived function graphs. Within an EPC, tasks, i.e. the elements depicting active behavior, are represented by functions. Therefore, we preserve the functions of an EPC as functions of the function graph. To preserve the process behav-

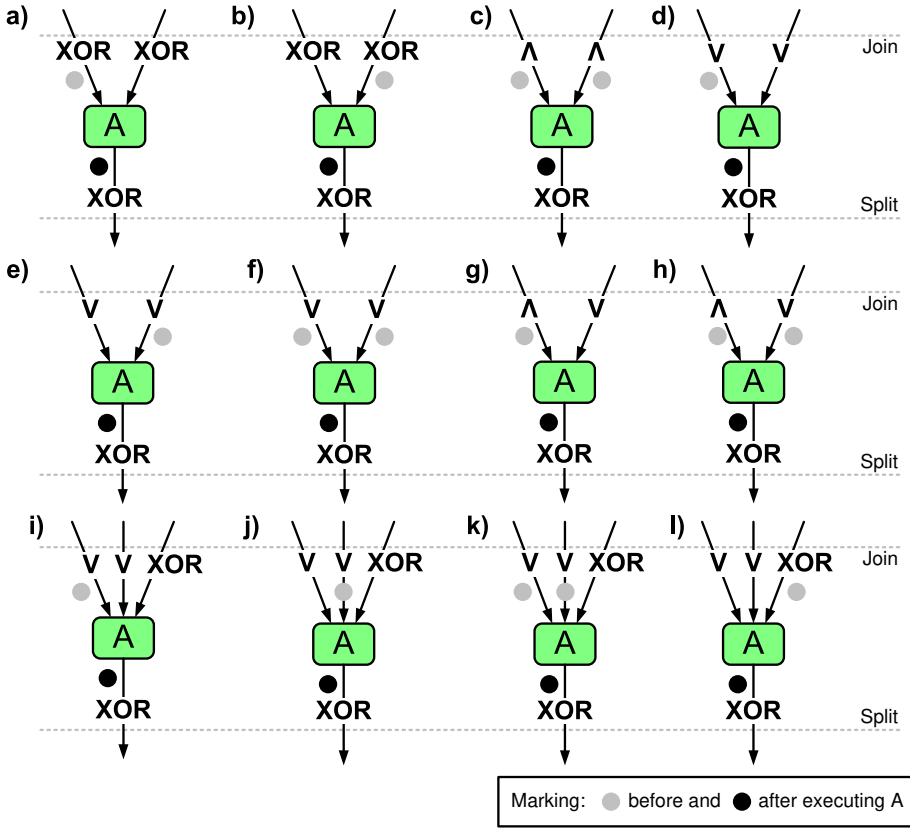


Figure 7.12: The marking of arcs necessary to execute a function in a function graph.

ior of the EPC, i.e. the order in which these functions can be executed, every function in the function graph is connected by arcs to exactly those functions which could be reached in the EPC through a path which does not pass any other function. The arcs' join and split types are afterwards calculated based on the EPC's split and synchronization behavior along these paths, i.e. based on the values of the join/split connectors. If no connectors exist along the path or if all these connectors are of type *XOR*, the corresponding arc is assigned the type *XOR*. If all the connectors are of type \wedge , the corresponding arc is assigned the type \wedge . Otherwise the value \vee is assigned to the arc.

Definition 7.4 (Function graph from EPC) Let $EPC = (E, F, X, m, A)$ be a well-formed EPC. Then $FG = (F^{FG}, A^{FG}, l_{join}^{FG}, l_{split}^{FG})$ is a function graph with

- $F^{FG} = F$,
- $A^{FG} = \{(x, y) \in (F \times F) \mid \exists z_1, \dots, z_n \in X \cup E \langle x, z_1, \dots, z_n, y \rangle \in A^*\}$,

- $l_{join}^{FG} \in A^{FG} \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a join relation type as follows:

$$l_{join}^{FG}((x, y)) = \begin{cases} XOR & \text{if } \forall_{n \geq 0} \forall_{z_1, \dots, z_n \in X \cup E} \langle x, z_1, \dots, z_n, y \rangle \in A^* \Rightarrow \\ & \quad \forall_{1 \leq i \leq n, z_i \in X_{join}} m(z_i) = XOR \\ \wedge & \text{if } \forall_{n \geq 1} \forall_{z_1, \dots, z_n \in X \cup E} \langle x, z_1, \dots, z_n, y \rangle \in A^* \Rightarrow \\ & \quad (\exists_{1 \leq i \leq n} z_i \in X_{join}) \wedge \\ & \quad (\forall_{1 \leq i \leq n, z_i \in X_{join}} m(z_i) = \wedge) \\ \vee & \text{otherwise,} \end{cases}$$

- $l_{split}^{FG} \in A^{FG} \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a split relation type as follows:

$$l_{split}^{FG}((x, y)) = \begin{cases} XOR & \text{if } \forall_{n \geq 0} \forall_{z_1, \dots, z_n \in X \cup E} \langle x, z_1, \dots, z_n, y \rangle \in A^* \Rightarrow \\ & \quad \forall_{1 \leq i \leq n, z_i \in X_{split}} m(z_i) = XOR \\ \wedge & \text{if } \forall_{n \geq 1} \forall_{z_1, \dots, z_n \in X \cup E} \langle x, z_1, \dots, z_n, y \rangle \in A^* \Rightarrow \\ & \quad (\exists_{1 \leq i \leq n} z_i \in X_{split}) \wedge \\ & \quad (\forall_{1 \leq i \leq n, z_i \in X_{split}} m(z_i) = \wedge) \\ \vee & \text{otherwise.} \end{cases}$$

The two examples in figures 7.13a and 7.13b show the relation between the splits of the control-flow in an EPC and in the function graph as well as the possible behaviors of these control-flow splits. In both figures, we analyze the behavior that can happen after a single execution of A . If all the split connectors between two functions of an EPC are of type \wedge , then the execution of the first function implies the execution of the second function as it is the case for functions A and D in Figure 7.13a. The \wedge split type of the corresponding arc in the function graph therefore requires the marking of the arc after the execution of A . As this arc is the only incoming arc of D , D requires only this token for its execution. Hence, D can be executed after A in the function graph as in the EPC. If all the split connectors between two functions of an EPC are of type XOR , then the execution of the first function implies the exclusive execution of the second function among all its subsequent functions. This is the case for functions A and D in Figure 7.13b. The corresponding function graph thus requires that after A 's execution the arc to D is marked exclusively as indicated by its XOR split type.

If the path between two functions of an EPC contains \vee connectors, or if it contains connectors of different types, then the succeeding function might or might not be triggered. As an example, in the model in Figure 7.13a either B or C can follow A (but always in combination with D). Within the function graph, this behavior is reproduced by assigning the split value \vee to the arc between A and B and also to the arc between A and C . In this way, it is optional for each of these arcs if it is marked after A 's execution. The function graph thus allows for executing B or C after A as the EPC. But in addition to marking one of the two arcs, the function graph also allows marking both or none of them as the marking of an arc in a function graph is independent from the marking of other arcs. Thus, in addition to the behavior of the EPC, behaviors where both B and C or none of them follow A are also possible in the function graph.

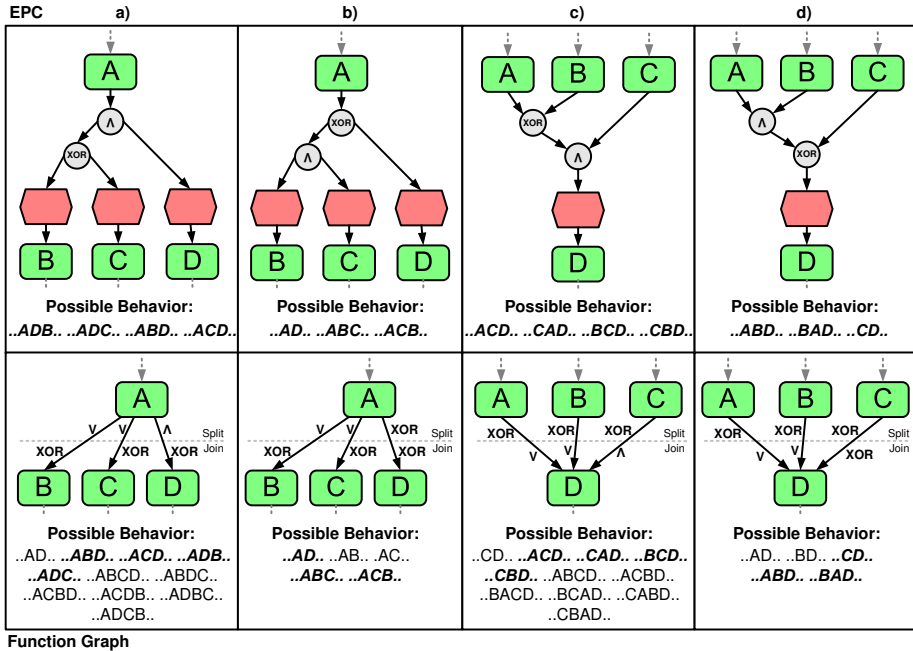


Figure 7.13: The relation between EPCs and function graphs (extracts)

Figures 7.13c and 7.13d show the corresponding behavior relation for join connectors of an EPC. Here we analyze the behavior that leads to a single execution of D . If all the connectors join the control-flow between two functions are of type \wedge , then the first function always has to precede the second one, as e.g. in the case of functions C and D in Figure 7.13c. The corresponding arc's \wedge join type in the function graph also ensures that D can only be executed after C has been executed and produced a token on this arc. In case all join connectors between two functions in an EPC are of type XOR , the first function always precedes the second one exclusively. Functions C and D in Figure 7.13d provide an example for this. In the function graph an arc's XOR join type implies exactly this behavior. The target function requires and exclusively consumes a token from that arc. In case different join connectors exist between two functions in an EPC (or if there are only \vee connectors), there are several combinations of preceding function executions possible before the execution of the succeeding function. The join type \vee in the corresponding function graph thus specifies that the consumption of tokens from each corresponding arc is optional for the execution of the target function. However, although this covers the behavior of the EPC, it might also allow for executing the target function in cases for which this was not allowed in the EPC.

All in all, whenever the split or join type of an arc in the function graph is XOR or \wedge , the behavioral relation between those two functions corresponds to the one in the EPC. When the split/join type is set to \vee the behavior of

the EPC is also covered by the function graph. Then, however, also additional behavior might be possible as we saw in Figure 7.13.

7.2.3 Merging Function Graphs

Two function graphs can be merged by *merging the sets of functions, merging the sets of arcs, and calculating the split and join types of the arcs* based on the values in the two function graphs.

To do this, we add a unique initial, i.e. first executed, function τ_I to each function graph. This function is connected to all initial functions of the particular function graphs via arcs with split type \vee and join type *XOR*. In the same way, a unique final, i.e. last executed, function τ_O is added to each function graph. All originally final functions of the particular function graph are connected via an arc with split type *XOR* and join type \vee to this new final function.

Adding such unique start and end functions is essential for preserving the characteristics of initial and final functions. If an initial function of a process is preceded by other functions in another process (like function *B* in the function graphs FG_1 and FG_2 of Figure 7.14a), the simple merge of the sets of arcs of the two function graphs would otherwise add the incoming arc to the function in the merged model (see FG_3 of Figure 7.14a). According to the semantics of the function graph, this implies that the function can only be triggered if the incoming arcs are marked with tokens. Thus, the function would lose its initial character which it had in one of the models. In the same way, final functions might become incorporated into a further process flow. This then prevents that the process execution can complete with this function (see function *C* in Figure 7.14a). If all executions are, however, started through the function τ_I (Figure 7.14b), the initial functions remain the first functions that represent some real behavior as all executions start through the silent function. Then the arcs' \vee split types provide a choice which of the originally initial functions are used to really start the process execution. Similarly, all process executions are completed through τ_O which can be reached from any set of originally final functions. Hence, without adding new functionality, these silent functions guarantee the presence of unique entry and completion points of the process and thus preserve the behavioral requirements of initial or final functions (see FG_6 of Figure 7.14b).

For calculating the split types of arcs, it is needed to analyze if functions are succeeded by the same functions in both models:

- If an arc is of split type *XOR* in one of the two function graphs and does not exist in the other graph, or if it has type *XOR* in both graphs, then the arc to the target function is either marked exclusively or not marked (a non-existent arc cannot be marked). This corresponds to the behavior of an *XOR* split type which is thus assigned to the arc in the resulting function graph as well.

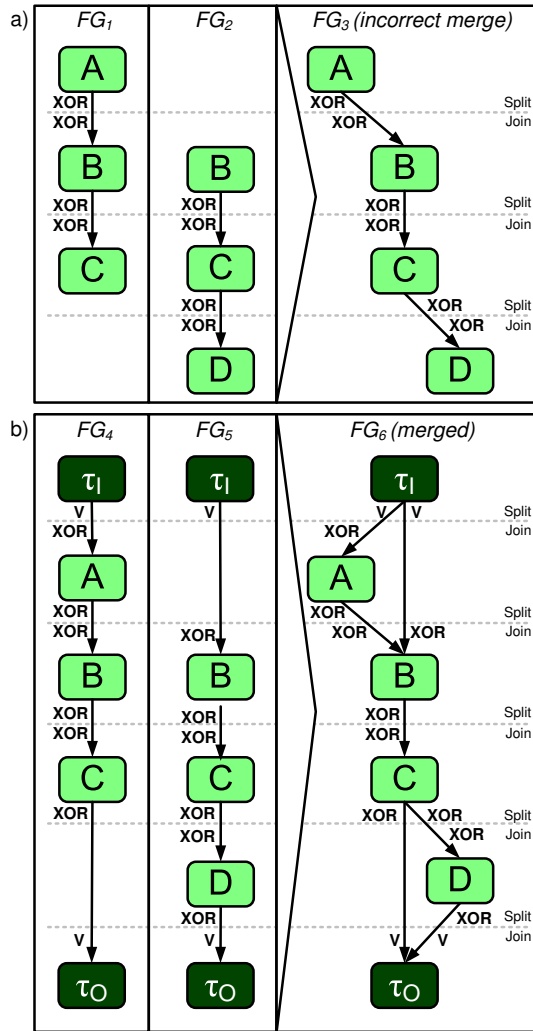


Figure 7.14: Initial and final functions must be unique among the models

- The split type \wedge is only assigned to an arc if it either is of split type \wedge in both input function graphs or if it has the value \wedge in one of the input function graphs and there is no outgoing arc at all from the arc's source function in the other function graph.
- In case there is an \wedge split type assigned to the arc in one model and the arc does not exist in the other model, but the source function of the arc has other successor functions in the other model, the \wedge value cannot be assigned to the resulting arc. It would imply that the arc must always be marked in the resulting model. This conflicts with the model without the

arc where a non-existent arc can obviously not be marked. Thus, in such cases the arc gets the more general split type \vee assigned as it gets in all other cases.

The join types of arcs are calculated in line with the split types. Thus, the corresponding line of argumentation also holds for the three join types of arcs:

- If an arc is of join type XOR in one function graph and does not exist in the other, or if it has type XOR in both graphs, the arc is assigned the join type XOR .
- The join type \wedge is assigned if the arc is either of join type \wedge in both function graphs or if it is of type \wedge in one of them and the arc's target function has no predecessors in the other function graph.
- In all other cases the arc is assigned join type \vee .

Definition 7.5 (Merging function graphs) *Two function graphs $FG^1 = (F^1, A^1, l_{join}^1, l_{split}^1)$, $FG^2 = (F^2, A^2, l_{join}^2, l_{split}^2)$ can be merged to a new function graph $FG^3 = (F^1 \cup F^2, A^1 \cup A^2, l_{join}^3, l_{split}^3)$ where:⁵*

- $l_{split}^3 \in (A^1 \cup A^2) \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a split relation type as follows:

$$l_{split}^3((x, y)) = \begin{cases} XOR & \text{if } ((l_{split}^1((x, y)) = XOR) \wedge (x, y) \notin A^2) \vee \\ & ((l_{split}^1((x, y)) = XOR) \wedge \\ & (l_{split}^2((x, y)) = XOR)) \vee \\ & ((x, y) \notin A^1 \wedge (l_{split}^2((x, y)) = XOR)), \\ \wedge & \text{if } ((l_{split}^1((x, y)) = \wedge) \wedge x^{FG^2} = \emptyset) \vee \\ & ((l_{split}^1((x, y)) = \wedge) \wedge (l_{split}^2((x, y)) = \wedge)) \vee \\ & (x^{FG^1} = \emptyset \wedge (l_{split}^2((x, y)) = \wedge)), \\ \vee & \text{otherwise,} \end{cases}$$

- $l_{join}^3 \in (A^1 \cup A^2) \rightarrow \{\wedge, XOR, \vee\}$ is a function which maps each arc onto a join relation type as follows:

$$l_{join}^3((x, y)) = \begin{cases} XOR & \text{if } ((l_{join}^1((x, y)) = XOR) \wedge (x, y) \notin A^2) \vee \\ & ((l_{join}^1((x, y)) = XOR) \wedge \\ & (l_{join}^2((x, y)) = XOR)) \vee \\ & ((x, y) \notin A^1 \wedge (l_{join}^2((x, y)) = XOR), \\ \wedge & \text{if } ((l_{join}^1((x, y)) = \wedge) \wedge y^{FG^2} = \emptyset) \vee \\ & ((l_{join}^1((x, y)) = \wedge) \wedge (l_{join}^2((x, y)) = \wedge)) \vee \\ & (y^{FG^1} = \emptyset \wedge (l_{join}^2((x, y)) = \wedge)), \\ \vee & \text{otherwise.} \end{cases}$$

Figure 7.15 shows an example for the arc values resulting from merging two function graphs. If an arc between two functions has the split type XOR assigned (as, e.g., the arc between A and C in Figure 7.15a), then this succeeding arc can be marked either exclusively or not at all after the execution of its source

⁵Note that if $x \notin FG^2$, then $x^{FG^2} = \emptyset$ and $y^{FG^2} = \emptyset$ (Definition 2.11, p. 19)

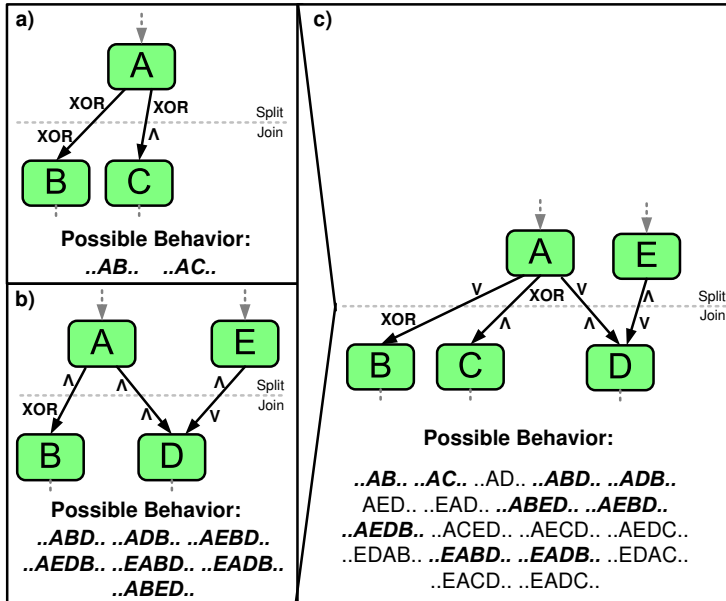


Figure 7.15: Merging two function graphs (extracts) and the behavior possible according to each of the function graphs such that no tokens remain on the arcs

function. This behavior must be preserved in the resulting function graph. As there is no arc from *A* to *C* in Figure 7.15b, this function graph only requires that the arc between *A* and *C* cannot be marked during execution which is covered by the opportunity not to mark the arc of the *XOR* split type. Thus, the corresponding arc of the resulting function graph gets the type *XOR* assigned. The same would hold if both input function graphs would require an *XOR* split type here as this would still cover all possibilities allowed by the two input graphs.

If, however, one model requires that the arc succeeding a function is exclusively marked after its execution (as the arc from *A* to *B* in Figure 7.15a) while the other model allows for a combined marking of the arc with other arcs through an \wedge or \vee split type (as in Figure 7.15b), then the arc is assigned the split type \vee in the resulting model (see the arc from *A* to *B* in Figure 7.15c). In this way, both combining the marking of this arc with other arcs (as required by Figure 7.15b) as well as its exclusive marking (as required by Figure 7.15a) remains possible as long as none of the other arcs leaving the first function is assigned the split type \wedge .

According to the definition of function graphs (Definition 7.3, p. 159), a single function that is the source of an arc with split type *XOR* cannot be the source of an arc with type \wedge in the same graph. However, it is well possible that a function like function *A* in Figure 7.15 is source of an arc of type *XOR* in one of the merged function graphs (e.g., the arc to *C* in Figure 7.15a) and in

the other graph it is the source of an arc non-existent in the first graph which is of type \wedge (the arc to D in Figure 7.15b). In this case, the function is the source of the arc between functions A and D which must be marked after A 's execution in the function graph of Figure 7.15b while this arc is not part of the graph in Figure 7.15a. Thus, the marking of the arc cannot be made obligatory in the resulting model in Figure 7.15c because the obligatory marking of the arc would then conflict with the behavior of the function graph in Figure 7.15a. Here, the marking of the arc between A and D was not necessary. Hence, in such cases the marking of the arc becomes optional by assigning it the \vee split type. In this way, it is also guaranteed that functions cannot be succeeded by arcs of split type \wedge and of split type XOR at the same time.

In case of the join type of the arc between A and D , there is no such function D in the function graph of Figure 7.15a and thus also no incoming arc to D . Hence, the only way in which D can be executed is depicted in Figure 7.15b where it always requires a token on the arc from A as specified by the arc's \wedge join type. Therefore, the corresponding arc in Figure 7.15c preserves the \wedge join type.

As the \vee split and join types are less restrictive than the \wedge and the XOR split/join types, using the \vee for an arc which had type \wedge or type XOR in one of the initial function graphs not only allows for the same behavior as in the original graphs. *It also allows for a number of additional behaviors that were not possible in any of the original graphs.* The behavior depicted in Figure 7.15 provides an example for this. The figure shows the behavior required and possible such that A is executed at least once and such that no tokens remain on the arcs of the function graph. The behavior possible in both initial graphs is shown in boldface while the new behavior is shown using non-boldface characters.

7.2.4 From Function Graphs to EPCs

A function graph can be transformed back into an EPC. For this, we first generate an EPC where each non-initial function is preceded by a dedicated join connector and each non-final function is succeeded by a dedicated split connector (see Figure 7.16b). Arcs connect these connectors in line with the arcs connecting the functions of the function graph via an event for each such arc. Start events are directly connected to initial functions while final functions are directly connected to end events. The connector types are calculated for each connector based on the values of all arcs originating or ending in the corresponding function in the function graph. If all arcs originating from a function in the function graph are of split type XOR , the split connector of this function in the EPC becomes an XOR connector (as for function B in Figure 7.16). If all arcs are of split type \wedge , it becomes an \wedge connector (as for A in Figure 7.16). In all other cases, it becomes a \vee connector. In the same way the join connector before each function in the EPC becomes an XOR connector if all arcs pointing at this function in the function graph are of join type XOR , it becomes an \wedge connector if the arcs are of join type \wedge , and it becomes a \vee connector otherwise (as for E in Figure 7.16).

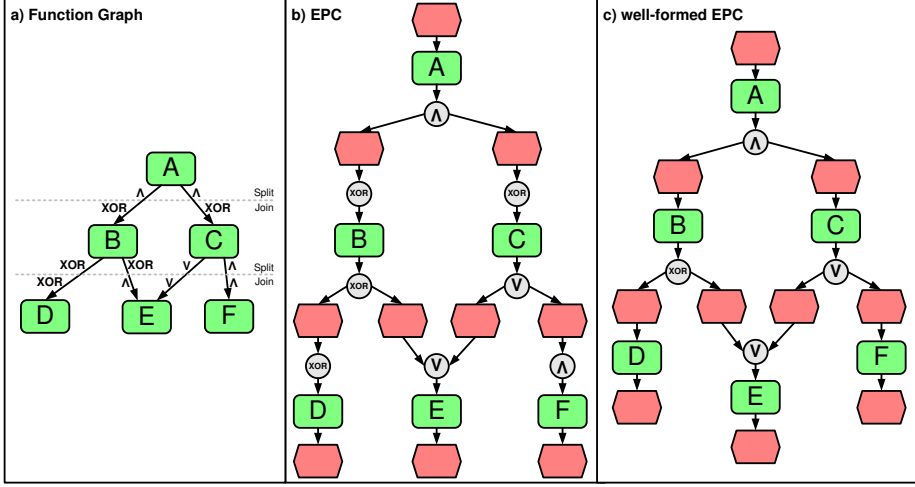


Figure 7.16: Transforming a function graph back into an EPC

Definition 7.6 (Generating EPCs from function graphs) A function graph $FG = (F, A^{FG}, l_{join}^{FG}, l_{split}^{FG})$ can be converted into an event-driven process chain $EPC = (E, F, X_{join}^{EPC} \cup X_{split}^{EPC}, m, A)$ as follows:

- $X_{join}^{EPC} = \{c_{join}^f | f \in F \wedge | \bullet f | \geq 1\}$ assigns a join connector to each function,
- $X_{split}^{EPC} = \{c_{split}^f | f \in F \wedge | f \bullet | \geq 1\}$ assigns a split connector to each function,
- $E = \{e^{(f_1, f_2)} | (f_1, f_2) \in A^{FG}\} \cup \{e_{start}^f | f \in F \wedge \bullet f = \emptyset\} \cup \{e_{end}^f | f \in F \wedge f \bullet = \emptyset\}$ is the set of events,
- $A = \{(f, c_{split}^f) | f \in F \wedge c_{split}^f \in X_{split}^{EPC}\} \cup \{(c_{join}^f, f) | f \in F \wedge c_{join}^f \in X_{join}^{EPC}\} \cup \{(c_{split}^{f_1}, e^{(f_1, f_2)}) | (f_1, f_2) \in A^{FG}\} \cup \{(e^{(f_1, f_2)}, c_{join}^{f_2}) | (f_1, f_2) \in A^{FG}\} \cup \{(e_{start}^f, f) | f \in F \wedge \bullet f = \emptyset\} \cup \{(f, e_{end}^f) | f \in F \wedge f \bullet = \emptyset\}$,
- for all $c_{join}^f \in X_{join}^{EPC}$: $m(c_{join}^f) = \begin{cases} XOR & \text{if } \forall_{x \in F} (x, f) \in A^{FG} \Rightarrow \\ & l_{join}^{FG}((x, f)) = XOR \\ \wedge & \text{if } \forall_{x \in F} (x, f) \in A^{FG} \Rightarrow \\ & l_{join}^{FG}((x, f)) = \wedge \\ \vee & \text{otherwise,} \end{cases}$
- for all $c_{split}^f \in X_{split}^{EPC}$: $m(c_{split}^f) = \begin{cases} XOR & \text{if } \forall_{x \in F} (f, x) \in A^{FG} \Rightarrow \\ & l_{split}^{FG}((f, x)) = XOR \\ \wedge & \text{if } \forall_{x \in F} (f, x) \in A^{FG} \Rightarrow \\ & l_{split}^{FG}((f, x)) = \wedge \\ \vee & \text{otherwise.} \end{cases}$

When transforming a function graph into an EPC, the connector values determining the possible behavior of the resulting EPC are calculated from the split

and join type values of the function graph's arcs. Only if each arc originating from a function is assigned the split type *XOR*, then also the corresponding EPC connector determining the successors of the function in the EPC is of type *XOR* allowing for an exclusive choice of one of the succeeding functions. This corresponds exactly to the behavior possible in the function graph. If all these arcs are of type \wedge , also the corresponding connector in the EPC becomes an \wedge connector. Again, the behavior of triggering all succeeding paths is in line with the behavior of the function graph. If the arcs are of type \vee or if there is a mixture of different split types among the arcs originating from a function of a function graph, then the corresponding EPC connector is assigned type \vee . The resulting behavior then corresponds to exactly the behavior if all the corresponding arcs of the function graph are of type \vee . Each other combination of arc types in the function graph allows for a subset of this behavior. Thus, by assigning the EPC connector the type \vee , additional behavior might become possible in the EPC compared to the behavior allowed in the function graph.

The argumentation for the behavior allowed by join connectors in the resulting EPCs is in line with the behavior of split connectors.

The EPC generated in this way might not be well-formed because it can violate the Requirement 7 of Definition 2.23 (p. 31): It may contain connectors which have only one incoming and at the same time only one outgoing arc (e.g., see functions *B*, *C*, *D*, and *F* in Figure 7.16b). However, such connectors can simply be eliminated from the net by replacing each of these connectors with a direct arc from its predecessor node to its successor node (see Figure 7.16c).

Definition 7.7 (Generating well-formed EPCs from function graphs)

If an EPC $^\diamond = (E^\diamond, F^\diamond, X^\diamond, m^\diamond, A^\diamond)$ was derived from a function graph, it can be converted into a well-formed EPC (E, F, X, m, A) as follows:

- $E = E^\diamond$,
- $F = F^\diamond$,
- $X = \{c \in X^\diamond \mid (|\bullet c| \geq 2) \vee (|c \bullet| \geq 2)\}$,
- $m \in X \rightarrow \{\wedge, \vee, XOR\}$ such that $m(c) = m^\diamond(c)$ for all $c \in X$,
- $N' = E^\diamond \cup F^\diamond \cup X$,
- $A = (A^\diamond \cap (N' \times N')) \cup \{(x, y) \in N' \times N' \mid \exists z_1, \dots, z_n \in (X^\diamond \setminus X) \langle x, z_1, \dots, z_n, y \rangle \in (A^\diamond)^*\}$.

Using the transformation algorithm of van Dongen and van der Aalst [57], the resulting EPC can also be translated back into a Petri net. OR-splits and OR-joins are then simply replaced by a set of transitions describing all possible combinations of paths that can be triggered respectively synchronized.

All in all, in each of the merge algorithm's three steps the possible behavior of the input model(s) is at least preserved in the resulting model, but usually even extended with additional behavior. This even holds if we only transform a single EPC into a function graph and back into an EPC without merging it with any other process model in-between. As explained earlier, for the creation of a configurable process model, this over-approximation of the behavior is a desired

property aimed at during the development of this algorithm. We prefer such generalized models over models that match the behavior of the input models exactly. For that reason, we also rather stick to the ‘simple’ EPC derived in the last transformation step although introducing additional connectors to create an EPC that preserves the behavior of the function graph more exactly would well be possible. Still, it must also be noted that in extreme cases, this over-approximation might lead to over-generalized models, e.g., if all connectors end up being V-connectors. Thus, if a merged model and thus the depicted algorithm is appropriate or not always depends on the individual context. In any case, the resulting model is a subclass of the original models. Hence, through process configuration the additionally allowed behavior can always be re-restricted to a desired amount of behavior.

In Section 7.1.2 we generated the basic process model of a configurable process model $M_{1..n}$ directly from log files through process mining. Originally, process mining techniques have been developed to generate individual models from individual log files as indicated through the dashed arrows from L_1 to M_1 , from L_2 to M_2 , etc. in Figure 7.1 (p. 144). Hence, in principle, it is also possible to first generate individual models through process mining and then use the model merging algorithm depicted in this section to generate a merged model $M_{1..n}^+ = M_1 \oplus M_2 \oplus \dots \oplus M_n$. Whether $M_{1..n}$ and $M_{1..n}^+$ depict exactly the same behavior as implied in Figure 7.1 depends on the generalization of the mining and merging algorithms. As the depicted merge algorithm is inspired by the multi-phase miner, we achieved an approximately identical behavior of the two models here by also using the multi-phase miner for the mining of process models. Both these techniques guarantee to preserve all the behavior represented in the log files and process models and over-approximate this behavior through using OR-joins and OR-splits whenever necessary.

7.2.5 Tool Support

We have implemented the merge algorithm as a plug-in of ProM which provides the necessary functionality to transform Petri nets to EPCs and vice versa, to import EPCs created with various software tools, to illustrate both Petri nets and EPCs, and to re-use EPCs with other data mining techniques. Figure 7.17 shows two EPCs for travel approval processes loaded in ProM, which we want to merge in the following for illustration purposes. The model on the upper right is the process from Figure 2.7 (p. 29) while the one on the left is a new process variant. In this new variant, reservations are made directly before the travel form is filled in and submitted. The travel is afterwards either approved and the form forwarded to the clearing center or rejected and the form is archived.

Before performing the actual merge of two selected EPCs as described in sections 7.2.2 to 7.2.4, the ProM plug-in allows users to create a mapping between the functions of the two input EPCs (see Figure 7.18) as well as between their events. In this way, it can be avoided that different names for the same functions or events (as e.g. defined through ontology classes or caused by typos etc.) cause superfluous additional elements in the resulting model. In the

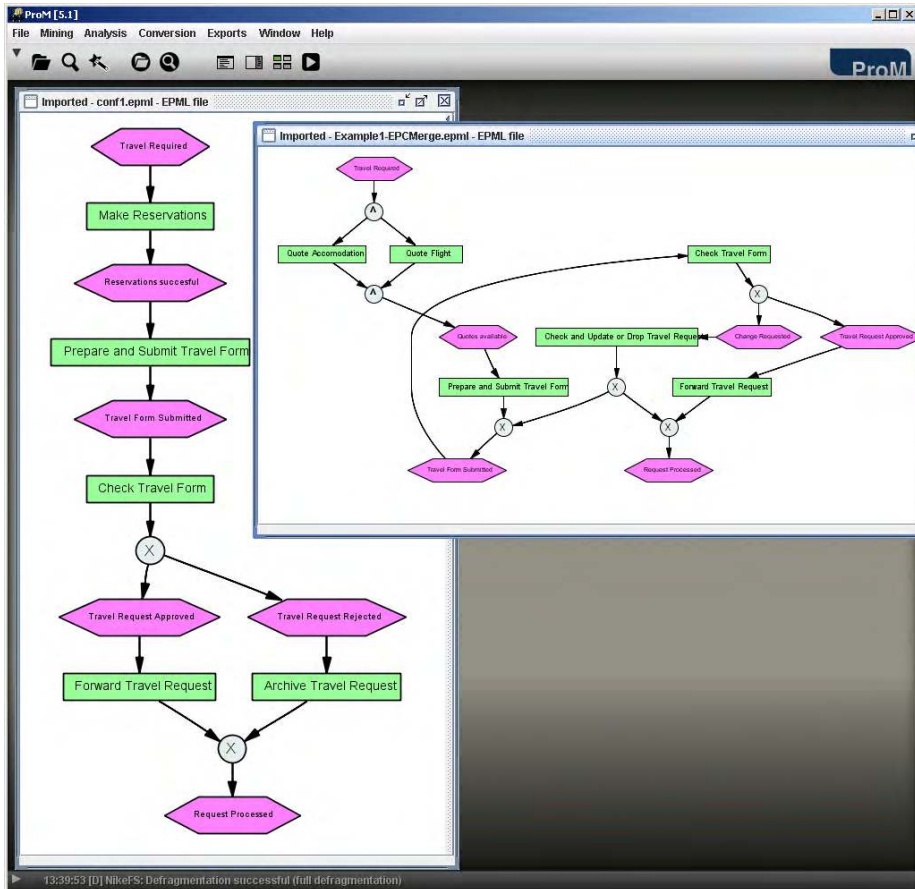


Figure 7.17: The two EPCs depict both a travel approval process. The goal is to merge them using the implementation of the EPC merge in ProM.

example, the functions of the EPC on the left in Figure 7.17 are listed on the left of the screen. In the middle, a corresponding function from the right EPC can be selected. To help the user with this mapping, the plug-in automatically suggests possible corresponding elements using a library provided by ProM for matching identifiers. The right column provides the opportunity to provide a name that will be used for the mapped functions in the resulting process model. In Figure 7.18, the functions *Prepare and Submit Travel Form*, *Check Travel Form*, and *Forward Travel Request* from the EPC on the left can in this way be mapped to equivalent functions of the EPC on the right. However, there are no corresponding elements for the functions *Archive Travel Request* and *Make Reservations*. Thus, we simply select *None* from the corresponding list of functions from the right EPC. Based on this mapping of functions and events

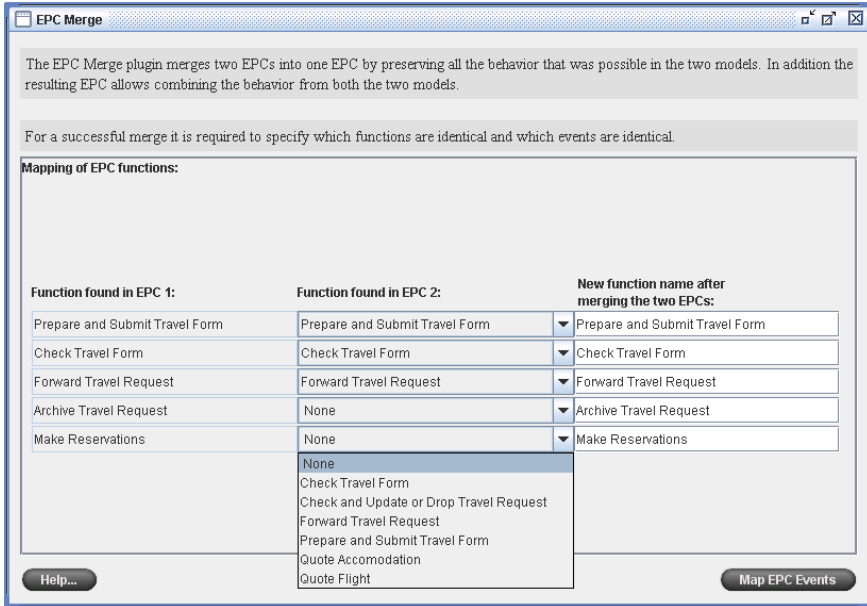


Figure 7.18: Before two EPCs can be merged, function and events names of both models have to be mapped onto a commonly agreed upon set of names.

between the two EPCs, the merge algorithm is then executed.⁶

As the behavior of an EPC is determined by the order of executing its functions, the names of the events between the functions are irrelevant from a behavioral point of view and thus ignored in the algorithm presented in sections 7.2.2 to 7.2.4. The algorithm simply creates new events when transforming the function graph back into an EPC. However, for the depiction and understanding of the process, the event names are of course important. Thus, after merging the two models, the implementation of the algorithm re-names each event of the resulting model based on the event names of the corresponding event(s) in the original models. Considering that the implementation introduces unique initial functions before each start event in the original model, and unique final functions after each end event of the original models, an event is always located on a path between two functions. Thus, for each event in the resulting model the algorithm can identify its preceding and succeeding functions. Then, it looks up these functions in the original models and identifies which events are located on the path between the two functions in the original models. If there is a unique event among all the original models, the event in the resulting model is named accordingly. If there are varying events on this path among the different EPCs, an *XOR* connector is added after the split connector of the preceding function in the resulting model. This connector splits the control-flow to all the different

⁶An illustrated description of how to use the EPC merge plug-in can be found at <http://www.floriangottschalk.de/epcmerge>.

events from the original models. Afterwards, another *XOR* connector re-joins the control-flow before the join connector of the succeeding function.

The EPC resulting from this merge is shown in Figure 7.19. In the integrated model, an \vee connector allows in the beginning for both the request of quotes for accommodation and flights as in the model on the right of Figure 7.17 as well as for making directly a reservation as in the model on the left of Figure 7.17. Due to the use of the \vee connector even solely quoting a flight or combining such a quote with a reservation of an accommodation would in contrast to the original models be possible according to this new model. While in each of the original models the check of the travel form could result in an acceptance of the travel request, the alternative to this was a rejection in one of the models, while it was the request for a change in the other model. Thus, the new, merged model in Figure 7.19 allows for a choice of one of these three options.

The algorithm, and thus also its implementation in ProM, is very efficient. All iterations over arcs to calculate the new arcs and connector values are limited to the paths in between two functions. In the transformation from EPCs to function graphs, the algorithm iterates over two trees per function, one for incoming paths from preceding functions to calculate the join type for the function graph, and one for outgoing paths to succeeding functions to calculate the split type of the corresponding arcs. Thus, the time complexity of this transformation is linear to the number of functions times the maximal number of arcs in these trees leading to predecessor and successor functions. As the number of a function's predecessor and successor functions is usually very limited compared to the overall number of functions in an EPC, and as this number usually remains constant even with an increasing number of functions in larger EPCs, i.e., the ratio even decreases, the time complexity of this transformation step can even be considered as linear to the number of functions for large models. The same argumentation holds for merging two function graphs. Again, the new split types and join types are calculated based on the incoming and outgoing arcs of each function. As their numbers remain limited compared to the overall number of functions in the merged model, also this can be performed in quasi linear time compared to the number of functions. Also, the transformation of function graphs back into EPCs implies just an iteration over the arcs of the merged function graph. Even determining the event names corresponds to determining the events along the paths between two functions in the original EPCs which has the same computational complexity as the determination of the split types and join types.

By implementing the merge of EPCs as a ProM plug-in, it can be used and combined with existing process mining techniques like the ones mentioned in Section 7.1 to create integrated process models from both models and log files. Moreover, ProM provides a range of conversion tools to and from EPCs. Thus, the algorithm is in this way also applicable to other notations like Protos or YAWL. Furthermore, as mentioned before, ProM plug-ins allow applying the algorithm to Petri nets although Petri nets do not support OR-joins and OR-splits directly. In the conversion from EPCs to Petri nets, OR-join and OR-split connectors are simply replaced by a range of silent transitions allowing

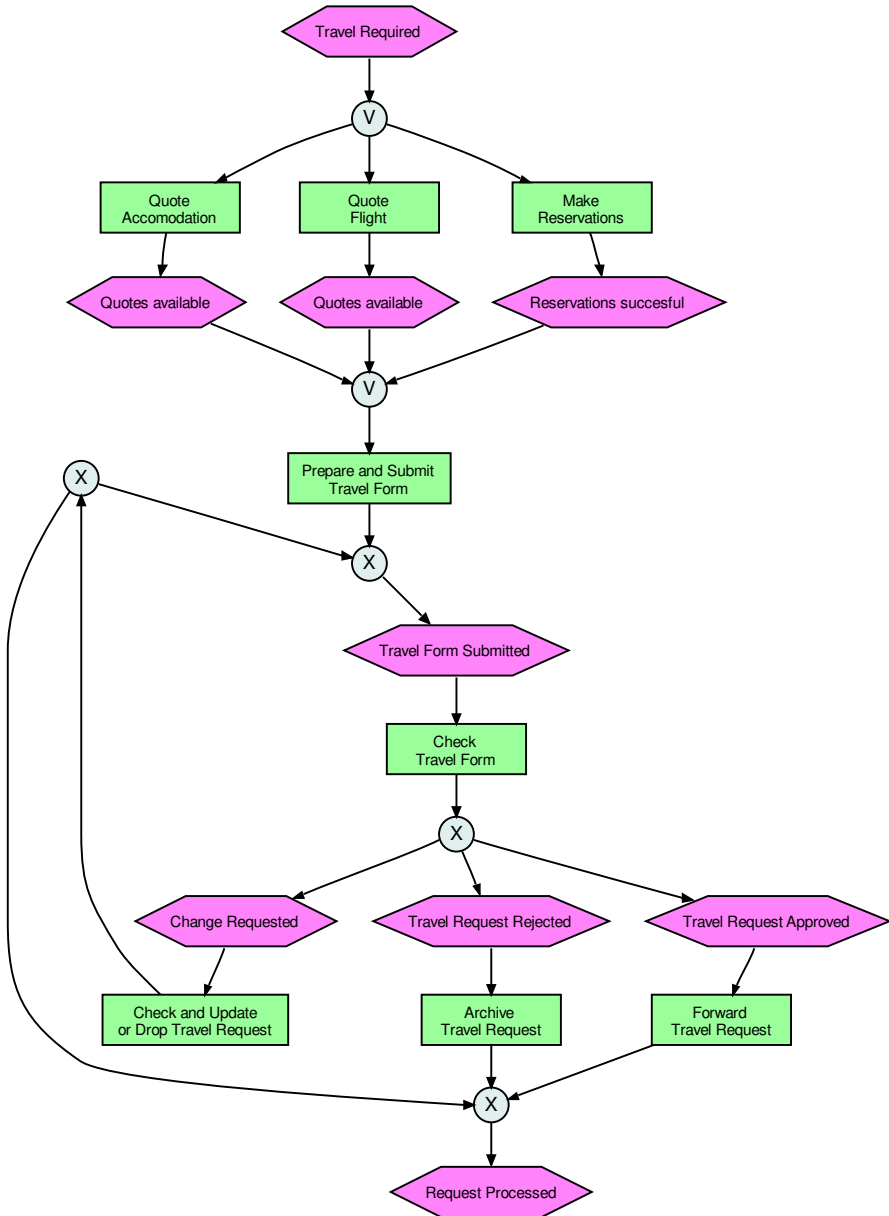


Figure 7.19: The EPC resulting from merging the two EPCs of Figure 7.17

the marking of each possible combination of subsequent places. This can be seen in the beginning of the Petri net in Figure 7.20 which shows the Petri net transformation of the EPC from Figure 7.19.

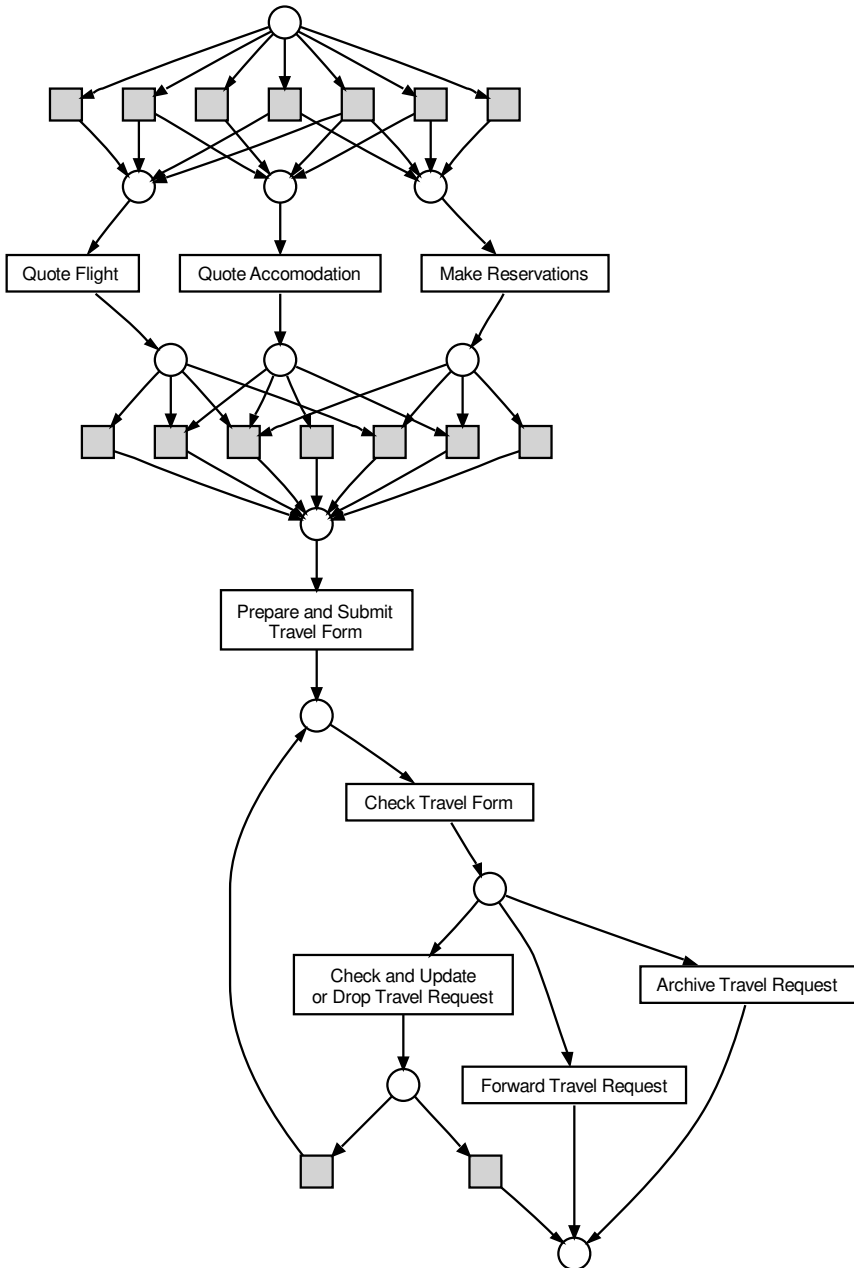


Figure 7.20: The merged EPC from Figure 7.19 transformed into a Petri net using ProM: V connectors are replaced by a set of transitions describing all possible subsets.

7.3 Deriving Configurations

The basic process model of a configurable process model — regardless whether it is generated through process mining, model merging, or manual construction — allows for the execution of all the process’s various variants. It covers all possible combinations of process parts, which is usually impossible to achieve by providing a set of individual process variants. But naturally, the model users do not need all these variations. Instead, they like to have a specific model covering exactly the required process behavior. Hence, the user of a configurable process model needs to configure the integrated model to that subset which depicts this desired behavior. That means, for a workflow net $WF = (P, T, A, L, l)$ we need to find a configuration $\mathcal{C} : T \rightarrow \{allow, hide, block\}$ such that the workflow net resulting from \mathcal{C} only depicts the behavior necessary for a particular application.

Examples for such configurations are the systems used to create the basic process model. These established variants of the process could thus provide a better starting point for the users of the configurable process model than the complete integrated model of all process variants can be. If we know the configurations of the basic process model leading to the selected, established process variants, and if we know which of these variants might probably be the closest to our requirements (e.g. because of a comparable company size and target market), then the derivation of an individual process would normally be limited to amending this given configuration by adding a number of elements from the basic process model to the configuration and/or removing some of them. In this way, the risky and time-consuming task of configuring the process from scratch can be avoided.

To determine such a configuration, we can (re-)use log files of the particular system. These log files contain information about all behavior that was used in the particular system. Thus, we can utilize this information for identifying which tasks of the basic process model are used in the particular system and which are not. That means that the configuration of the basic process model (and hence a configured process model) can be derived from the log files of successful process implementations (see Figure 7.21).

For this, we simply need to map the event identifiers from the log file onto tasks depicted in the basic process model. We assume here that all log events from the log file correspond to tasks in the basic process model as otherwise its configuration would obviously not be able to cover the behavior in the log file. If this is not the case, the integrated model must either be adapted to also include this additional behavior, this additional behavior must be eliminated from the log file in a pre-processing step by removing the non-covered log events, or all event traces with such additional behavior are simply discarded. This issue has, e.g., been addressed by Rozinat and van der Aalst [144] in detail and is thus of no further concern here. For simplicity, let us in the following just discard all event traces which are not covered by the basic process model.

While the mapping of log events to tasks already provides an indication about which visible tasks are required in the configuration, it alone does not provide details on the concrete configuration of the invisible and the non-required

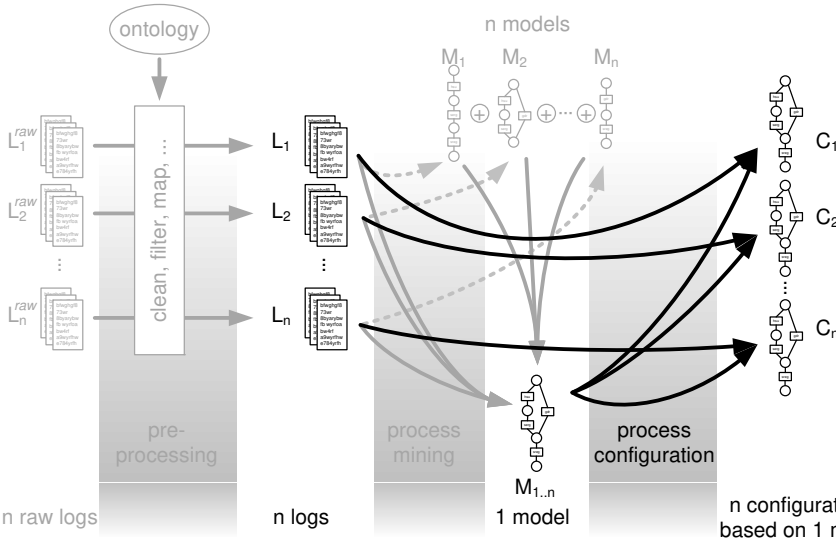


Figure 7.21: After pre-processing the log files, process mining generates the basic process model.

tasks of the model, i.e. must their use be configured as blocked or as hidden. To identify these configuration decisions, we need to ‘replay’ the log file on the basic process model. That means that we need to take each event trace from the log file as ‘a guide’ through the basic process model, while we execute the tasks listed in the trace within the basic process model. Considering a workflow net, this means that the event traces have to lead from its initial marking to its final marking.

Definition 7.8 (Complete Firing Sequences) Let WF be a workflow net, M_I be the initial marking of WF , and M_O be the final marking of WF . Then $\Phi_{WF}^{IO} = \{\sigma | M_I \xrightarrow{\sigma}_{WF} M_O\}$ is the set of all complete firing sequences of WF from M_I to M_O .

The log replay works as follows: Starting with the initial marking M_I for each trace, the log replay searches for a way to fire the transition which conforms to the first log event without firing any other visible transitions before that. If the initial marking does not enable the corresponding transition directly, the replay has to find a firing sequence of silent transitions that enables the firing of the transition conforming to the first log event. After the firing of this first transition, the replay continues with finding a firing sequence of silent transitions which enables the transition conforming to the second event in the log file and fires this one. This continues until the transition conforming to the last log event has been fired. If the marking of the workflow net resulting from the firing of this last transition is not yet the final marking M_O , then an additional firing sequence of silent transitions has to be found that leads to the final marking. For any event trace θ of the log file, θ must thus correspond to the identifiers

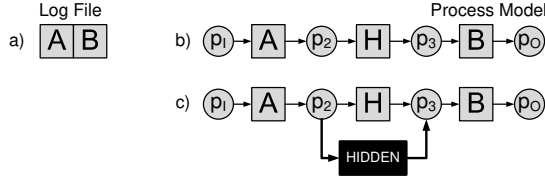


Figure 7.22: For replaying a log file, the potential of hidden transitions must be taken into account.

of a complete sequence of transition firings σ in the workflow net while ignoring all the firings of silent transitions in σ , i.e. $\theta = \pi_{L \setminus \{\tau\}}(l(\sigma))$.

By remembering all the transitions that we fired while replaying all the traces in a log file, we do not only know which visible transitions have to remain possible in the configured net, but also which silent transitions have to be preserved in the configuration. However, this replay fails if transitions of the basic process model are hidden in the configuration that leads to the replayed log file. For example, take the simple event trace $\langle A, B \rangle$ as shown in Figure 7.22a and the workflow net shown in Figure 7.22b. Obviously, transition H needs to be hidden during the execution of the workflow net to generate the event trace $\langle A, B \rangle$. However, if we replay $\langle A, B \rangle$ on the workflow net, we only reach a marking of p_2 . There is no sequence of silent transitions that could lead to the marking of p_3 and thus enable B . To be able to replay the log file, we therefore need to add a silent transition as an alternative to transition H , enabling the replay to bypass the firing of the transition that is configured as hidden (see Figure 7.22c).

To replay a log file, generated from a configuration of a basic process model on the basic process model, we hence have to introduce such bypasses for any visible transition, i.e. for all transitions which are not labeled τ .

Definition 7.9 (Expanded Workflow Net) *Let $WF = (P, T, A, L, l)$ be a workflow net. Then $WF^{exp} = (P, T \cup T_{hid}, A \cup A_{hid}, L \cup \{\tau\}, l^{exp})$ is the expanded workflow net of WF with*

- $T_{hid} = \{\tau_t | t \in T \wedge l(t) \neq \tau\}$ being a set of silent transitions of which each silent transition corresponds to a visible transition of WF ,
- $A_{hid} = \{(p, \tau_t) | p \in P \wedge \tau_t \in T_{hid} \wedge (p, t) \in A\} \cup \{(\tau_t, p) | \tau_t \in T_{hid} \wedge p \in P \wedge (t, p) \in A\}$ being the set of arcs incorporating the transitions $\tau_t \in T_{hid}$ into WF ,
- $l^{exp} : T \cup T_{hid} \rightarrow L \cup \{\tau\}$ being a function assigning labels to transitions such that $l^{exp}(t) = \begin{cases} l(t) & \text{if } t \in T \\ \tau & \text{otherwise (i.e. } t \in T_{hid}), \end{cases}$

By adding such a potential bypass to any visible transition in the workflow net for the replay, we can clearly identify and distinguish transitions that are allowed to be used, transitions that are blocked, and transitions that are hidden in the configuration that has led to the particular log file. All visible and silent transitions of the original workflow net, i.e. the net without the bypasses added

for the replay, which are used during the replay of all the traces of the log file are configured as being allowed to be used. If a transition's bypass is used during the log replay, but the transition itself is not used during the replay, the transition is configured as hidden. If both the transition and the bypass are used, the transition's use remains allowed. In this case, the original workflow net should already contain a silent transition as bypass to the transition to be able to produce the behavior that both the transition and its bypass are used. Hence, the use of the bypass already included in the original workflow net then remains allowed as well while this silent transition will be configured as blocked if all traces use the visible transition. Finally, if neither a transition nor its bypass is used during the replay, the transition must be configured as blocked.

Definition 7.10 (Configuration of Log) Let $WF = (P, T, A, L, l)$ be a workflow net, $WF^{exp} = (P, T \cup T_{hid}, A \cup A_{hid}, L \cup \{\tau\}, l^{exp})$ be the expanded workflow net of WF , and $\Phi_{WF^{exp}}^{IO}$ be the set of all complete firing sequences of WF . Moreover, let LOG be a log file such that $\bigcup_{\theta \in LOG} \text{events}(\theta) \subseteq \{l(t) \in L \mid t \in T \wedge l(t) \neq \tau\}$, i.e. all events of LOG correspond to a label of a visible transition of WF . Then

$$\Phi_{WF^{exp}}^{LOG} = \{\sigma \in \Phi_{WF^{exp}}^{IO} \mid \exists \theta \in LOG \theta = \pi_{L \setminus \{\tau\}}(l(\sigma))\}$$

is the set of all firing sequences that correspond to behavior depicted in the log file and

$$V = \bigcup_{\sigma \in \Phi_{WF^{exp}}^{LOG}} \{t \in T \cup T_{hid} \mid t \in \sigma\}$$

is the set of transitions of WF^{exp} that can fire when replaying LOG . The corresponding configuration of WF is $\mathcal{C} : T \rightarrow \{\text{allow}, \text{hide}, \text{block}\}$ such that

$$\mathcal{C}(t) = \begin{cases} \text{allow} & \text{if } t \in T \cap V \\ \text{hide} & \text{if } t \in T \setminus V \wedge \tau_t \in V \\ \text{block} & \text{otherwise.} \end{cases}$$

In Figure 7.23 we replayed a log file of a concrete travel approval process on the workflow net from Figure 7.8 (p. 155) and marked all the used, visible transitions with a bold border and all used, silent transitions as completely black. In addition, we also marked paths that were followed in bold, while all paths that are not followed are played down through dashed arcs. The visible transitions *Prepare Travel Form (Secretary)*, *Prepare Travel Form (Employee)*, *Submit Travel Form*, *Decision Making*, *Drop Travel Request*, *Check and Update Travel Form*, and *End Process* are used in the log and thus allowed in the corresponding configuration of the workflow net. *Prepare Travel Form (Secretary)* and *Prepare Travel Form (Employee)* are the initial transitions according to the log file. As both are preceded by other visible transitions in the original workflow net (*Quote Accommodation*, *Quote Flight*, and *Request Quotes*), the sequences of transition firings enabling these transitions have to bypass these preceding transitions through the added, silent transitions labeled *HIDDEN*. As the corresponding transitions are not used during the replay these transitions must be

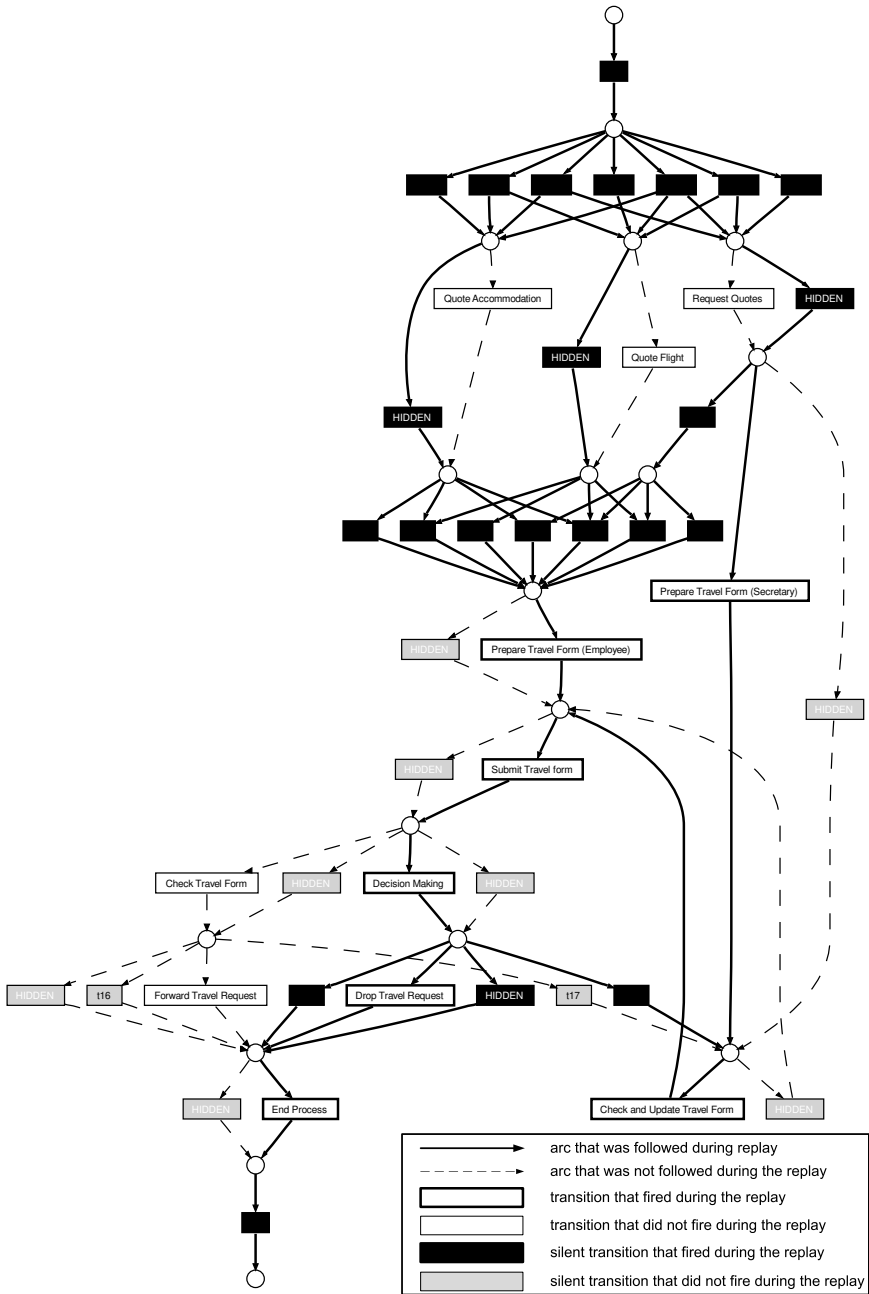


Figure 7.23: A log file replayed on the workflow net of Figure 7.8.

configured as hidden while all the silent transitions along these paths that belong to the original workflow net, remain configured as allowed. The transitions *Check Travel Form* and *Forward Travel Request*, as well as the silent transitions *t16* and *t17* are not used at all and thus configured as blocked. As a last remark, note that for the transition *Drop Travel Request* both the visible transition as well as the bypass is used. This is the case because the model already contained a bypass to this transition and traces from the log file can thus use either the bypass or the visible transition. Hence, both the silent transition of the original workflow net as well as the visible transition are configured as allowed and the fact that the bypass of the expanded workflow net was also used during the replay is irrelevant.

Using this configuration, the individual process model corresponding to this behavior can then be derived from the basic process model as depicted in Definition 3.5 (p. 56). The resulting workflow net is shown in Figure 7.24.

7.4 Case Study Re-visited

To test the practical applicability of the tools and algorithms discussed in this chapter, let us re-visit the processes from the case study described in Chapter 6. In this chapter we manually merged the Protos process models of the individual municipalities into an integrated basic process model, which (at first) was also created in Protos. This approach turned out to be cumbersome and error-prone. Both the mining of the basic process models if log files of various systems are available (discussed in Section 7.1) and the merging of existing process models (discussed in Section 7.2) aim at improving and automating the creation of the basic process model. Therefore, let us test these approaches with the models from the case study in this section. Afterwards, we will also show how the approach from Section 7.3 can be used to identify configurations for the configurable process models from the case study.

7.4.1 Mining Models from Log Files

In the case study we used individual models as input to create the basic process model, while the mining approach from Section 7.1 requires log files of the executed processes. Process simulation engines are able to generate log files from process models by simulating the execution of the particular processes. For verifying the applicability of process mining to generate configurable process models, simulated log files are better suited than log files taken directly from the computer systems of the municipalities: As the simulated log files conform exactly to the behavior we want to have in the resulting model, we do not need to deal with the noise occurring in log files from real systems (i.e. we save most of the pre-processing efforts described in Section 7.1.1). Furthermore, we can in this way use a common approach for all models and do not have to deal with the completely different log formats of the different systems used by the municipalities. Hence, in this way we avoid having to perform extensive pre-

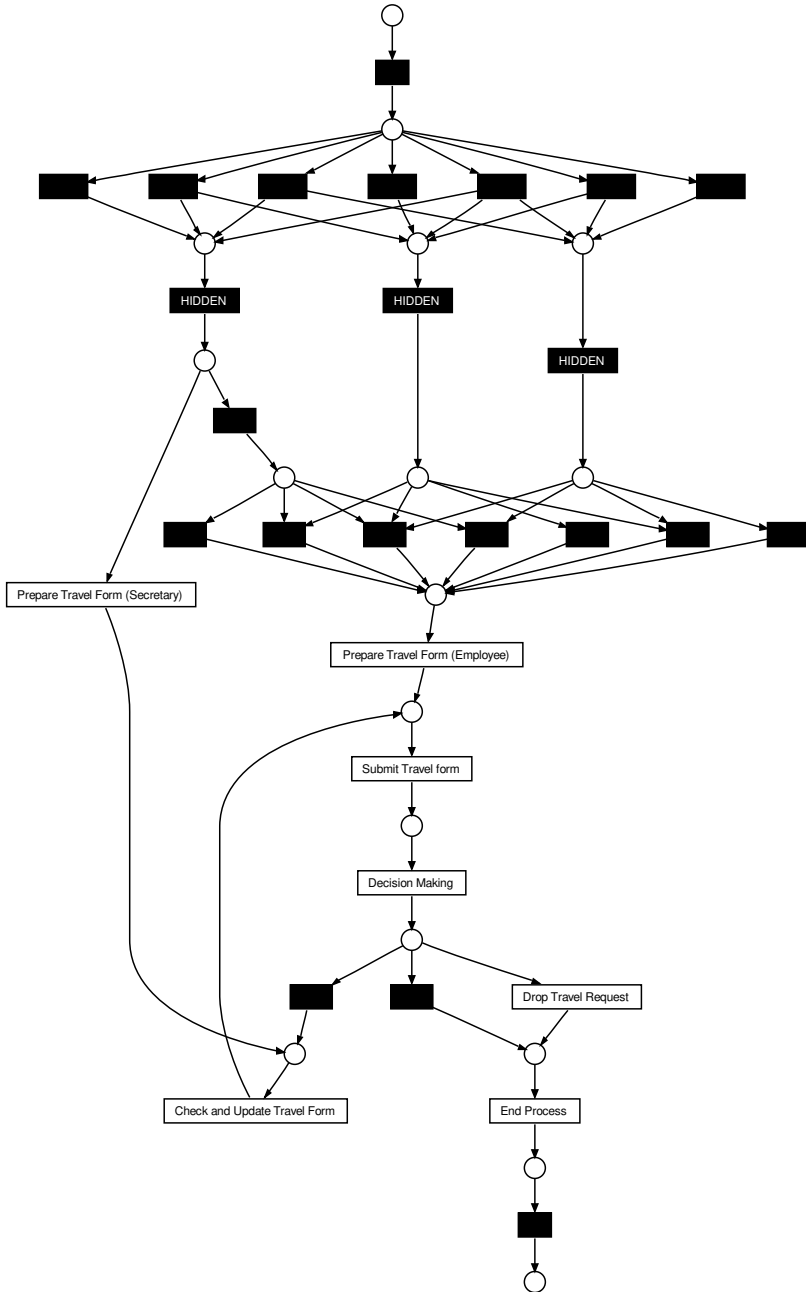


Figure 7.24: The configured workflow net derived from the configuration highlighted in Figure 7.23.



Figure 7.25: Key data of the combined log files of the various municipalities for acknowledging an unborn child (ProM screenshot).

processing steps, while still having ‘optimal’ log files for the verification of the mining approach available.

To generate log files for the case study’s Protos models, Protos2CPN [75] (see Section 2.4.2, pp. 32f) can translate the Protos models into colored Petri nets which write log files during their simulation. Using the ProM import framework [83, 146], these log files can be translated into the MXML format required by ProM.

The individual log files generated from the different variants of a process can then simply be merged. For example, we generated in this way logs for in total over 3800 cases of the various process variants for acknowledging an unborn child (see Figure 7.25). Using the multi-phase miner [56, 57] on the combined log file, we generated an EPC from this log file. The resulting EPC was translated into a Protos model using ProM. This allows an easier comparison of the resulting models with the process from Figure 6.3 (p. 128). Figure 7.26 shows the model mined from the combined log file.

When looking at the mined model, it becomes immediately apparent that some log events with different names represent the same behavior. For example, the events *identify*, *identify*, and *Confirm Identity* which are highlighted on the top-left of the model all represent the same identification task — they were just named differently in the various models. In the same way *Decide choice of name (foreign country)* and *Decide choice of name (for foreign country)* represent the same tasks, as well as *Decide choice of name (Dutch law)* and *Decide choice of name (under Dutch law)*. For that reason, such log events should be harmonized before mining the model (again). The model mined after harmonizing these (and further) log events is depicted in Figure 7.27.

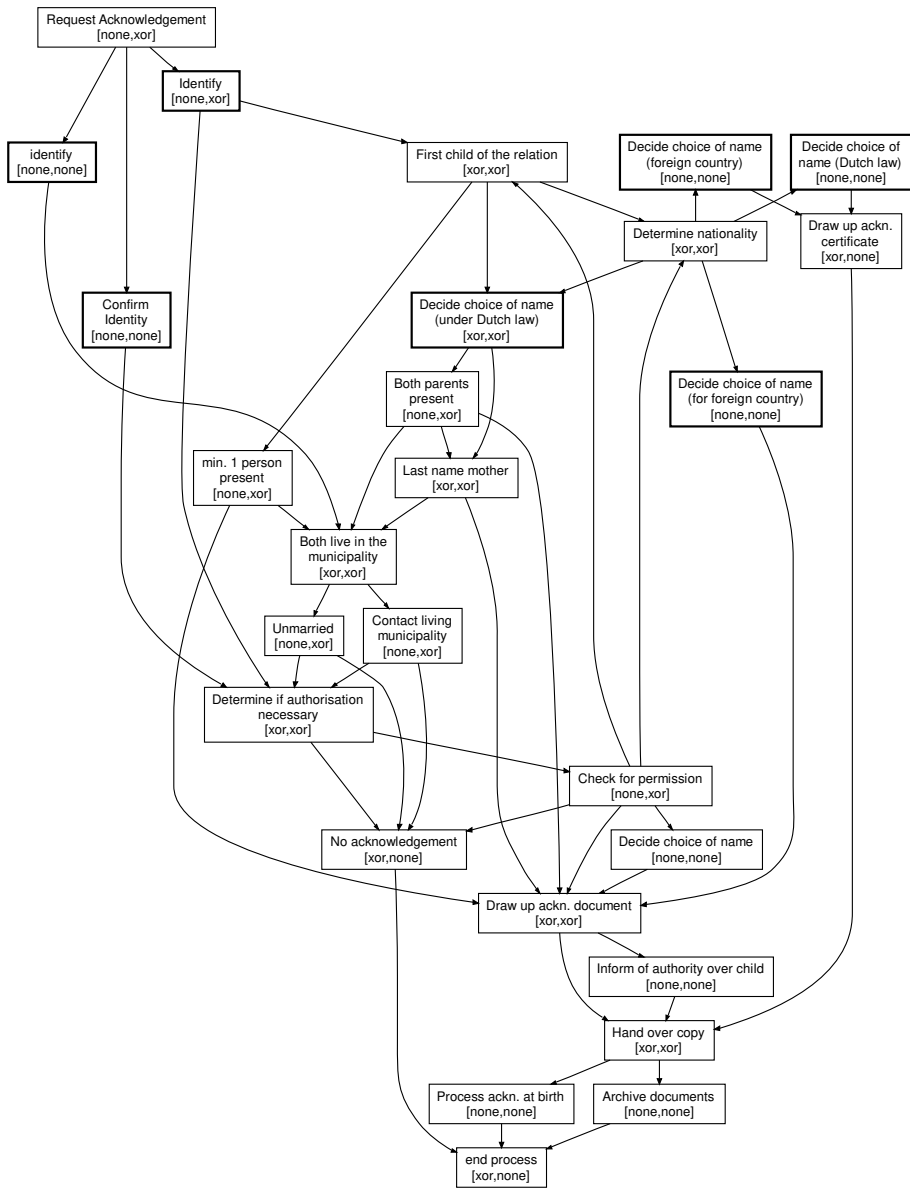


Figure 7.26: Protos model for acknowledging an unborn child, mined from the combined log files of the various municipalities using the multi-phase miner [56, 57]. Note that some tasks seem identical but have different labels.

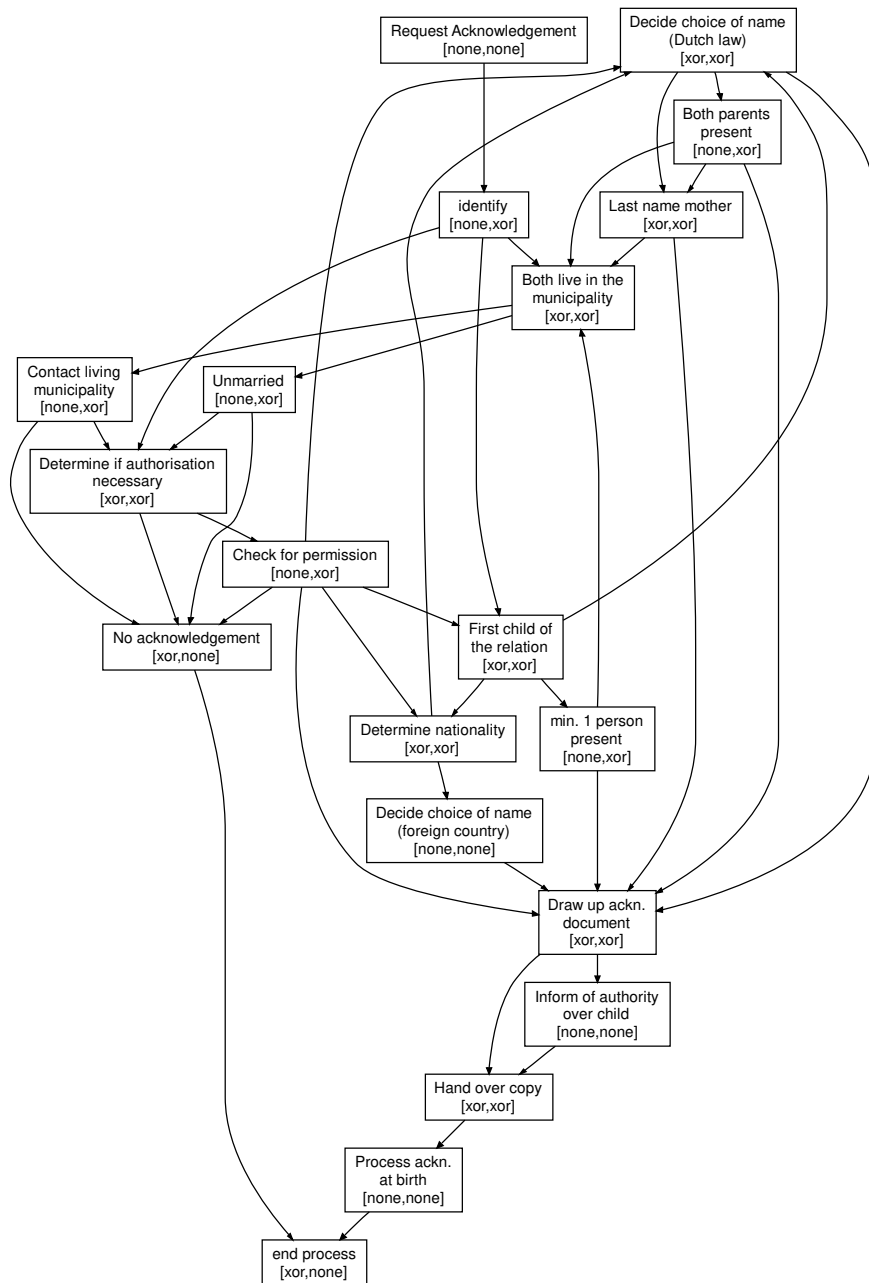


Figure 7.27: Protos model for acknowledging an unborn child, mined from the combined log files of the various municipalities after unifying equivalent log events.

When carefully comparing it with the manually integrated Protos model from Figure 6.3 (p. 128), it turns out that both models are equivalent, i.e. each graph node has a corresponding node in the other graph, and the equivalent nodes of two connected nodes in one graph are also connected in the other graph and vice versa. Given that we have simulated in average over 750 process instances per process model, and given that all decisions during this simulation were made randomly with an equal distribution of choosing possible process continuations, we assume that the used log files represented the complete possible behavior of the individual models. Furthermore, van Dongen and van der Aalst [56, 57] have proven that the multi-phase mining algorithm preserves all behavior when building a process model. Hence, it is not surprising that the model generated through process mining conforms to the model that was manually created with the same goal.

Still, it also shows that with good quality log files available, process mining is a useful tool for creating the basic process model of a configurable process model. In the example of the case study, the manual creation of the merged model including elimination of errors took the process designer several days. Producing the log files, identifying and mapping identical log events, and mining the integrated process model was however just a matter of several hours.

7.4.2 Merging Individual Models

A direct merge of the process models from the case study according to the merge algorithm presented in Section 7.2 seems to be even more attractive as we can avoid generating log files of the individual models. For this, we first need to convert the Protos models from Chapter 6 into EPCs as the merge algorithm is based on EPCs. As mentioned before, ProM provides the necessary functionality for this. After all individual Protos models have been converted into EPCs, we can use the implementation of the merge algorithm as described in Section 7.2.5. In this way, we can add one EPC after the other to the merged EPC. In each step, the merge algorithm provides the opportunity to match the various function names. Thus, it can easily be seen and specified that, for example, the function *Identify* of one of the models is identical to the function *Confirm identity* in the other model (see Figure 7.28).

After merging all four individual models and the model of the NVVB, the final EPC is converted back into a Protos model. The resulting model for the process of acknowledging an unborn child is depicted in Figure 7.29. It is again equivalent to the manually merged model shown in Figure 6.3 (p. 128), and thus to the model generated using the multi-phase miner shown in Figure 7.27.

Also this is not a surprising result. The merge algorithm is inspired by the multi-phase miner [56, 57] which was used to create the model from Figure 7.27. In fact, the merge algorithm merges behavior locally in an identical manner as the multi-phase miner. The difference is that the merge algorithm performs this local merge systematically, i.e. once per any combination of functions. Thus, it guarantees that any behavior possible according to one of the input models is preserved in the resulting model. When using the multi-phase miner, such

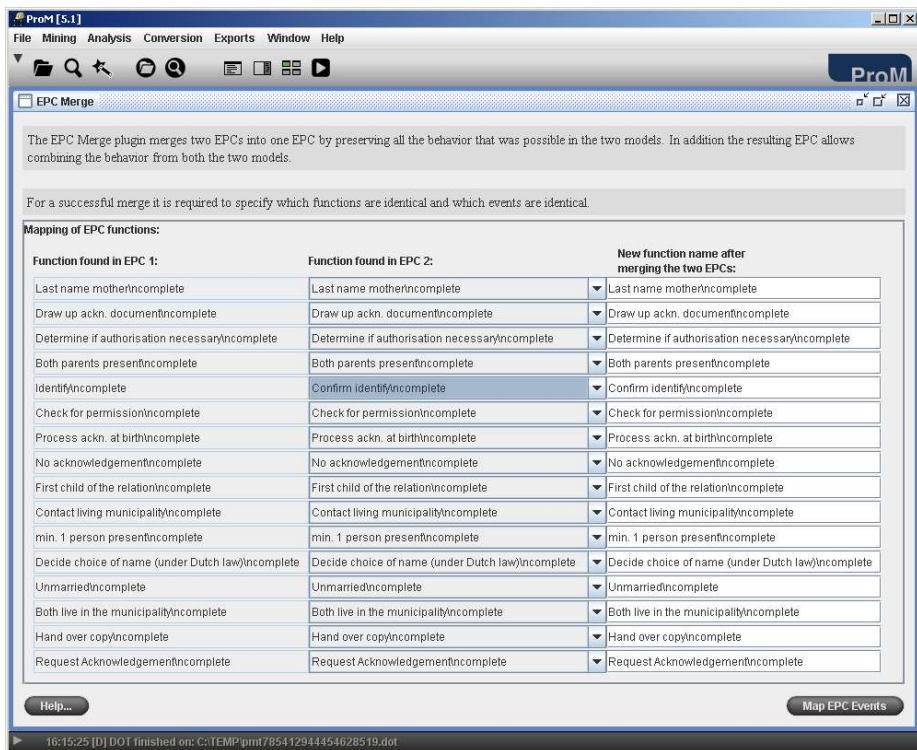


Figure 7.28: Mapping functions from different EPCs before merging the EPCs into a single model.

behavior is only merged if it occurs in one of the process instances contained in one of the log files. Hence, it is necessary to assume completeness of the log files in order to assume a complete merge of the possible behaviors. Furthermore, the multi-phase miner re-checks already merged behavior with any new process instance following the same path as previous instances.

We can therefore conclude that if correct models of process variants that should be combined to a basic process model are available, then the merge algorithm presented in Section 7.2 is able to generate models that are at least as good as the models that we can create by mining a process model from log files using the multi-phase miner. In scenarios similar to the case study from Chapter 6 the merge algorithm can therefore help in significantly reducing the workload of creating a basic process model. For example, creating the process model depicted in Figure 7.29 from the original Protos models took by automatically merging the individual models less than half an hour. This is again significantly less time than the time needed using the mining approach which, in turn, took less time than merging the models manually.

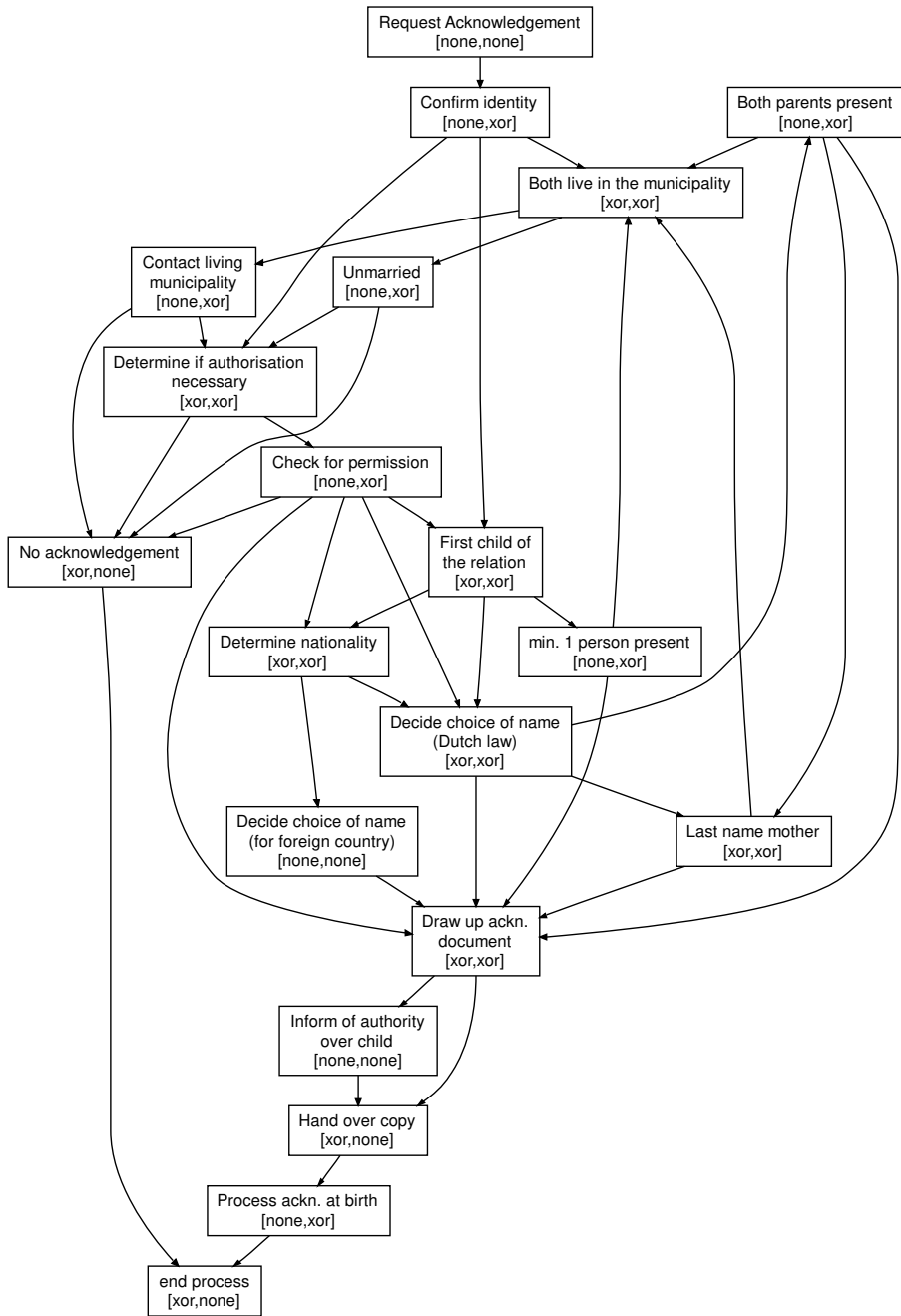


Figure 7.29: Protos model resulting from the merge of all five process variants for acknowledging an unborn child.

7.4.3 Identifying Individual Configurations

To identify the configuration of a process model that is used by a particular system, we suggested in Section 7.3 to replay a log file of this system on the basic process model. To be able to do this on the basic process models derived in the case study from Chapter 6, let us use again ProM to convert the Protos models into Petri nets. ProM's conformance checker [144] then provides the opportunity to replay the individual log files which we generated for mining basic process models in Section 7.4.1 on these Petri nets.

Figure 7.30 shows the Petri net that is derived from the integrated Protos model for the various variants of acknowledging an unborn child which was shown in Figure 6.3 (p. 128). As Petri net transitions do not support an XOR-join behavior, each transition that has a corresponding join behavior in the Protos model is preceded by a dedicated place for this. Thus, the transitions leading to such a place represent the different ports of the Protos task that corresponds to the particular transition. For example, the silent transitions *t130* and *t185* in Figure 7.30 represent in this way the input ports of the task *First child of the relation*. In the same way, there is also a dedicated place after each transition in the Petri net in case of an XOR-split behavior of the corresponding Protos task. The transitions subsequent to this place thus represent the different output ports of the task.

To detect the configuration of this process that is, for example, used by the NVVB in their reference model, the transitions and arcs that were visited during the replay of the corresponding log file are highlighted in Figure 7.30. In this way, it is easy to see that neither transition *t130* nor transition *t185* was used when replaying the log file. For that reason, we can assume that both these input ports of the task *First child of the relation* must be blocked in order to achieve a configuration of the basic process model that matches the NVVB model. Of course, this means that the task's output ports are not used either and can thus be blocked as well. In the same way, none of the transitions leading to the transitions *Both live in the municipality*, *min. 1 person present*, *Contact living municipality*, *Unmarried*, or *Last name mother* has fired during the replay. Hence, all their input and output ports must be blocked as well.

Also, the transition *Inform of authority over child* has not fired. However, the transition *Draw up ackn. certificate* that leads to the preceding place did fire. All cases then just follow a silent transition that represents a simple alternative to the transition *Inform of authority over child*. This alternative path also exists in the Protos model from Figure 6.3. Hence, it is fine to block the input and output ports of task *Inform of authority over child*. Still, it could be investigated if there is a configuration of this integrated model that uses both the task *Inform of authority over child* as well as its bypass. If this is not the case, the input port of task *Inform of authority over child* could also be configured as hidden which then allows deleting the bypass in the configurable process model.

The same argumentation as for task *Inform of authority over child* holds for task *Both parents present* as it is bypassed using a silent transition as well. Special attentions needs to be paid to the silent transitions *t183* and *t187* which

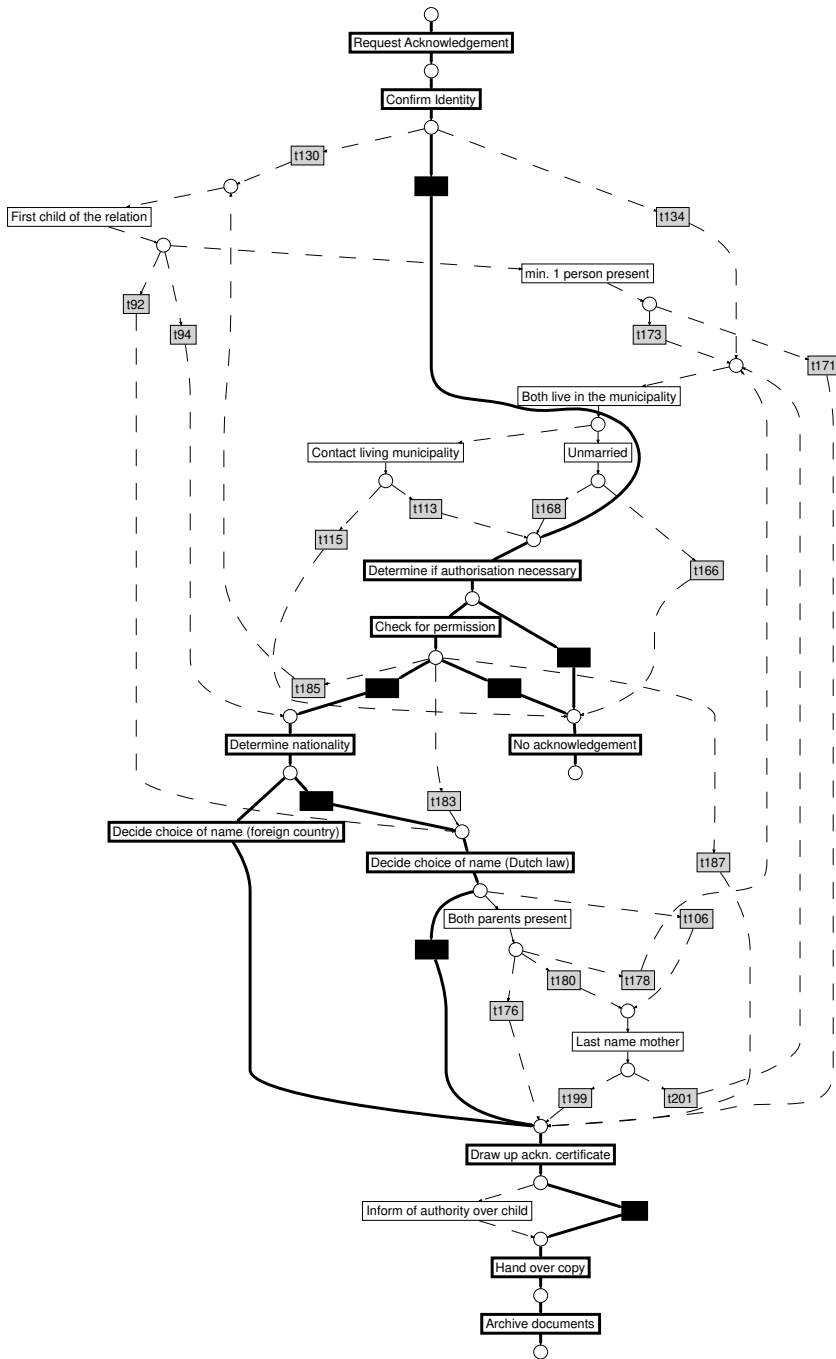


Figure 7.30: Replaying the logs on the Petri net which conforms to the Protos model for acknowledging an unborn child, reveals the parts actually used. These are highlighted.

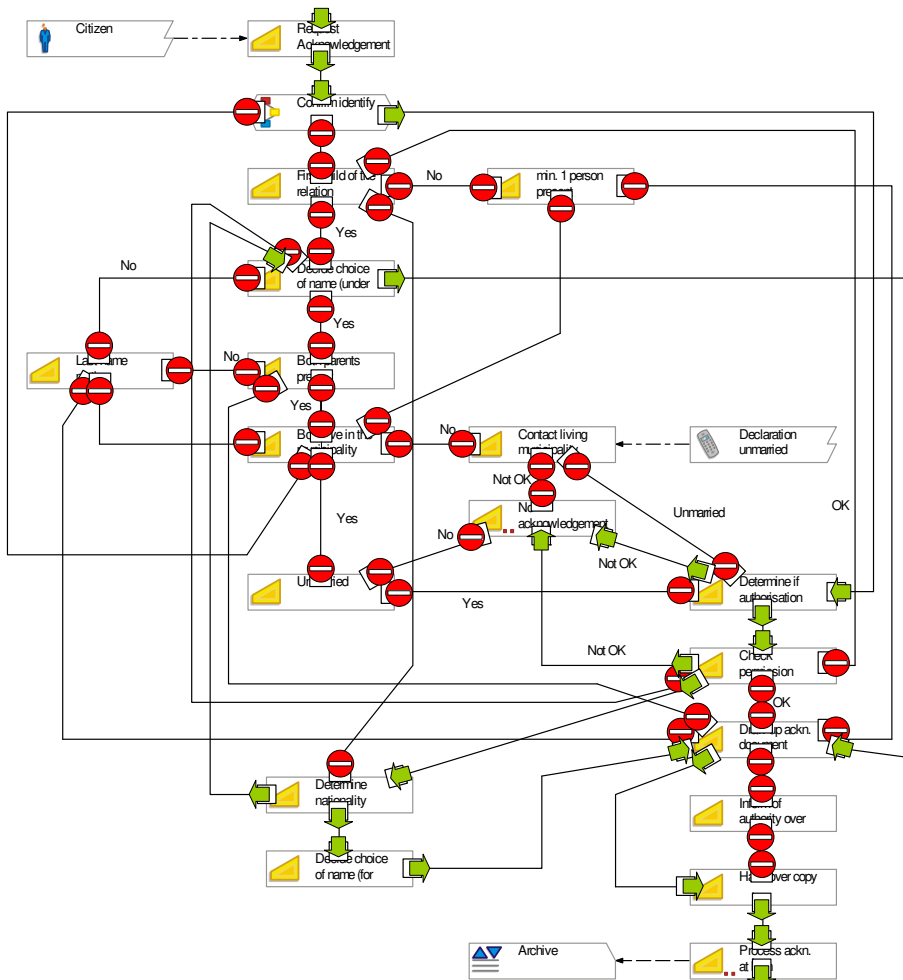


Figure 7.31: Configuration of the process for acknowledging an unborn child according to the reference model of the NVVB.

connect transition *Check for permission* with the transitions *Decide Choice of name (Dutch law)* and *Draw up ackn. certificate* respectively. Although these surrounding transitions do fire, the silent transitions themselves do not fire during the replay. Hence, the corresponding output ports of task *Check for permission* (and the input ports of the other two tasks) must be configured as blocked as well.

Figure 7.31 shows all these configuration decisions in the integrated Protos model from Figure 6.3.

7.5 Related Work

The techniques suggested in this chapter use and build up on ideas developed for generating process models from log files, by merging several process models, and through synthesizing state based models. Moreover, they use ideas developed to check if a log file conforms to a process model or not. Therefore, we will in the following depict the relation between this related work and the techniques presented here as well as point the interested reader to the relevant sources for further details.

7.5.1 Process Mining

Process mining techniques aim at extracting information from log files that are generated by information systems. In this chapter we suggested the use of algorithms that derive a process model from such log files, depicting the process behavior in a graphical manner. For our example, we used the so-called multi-phase miner, developed by van Dongen and van der Aalst [56, 57]. This algorithm guarantees that any event trace of the log file can be replayed on the process model that was generated using this algorithm, i.e. the model fits 100 percent to the log file. While this is nice for our examples, this is not always a necessary property, especially if a log file not only contains desired information, but also undesired information. This so called ‘noise’ can, e.g., be the result of uncompleted, failed, or exceptionally deviating executions of the particular process. Thus, most of the time such noise should be ignored.

Figure 7.32 provides an overview of process mining algorithms and their characteristics: *Event log* determines if the algorithm considers each log event as an atomic task, or if the algorithm is also able to deal with separate log events for the start and the completion of a task. *Mined model* provides information on the nature of the generated model, i.e. if only dependencies between tasks are depicted, if the branching and synchronization behavior of splits and joins is determined, or if a process model is derived covering all events of the log or only a subset. *Sequence*, *Choice*, *Parallelism*, *Loops*, and *Non-free-choice* refer to the corresponding workflow patterns [11]. *Invisible tasks* can be used to skip tasks (as we did for hidden transitions), or they can be used for silently branching and synchronizing the control-flow. The most common way to deal with noise is pruning of dependencies that occur only rarely, i.e. less often than a certain threshold. A more detailed discussion on the different issues and approaches to process mining can be found in the work of van der Aalst et al. [10], Alves de Medeiros [25], van Dongen [55], Günther [82].

The ProM process mining framework which we already used several times in this chapter supports most of the listed process mining algorithms through about 50 plug-ins dedicated to process mining. It furthermore enables the combined use of these and a wide variety of other process analysis techniques through a total of more than 250 process mining, analysis, import, export, conversion, and filter plug-ins [15].

	Cook et al. [45]	Agrawal et al. [22]	Pinter and Golani [128]	Herbst and Karagiannis [90]	Schimm [165]	Greco et al. [81]	van der Aalst et al. [14]	Weijters and van der Aalst [186]	van Dongen and van der Aalst [57]	Wen et al. [188]	Alves de Medeiros [25]	Günther and van der Aalst [84]	van der Werf et al. [189]	van der Aalst et al. [18]
Event log:														
- Atomic tasks	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
- Non-atomic tasks			✓		✓									
Mined model:														
- Dependencies	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Nature split/join	✓			✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
- Whole model		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sequence:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Choice:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Parallelism:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Loops:														
- Structured	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
- Arbitrary		✓	✓				✓	✓	✓	✓	✓	✓	✓	✓
Non-free-choice:														
- Local	✓	✓	✓			✓	✓	✓	✓	✓	✓		✓	✓
- Non-local										✓	✓	✓	✓	✓
Invisible Tasks:														
- Skip	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓		
- Split/join	✓			✓										
Noise:														
- Depend. pruning	✓	✓	✓	✓							✓	✓		
- Other								✓			✓	✓		

Figure 7.32: An overview of process mining algorithms and their characteristics (adapted from Alves de Medeiros [25]).

7.5.2 Model Merging

Process models are used to provide abstract views on the process behavior of complex systems. As various models are usually inconsistent [152] even if they are intended to depict the same behavior, several frameworks to align the depicted system descriptions for being able to merge them are suggested in liter-

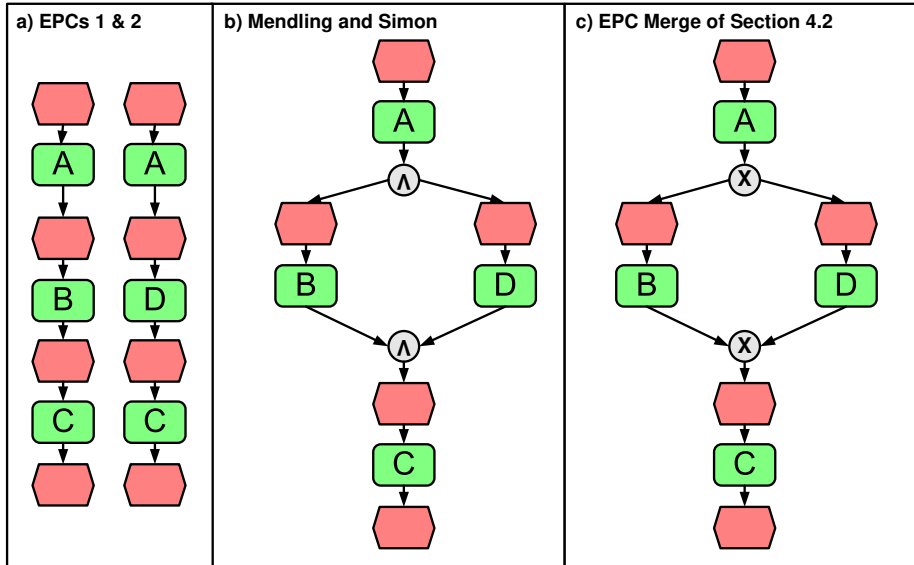


Figure 7.33: Merging the EPCs from (a) according to the algorithm of Mendling and Simon [118](b) and as depicted in Section 7.2 (c).

ature, e.g. by Brunet et al. [39] as well as in Sabetzadeh and Easterbrook [152] and Sabetzadeh et al. [153].

Algorithms that aim at merging two process models are typically developed for a particular purpose. For example, it is interesting to compare the approach suggested here with an algorithm suggested by Mendling and Simon [118]. Both algorithms are capable of merging process models that are depicted as EPCs. Our goal is to merge EPCs depicting different processes that are executed similarly. Thus, whenever the merged processes deviate, our approach here provides a choice between the varying behavior. The goal of Mendling and Simon [118] is to integrate different views on the same process. Thus, they assume that all the tasks from all the models need to be executed. If the same task occurs in different views, i.e. different models, the overall process has to be synchronized, and if tasks vary between the views, all those tasks can be executed. Hence, the algorithm of Mendling and Simon [118] synchronizes and de-synchronizes (see Figure 7.33b) through \wedge connectors whenever the algorithm depicted in Section 7.2 provides a choice between the varying behavior through XOR or \vee connectors (see Figure 7.33c).

7.5.3 Synthesis

At the beginning of Section 2.2 (pp. 19ff), we showed how behavior can be represented as a state-transition system which explicitly shows each state a system can be in and how it can change between these states. Thus, besides generating integrated models through the mining from event logs, and the merging of

the models, also the synthesis of such state-based models into process models might be able to generate a process model covering multiple variants of how a process can be executed. The goal is here, to find a process model that covers exactly the behavior of all the considered transition systems. Thus, while being as compact as possible — like in our approach — the resulting process model should allow for all the behavior that is represented by the state transition systems. But different from our approach, it should not over-approximate, i.e. not allow for any additional behavior. For example, Cortadella et al. [46] present an algorithm for the synthesis of state-based models into Petri nets by using minimal regions which is also implemented in a tool called *Petrify* [47]. Also ProM provides several region-based approaches to synthesis [15]. Details of the merge of transition systems and properties of such a merge of behavior are discussed in the work of Brunet et al. [39] and Uchitel and Chechik [180].

7.5.4 Identifying Configurations and Conformance

Identifying a configuration for a process model that matches a log file is closely related to research on checking the conformance of a log file to a process model. Besides checking for the fitness of a log file to a process model, conformance checking also tests the **appropriateness** of a process model to a log file, i.e. how much additional behavior does the process model allow compared to the behavior occurring in a log file. If we compare a process variant with the basic process model, this additional behavior corresponds to the behavior we disallow through configuration.

Metrics to identify the fitness and the appropriateness of a log file to a process model have been developed by Rozinat and van der Aalst [144, 145]. To determine the fitness and a simple appropriateness measurement, Rozinat and van der Aalst use an advanced log replay which has been implemented in ProM. In addition to marking visited arcs and nodes of a process model, it precisely counts the number of visits as well as it searches for partial matches of a log file to a process model. Still, the implementation of the log replay as part of the conformance checker in ProM can also be used to determine configurations as explained in Section 7.3.

Instead of replaying a log file on the model, Cook and Wolf [44] generate event streams from a process model and then compare these event streams with the event streams in the log files. However, for detecting configurations of a process model a comparison of behavior on the log level instead of on the model level is less suitable as it would be necessary to translate the differences back onto the model.

While not completely replaying log files, Jansen-Vullers et al. [97] use frequency profiles, depicting how many times each task is executed according to a particular log file. By generating different frequency profiles from different log files, they determine optional tasks and routings, i.e. configuration options, for EPCs. Different from the complete replay of a log files, generating frequency profiles does not lead to a concrete configuration, but produces a range of config-

uration variants. The approach of Jansen-Vullers et al. [97] therefore determines the ‘best’ solution by resolving an integer programming problem.

7.6 Conclusions

In this chapter, we discussed tools that can help the developer of a configurable process model in constructing such models. As configurable process models aim at integrating and providing the variations among several variants of a business process, the different variants of the process are often already in place somewhere. In such cases, information about the individual process variants is most of the time available, typically in the form of log files that have recorded executed work, or as process models documenting the work that should be executed.

Log files of a business process are composed of traces that document each execution of the business process. Process mining can be used to generate process models from these traces. To generate a process model that covers multiple process variants through process mining, the traces of the individual variants need to be combined and aligned such that semantically equivalent log events are mapped onto each other. A process mining algorithm can then build a process model such that it is valid for all traces. Hence, the resulting model then integrates the behavior of the different process variants.

For providing good results, process mining algorithms require a proper, but often difficult and cumbersome, pre-processing, i.e. cleaning and aligning, of the log files of the different systems. Therefore, if process models are already available for different process variants, directly merging these models often returns better results. When constructing a configurable process model, it is important that the merge algorithm preserves at least the behavior depicted by the individual models as it would otherwise be impossible to derive this behavior through process configuration later on. The presented merge algorithm achieves this through generalizing specific AND-join, AND-split, XOR-join, and XOR-split synchronization and branching patterns to more general OR-join and OR-split patterns whenever necessary.

Of course, a process model integrating various process models can also be constructed from log files by first mining individual process models and afterwards merging the individual models. However, note that in practice the resulting model will hardly depict exactly the same behavior as a model which is created by combining the log files first and then mining the integrated model directly from the combined log file. The reason for this is that process mining as well as the model merging algorithms both over-approximate resulting behavior, but hardly in an identical manner. Still, by using similarly over-approximating algorithms, like combining the depicted model merging algorithm with the multi-phase miner of van Dongen and van der Aalst [56, 57], a similar behavior can be captured. For example, we generated identical process models by merging process models from the municipality case study from Chapter 6

and by mining a process model from log files of the various input models using the multi-phase miner.

The additional behavior made possible through integrating and over-approximating the behavior of different process variants can be re-restricted through configuring the integrated model, i.e. the basic process model. To identify a complete configuration of the basic process model, log files can also be used. For this, the log file of an individual system can be replayed on the integrated model. The configuration then results from the transitions that are used, skipped, or completely avoided during the replay.

The configuration that is derived from the log file of an existing system is then obviously a feasible configuration of the process model (otherwise it would not have been possible to create the log file). However, configurable process models also aim at providing the model users with options to deviate from such universal solutions by combining options from different systems. Thus, the configuration of an individual process variant provides a good starting point for deriving an individual process variant, but we need to determine in the following which changes to this configuration are possible and which should better be avoided as they would lead to incorrect process models.

*The more constraints one imposes, the more one
frees one's self of the chains that shackle the spirit.
Igor Stravinsky (1942)*

Chapter 8

Executability of Configurations

Tasks of a basic process model cannot be configured freely as being allowed, blocked or hidden. Usually, some tasks are essential for a process's execution. For example, a travel approval process that does not allow for executing the approval of the request would be rather meaningless. Besides such content issues, a configured process model can also become structurally or semantically incorrect, i.e. it is technically no longer executable. For example, let us have a look at the top half of Figure 8.1. It shows a configuration for the workflow net from Figure 2.2, but the blocking of the transition *Compare Accommodation Quotes* is problematic here. In the resulting process model (lower half of Figure 8.1), p_2 and p_4 are not on some path from p_I to p_O . Thus, the resulting Petri net is not a correct workflow net (see Definition 2.15, p. 23). Moreover, the Petri net deadlocks as soon as t_1 and τ_{t_3} have fired because both t_5 or t_6 would require a token in p_4 to be able to fire.

To restrict the configuration space and thus prevent such improper configurations, configurable process modeling languages as we defined them in Chapter 4 allow for the specification of configuration constraints. Only if the configuration constraint of a configurable process model evaluates to *true*, a configuration is valid. Still, the question that remained open so far is how to find such constraints.

In the following, we will therefore explore, how we can setup constraints on configurations which ensure the executability of a configured process model. Thus, we will have a look on the correctness of a process model's configuration from different perspectives. As we have clearly defined syntactical and semantical correctness of process models using the formal workflow net notation in Chapter 2 (Definitions 2.15, p. 23, and 2.18, p. 24), we will first analyze how a correct workflow net syntax can be preserved during configuration as well as

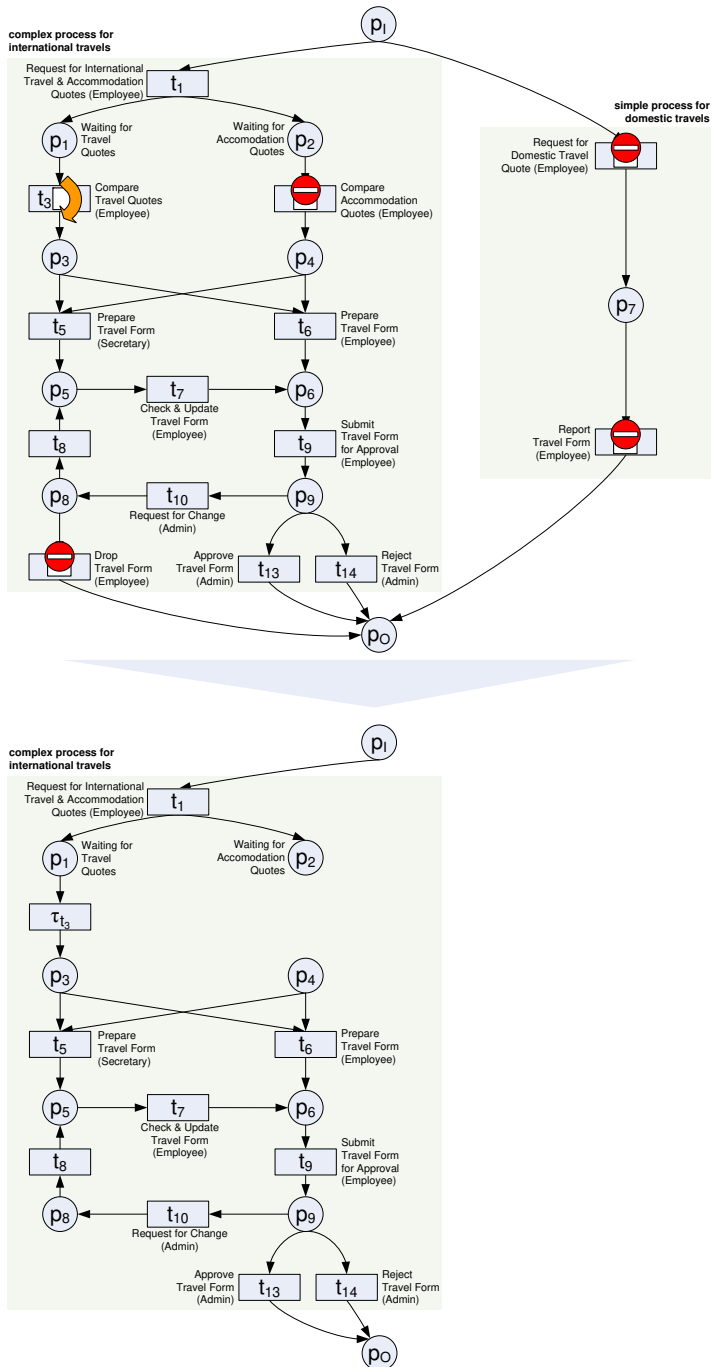


Figure 8.1: This configuration leads to a non-executable process model

for which models a correct semantics, i.e. soundness, can also be preserved in this way. Thereafter, Section 8.3 will briefly analyze how these results can be applied in the context of C-YAWL and the configuration framework presented in Chapter 5. How it can be ensured that missing data or resources cannot cause a deadlock of the process is subject of Section 8.4.

8.1 Preserving Syntactic Correctness

Let us start this chapter with analyzing how we can preserve a correct workflow net syntax for configured workflow nets. Thus, we assume that we want to configure a workflow net which is syntactically correct. The configured Petri net (as derived according to Definition 3.5, p. 56) thus needs to have unique source and sink places, as well as each node of the configured net needs to remain on a path between these two places (Definition 2.15, p. 23). Only then the configured net is also a workflow net.

By definition of a configured workflow net, the source place p_I and the sink place p_O remain in the configured net. As no arcs are added during the configuration, these places also preserve their character as source place and sink place. Thus, during configuration we only need to test if a configuration violates the requirement that each node of the resulting, configured workflow net is on a path between p_I and p_O .

Transitions that are configured as hidden are replaced with silent transitions during the configuration. For example, the hidden transition *Compare Travel Quotes (Employee)* is replaced with the silent transition τ_{t_3} in Figure 8.1. Incoming and outgoing arcs are identical to the ones of the original transitions. Thus, these replacements do not interrupt any of the paths between the source place and the sink place.

Blocked transitions, however, are removed from the net including their incoming and outgoing arcs. Thus, this removal can disconnect parts of previously existing paths between the source and the sink place as we have seen with removing transition *Compare Accommodation Quotes (Employee)* in Figure 8.1. Due to the removal of this transition, both p_2 and p_4 are no longer on such a path.

But not every blocking really causes problems. As both transitions *Request for Domestic Travel Quote (Employee)* and *Report Travel Form (Employee)* of the simple process for domestic travels are blocked, both transitions are removed including their surrounding arcs. In this way, p_7 is not connected to any arc any longer and is removed from the net as well (Definition 3.5, p. 56). Also, the blocking of the transition *Drop Travel Form (Employee)* is unproblematic. Although its surrounding arcs are removed and thus the paths from p_8 to the sink place p_O via this transition, there is a second path from p_8 to p_O via t_8 , p_5 , t_7 , p_6 , t_9 , p_9 , and t_{13} or t_{14} . Hence, we only have to forbid the blocking of transitions that would remove the last paths from p_I to any node remaining in the configured net, or from any node remaining in the net to p_O .

Process configuration is usually a staged process, i.e. not all configuration decisions are made at one point in time but users make them one after another in a stepwise manner. Thus, the viable configuration options change after each step. To determine the viable configuration options at any point in time, boolean expressions can be used. Here, we distinguish nodes which remain in the net from nodes which do not by using a boolean variable for each node. If the variable is set to *true*, the node remains part of the net; if it is set to *false*, the node is dropped in the configured net. Accordingly, we assign a blocked transition the value *false*, while a transition that is allowed or hidden is assigned the value *true*. All transitions that are not explicitly configured remain as variables (i.e. unset).

According to Definition 3.5, any internal place remains in the net if there is a non-blocked transition in its pre-set or post-set. This can be translated into a **configuration constraint** in boolean logic as follows. If any transition variable is set to *true*, i.e. it remains in the net, the places both in its pre-set as well as in its post-set must also be set to *true* to remain in the net, formally: $\bigwedge_{t \in T^c} (t \Rightarrow (\bigwedge_{p \in \bullet t} p \wedge \bigwedge_{p \in t \bullet} p))$.¹

By valuating places and transitions in this way, we can now ensure the requirement that any node remains on a path from p_I to p_O . For this, we test for each node that remains in the net if there is a path from the source place p_I to this node of which no node is assigned the value *false*, and if there is a path from the node to the sink place p_O of which no node is assigned the value *false*. If both these paths exist, then none of the nodes on these paths are removed, i.e. the path from p_I via this node to p_O is preserved. In fact, if a non-blocked transition t has at least one place in its pre-set which is on a directed path from p_I , then t is also on such a path. In the same way, if it has at least one place in its post-set which is on a directed path to p_O , then a path exists that leads from t to p_O . Thus, while the user configures the model and thus sets the values of the transitions to *true* or *false*, the configuration constraint implies that we only need to test the paths from p_I to the places preceding the transitions that are unset or set to *true*, respectively the paths from the places succeeding these transitions to p_O . Furthermore, when searching for such paths we can restrict our analysis to acyclic paths as a cycle² always leads back to the same node, therefore not providing any valuable progress on a path from p_I to p_O . Formally, we define an acyclic path as follows:

Definition 8.1 (Acyclic Path) Let $PN = (P, T, A, L, l)$ be a Petri net:

- $\psi = \langle n_1, n_2, \dots, n_k \rangle$ is an **acyclic path** of PN if $(n_i, n_{i+1}) \in A$ for $1 \leq i \leq k - 1$ and $1 \leq i < j \leq k \Rightarrow n_i \neq n_j$ for any i and j ,
- Ψ_{PN} is the set of all acyclic paths of PN .

Furthermore, if PN is a workflow net then

¹Where with t, p we indicate a transition, respectively a place, which is set to *true*.

²Note that in this context cycles refer to the net structure and not to the dynamic behavior. Therefore, it suffices to consider only acyclic paths.

- for all $n \in P \cup T$, $\psi_I(n) = \{\langle n_1, n_2, \dots, n_k \rangle \in \Psi_{PN} \mid n_1 = p_I \wedge n_k = n\}$ is the set of all acyclic paths from p_I to n ,
- for all $n \in P \cup T$, $\psi_O(n) = \{\langle n_1, n_2, \dots, n_k \rangle \in \Psi_{PN} \mid n_1 = n \wedge n_k = p_O\}$ is the set of all acyclic paths from n to p_O .

The following definition then summarizes the depicted constraint on the correctness of a configured workflow net.

Definition 8.2 (Process Correctness Constraint) *Let $WF = (P, T, A, L, l)$ be a workflow net. Treating each place and each transition of WF with a propositional variable, the process correctness constraint $PCC(WF)$ is a propositional logic formula over these variables, given by the conjunction of the following expressions:*

- p_I and p_O are always true, i.e. $p_I \wedge p_O$;
- each place p that evaluates to true implies the disjunction of all acyclic paths from p_I to p and the disjunction of all acyclic paths from p to p_O :

$$\bigwedge_{p \in P} [p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)].$$

In the way the process correctness constraint PCC is constructed, it guarantees that any process configuration of a workflow net whose valuation satisfies PCC , results in a syntactically correct configured workflow net:

Theorem 8.1 *Let $WF = (P, T, A, L, l)$ be a workflow net and $PCC(WF)$ be its process correctness constraint. Let \mathcal{C} be a configuration of WF and let $WF^{\mathcal{C}} = (P^{\mathcal{C}}, T^{\mathcal{C}}, A^{\mathcal{C}}, L^{\mathcal{C}}, l^{\mathcal{C}})$ be the resulting configured net. Let $v \in T \cup P \rightarrow \{\text{true}, \text{false}\}$ be such that $v(q) = \text{true}$ iff $q \in T^{\mathcal{C}} \cup P^{\mathcal{C}}$. Then $WF^{\mathcal{C}}$ is a workflow net $\Leftrightarrow v \models PCC(WF)$.*

PROOF

(\Rightarrow) Let $WF^{\mathcal{C}}$ be a workflow net and let $v \in T \cup P \rightarrow \{\text{true}, \text{false}\}$ such that $v(n) = \text{true}$ iff $n \in T^{\mathcal{C}} \cup P^{\mathcal{C}}$. As $p_I \in P^{\mathcal{C}}$ and $p_O \in P^{\mathcal{C}}$ (Definition 3.5, p. 56), $v(p_I) = \text{true}$ and $v(p_O) = \text{true}$, hence $v \models p_I \wedge p_O$.

Since $WF^{\mathcal{C}}$ is a workflow net, for all $p \in P^{\mathcal{C}}$ there exists at least one directed path from p_I to p . Let $\psi \in \psi_I(p)$ be such a path, thus for all $n \in \psi$ we have $n \in P^{\mathcal{C}} \cup T^{\mathcal{C}}$, hence $v(n) = \text{true}$. Therefore, $v \models \bigwedge_{n \in \psi} n$. Hence, $v \models \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n)$. Similarly, as there is at least one path from p to p_O , $v \models \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)$, hence $v \models \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)$. Thus, for all $p \in P^{\mathcal{C}}$: $v \models \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)$ and therefore for all $p \in P^{\mathcal{C}}$: $v \models p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)$. If $p \in P \setminus P^{\mathcal{C}}$ then $v(p) = \text{false}$ and thus $v \models p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)$. Hence $v \models \bigwedge_{p \in P} [p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n \in \psi} n) \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n \in \psi} n)]$.

(\Leftarrow) Let $v \models PCC(WF)$. Assume $WF^{\mathcal{C}}$ is not a workflow net. Since p_I and p_O belong to $WF^{\mathcal{C}}$ by definition, choose $p \in P^{\mathcal{C}}$ such that there is either (1) no path from p_I to p or (2) no path from p to p_O .³ If (1) then for

³This covers both the case if a node is not on path from p_I to p_O as well as the case of multiple source or sink places.

all $\psi \in \psi_I(p)$ there is a node $n \in \psi$ such that $n \notin P^C \cup T^C$ and thus $v(n) = \text{false}$, $v \not\models n$ and hence for all $\psi \in \psi_I(p)$ $v \not\models \bigwedge_{n' \in \psi} n'$ and thus $v \not\models \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n' \in \psi} n')$. If (2) then for all $\psi \in \psi_O(p)$ there is a node $n \in \psi$ such that $n \notin P^C \cup T^C$ and thus $v(n) = \text{false}$, $v \not\models n$ and hence for all $\psi \in \psi_O(p)$ $v \not\models \bigwedge_{n' \in \psi} n'$ and thus $v \not\models \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n' \in \psi} n')$. From both cases we can conclude $v \not\models \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n' \in \psi} n') \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n' \in \psi} n')$. Given that $v \models p$, $v \not\models p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n' \in \psi} n') \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n' \in \psi} n')$. This implies that $v \not\models \bigwedge_{p \in P} [p \Rightarrow \bigvee_{\psi \in \psi_I(p)} (\bigwedge_{n' \in \psi} n') \wedge \bigvee_{\psi \in \psi_O(p)} (\bigwedge_{n' \in \psi} n')]$, hence $v \not\models PCC(WF)$ (Contradiction). \square

A valuation of the transitions of a configurable workflow net is therefore correct if the conjunction of the configuration constraint and *PCC* can be satisfied. Checking the satisfiability is an NP-complete problem. To tackle this complexity, we propose to use a SAT solver⁴ based on Shared Binary Decision Diagrams (SBDDs). Existing SBDD solvers can easily deal with systems made up of around one million possibilities [121]. Hence they are reasonably adequate to handle all the configurations produced by a configurable process model.

During the staged configuration, we propose to use the solver to obtain a reduced representation of the conjunction of the configuration constraint and *PCC* in conjunctive normal form, where each variable is initially unset. For example, Figure 8.2 shows the constraints derived for the workflow net of Figure 2.2 (p. 22). Using the solver, we can reduce this constraint to its conjunctive normal form:

$$\begin{aligned} & \mathbf{PI} \wedge \mathbf{PO} \wedge (\overline{p_1} \vee t_1) \wedge (p_1 \vee \overline{t_1}) \wedge (\overline{t_1} \vee p_2) \wedge (t_1 \vee \overline{p_2}) \wedge (\overline{p_2} \vee p_3) \wedge (p_2 \vee \overline{p_3}) \wedge (\overline{p_3} \vee \\ & t_3) \wedge (p_3 \vee \overline{t_3}) \wedge (\overline{t_3} \vee p_4) \wedge (t_3 \vee \overline{p_4}) \wedge (\overline{p_4} \vee t_4) \wedge (p_4 \vee \overline{t_4}) \wedge (\overline{t_4} \vee p_5 \vee p_6) \wedge (t_4 \vee \overline{p_5}) \wedge \\ & (t_4 \vee \overline{p_6}) \wedge (\overline{p_5} \vee t_5) \wedge (p_5 \vee \overline{t_5}) \wedge (\overline{t_5} \vee p_6) \wedge (t_5 \vee \overline{p_6}) \wedge (\overline{p_6} \vee t_7 \vee t_6) \wedge \\ & (p_6 \vee \overline{t_7}) \wedge (p_6 \vee \overline{t_6}) \wedge (\overline{t_7} \vee \overline{p_7} \vee p_8 \vee p_9) \wedge (t_7 \vee t_6 \vee p_7) \wedge (t_7 \vee t_6 \vee \overline{p_8}) \wedge (t_7 \vee t_6 \vee \overline{p_9}) \wedge \\ & (t_7 \vee \overline{p_8} \vee t_{12}) \wedge (t_7 \vee \overline{p_8} \vee \overline{t_8}) \wedge (\overline{t_6} \vee \overline{p_7} \vee p_8 \vee p_9) \wedge (\overline{p_7} \vee t_2) \wedge (p_7 \vee \overline{t_2}) \wedge (\overline{t_2} \vee \overline{p_8} \vee \\ & t_{11}) \wedge (\overline{t_2} \vee \overline{p_9} \vee t_{11}) \wedge (t_2 \vee p_8 \vee p_9) \wedge (t_2 \vee \overline{t_{11}}) \wedge (\overline{p_8} \vee t_{10}) \wedge (p_8 \vee \overline{t_{10}}) \wedge (\overline{t_{10}} \vee p_9) \wedge \\ & (\overline{t_{10}} \vee t_{12} \vee t_8) \wedge (t_{10} \vee \overline{p_9} \vee \overline{t_{12}}) \wedge (t_{10} \vee \overline{p_9} \vee \overline{t_8}) \wedge (\overline{p_9} \vee t_9) \wedge (p_9 \vee \overline{t_9}) \wedge (\overline{t_9} \vee t_{13} \vee t_{14}) \wedge \\ & (t_9 \vee \overline{t_{12}}) \wedge (t_9 \vee \overline{t_{13}}) \wedge (t_9 \vee \overline{t_{14}}) \wedge (\overline{t_{12}} \vee t_{13} \vee t_{14}) \wedge (t_{13} \vee t_{14} \vee t_{11}) \wedge (t_{13} \vee t_{14} \vee \overline{t_8}). \end{aligned}$$

Then we conjunct this formula with each new transition valuation as provided by the user during the configuration process, and further reduce the formula. However, the solver can only reduce the formula if it is satisfiable, i.e. if the configuration can yield a syntactically correct process model. This may imply to automatically force to *true* or *false* the conjunction or disjunction of other transitions that are still unset, in order to keep the formula satisfiable.

For example, let us start the configuration of the process from Figure 2.2 with blocking the transition *Request for Domestic Travel Quote (Employee)*. Thus, the transition is assigned the value *false*, i.e. we concatenate the constraint above with $\wedge \overline{t_2}$ and again use the SBDD solver to reduce the constraint further:

⁴Available at <http://www-verimag.imag.fr/~raymond/tools/bddc-manual>.

$$\mathbf{p_I} \wedge \mathbf{p_O} \wedge \mathbf{p_1} \wedge \mathbf{t_1} \wedge \mathbf{p_2} \wedge \mathbf{p_3} \wedge \mathbf{t_3} \wedge \mathbf{p_4} \wedge \mathbf{t_4} \wedge (\overline{p_5} \vee t_5) \wedge (p_5 \vee \overline{t_5}) \wedge (p_6) \wedge (\overline{t_5} \vee t_7) \wedge (t_5 \vee \overline{t_7}) \wedge (t_7 \vee t_6) \wedge (t_7 \vee \overline{p_8} \vee t_{12}) \wedge (t_7 \vee \overline{p_8} \vee \overline{t_8}) \wedge \overline{\mathbf{p_7}} \wedge (\overline{p_8} \vee t_{12} \vee t_8) \wedge (p_8 \vee \overline{t_{12}}) \wedge (p_8 \vee \overline{t_8}) \wedge \mathbf{p_9} \wedge (\overline{t_{12}} \vee t_{10}) \wedge (t_{12} \vee t_8 \vee \overline{t_{10}}) \wedge (\overline{t_8} \vee t_{10}) \wedge \overline{\mathbf{t_2}} \wedge \overline{\mathbf{t_{11}}} \wedge \mathbf{t_9} \wedge (t_{13} \vee t_{14}).$$

As we can see, when illustrating the result in the net (see Figure 8.3) setting t_2 to *false* also causes setting the place p_7 to *false* as there is no longer any path that leads from p_I to p_7 , i.e. it cannot be reached from p_I any longer. As p_7 is set to *false*, also t_{11} must be set to *false* because p_7 is the only place in the pre-set of t_{11} , i.e. there is no longer a place in the pre-set of t_{11} that can be set to *true*. Hence, the blocking of the transition *Request for Domestic Travel Quote (Employee)* makes the complex process for international travels the only feasible way to get from p_I to p_O which must always be *true*. The only transition in the post-set of p_I which can actually cause p_I to be *true* according to the process correctness constraint *PCC* is t_1 . Thus, the truth value of t_1 is set to *true*. Together with this, the configuration constraint sets all the variables of places in t_1 's pre-set and post-set to *true*, i.e. p_1 and p_2 become *true* as well. Any path that leads from p_1 to p_O includes t_3 and p_3 , while any place from p_2 to p_O includes t_4 and p_4 . Thus, these nodes are automatically set to *true* as well. Furthermore, also p_6 , t_9 , and p_9 are on a critical path between p_I and p_O . All remaining nodes remain unset, as there are multiple paths from all of them to p_O or from p_I to them.

As hiding a transition does not change the value assigned to a transition, we can in the next configuration step hide the transition *Compare Travel Quotes (Employee)* without causing any further valuations of nodes in the net (see Figure 8.4). If we block in a third configuration step the transition *Request for Change (Admin)* as travel request should only be accepted or be rejected, t_{10} is set to *false*. Now, we do not need to re-evaluate the whole, original constraint, but we can add $\wedge \overline{t_{10}}$ to the reduced form of the constraint that was the result of the previous configuration step which results in the following constraint:

$$\mathbf{p_I} \wedge \mathbf{p_O} \wedge \mathbf{p_1} \wedge \mathbf{t_1} \wedge \mathbf{p_2} \wedge \mathbf{p_3} \wedge \mathbf{t_3} \wedge \mathbf{p_4} \wedge \mathbf{t_4} \wedge (\overline{p_5} \vee t_5) \wedge (p_5 \vee \overline{t_5}) \wedge \mathbf{p_6} \wedge (\overline{t_5} \vee t_7) \wedge (t_5 \vee \overline{t_7}) \wedge (t_7 \vee t_6) \wedge \overline{\mathbf{p_7}} \wedge \overline{\mathbf{p_8}} \wedge \mathbf{p_9} \wedge \overline{\mathbf{t_{12}}} \wedge \overline{\mathbf{t_8}} \wedge \overline{\mathbf{t_2}} \wedge \overline{\mathbf{t_{11}}} \wedge \overline{\mathbf{t_{10}}} \wedge \mathbf{t_9} \wedge (t_{13} \vee t_{14}).$$

Setting t_{10} to *false* thus results in p_8 becoming *false* as well (see Figure 8.5), and therefore also t_8 and t_{12} are set to *false*. However, note that p_5 still remains unset as it can be required if t_5 is allowed or hidden, but it should become *false* if t_5 would be set to *false* in further configuration steps.

If all transitions for which the variables are still unset should remain in the net, i.e. they are set to *true*, this configuration leads to the syntactically correct workflow net shown in Figure 8.6. Thus, as nodes that are automatically set to *true* by using and evaluating the depicted constraints cannot be configured as blocked any longer, a configuration decision leading to a syntactically wrong workflow net (like the one shown in Figure 8.1 at the beginning of this chapter) can be prevented.

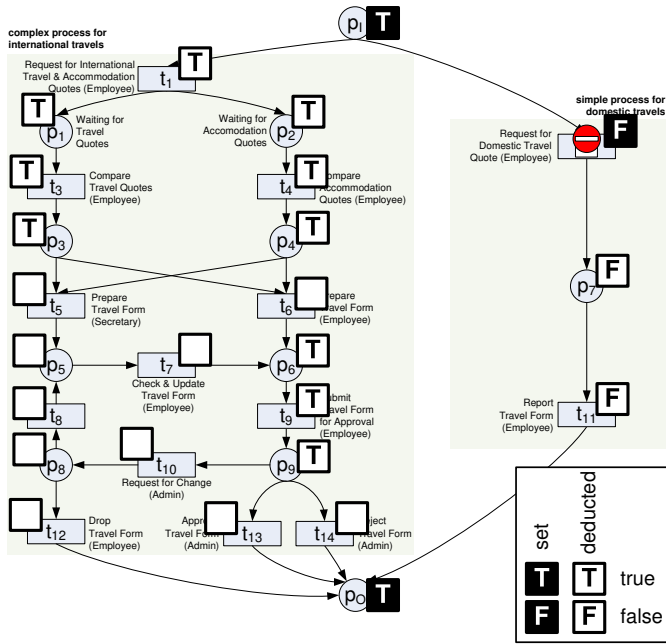


Figure 8.3: Configuration step 1: Blocking the transition *Request for Domestic Travel Quote (Employee)*.

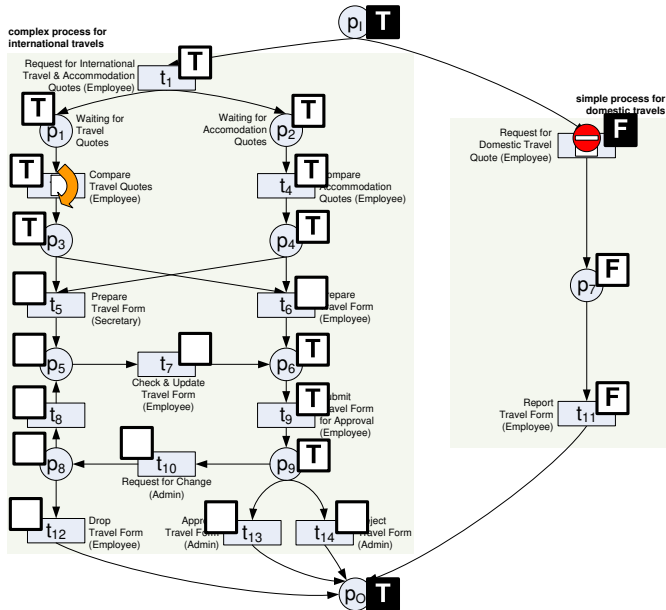


Figure 8.4: Configuration step 2: Hiding the transition *Compare Travel Quotes (Employee)*.

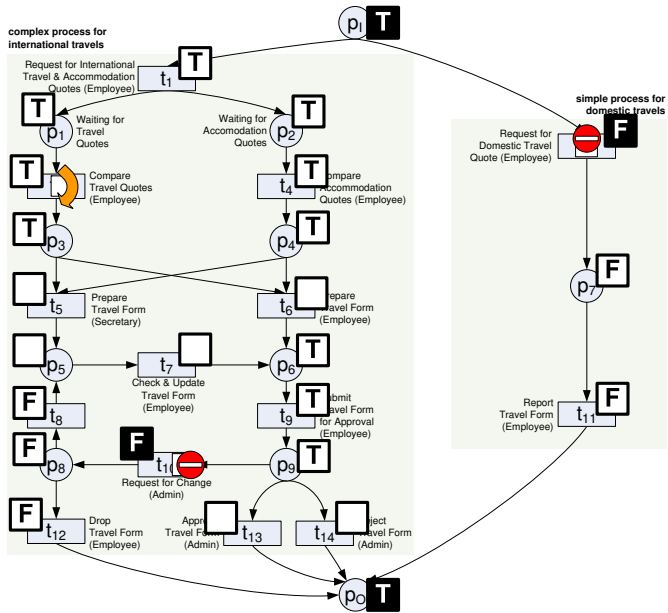


Figure 8.5: Configuration step 3: Blocking the transition *Request for Change (Admin)*.

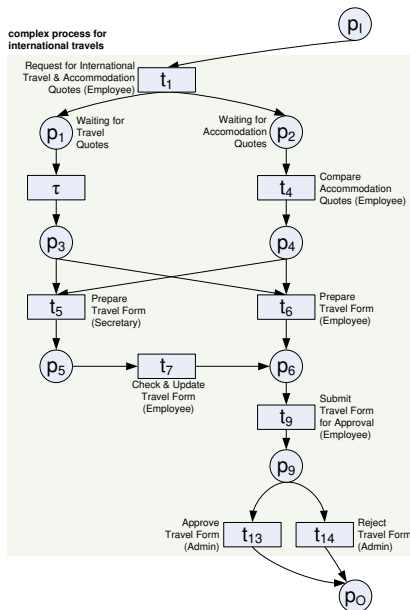


Figure 8.6: The workflow net resulting from the configuration shown in Figure 8.5.

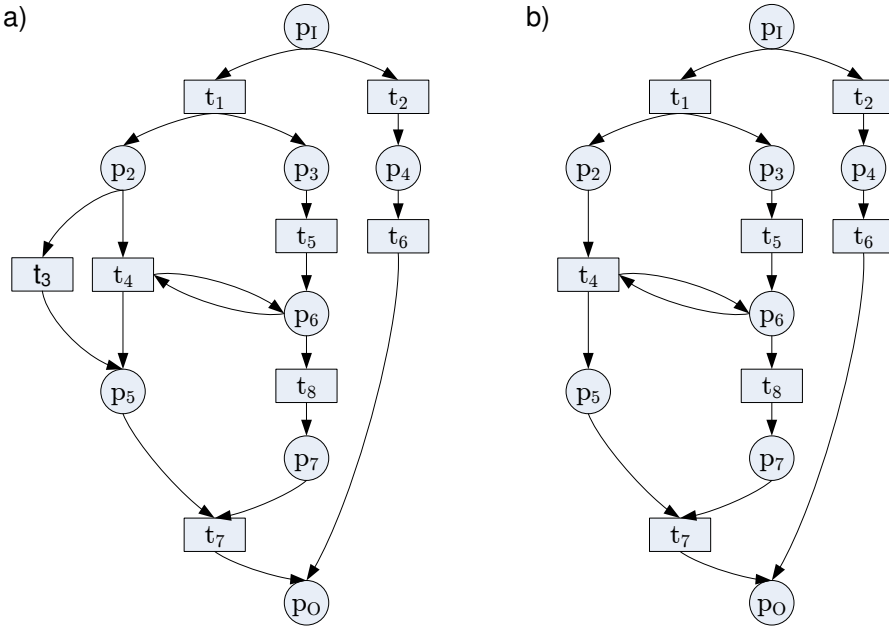


Figure 8.7: Blocking t_3 in (a) leads to an unsound workflow net (b)

8.2 Preserving Semantic Correctness

In addition to being structurally correct (i.e. being a workflow net), a configured process model must also be executable, i.e. sound. Not every model that is syntactically a correct workflow net is indeed sound, even if it is derived from a sound workflow net. The example in Figure 8.7 shows this. The workflow net in (a) is a sound workflow net: if p_2 and p_6 are marked, and t_8 fires before t_4 , the token in p_2 can reach p_5 via t_3 . However, if t_3 is blocked (b), t_4 needs to fire before t_8 as t_4 depends on the token in p_6 . However, this token is removed when t_8 fires before t_4 has fired. Since it is not enforced that t_4 indeed fires before t_8 , the process in (b) might deadlock. Thus, it is not sound, although (b) is still a syntactically correct workflow net.

While the example in Figure 8.7 shows that syntactic correctness is not a sufficient requirement for preserving soundness for all workflow nets, it can be shown that this is sufficient for a certain subclass of workflow nets, namely free-choice workflow nets (see Definition 2.20, p. 25). The restriction to this class of Petri nets often provides a good compromise between expressiveness and verification complexity. Not only do free-choice workflow nets have several desirable properties [54], but the most frequently used constructs of process modeling languages such as EPCs or BPEL can be mapped to Petri nets in this class. Thus, let us show here, how the process correctness constraint that preserves syntactic correctness for these models can also preserve their semantic correctness.

Assuming the basic process model which should be configured is a sound, free-choice workflow net, we are able to identify several configuration properties relevant for the preservation of soundness during the configuration process:

Proposition 8.1 (Properties of Configuration) *Let $WF = (P, T, A, L, l)$ be a sound, free-choice workflow net with source place p_I and sink place p_O , let \mathcal{C} be a configuration of WF , and let $WF^{\mathcal{C}} = (P^{\mathcal{C}}, T^{\mathcal{C}}, A^{\mathcal{C}}, L^{\mathcal{C}}, l^{\mathcal{C}})$ be the configured net resulting from \mathcal{C} . If $WF^{\mathcal{C}}$ is a workflow net (i.e. $PCC(WF)$ evaluates to true), then:*

- a) $\forall t \in T^{\mathcal{C}} [(\bullet^{WF} t = \bullet^{WF^{\mathcal{C}}} t) \wedge (t^{\bullet WF} = t^{\bullet WF^{\mathcal{C}}})]$,
- b) $p_I \in P^{\mathcal{C}}$ and $p_O \in P^{\mathcal{C}}$,
- c) $\forall t \in \{t \in T \mid \mathcal{C}(t) = \text{block}\} [(\bullet^{WF} t \cap P^{\mathcal{C}} = \emptyset) \vee \exists t' \in T^{\mathcal{C}} (\bullet^{WF} t = \bullet^{WF} t')]$ (a blocked transition is either not consuming any tokens from $P^{\mathcal{C}}$ or there is a transition in $T^{\mathcal{C}}$ with the same pre-set),
- d) $\forall \sigma \in T^{\mathcal{C}*} (M_I \xrightarrow{\sigma}_{WF} \Leftrightarrow (M_I \xrightarrow{\sigma}_{WF^{\mathcal{C}}}))$ (the pre-sets and post-sets of transitions in $T^{\mathcal{C}}$ are the same in both nets, therefore, the respective behaviors are identical when considering only firing sequences $\sigma \in T^{\mathcal{C}*}$),
- e) $\forall \sigma \in T^{\mathcal{C}*} \forall M [(M_I \xrightarrow{\sigma}_{WF} M) \Leftrightarrow (M_I \xrightarrow{\sigma}_{WF^{\mathcal{C}}} M)]$,
- f) $WF^{\mathcal{C}}[M_I] \subseteq WF[M_I]$ (all firing sequences of $WF^{\mathcal{C}}$ are also possible in WF),
- g) $WF^{\mathcal{C}}$ is free-choice,
- h) $\forall M \in WF^{\mathcal{C}}[M_I] \setminus \{M_O\} \exists t' \in T^{\mathcal{C}} [M[t']]$ ($WF^{\mathcal{C}}$ has no deadlock markings).

PROOF

- a) Follows directly from the construction of $WF^{\mathcal{C}}$.
- b) Idem.
- c) Suppose that some blocked $t \in T$, i.e. $\mathcal{C}(t) = \text{block}$, consumes a token from a place $p \in P^{\mathcal{C}}$ in WF . Because $WF^{\mathcal{C}}$ is a workflow net with source place p_I and sink place p_O , there has to be a path from p to p_O . Hence, there is a transition $t' \in T^{\mathcal{C}}$ consuming a token from p . Hence $\bullet^{WF} t \cap \bullet^{WF} t' \neq \emptyset$, thus $\bullet^{WF} t = \bullet^{WF} t'$ (because WF is free-choice and places are only removed if unconnected).
- d) Follows directly from (a).
- e) Follows directly from (d).
- f) Follows directly from (e).
- g) Let $t, t' \in T^{\mathcal{C}}$ such that $\bullet^{WF^{\mathcal{C}}} t \cap \bullet^{WF^{\mathcal{C}}} t' \neq \emptyset$. Given that $\bullet^{WF} t' = \bullet^{WF^{\mathcal{C}}} t'$ and $\bullet^{WF} t = \bullet^{WF^{\mathcal{C}}} t$, we have $\bullet^{WF^{\mathcal{C}}} t \cap \bullet^{WF^{\mathcal{C}}} t' = \bullet^{WF} t \cap \bullet^{WF} t' \neq \emptyset$. Hence $\bullet^{WF} t = \bullet^{WF} t'$ and thus $\bullet^{WF^{\mathcal{C}}} t = \bullet^{WF^{\mathcal{C}}} t'$. Therefore $WF^{\mathcal{C}}$ is free-choice.
- h) Let $M \in WF^{\mathcal{C}}[M_I] \setminus \{M_O\}$. Then using (e) we can deduce that there is a σ such that $M_I \xrightarrow{\sigma}_{WF^{\mathcal{C}}} M$ and $M_I \xrightarrow{\sigma}_{WF} M$, thus there exists a $t \in T$ such that $M[t]$ (as WF is sound). If $t \in T^{\mathcal{C}}$ then we are done. If $\mathcal{C}(t) = \text{block}$ then there exists a $t' \in T^{\mathcal{C}}$ such that $\bullet^{WF} t = \bullet^{WF^{\mathcal{C}}} t'$ (c). Therefore $M[t']$. \square

While propositions a, b, d, e and f follow directly from the construction of configured nets and also hold for workflow nets that are not free-choice, propositions c, g, and h are particularly interesting regarding the preservation of soundness during process configuration. The problem in the example of Figure 8.7 is that the configuration may yield an unsound model when a transition is blocked whose pre-set is also partly the pre-set of another transition (in this case t_3 shares p_2 in its pre-set with t_4 whose pre-set also includes p_6). By definition, in a free-choice workflow net such a situation cannot exist and therefore a deadlock marking cannot occur (as stated by propositions 8.1c and h). Furthermore, the deadlock in the example prevents all tokens from reaching the final place. But if a configured net is derived from a free-choice workflow net, also the configured net will be free-choice (Proposition 8.1g). The free-choice property, however, permits any token to freely move towards the final place. Thus, it prevents the problem occurring in the example (see also the works of Kiepuszewski et al. [105] and van der Aalst [2]).

Therefore, these properties allow us to prove that if a configured net which was derived from a sound, free-choice workflow net is still a workflow net, then it fulfills the soundness criteria. Formally:

Theorem 8.2 *Let $WF = (P, T, A, L, l)$ be a sound, free-choice workflow net with source place p_I and sink place p_O , let \mathcal{C} be a configuration of WF and let $WF^{\mathcal{C}} = (P^{\mathcal{C}}, T^{\mathcal{C}}, A^{\mathcal{C}}, L^{\mathcal{C}}, l^{\mathcal{C}})$ be the resulting configured net. If $WF^{\mathcal{C}}$ is a workflow net, then $WF^{\mathcal{C}}$ is sound.*

PROOF

- *proper completion*: since $WF^{\mathcal{C}}[M_I] \subseteq WF[M_I]$ (Proposition 8.1f), M_O is the only state marking p_O .
- *option to complete*: because $WF^{\mathcal{C}}$ is a free-choice workflow net (Proposition 8.1g), any token can decide to move towards p_O . If p_O is marked, all other places are empty ($WF^{\mathcal{C}}$ has proper completion). Hence, marking M_O can be reached (and the property holds) or the net is in a deadlock M . However, this is not possible as $WF^{\mathcal{C}}$ has no deadlock markings (Proposition 8.1h).
- *no dead transitions*: we define a length function as follows: $length : T^{\mathcal{C}} \rightarrow \mathbb{N}$. If $p_I \in \bullet t$ then $length(t) = 0$. Otherwise $length(t) = 1 + \min_{p \in \bullet t, t' \in \bullet p} length(t')$. Given that every transition in $WF^{\mathcal{C}}$ is on a path from p_I , the function is well-defined. Using induction we prove that for all $n \in \mathbb{N}$: $\forall_{t \in T^{\mathcal{C}}} [length(t) = n \Rightarrow t \text{ is not dead in } WF^{\mathcal{C}}]$.
 - *Base case*: If $n = 0$ then $\bullet t = \{p_I\}$ and as $p_I \in P^{\mathcal{C}}$ (Proposition 8.1b), $M_I[t]$, hence t is not dead.
 - *Induction hypothesis*: If $t \in T^{\mathcal{C}}$ is such that $length(t) = n + 1$, there exists a transition t' such that $length(t') = n$ and a place $p' \in t' \bullet \cap \bullet t$. t' is not dead (Induction Hypothesis), hence there exists a $M \in WF^{\mathcal{C}}[M_I]$ such that $M[t']$. Let M' be such that $M \xrightarrow{t'} M'$, then M' marks place p' . As $WF^{\mathcal{C}}$ has the option to complete, $M' \rightarrow M_O$.

This implies that some transition t'' exists which removes the token from p' in some marking M' , hence $p \in \bullet t''$. Therefore $\bullet t \cap \bullet t'' \neq \emptyset$, and thus, given that WF^C is free-choice (Proposition 8.1g) $\bullet t = \bullet t''$. Therefore $M'[t]$ and t is not dead. \square

Theorems 8.1 (p. 205) and 8.2 can be combined to show for free-choice workflow nets that a configured net is sound if and only if the process correctness constraint PCC is satisfied for the corresponding configuration. If the configured net is not a free-choice workflow net, PCC is still a necessary requirement as the syntax of a workflow net is the basis for the soundness property⁵. However, as the example from Figure 8.7 shows, it is not a sufficient requirement for such nets. The process correctness constraint PCC can in this case still be used during the staged configuration process to rule out any syntactically incorrect process models, but it is necessary to use conventional soundness analysis tools such as Woflan [182] to verify the executability, i.e. soundness, of the configured net.

8.3 Correctness in C-YAWL

In principle, the transformation from C-YAWL to YAWL as depicted in Section 4.3.5 (p. 91) ensures the same correctness in the configured process model as the process correctness constraint and the configuration constraint do for the configuration of workflow nets. In fact, what we described as cleanup algorithm in the second part of Section 4.3.5 (see p. 96) identifies and removes behavior in a comparable way to the blocking of transitions enforced by the two constraints.

The cleanup algorithm first identifies all flows and nodes in a YAWL model that are not on a path between the input condition \mathbf{i} and the output condition \mathbf{o} . This is exactly the same as what the process correctness constraint does: It assigns the configuration value *false* to those nodes of a workflow net that after the application of the other configuration decisions are no longer on a path between the workflow net's input place p_I and its output place p_O .

Thereafter, the cleanup algorithm for YAWL removes all those flows and nodes that are not a path from \mathbf{i} to \mathbf{o} . At the same time, it checks if any removed flow targets a task with an AND-join semantics or has its source at a task with an AND-split semantics. In these cases, removing only this single flow would change the behavior of the task. Hence, the cleanup algorithm removes all flows and the whole task even if the task would otherwise be on a path between \mathbf{i} and \mathbf{o} . This is in fact the same removal behavior as what is implied by the configuration constraint we introduced for configurable workflow nets in Section 8.1. The configuration constraint requires that if a transition (which has AND-join and AND-split semantics in workflow nets) remains in the net,

⁵Soundness is only defined for workflow nets (Definition 2.18, p. 24). Even if we would generalize it to any Petri net with a designated source and sink place, any node that is either not on a path from p_I or not on a path to p_O would cause the violation of at least one of the three soundness requirements. Thus, even in such a generalized soundness notation, PCC would still be a necessary requirement.

i.e. is assigned the value *true*, then all the subsequent places (and thus also the paths to them) must remain in the net as well, i.e. *true* must be assigned to them as well. The other way around, this means, if we want to set the value of one of these places to *false* because it is not on a path between p_I and p_O , then we have to set the transition itself to *false*, i.e. we have to remove the transition — exactly as the cleanup algorithm does in YAWL.

Next, the cleanup algorithm checks again if all nodes that remain are still on a path between **i** and **o** and repeats the whole removal procedure until all nodes are on such a path after the removal of elements with AND-join or AND-split semantics. Thus, it stops when both criteria are fulfilled. This is exactly what we required for the correctness of a configured workflow net: both the configuration constraint and the process correctness constraint must be fulfilled at the same time. The cleanup algorithm therefore guarantees syntactic correctness for YAWL models.

Furthermore, Theorems 8.1 and 8.2 show that we can also apply process correctness constraints and configuration constraints as process constraints in the context of the configuration framework from Chapter 5. In Section 4.3.5 (pp. 91ff), we suggested to use the cleanup algorithm to automatically enforce correctness of configured YAWL models. That means, even if a process configuration that results from answering a questionnaire allows the use of tasks, the cleanup algorithm removes them if they harm the correctness of the configured process. Thus, the cleanup algorithm works independently of the constraints imposed on a configurable YAWL model.

By adding (YAWL versions of) process correctness constraints and configuration constraints to the process constraint of a C-YAWL model, we could however establish a feedback link from the process configuration on the domain configuration through the mapping of the configuration framework (see Figure 5.4, p. 111). If setting a domain fact, i.e. answering a question, implies the blocking of a task, the process correctness constraint is then able to identify which other tasks must be removed as well and sets the corresponding process facts to *false*. Thus, through the framework no domain fact (or combination thereof) can set this process fact to *true* anymore, and the questionnaire will inhibit any corresponding answer(s). In this way, it becomes impossible that the domain expert selects options in the questionnaire, which will never be available because they are automatically removed by the cleanup algorithm.

8.4 Constraints from Resource- and Data-flows

So far, we only looked on the correctness of a configured process model from a control-flow perspective. Thus, it was possible to assume that the only dependency of tasks is that they have to be potentially used according to the process flow. As long as this criteria is fulfilled, a task could be included in the process model, i.e. it was possible to set the corresponding variable to *true*. Most tasks, however, are not only dependent on the execution of previous tasks, but also on the availability of some resources and some data. For example, the approval of

a travel request, requires the presence of a person with the necessary authorization to approve the request which also needs to have the corresponding, filled-in form available. Therefore, we have to analyze in the following in which ways process configuration might prevent the availability of necessary resources and data and, as a result, a successful process execution.

Note, that we do not look into the configurability of data or resources themselves here, i.e. we do not analyze if we could change the resources involved in the execution of a task, or the data that is necessary for a task execution. Readers interested in ideas on the configurability of these perspectives should have a look at the work of La Rosa et al. [112].

8.4.1 Data-flow Correctness

Trčka et al. [179] identified and formalized a set of nine data-flow anti-patterns, i.e. patterns in the modeling of the data-flow which potentially result in problems or errors in the execution of business processes. In [179] these are formalized in temporal logic (Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and CTL*). To identify potential issues that can arise in the data-flow after a process model has been configured, let us thus in the following analyze these patterns from a configuration perspective.

For simplicity, we assume here three different ways how a task of a process model can handle data: it can create (write), read, or delete data.

Definition 8.3 (Anti-pattern 1: Missing data [179])

Data is missing if some data needs to be read which has either never been created or which has been deleted without having been re-created.

The missing data anti-pattern describes a situation that must be avoided in any process model as the need to access data which does not exist leads almost for sure to issues like undefined data values or the deadlock of the process. There is a true dependency that data can only be read if it has been created beforehand. As configuration does not add any new tasks to a process which could create data, any data that is read according to a task of the basic process model must also be created by a task which is according to the basic process model executed beforehand. If there is no task creating the necessary data, there is no scenario in which the task could be executed, meaning it should not be added to the process in the first place. Hence, the missing data anti-pattern should not occur in the basic process model.

Furthermore, it should of course not occur in any configured model. Hence, during the process configuration it is not permitted to remove any task from the basic process model which creates data that is subsequently read by other tasks, unless either alternatives to the removed task guarantee that the required data is created before the data is read, or the ‘reading’ task is removed as well.

If a task is configured as blocked during the process configuration, any process execution needs to follow an alternative to the path via this task. These alternative paths already exist and can be followed in the basic process model because — as mentioned before — no additional process branches are added

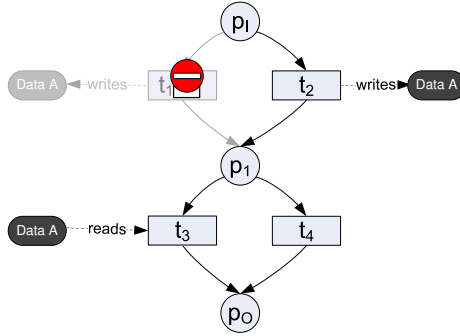


Figure 8.8: Blocking a task does not lead to missing data.

when configuring the model. Hence, if an alternative path leads to a task which reads data that is created by the blocked task, then this data is also created along the alternative path (e.g., task t_2 is an alternative to task t_1 in Figure 8.8). Thus, the blocking of a task's execution cannot lead to missing data during the execution of the corresponding process.⁶

If a task is configured as hidden, the task's execution and thus any creation, reading, or deleting of data by this task is skipped. The process execution continues after the task, simply omitting the task (without executing alternatives) and continuing with the subsequent tasks. However, due to the lack of executing alternative tasks, the availability of data usually created by the now hidden task is no longer guaranteed. This leads to missing data if a subsequent task needs to read exactly this data. For example, the hiding of task t_1 in Figure 8.9a can result in missing data at t_4 in case an execution path via t_2 is chosen. Thus, if a task which creates data should be configured as hidden, we need to check if any subsequent task accesses this data. If this is the case (like for t_4) and there is no alternative task which produces the data and which is guaranteed to be executed (in the example, t_3 is not guaranteed to be executed), then there are two options to ensure the absence of missing data: either the 'reading' task's execution is prevented as well, i.e. it is configured as hidden (like in Figure 8.9b) or blocked, or any execution sequence that leads to the 'reading' task is prevented through blocking one or more tasks along this sequence (like in Figure 8.9c).

Definition 8.4 (Anti-pattern 2: Strongly redundant data [179])

Data which is created during the process execution is strongly redundant if in all possible continuations of the process it is never read again.

It is not necessary to create strongly redundant data as it is never used later on. As we do not add any tasks which could read redundant data during the process configuration, there is no reason that the basic process model of the configurable process model should contain tasks that create strongly redundant data.

⁶Note, that this does not hold the other way, i.e. it might be necessary to block particular tasks to deal with missing data.

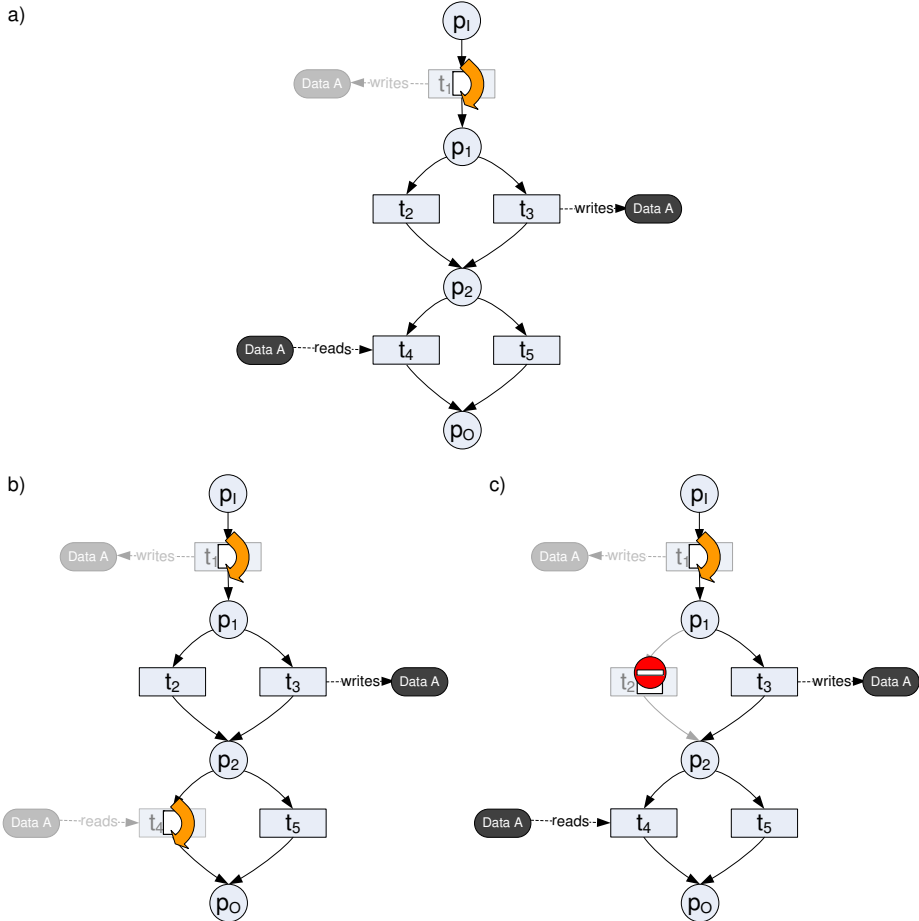


Figure 8.9: Hiding a task might lead to missing data (a) which can be resolved by eliminating the reading task(s) (b) or blocking all paths that do not create the necessary data in time (c).

As process configuration eliminates the execution of tasks, all the tasks that read a certain data element can be eliminated while the task that creates it remains in the net. Then, the created data becomes strongly redundant. Although the creation of such data is unnecessary, and a hiding of the tasks which creates the data might be recommended, strictly speaking, such a configuration does not prevent a successful execution of the process. Also, a configuration of the data perspective as suggested by La Rosa et al. [112] could be recommended here. However, as the execution of the process itself is not endangered, we do not need to enforce these suggestions through configuration constraints.

Definition 8.5 (Anti-pattern 3: Weakly redundant data[179])

Data which is created during the process execution is weakly redundant if in some continuations of the process it is never read again.

The creation of weakly redundant data can be a design decision to standardize a process with various outcomes. Thus, we will not ask for avoiding weakly redundant data in the basic process model or in one of its configurations. Through configuration, weakly redundant data of the basic process model can become strongly redundant if all tasks accessing the data are eliminated from the process. Then, what we discussed above for data that becomes strongly redundant through configuration holds.

Definition 8.6 (Anti-pattern 4: Strongly lost data [179])

Data is strongly lost if data is created, but never read before it is overwritten by another task in all possible continuations of the process.

Although data that is overwritten without being read does not need to be created during a process execution this anti-pattern might well be desired in the basic process model to avoid missing data in configured nets. If a configuration eliminates one of the tasks which create the data, the other data-creating task can still guarantee the absence of missing data.

If strongly lost data can still be found in the configured net, the first data creation is obviously superfluous. Hence, like for strongly redundant data, a hiding of the first data-creating task or a configuration of the data-flow of this task might be recommended. But, again, strongly lost data does not hinder the process execution itself and thus its absence does not need to be enforced through configuration constraints.

Definition 8.7 (Anti-pattern 5: Weakly lost data [179])

Data is weakly lost if data is created, but never read before it is overwritten by another task in some continuations of the process.

Like strongly lost data, weakly lost data might be introduced into a basic process model to guarantee the existence of certain data in some configurations of the process. Moreover, like for weakly redundant data, it might be a design decision to standardize or guarantee the presence of data by first loading it, but updating it later on if certain conditions apply. Thus, weakly lost data is usually not a problem in the basic process model or in one of its configurations. Of course, weakly lost data might become strongly lost data if all tasks reading the data in between the data creation tasks are eliminated from the net through configuration. Then, obviously the remarks for strongly lost data from above apply.

Definition 8.8 (Anti-pattern 6: Inconsistent data [179])

Data is inconsistent if a task is reading data while some other task instance is writing to this data in parallel.

Inconsistent data is problematic because it remains open if the reading of the data happens before or after the writing, i.e. there is ‘race condition’ making it unclear whether the new or the old values are read [80, 85]. For example, have a look at Figure 8.10a. Here it is completely random, if t_3 reads the data value that was created by t_1 or the data value that was created by t_2 . Hence, a process model should always synchronize the parallel process branches and

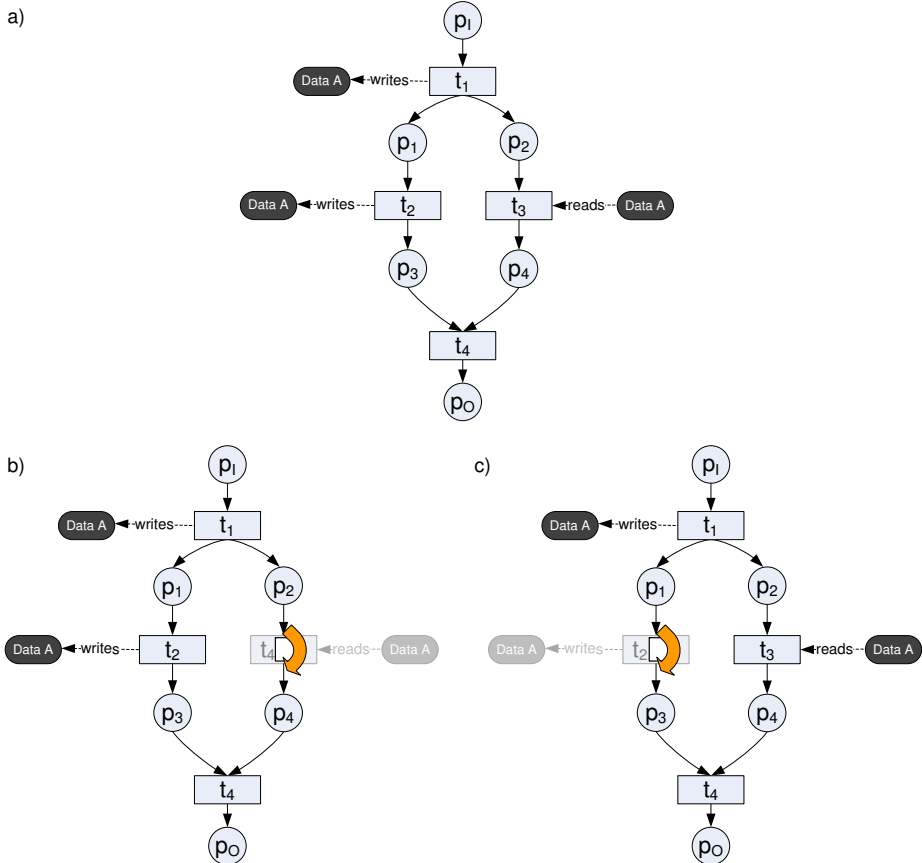


Figure 8.10: Inconsistent data in the basic process model (a) can be eliminated by process configuration (b or c)

thus determine a clear order. If a basic process model contains inconsistent data, this inconsistency can be eliminated by process configuration. For this, either the task reading the data must be configured as blocked or hidden as, e.g., in Figure 8.10b, or the task writing the data must be configured as blocked or hidden as, e.g., in Figure 8.10c. In the first case, the reading of data is eliminated in the configured process model while in the latter case the writing is eliminated. Hence, if we allow inconsistent data in a basic process model, a configuration constraint must guarantee one of these configurations. If we do not allow inconsistent data in the basic process model, the configured process model is guaranteed to have no data inconsistency as we cannot add any process behavior during the configuration, and thus also no parallel behavior.

Definition 8.9 (Anti-pattern 7: Never deleted data [179])

Data is never deleted if there is a process execution where data is created but not deleted before the process completes.

While never deleted data might be considered as garbage that is left behind after the completion of a process, it does not influence if a process can complete and thus its executability. Hence, we do not consider this anti-pattern for configuration.

Definition 8.10 (Anti-pattern 8: twice deleted data [179])

Data is deleted twice if data that was already deleted should be deleted without having been re-created in the meantime.

This pattern is similar to the missing data pattern as accessing data that was already deleted is not possible, and hence it cannot be deleted again. However, considering that the outcome of any deletion should be the non-existence of the data afterwards, one could consider a deletion of data as ‘delete the data if it exists’. Then, multiple deletion of data is unnecessary, but not problematic. Also, similar as strongly lost data which is created twice, twice deleted data could be part of a basic process model to make sure that the data is deleted even if one of the deletion tasks is eliminated from the process model by process configuration.

Definition 8.11 (Anti-pattern 9: Not deleted on time [179])

If data is read for the last time and there is no potential further reading of this data, the data should be deleted immediately. Otherwise, it is not deleted on time.

While an on-time deletion of data aims at storing the minimal necessary amount of data, a late deletion does not hinder a successful process execution as long as sufficient storage space is available. A late deletion of data might be the result of process configuration: if a task reading the data is eliminated from the process by configuration, the data could be deleted after the previous reading already. This could only be avoided by giving the potential of deleting the data to all tasks that might become the last ones to access the data, and configuring the data perspective such that the data is deleted as soon as it will for sure not be accessed in the future (see [112] for the configuration of the data perspective).

All in all, *configuration constraints only need to be introduced to enforce the absence of the missing data anti-pattern and the inconsistent data anti-pattern during the process configuration.*

To verify if an anti-pattern occurs in a process model temporal logic (LTL, CTL, CTL*) can be used [179]. Temporal logic extends propositional logic with temporal operators to address the changing of variable values over time, i.e. in our case over the progress of a process execution. For example, it can be checked in this way, if a property will hold after the next step of the process execution, if it will always hold during a process execution, or if it will eventually hold at some point in time during the process execution.

In this way, Trčka et al. [179] show for example that a missing data element d conforms to an execution path along which d is either not written before being read, or along which it is deleted and not written again before being read

(Definition 8.3). This can be formally expressed in CTL* as

$$E[(\overline{\text{write}(d)} \text{ U } \text{read}(d)) \vee F[\text{write}(d) \wedge (\overline{\text{write}(d)} \text{ U } \text{read}(d))]].^7$$

A model checker [41] can use this formula for testing if a model contains missing data. For this, it requires not only the temporal logic formula, but also the model on which it should be tested. As the goal is that missing data is avoided in the configured process model, this process model must be the configured process model. For that reason, checking for violations of data-flow constraints can only happen after the model has been adapted according to the process configuration and corresponding tests should only be performed if the process correctness constraint has guaranteed control-flow correctness of the derived model.

8.4.2 Resource Availability

A task of a process can only be executed if all required resources like employees or machines are available. Hence, to successfully execute a process, it must be guaranteed that a sufficient amount of the required resources is available. Both, van Hee et al. [88] and Prisecaru [131] developed notions of soundness for such resource-constrained workflow nets. If these soundness notions are fulfilled, the workflow net is guaranteed to be executable under the imposed resource constraints. Like we required that any basic process model should fulfill the control-flow soundness criteria, we thus also require that if a basic process model is resource-constrained, it should fulfill the soundness requirements of resource-constrained process models.

As we aim at guaranteeing the executability of configurations of the basic process model, let us have a look how process configuration influences the availability of resources. That means, we are not looking at adjusting the resources involved in executing tasks, but rather look at the influence of process configuration on resource assignments.

For this, we assume that any tasks of the basic process model specifies and claims the necessary resources when it starts being executed, and it releases them after its completion (like we showed in figures 2.3 to 2.6 (pp. 26f), also see Figure 8.11a). This is reasonable as any resource is only busy with executing tasks as long as these tasks are executed. If resources are supposed to perform several tasks in a row, one can simply specify that all these tasks should be executed by the same person in the resource requirements.

If we now block the execution of a task, the task will not be executed. Hence, it will not claim any resources, i.e. the resources remain available. Thus, the claiming of these resources by any other task that requires them before or after the now blocked task according to the basic process model is still able

⁷‘E x ’ means that there exists at least one path from the current state where x holds, ‘F x ’ means that x has to hold eventually, i.e. somewhere on the subsequent path (finally), and ‘ x U y ’ means that x has to hold in the subsequent path until at some point y holds. Further details on temporal logic, CTL*, and model checking can, e.g., be found in the work of Emerson and Halpern [63] and Clarke et al. [41].

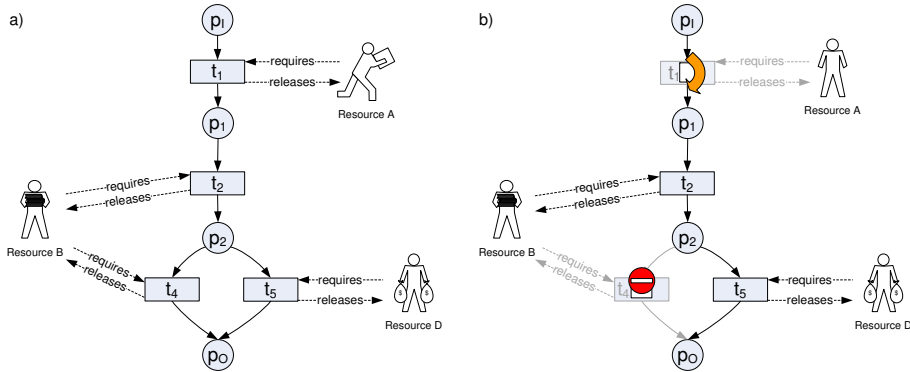


Figure 8.11: As long as sufficient resources are available for executing the basic process model (a), process configuration can not lead to a lack of resources in the configured model (b). Instead, it might lead to superfluous resources (resources A, C in (b)).

to do so (see Task t_4 in Figure 8.11b). The same holds if we configure a task as hidden: the process continues without claiming and releasing the required resources which remain available (see Task t_1 in Figure 8.11b).

For that reason, process configuration may lead to additional availability of resources (e.g., Resource A in Figure 8.11b does not have to do any work anymore). This might result in efficiency issues, but it is no problem from a process executability point of view. Therefore, we do not have to set up any additional configuration constraints for guaranteeing the availability of resources here. To deal with adjusting the resource requirements of tasks that remain in the net and the dependencies between them (and therefore to deal with the efficiency issues caused by process configuration), a configuration of the resource perspective itself would be required. For this, the interested reader should again look into the work of La Rosa et al. [112]. Initial ideas in this context can also be found in [72].

8.5 Related Work

In this chapter we addressed the feasibility of process model configuration, i.e. the executability of configured process models. The discussed topics are mainly related to research performed on the correctness of process models known as soundness from both the control-flow perspective as well as from a data-flow perspective. As noted earlier, there is surprisingly little related work on sound process models from a resource perspective [88, 131].

8.5.1 Soundness from a Control-flow Perspective

Soundness of process models as we have already introduced in Definition 2.18 (p. 24) was originally defined by van der Aalst [3]. Verbeek et al. [182] suggest a range of efficient techniques that can be used to test a process model on the

soundness property. They have been implemented in the process model verification tool Woflan. While the approach which we suggested in the first two sections of this chapter specifically aims at testing the soundness of the nets resulting from configuration decisions without deriving the particular net, the approach of Verbeek et al. was developed to test a specific net on its soundness. Thus, using Woflan the validity of a configuration decision can only be determined *after* deriving the configured net. Still, the usage of Woflan to test the correctness of configured nets has the advantage that it can guarantee semantic correctness for non-free-choice nets, while our approach only constitutes a necessary, but not a sufficient condition for the soundness of such nets.

The technique suggested here derives propositional logic constraints from process models. Similar techniques have been used for analyzing the coverability of Petri nets by Abdulla et al. [19], while Sadiq et al. [154] ensure that a specific process model can only be adapted within a certain freedom which is defined by the process developer through such constraints. Different from this specific, individual definition of constraints for a particular purpose, the constraints derived here aim at checking that any configuration step preserves the structural properties of any workflow net.

Pesic et al. [126] even go yet another step further and suggest defining the whole workflow purely based on constraints formulated in LTL. In this approach no process model in the classical sense is defined, i.e. only constraints are given in a graphical way. As long as none of the defined constraint restricts the execution of a task, its execution is allowed.

8.5.2 Soundness from a Data-flow Perspective

We based the analysis of issues that can occur from a data-flow perspective during process configuration on the data-flow anti-patterns of Trčka et al. [178, 179]. Some initial ideas to detect data dependencies in workflow nets have also been suggested by Huang et al. [93]. The ADEPTflex approach of Reichert and Dadam [136], which aims at ensuring data consistency when workflow are dynamically changed during the process's execution, is based on two rules that preserve the executability of workflows: (1) any data must be written before it can be read, and (2) tasks from different process branches which are executed in parallel should not write to the same data element. These rules therefore relate to the missing data and inconsistent data anti-patterns of Trčka et al.

Moreover, all these patterns and rules strongly relate to the problems of the read-after-write, write-after-read, and write-after-write data hazards which occur in instruction pipelines of modern computer processors that execute multiple instructions at the same time and thus also have to deal with data dependencies (further details on data hazards can, e.g., be found in the book of Hennessy and Patterson [89]). This is therefore a very relevant subject in the design of compilers for computer programs which should efficiently run on modern processors with multiple cores, aiming at the parallel processing of instructions. Detailed insights into how modern compilers deal with these data dependencies can, e.g., be found in the book of Kennedy and Allen [104]. The issues found here are also

similar to the ‘lost update’ problem in database theory and therefore relate to the ACID (atomicity, consistency, isolation, durability) principle for database transactions addressed by Gray [80] and Haerder and Reuter [85].

8.6 Conclusions

While the configuration of process models enables a well-defined way of restricting process behavior, not every configuration decision leads to a meaningful and still executable process. Thus, we discussed in this chapter how process configuration can be constrained such that it is forced to lead to ‘good’ process models. We checked whether the derived process model preserves a correct syntax, whether it still depicts executable behavior (i.e. is sound), whether all necessary data can still be processed, and whether the execution of the process model behaves in an intended way content-wise.

The correct syntax of a process model can be guaranteed through evaluating a propositional logic expression that is derived from the paths of the basic process model. In a staged configuration process, each configuration decision then corresponds to assigning a truth value to a propositional letter which reduces the overall formula. It automatically results in further configuration decisions if the syntactic correctness of the resulting process model can otherwise not be guaranteed any longer.

For C-YAWL, the transformation algorithm that forms new YAWL models according to configuration decisions already includes an implementation of the depicted constraints. Thus, for YAWL models derived from C-YAWL models syntactic correctness is guaranteed.

Furthermore, it was shown that for free-choice workflow nets (and any other process modeling language that can be mapped on such nets), this syntactic correctness also guarantees semantic correctness, i.e. the preservation of soundness of the process model. For non-free-choice models, it is a necessary but not a sufficient requirement. Hence, an additional verification with tools like Woflan [182] which are capable of testing the soundness of such models, is required for these types of models.

The configuration of a process model can also lead to missing data in case a task is eliminated from the process model which creates data, later required by other tasks. To detect missing data in a process model, temporal logic can be used. Thus, any configuration in which the corresponding check evaluates to *true* must be forbidden. Furthermore, a basic process model can purposefully contain inconsistent data, i.e. it allows for a reading and writing of the same data in independent, parallel process branches. Also, this can be detected by evaluating corresponding temporal logic formulas. In such cases, process configuration must resolve this conflict by eliminating either the reading tasks or the writing tasks.

*In politicall affaires, as well as mechanicall,
it is farre easier to pull downe, then build up.
James Howell (1644)*

Chapter 9

Conclusions

The aim of the research presented in this thesis was to improve the support for process model reuse through developing a better understanding of process model configuration. In this final chapter an overview of the results obtained is provided first (Section 9.1). Afterwards, Section 9.2 discusses both limitations of the used approach and suggestions for future work. For this, the section reflects on practical implications of process model configuration for the reuse of process models. The chapter ends by summarizing the most important conclusions for the use of configurable process models (Section 9.3).

9.1 Contributions

This thesis contributed to the goal of providing better support for process model reuse by providing a sound definition of process model configuration which led to the definition of configurable workflow modeling languages as a technique to provide dedicated process adaptation support. A framework that maps natural-language questionnaires to process configuration decisions was suggested in order to simplify process model configuration for domain experts. As the construction of such configurable models is far from trivial, a number of techniques supporting the construction of such models were discussed. This also resulted in the development of a new algorithm to merge multiple process models into a single model. Process configuration restricts the behavior allowed according to a process model. Hence, when configuring a process model, it is easy to implicitly inhibit more behavior than desired. To counter this, a boolean expression over process model nodes which guarantees sound process models was suggested.

Figure 9.1 provides an overview of how these techniques contribute to the construction and use of configurable process models. The key contributions are summarized in the following sections.

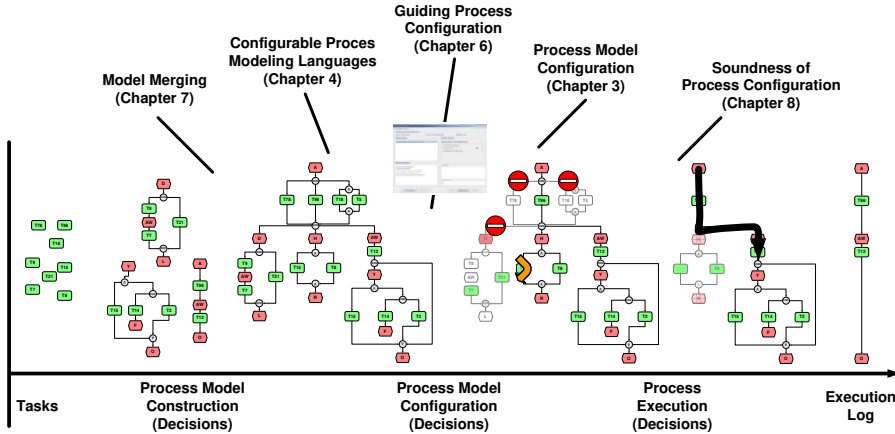


Figure 9.1: Contributions of this thesis for the construction and use of configurable process models.

9.1.1 Process Model Configuration

When configuring a process model, the behavior that is allowed according to the model is restricted, i.e. non-desired behavior is eliminated. While other approaches derive process configuration options based on observed needs for restricting the process behavior, we showed in Chapter 3 that eliminating behavior can be seen as the inverse of adding behavior. For that reason, process configuration can use techniques that have been defined for identifying inheritance relations among dynamic behavior. These techniques compare the behavior of two models and identify exactly that behavior that is added to a process model in comparison to the other process model. The two techniques to detect added behavior, called **blocking** and **hiding**, can thus also be used to remove added behavior. Therefore, they can be used for configuring process models. While blocking encapsulates behavior, i.e. inhibits it completely, hiding abstracts from behavior, i.e. it ignores that certain behavior has to be executed, quasi skipping it. When configuring process models, it is thus needed to analyze how these two operations can be applied in the corresponding process modeling language.

9.1.2 Configurable Process Modeling Languages

As blocking and hiding can be used to restrict any process behavior in any process model, any process model can serve as the basis for a configurable process model. However, as the configuration operations can inhibit all behavior in such a model, they can even inhibit more behavior than desired. Hence, we suggested in Chapter 3 that a configurable process modeling language must be able to limit the possible configuration space through configuration constraints. These constraints restrict the allowed application of blocking and hiding to desired configuration opportunities only.

In practice, the basic process model of a configurable process model is a model that contains the behavior of several process variants. Usually, it is not desired to execute all these variations in a single organization. Thus, by providing the opportunity to define a default configuration, a process model that is similar to a best-practice reference model can be provided along with any configurable process model. Configuring the process model then solely means changing configuration decisions where individual requirements deviate from the standard solution.

To extend an existing process modeling language with configuration options, one must identify how the operators of blocking and hiding behavior can be applied to the particular language. In Chapter 4, this was outlined for the workflow languages of SAP WebFlow, BPEL, and YAWL. A formalization of the concepts for C-YAWL allowed its implementation in the YAWL system environment, and thus an evaluation of the ideas in practice.

For this, a case study was performed and presented in Chapter 6. During the case study, the process variations among four Dutch municipalities of executing registration processes were identified, and a single process model that supports all these variations plus the suggestions of a best-practice reference model of these processes was built. Through process configuration it is possible to derive any of the individual process model variants from these models. For example, the configuration that results in a process model equivalent to the reference model may in this way be selected as the default configuration. By choosing YAWL for the implementation of the processes, i.e. a workflow notation rather than a less formal business process modeling language, the configured workflow models can directly steer the execution of the particular processes using YAWL's workflow engine.

9.1.3 Guiding Process Configuration

Chapter 5 introduced a framework that allows steering the configuration of process models through a natural-language questionnaire. This allows domain experts to adapt a reference process to individual needs without having to verify the correctness of the models resulting from process configuration decisions. In fact, they do not even need to understand the underlying process modeling notation.

In the questionnaire, the domain variability can be captured independently of any process complexity through questions and pre-defined answering options. The various answers are then mapped onto process configuration decisions. Each configuration decision can depend on one or more answers. Similarly, each answer can influence one or more configuration decisions. Through the mapping, constraints among answering the domain-related questions are added to the constraints that preserve the correctness of the configured process model. In this way, the questionnaire is able to identify remaining configuration options and thus offers only those questions and answering opportunities that lead to a correct process configuration.

The framework has been implemented for steering the configuration of C-YAWL models. This enabled the development of questionnaires for the configurable process models created in the case study presented in Chapter 6. In subsequent expert interviews, stakeholders in configurable process models were enthusiastic about using both configurable models and the configuration framework. In their opinion, the techniques visualize the variation options in such a way that they are easily understandable by domain experts. Their main concerns related to the effort required to build models integrating different process variants as well as to the models' completeness.

9.1.4 Model Merging

A range of tools that could help in reducing the effort required for constructing basic process models was discussed in Chapter 7. The tools aim at automatically integrating various process variants into a single process model. The use of process mining techniques on a set of log files from various existing systems executing the process in question was sketched, as well as an algorithm to merge existing process models was developed. The suggested algorithm preserves all behaviors of all individual models in the generated model. For this, it makes use of the capabilities of OR-splits and OR-joins when the individual models do not agree on a clear AND or XOR behavior. These constructs basically allow the execution of any combination of subsequent behavior or synchronize such behavior. In addition to the behavior of the individual models, the models constructed by the merge algorithm also allow for combining the behaviors from different individual models in a single process executions. Hence, the merged model allows for more behavior than the sum of the individual models, i.e. the resulting model generalizes behavior where appropriate. As this additional behavior can be restricted again by process configuration, these properties are desired when constructing configurable process models.

Both the mining of basic process models from log files as well as the merge algorithm for individual models have been tested on the same process models that served as input to the case study presented in Chapter 6. In both cases the automatically generated models were identical to the models that were generated manually during the case study. This shows that the depicted algorithm can indeed be beneficial for the construction of practically relevant configurable process models.

9.1.5 Soundness of Process Configuration

When restricting process behavior through process configuration, one can easily inhibit the execution of essential tasks. Then, process executions might get stuck in deadlocks or livelocks. The notion of sound process models ensures the absence of such control-flow issues, i.e. every case that is executed according to the process model has the chance to complete. In Chapter 8, we therefore discussed how a configuration constraint can guarantee that soundness is preserved while configuring a process model. The boolean constraint suggested proved to

preserve soundness for a sub-class of workflow nets called free-choice nets. The most frequently used constructs of languages like EPCs, BPMN, or BPEL can be mapped to this type of nets. Thus, the constraint can be applied to guarantee soundness of configured process models in any such process modeling language. For any other process model, the constraint is still a necessary requirement but in itself not sufficient. Then, advanced verification tools are required.

Besides control-flow soundness, the preservation of a correct data-flow as well as efficient and effective process executions are similarly important. Hence, Chapter 8 also pointed out that process configuration can lead to missing data, which can be detected through LTL expressions.

9.2 Limitations and Future Work

By focusing on process configuration techniques and tools, *the research presented in this thesis is limited to technology that restricts process behavior*. Still, providing process models with the aim that these models should be reused requires much more than just process configuration techniques. Hence, we need to put process configuration in the context of practical process model reuse in general. Let us therefore discuss in this section both the limitations of this thesis's approach as well as ideas for future research opportunities that connect process configuration to related challenges of process model reuse. In this context, four aspects are especially important:

- Besides process configuration, a number of further techniques for process model adaptation exist, all aiming at improving some aspects of process model reuse. In order to improve process model reuse further in the future, the interplay of process configuration with these existing techniques must be enhanced.
- Configurable process models provide plenty of variation opportunities how a process can be configured. To make a sensible decision on the optimal configuration for a particular context, the user therefore needs further support on the distinct characteristics and the impact of the various configuration options.
- Throughout the thesis we examined the use of configurable process models from the model construction to the execution of individual instances. However, a business process — and thus also a process model — has to be adjusted to changing business environments. That means, neither a configuration nor a configurable process model remains unchanged over time. For that reason, the role of process configuration within the process model life cycle has to be investigated.
- Last but not least, while the focus of this thesis was on technologies, no process model will be reused if it does not provide content that is worth being reused. Hence, this section concludes with pointing out the need for good-quality content in configurable process models.

9.2.1 Adapting Configured Models

Process configuration allows for reusing what is defined in the basic process model. Thus, different model variants can only be achieved from a configurable process model if all the variation options have been integrated within the model beforehand. A process model that depicts behavior which is not defined in the basic process model cannot be created purely through process configuration as defined in this thesis. For this, process configuration has to be enhanced with well-defined interfaces to generic adaptation mechanisms which then allow for adding new behavior to the process model. This can be a simple manual adaptation or it could be another technique specifically aiming at the improved reuse of process models, like the aggregation or the instantiation of process models (as outlined in the process model adaptation framework of Becker et al. [33, 34]).

As the case study from Chapter 6 has shown, it is hard to achieve a complete basic process model which really covers all potential variation options. Thus, further adaptations are probably needed more often than not. Hence, future research efforts should be put on investigating how process model reuse could be better supported by improving the interplay between process model configuration and more generic process model adaptation mechanisms. For example, one could possibly record all manual modifications made to a configured process model. Then, these log files of how the process model was manually changed could be used to improve the basic process model. For this, any added behavior could be added to the basic process model as well. Removed behavior could be captured in terms of process configuration. Then, an investigation why behavior was restricted can lead to an improvement of configuration recommendations and constraints.

9.2.2 Configuration Performance

When configuring a business process, the goal is to get a process model that performs according to the individual business requirements. When looking at the executability of configurations in Chapter 8, we only ensured that the process can be executed at all. In future work, it is therefore necessary to find those configurations that optimally meet the business requirements.

Thus, it is necessary to identify which configuration decisions influence the performance of the process in which way. For example, some configuration decisions might lead to less executed work, some to less required storage space, and some to a higher customer satisfaction, etc. As we can configure individual tasks, not all the decisions leading to these outcomes necessarily exclude each other.

To find the coherence between configuration decisions and the performance of the resulting process, it might, e.g., be possible to use a two-phased approach. In the first phase, the performance characteristics of differently configured process variants are identified. This can either be done by deriving performance indicators for the configurations of existing systems (which can be identified

as suggested in Chapter 7), or by simulating new configurations of the basic process model as, e.g., outlined in [75, 146]. Having generated or identified a sufficient amount of such data sets, relations between configuration decisions and the performance of the process could be identified using corresponding machine learning techniques. For example, association rules between configuration decisions and performance values could be generated using the *A priori* method of Agrawal et al. [21]. These could then be used for deriving process configuration decisions based on the aspired process performance.

9.2.3 Configuration in the Process Life Cycle

The assumption in this thesis was that configurable process models are built only once, the resulting models are configured hundreds of times, and any model resulting from a configuration is executed millions of times. However, in practice neither the configured models nor the basic process model will remain unchanged over time. That means, changes to the process environment also lead to changes in the process configuration or even to changes in the configurable process model. Such changes can, e.g., be driven by new law, evolving technology, etc. The change of a business process over time is usually depicted in the process model life cycle which is divided into four phases: (a) design of the process model, (b) system configuration to implement the process, (c) process enactment, i.e. executing the process according to the process model, and (d) a process diagnosis phase which gives insights into the quality of the process execution and leads to process adaptations, i.e. to the design of new process models.

When using a configurable process model, the diagnosis phase does not necessarily need to lead to the design of a new process model. Instead, the process can also be adapted by just changing the process configuration, skipping the process model design phase (see Figure 9.2). Especially in the context of reference models, the process design is usually done externally by the reference model provider while the model user configures and executes the procured process design. To improve the process model over time, the model provider relies on practical experiences of model users, i.e. both the model user as well as the model provider need to analyze the process executions in order to enhance models and configurations.

Hence, future research needs to investigate if the use of process configuration could also lead to simplifications in transferring process instances between different configurations compared to existing approaches like the ones of Rinderle et al. [140], Han et al. [87], or Kammer et al. [99]. For example, this would be helpful whenever a process configuration is changed. Furthermore, it must be analyzed, how existing configurations for an 'old' version of a configurable process model, can be applied to a new version of the configurable process model. Currently, the only options are to either adapt each configured model individually, or to start configuring the new configurable model from scratch. However, if it would be possible to carry configuration decisions forward, maintenance changes over time could be simplified. This would allow making changes on the configurable process model and propagating them to the individual models

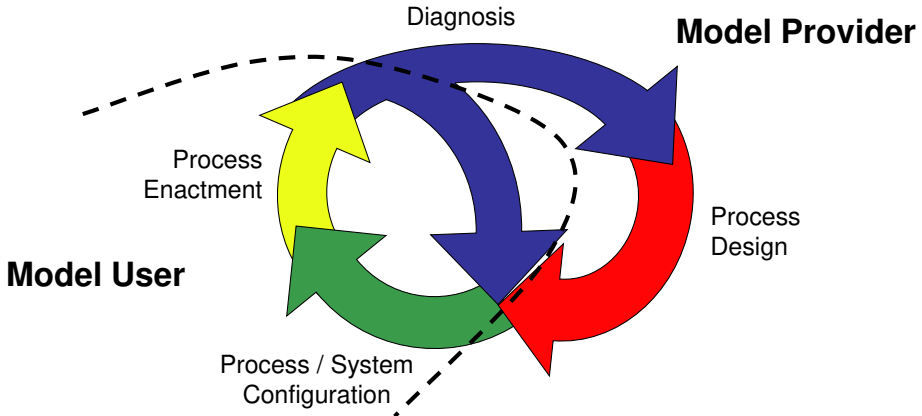


Figure 9.2: The lifecycle of a configurable process model.

afterwards. In this way, each ‘old’ configuration decision could be applied automatically to the new configurable process model, and only new configuration choices would still need to be addressed before the updated model can be put in place.

9.2.4 Process Model Content

Obviously, a process model will only be (re-)used if it provides information that is worthwhile to use. Hence, all techniques and tools suggested in this thesis are useless without resulting in good-quality content, which also requires high-quality information being available when developing configurable process models. In Chapter 7 we suggested a range of tools that help in constructing the basic process model of a configurable process model, aiming at helping the model designer with constructing good-quality models. However, if logs are missing or the quality of log files is not good, process mining cannot assist in obtaining satisfactory process models. In the same way, the depicted merge algorithm is not able to deliver a useful basic process model, if the quality of the input models is low. In fact, constructing a configurable process model is not about promoting techniques. Constructing a configurable process model means editing and combining existing process behavior such that the model reuse is facilitated.

Errors even in presumably good-quality reference process models like SAP’s reference model [58, 119] show that having high-quality input available is far from trivial to achieve. And, if the input contains errors already, these errors are usually propagated to the configurable model. Also, the case study in Chapter 6 has shown how cumbersome it is to collect information from various sources and aligning them to construct configurable models, even though desired results were achieved in the end. Moreover, as municipalities have to account for their activities publicly, the municipalities had no issues in sharing and explaining

their processes in detail for the case study. Obtaining such data from organizations that have to compete with other organizations is much more difficult. As they fear to lose competitive advantages, they are reluctant in sharing process information [172].

Organizations that have access to process information from various companies are consultancy firms. Furthermore, they often already use an existing ‘best-practice’ reference model as basis for process implementations which they adapt manually in each implementation project. Thus, they are likely candidates both for constructing configurable process model and improving their reference processes, as well as for using them. However, as they hardly sell the reference model as a product itself, the high costs of creating and updating a configurable reference model are a major barrier. Hence, the focus of future research must be in further reducing the efforts required in creating such models. For this, solutions that automatically record the variations when adapting the process model and which use these adaptation as further variation options of the configurable process model afterwards (as sketched in Section 9.2.1) could again be helpful. In this context, it will also be interesting to see how initiatives like Software AG’s *AlignSpace*¹ for the shared creation of public process repositories via the internet, similar to social communities and wikis [61], will be able to provide quality process models in the coming years. For example, the process configuration mechanisms of blocking and hiding could be used here to depict an integrated model that highlights various changes made to process models over time.

9.3 Summary

Configurable process models can facilitate the reuse of process models. They simplify the adaptation of process models by allowing model users to block or hide already modeled process behavior instead of requiring users to make cumbersome manual changes and additions to a standard template. That means that, although the easier process model adaptation is on the expense of an increase in the complexity of building the configurable process models compared to a simple template, developing configurable process models is useful whenever a process model is needed for a large amount of widely standardized process implementations with small variations. In this way, the savings in not having to do many manual adaptations exceed the efforts needed to construct the configurable model. Furthermore, tools supporting the construction of configurable process models like the automatic merge of existing models or the construction of process models from log files of existing systems through process mining contribute to reducing construction efforts.

Process configuration is a powerful tool for restricting any behavior allowed according to process models. Constraints among configuration decisions can be used to limit this configuration space, i.e. constraints can guarantee that only desired process variants can be achieved through process configuration.

¹see <http://www.alignspace.com>

For process models that are to be adapted by domain experts without the help of process modeling experts, process configuration decisions can be mapped to predefined answers to questions posed in natural language and organized in questionnaires. Answering the questions then leads to the definition of a process configuration and thus to an individually adapted process model without the need to work on the process model itself.

For a configurable process model to be successful, it is essential that its content provides benefits to the organizations using it, and that it provides the support necessary to get to the corresponding process configuration. Moreover, the configured model must be easily adaptable in case it does not already completely fulfill all individual requirements. In addition, both the configurable model itself as well as its configured variants have to be adjustable to a changing environment over time. Hence, future research should aim at providing further support for these challenges.

Appendix A

Case Study Process Models

This appendix contains the process models for the four processes from the case study described in Chapter 6 as summarized in Figure A.1.

The four selected processes are four out of the five most executed registration processes in the civil affairs departments of municipalities. The process excluded is the registration of a couple's divorce as the steps taken by the municipalities are rather trivial in this process (its main steps are a matter of judicature).

	Acknowledging an unborn child	Registering a newborn	Marriage	Issuing death certificate
NVVB reference model	A.2	A.9	A.16	A.23
Municipality 1 (26.000 inhabitants, no hospital, Software A)	A.3	A.10	A.17	A.24
Municipality 2 (42.000 inhabitants, no hospital, Software A)	A.4	A.11	A.18	A.25
Municipality 3 (117.000 inhabitants, has hospital, Software A)	A.5	A.12	A.19	A.26
Municipality 4 (200.000 inhabitants, has hospital, Software B)	A.6	A.13	A.20	A.27
Configurable Protos model	A.7	A.14	A.21	A.28
Configurable YAWL model	A.8	A.15	A.22	A.29

Figure A.1: Overview of the models contained in the appendix

The depicted reference models of these processes can be obtained from the NVVB. They document in detail recommendations of how municipalities should execute these processes. The depicted models are part of a package of more than 100 of such reference process models for common processes executed in municipalities.

The individual municipalities were chosen such that a variation in the resulting models was expected. For this, the municipalities primarily vary in the number of citizens. Furthermore, they use two different software applications to support the execution of the depicted processes. If a municipality has a hospital or not has a significant influence on the number of cases executed for both the process of registering a newborn as well as the issuing of death certificates.

The configurable Protos models are the results of manually merging the four individual process models and the model of the NVVB for each of the four processes. These models were then manually converted into the depicted YAWL models. Not visible in the figures of the YAWL models is that the underlying required and produced data is also specified for the various tasks. In this way, configurations of these models can be loaded and executed using the YAWL workflow system.

A.1 Acknowledging an Unborn Child

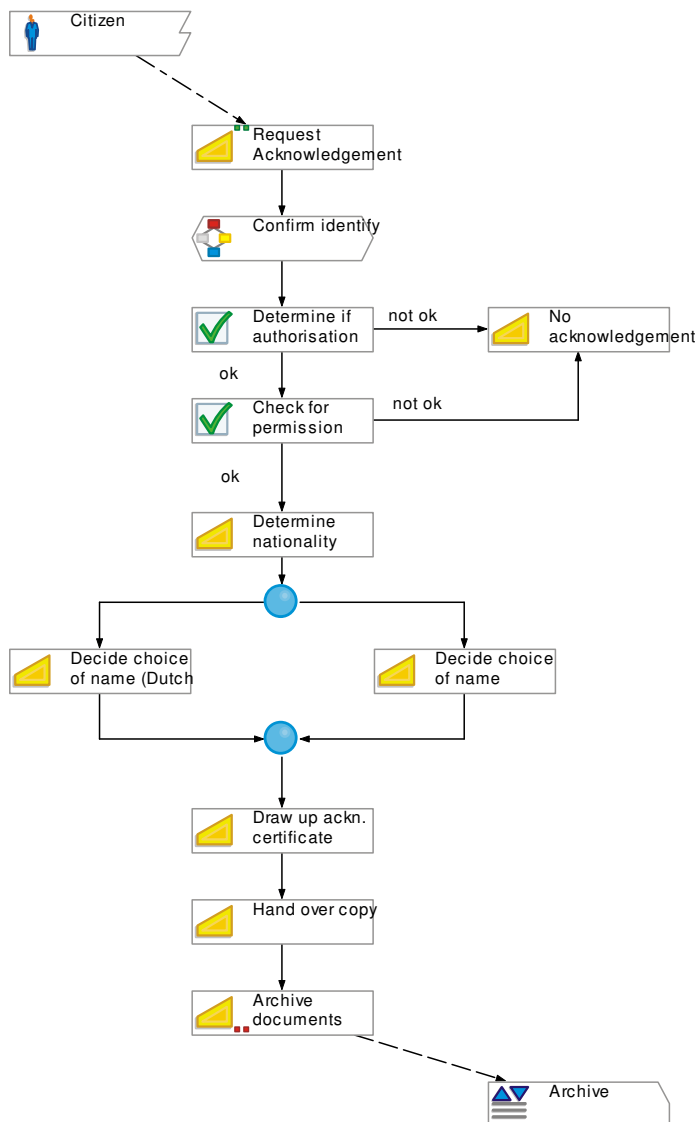


Figure A.2: Reference model of the NVVB for the process of *acknowledging an unborn child*. After the identity of the applicant has been checked, it is determined if he is authorized to acknowledge the child. Furthermore, the permission of the pregnant wife is checked. Based on the nationality, the choice of the last name for the baby is either performed based on Dutch or foreign law. The issued acknowledgement certificate is handed over to the applicant before the documents are archived.

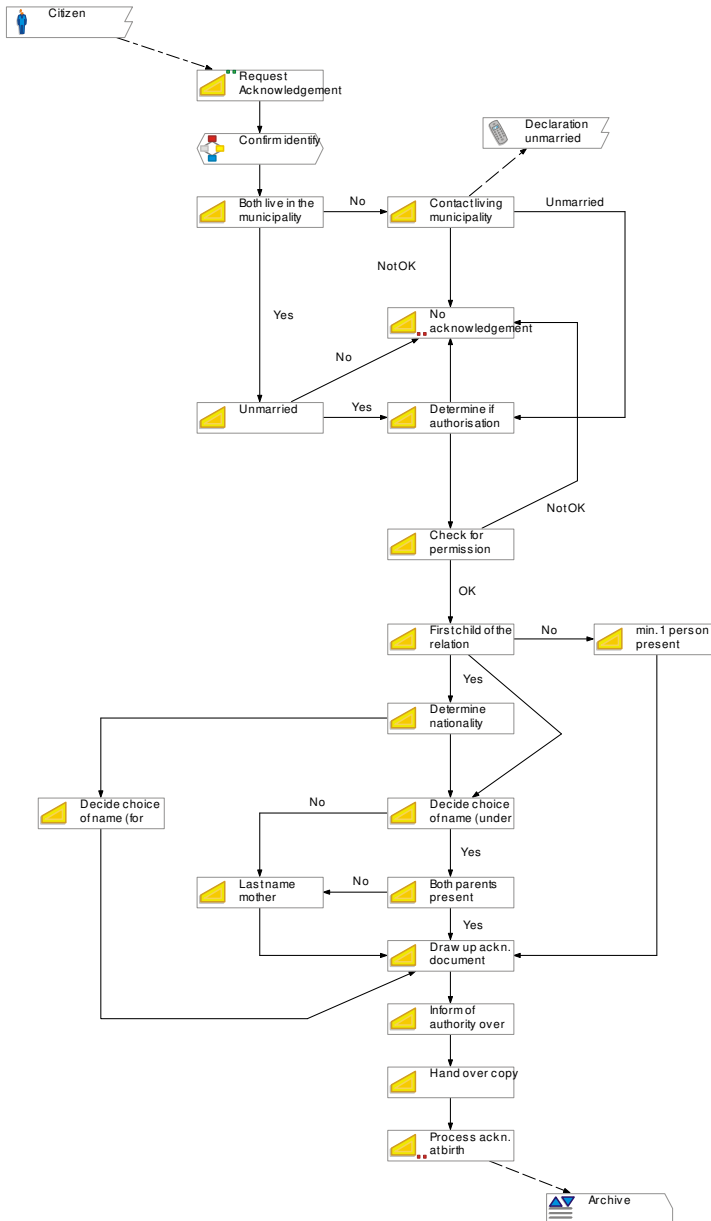


Figure A.3: Process model for *acknowledging an unborn child* according to Municipality 1 (small): In addition to the suggestions by the reference model, Municipality 1 checks if both parents live in the municipality and are unmarried before determining the authorization. Also, later on Municipality 1 checks if the child is the first out of the relationship because if not, the choice of the last name is identical to the previous child and does not have to be determined again.

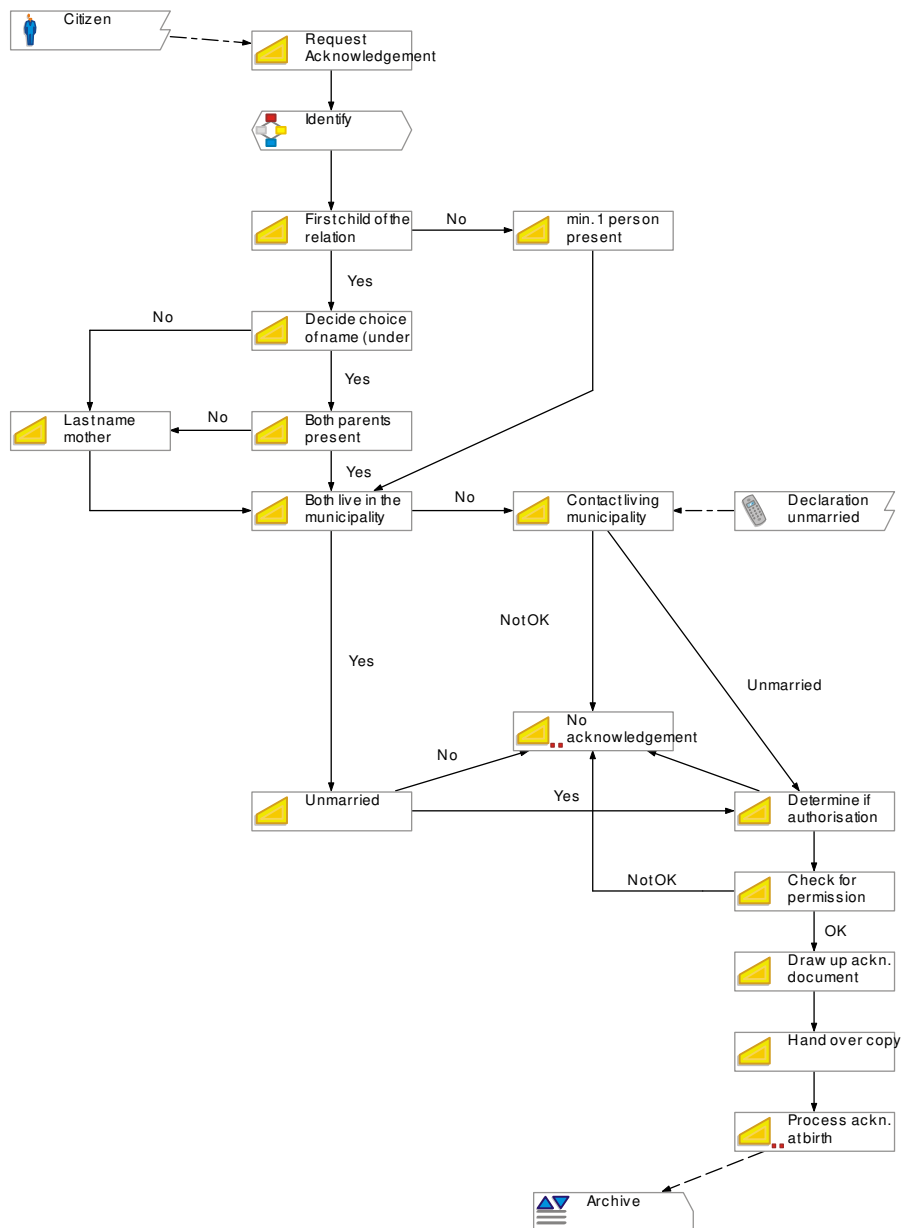


Figure A.4: Process model for *acknowledging an unborn child* in Municipality 2 (medium-size): Municipality 2 checks first if the child is the first of the relationship. Only afterwards it is checked if the parents live in the municipality, i.e. the order of these process parts is inverse compared to Municipality 1.

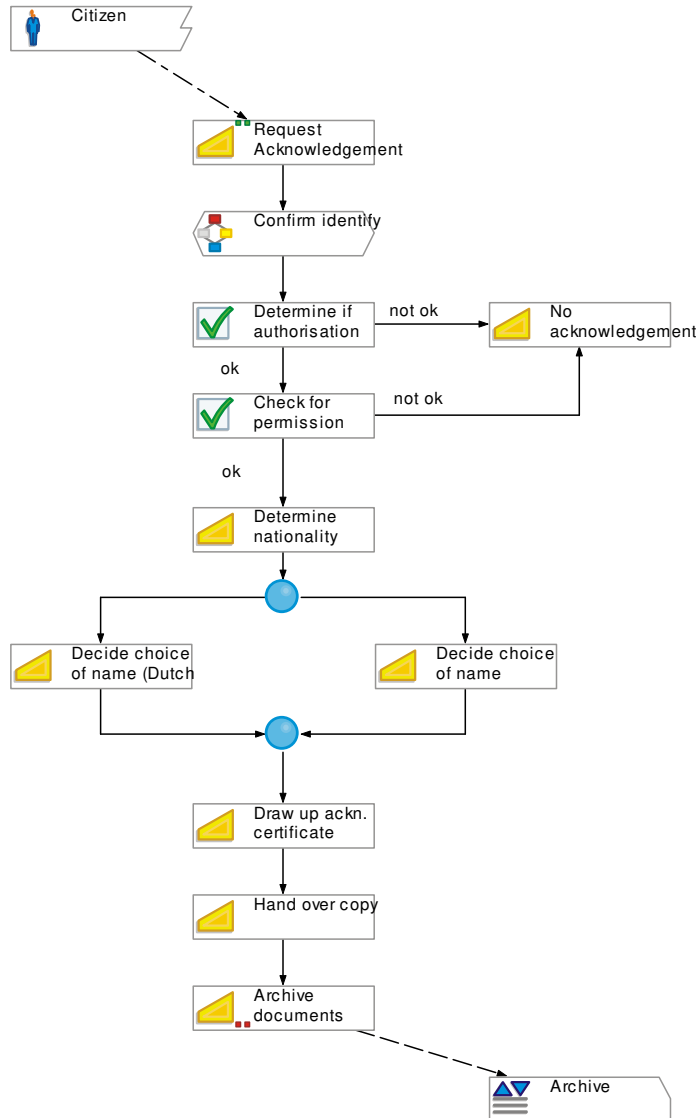


Figure A.5: Process model for *acknowledging an unborn child* in Municipality 3 (large): The process is executed in the same way as suggested by the NVVB.

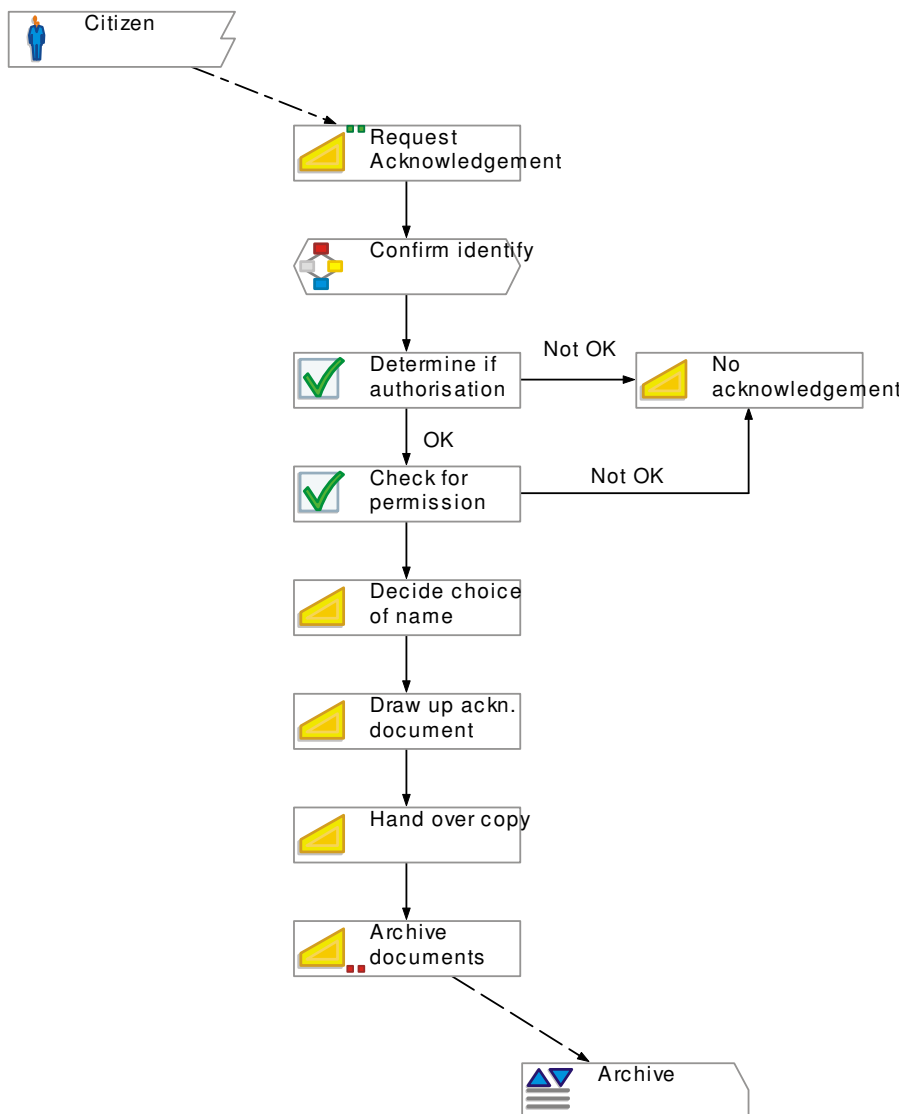


Figure A.6: Process model for *acknowledging an unborn child* in Municipality 4 (very large): The process model depicts even less steps than the process suggested by the NVVB as it does not handle acknowledgement requests from foreigners different from the ones of Dutch citizens.

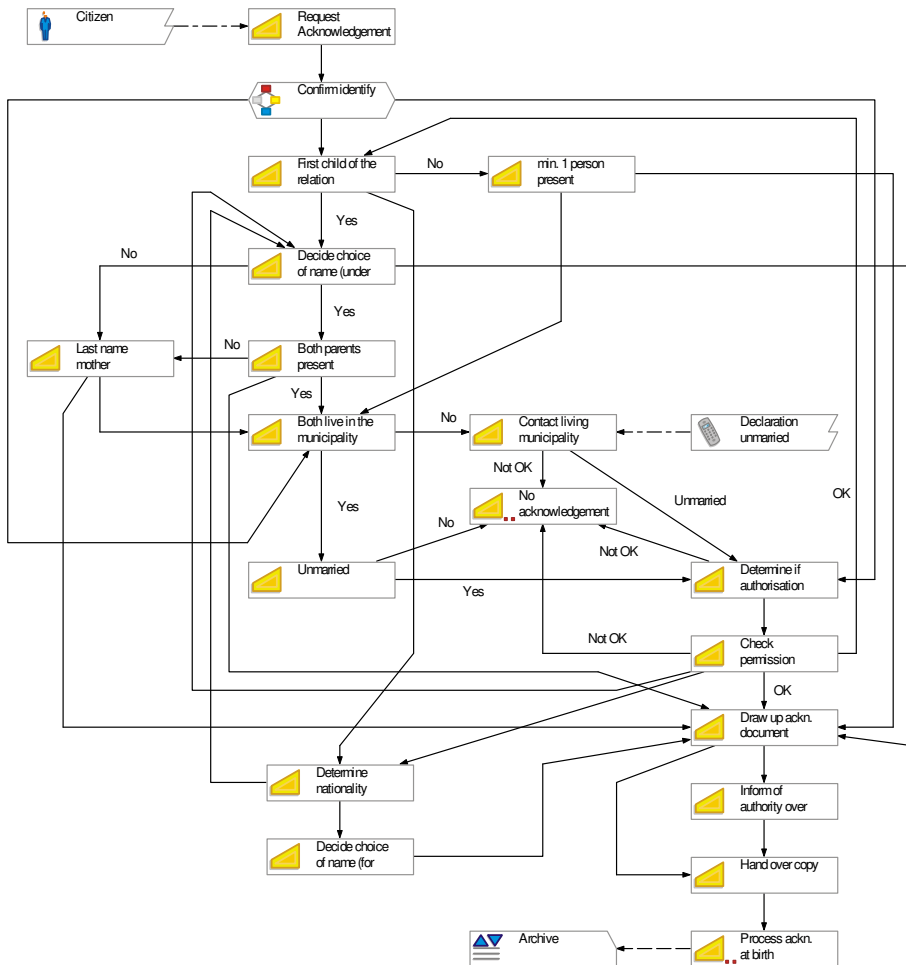


Figure A.7: Integration of the five process variants for *acknowledging an unborn child*: The increase in the number of arcs is obvious and mainly due to the variations in the order in which the different municipalities execute the various tasks.

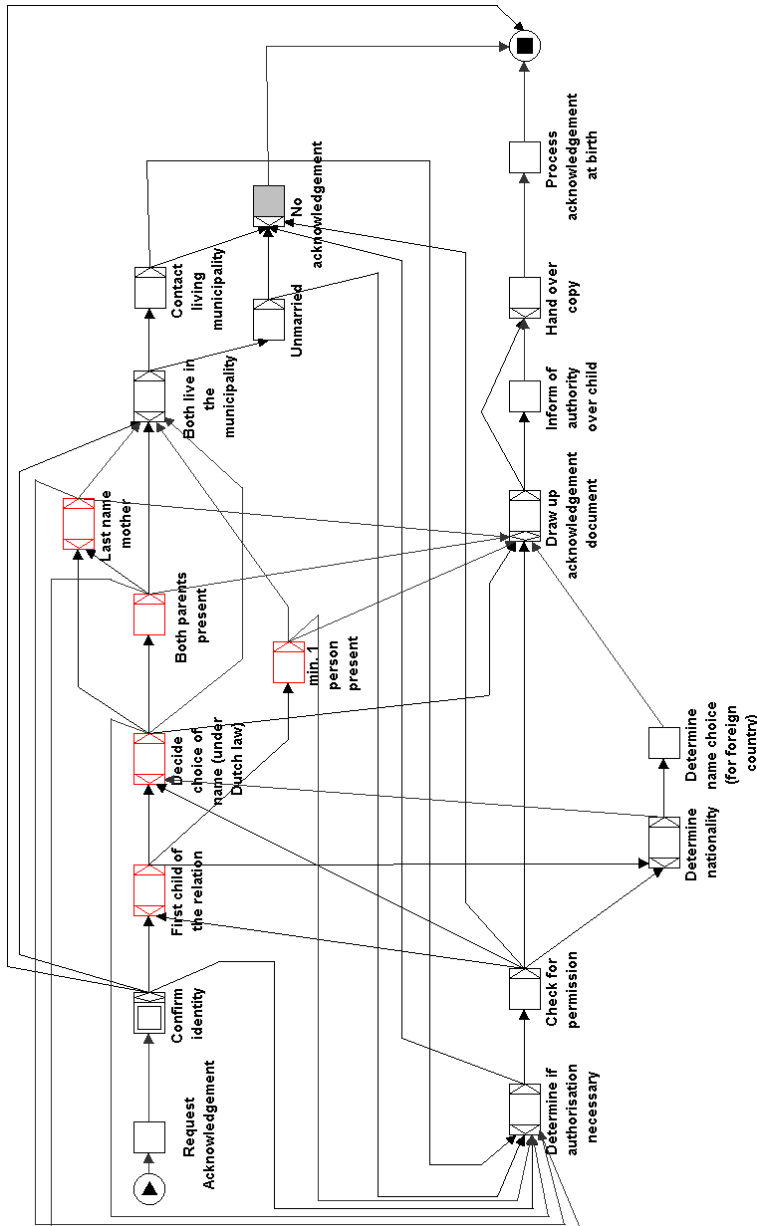


Figure A.8: Translation of the integrated process model for *acknowledging an unborn child* into YAWL: As it seems from looking at the individual process variants that the order between checking if a child is the first of a relationship and if both parents live in the municipality does not matter, the option to execute them in parallel was added by using an OR-split at task *Confirm identity*. The branches are synchronized again through the OR-join before executing the task *Draw up acknowledgement document*. If no acknowledgement can be performed, all tasks that can still be executed in parallel are canceled through the cancellation region of this task.

A.2 Registering a newborn

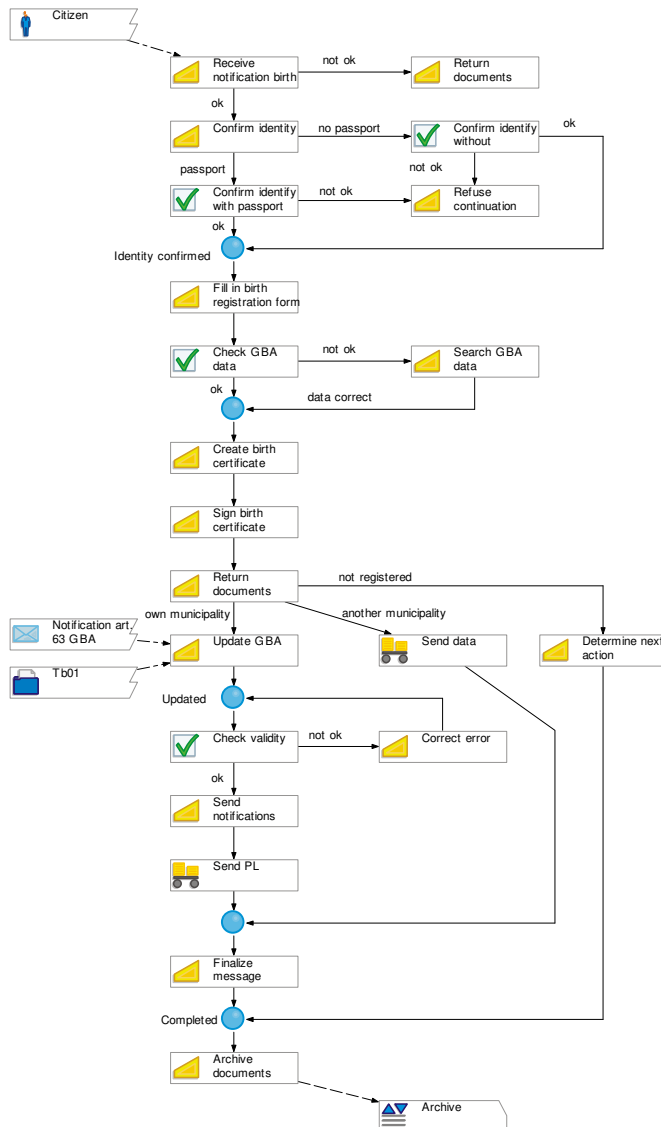


Figure A.9: Reference model of the NVVB for the process of *registering a newborn*: After the identity of the person registering the child has been confirmed, a birth registration form should be filled in. The details of the family members entered in this form are checked with the help of the municipality database (GBA) before the birth certificate is issued. In case the parents are registered in the municipality where the newborn is registered, the municipality database is updated with the information about the newborn. Otherwise the data is sent to the municipality where the parents are registered. If the parents are not registered at all, the process is escalated.

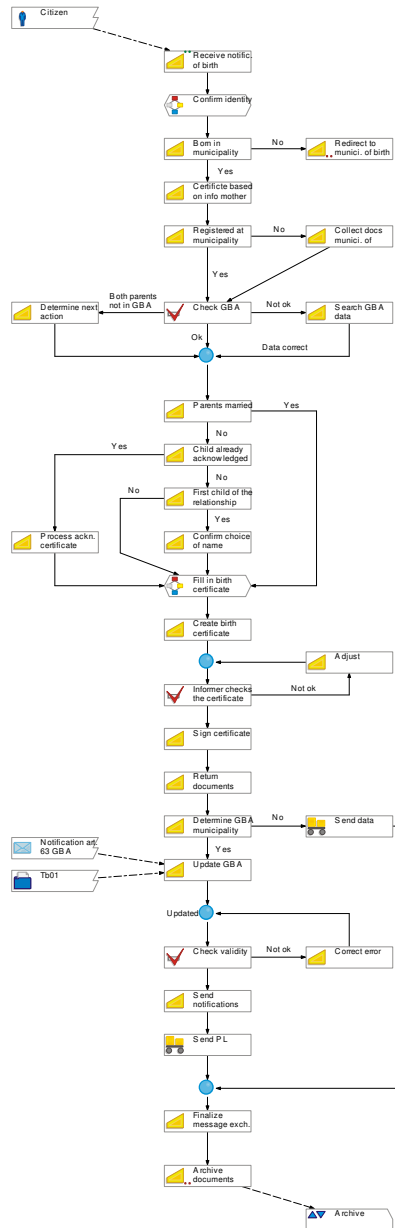


Figure A.10: Process model for *registering a newborn* in Municipality 1 (small): In comparison to the suggestions of the NVVB, Municipality 1 does not require filling in a form first but personally asks for information on the birth and if the parents live in the municipality. After the database check, it is checked if the parents are married. If not, the last name must be determined before the birth certificate can be created. Also, Municipality 1 requires that the person registering the child checks the certificate before it is signed and handed out. Like in the reference model, finally the database of the municipality is updated or the documents are forwarded to the responsible municipality.

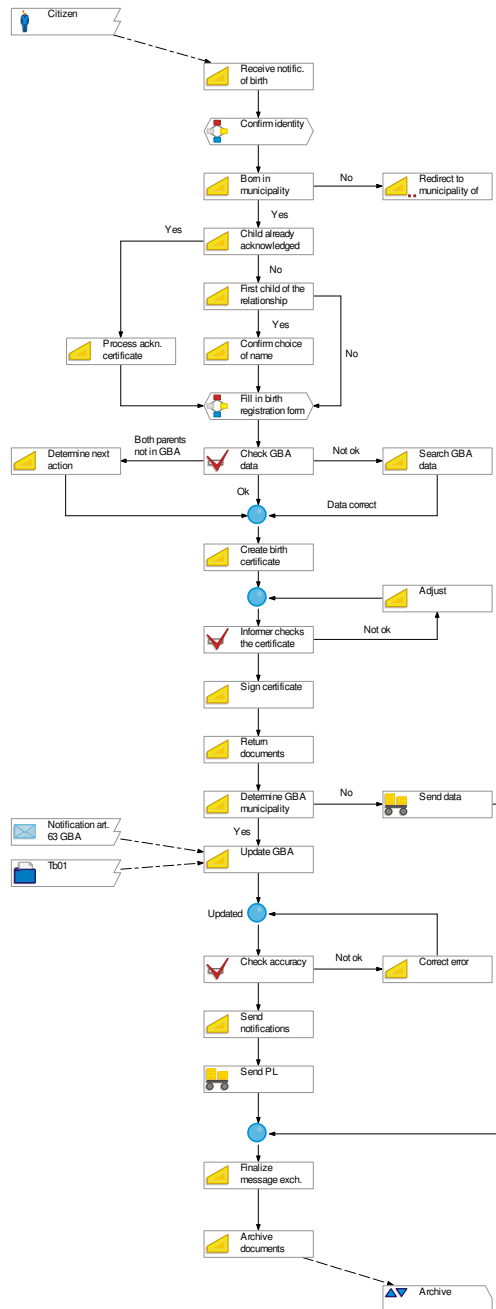


Figure A.11: Process model for *registering a newborn* in Municipality 2 (medium-size): The process of Municipality 2 is very similar to the process of Municipality 1. The main difference is that Municipality 2 determines the choice of name before checking the details of the parents in the municipality database while Municipality 1 did this only afterwards.

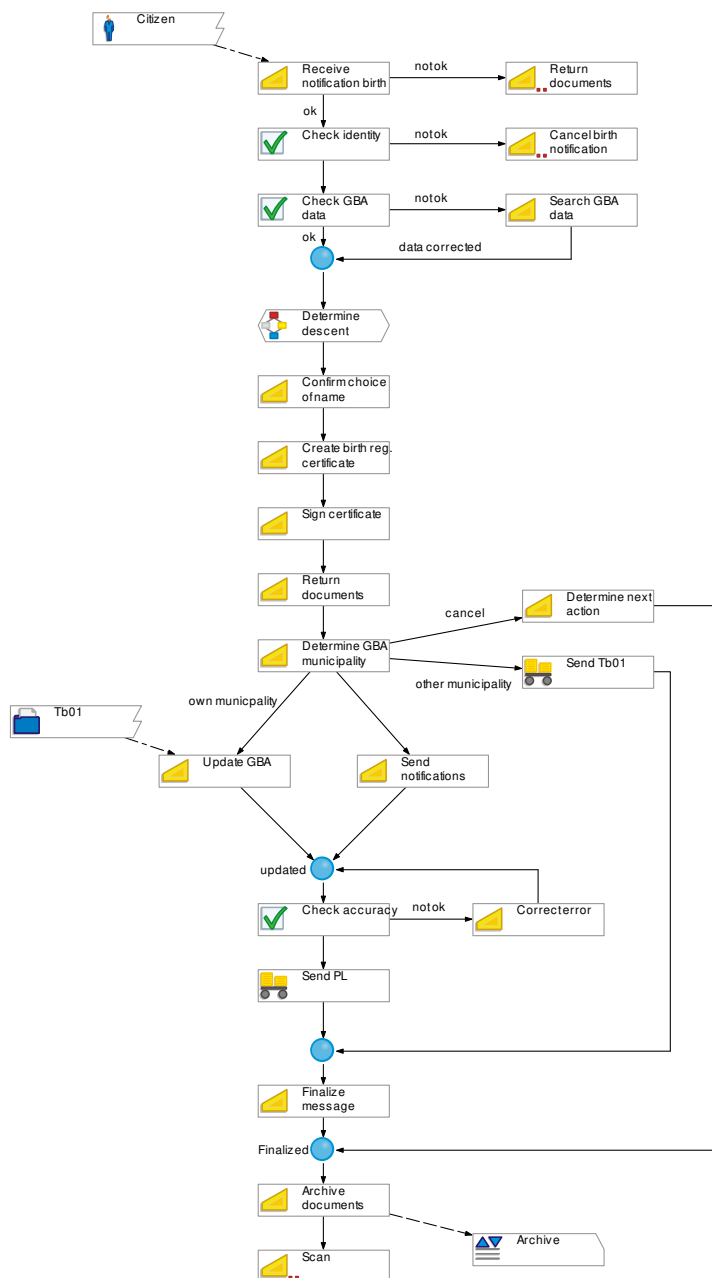


Figure A.12: Process model for *registering a newborn* in Municipality 3 (large): The process of Municipality 3 is similar to the reference model suggested by the NVVB, but not identical. Main differences are that the informer does not need to fill in a form before the database is checked, and that separate steps are performed to determine the descent and the name of the newborn. Furthermore, the model contains the option to send notifications to court in case the parents have no authority over the newborn.

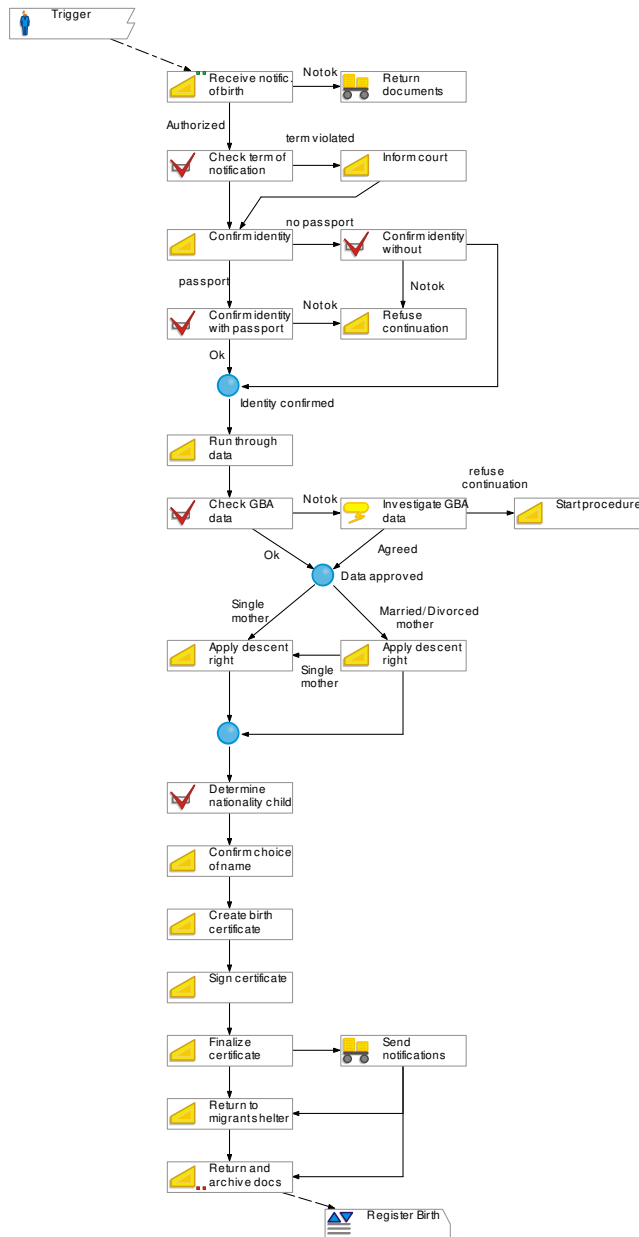


Figure A.13: Process model for *registering a newborn* of Municipality 4 (very large): Municipality 4 starts the process of registering a newborn with checking if the registration happens within the timeframe prescribed by law and informs a court if not. Identity check, and check of the parents details are performed in line with the suggestions of the NVVB. Afterwards the model describes in detail the applied descent rights before the name can be chosen and the birth certificate can be created. Notably, the process of Municipality 4 does not include the update of the municipality database contained in the other models of this process.

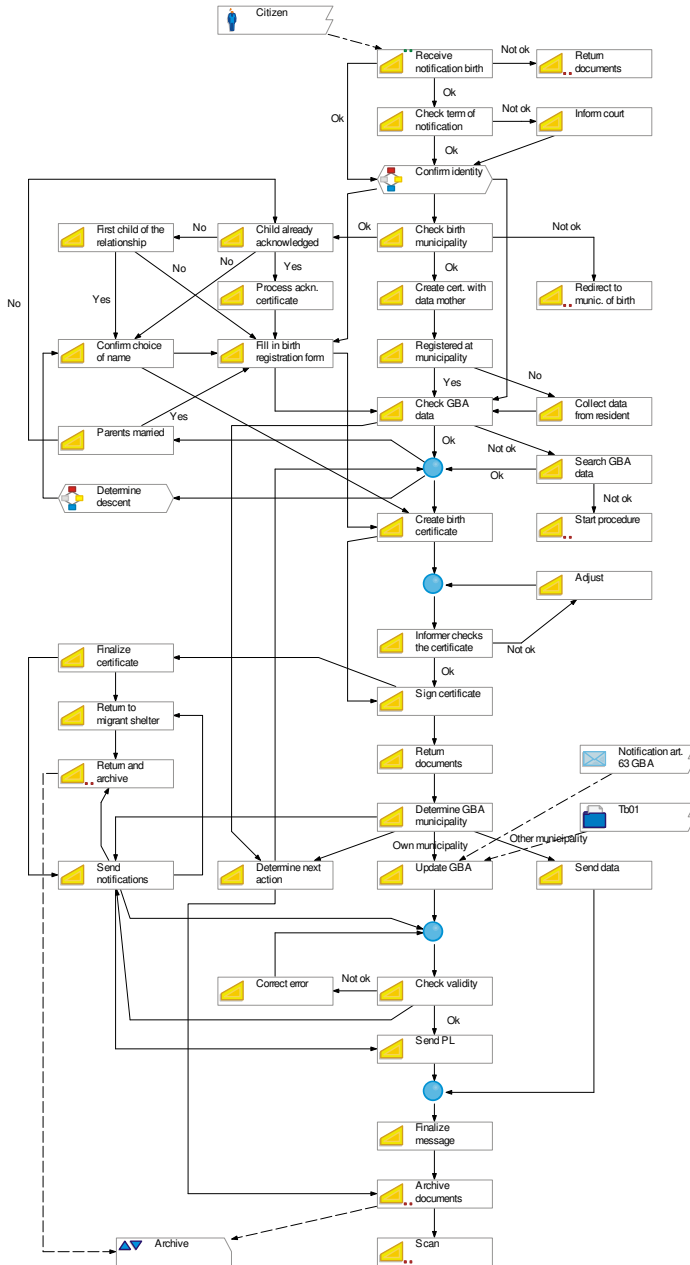


Figure A.14: Integration of the five process variants for *registering a newborn*: Again, the number of arcs increases significantly, but in this model also the number of tasks increases significantly compared to the individual models. That means, the tasks executed vary more than in the process of acknowledging an unborn child. Also quite a number of arcs are introduced to skip other tasks, i.e. in a configurable process models these arcs might also be replaceable by simply hiding the skipped tasks.

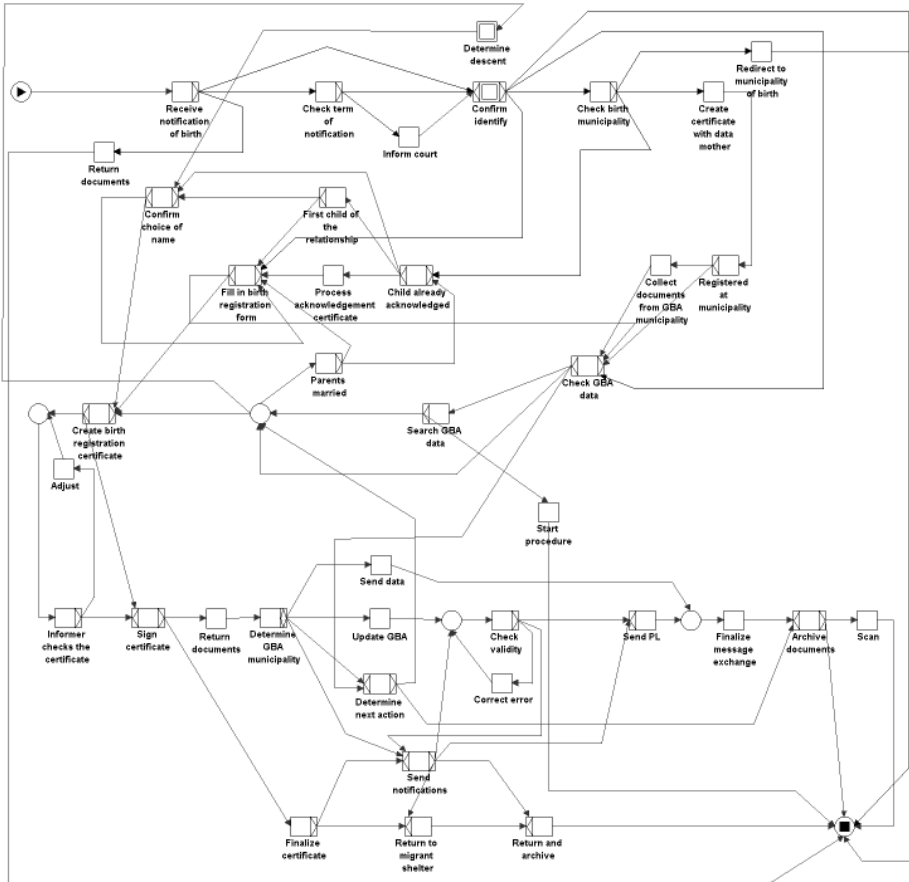


Figure A.15: YAWL model of the process for *registering a newborn* from Figure A.14: Note that all choices in the YAWL model are XOR-splits which are synchronized again either through conditions or through XOR-joins.

A.3 Marriage

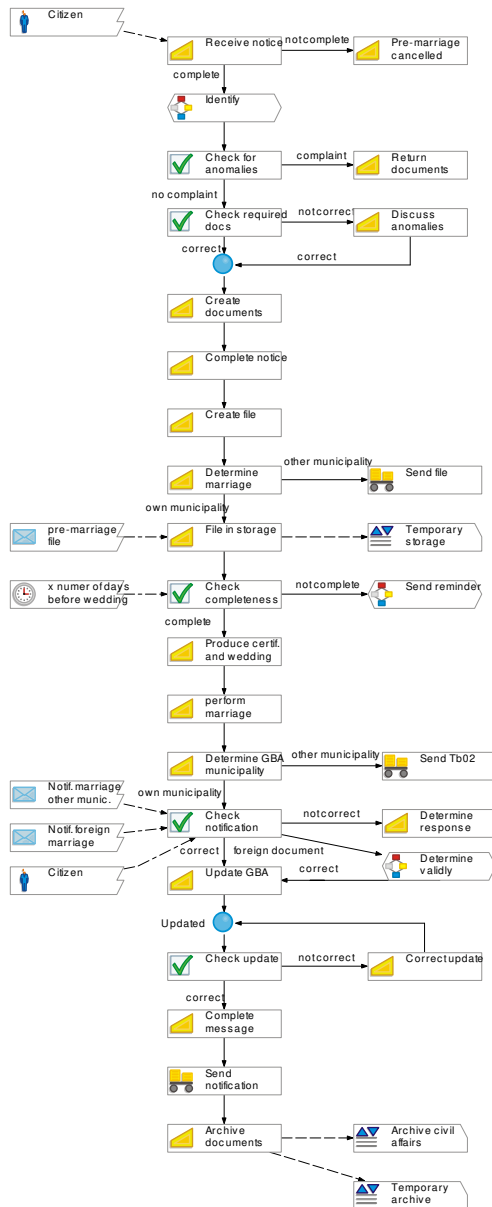


Figure A.16: Reference model of the NVVB for processing a *marriage*: The marriage is the process involving executing the most steps out of the four processes considered during the case study and can be divided in three phases: the creation and checking of documents before the marriage, the wedding itself, and the registration of the marriage in the municipality database (GBA).

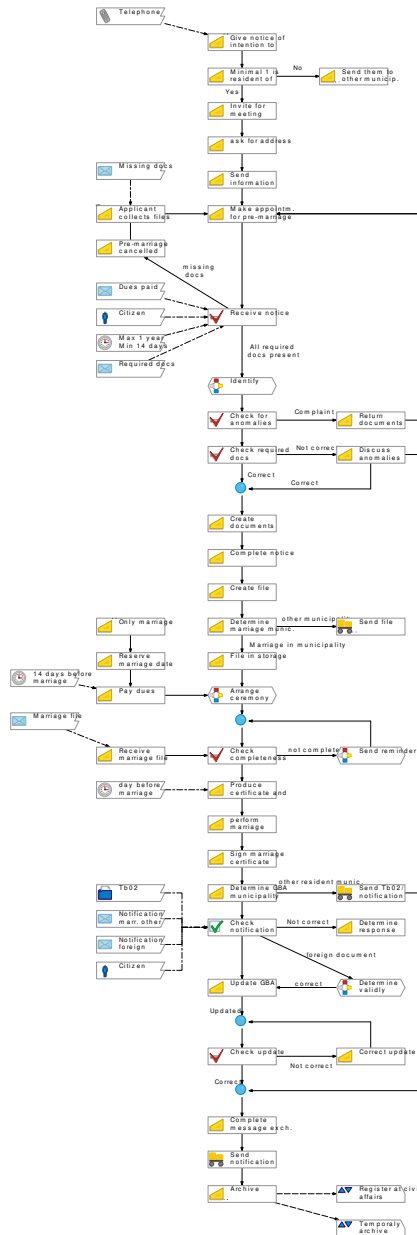


Figure A.17: Process model for the execution of a *marriage* in Municipality 1 (small): The process of Municipality 1 deviates from the process suggested by the NVVB mainly in service parts. For example, Municipality 1 requires the couple to make an appointment for processing the pre-marriage part of the process. Moreover, Municipality 1 sends the couple information material about getting married. The process model also contains a second starting point for the process in case the pre-marriage part of the process was performed in a different municipality than Municipality 1.

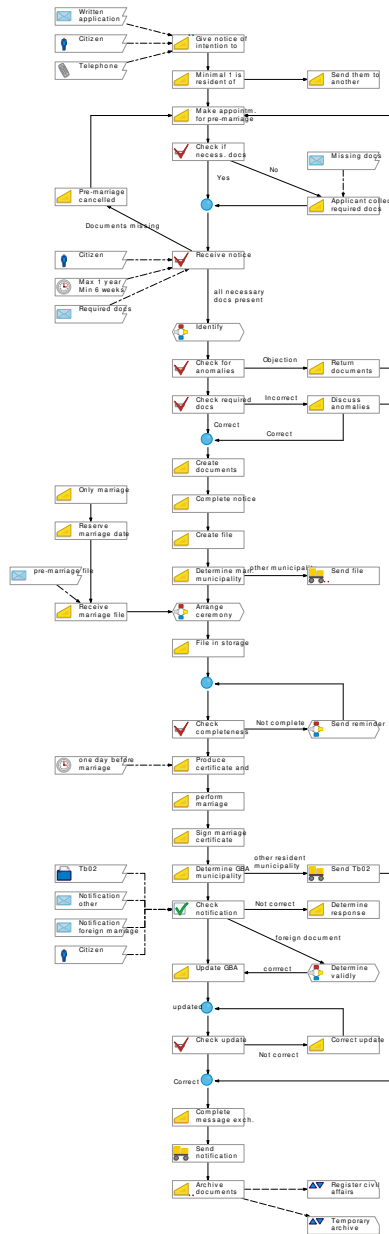


Figure A.18: Process model for the execution of a *marriage* in Municipality 2 (medium-size): Also Municipality 2 requires citizens to make an appointment for executing the pre-marriage part of the process, but it does not send out information material like Municipality 1. Instead, it provides the opportunity to the couple to check if the couple has all necessary documents before executing the pre-marriage process. In this way, the applicants can still collect the documents. Besides this, the process of Municipality 2 is almost identical to the process of Municipality 1, including the option to start the process directly with the wedding in case the pre-marriage part of the process was executed in a different municipality.

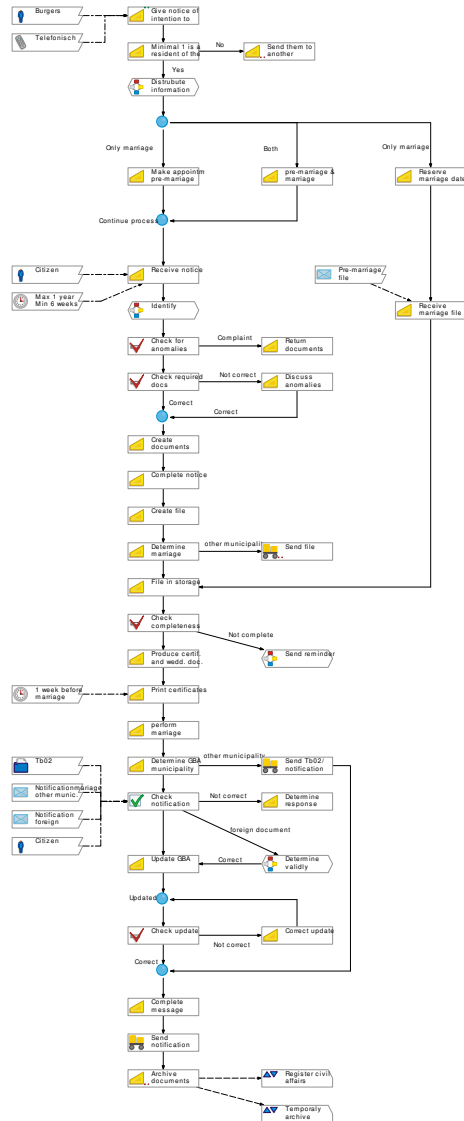


Figure A.19: Process model for the execution of a *marriage* in Municipality 3 (large): While the wedding and the database registration parts of the process are very similar to Municipality 1 and Municipality 2, Municipality 3 organized the pre-marriage part again differently from the two other municipalities. Here, the process can in any case only be executed if at least one person of the couple lives in the municipality. If this is the case, the couple gets first some information material before there is the chance for making appointments either for the pre-marriage, for the pre-marriage and the wedding, or for the wedding only — dependent on which activities should happen in Municipality 3. If only the wedding happens in Municipality 3, the pre-marriage part of the process is skipped and instead the municipality waits to receive the corresponding file from another municipality. If only the pre-marriage happens in the municipality, the documents are sent to the municipality responsible for the wedding after the pre-marriage part of the process is completed.

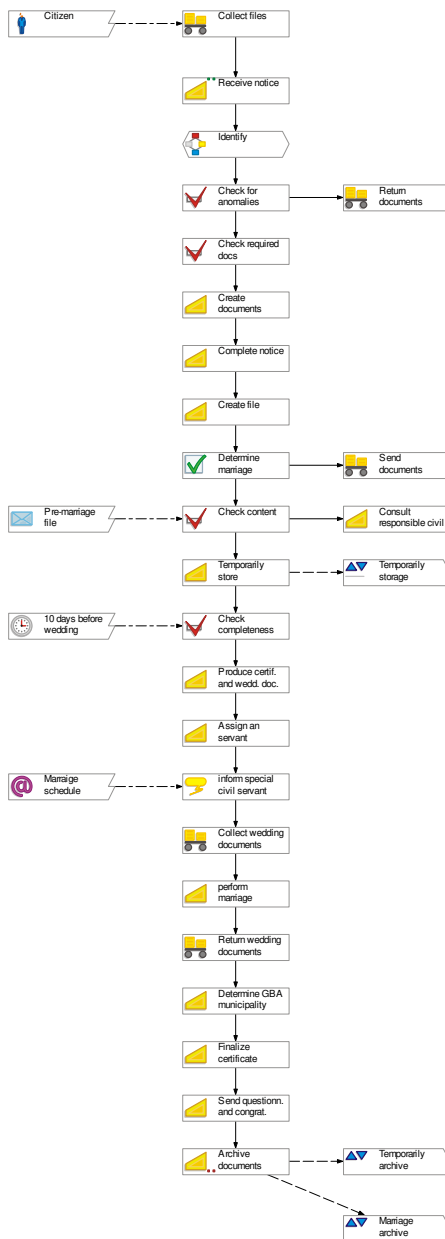


Figure A.20: Process model for the execution of a *marriage* in Municipality 4 (very large): Municipality 4 organizes the marriage process in a very sequential way. The only variations of the process are that after the pre-marriage the documents can be sent to a different municipality, or that if any documents contain anomalies they are either returned or the responsible organizations or departments are contacted.

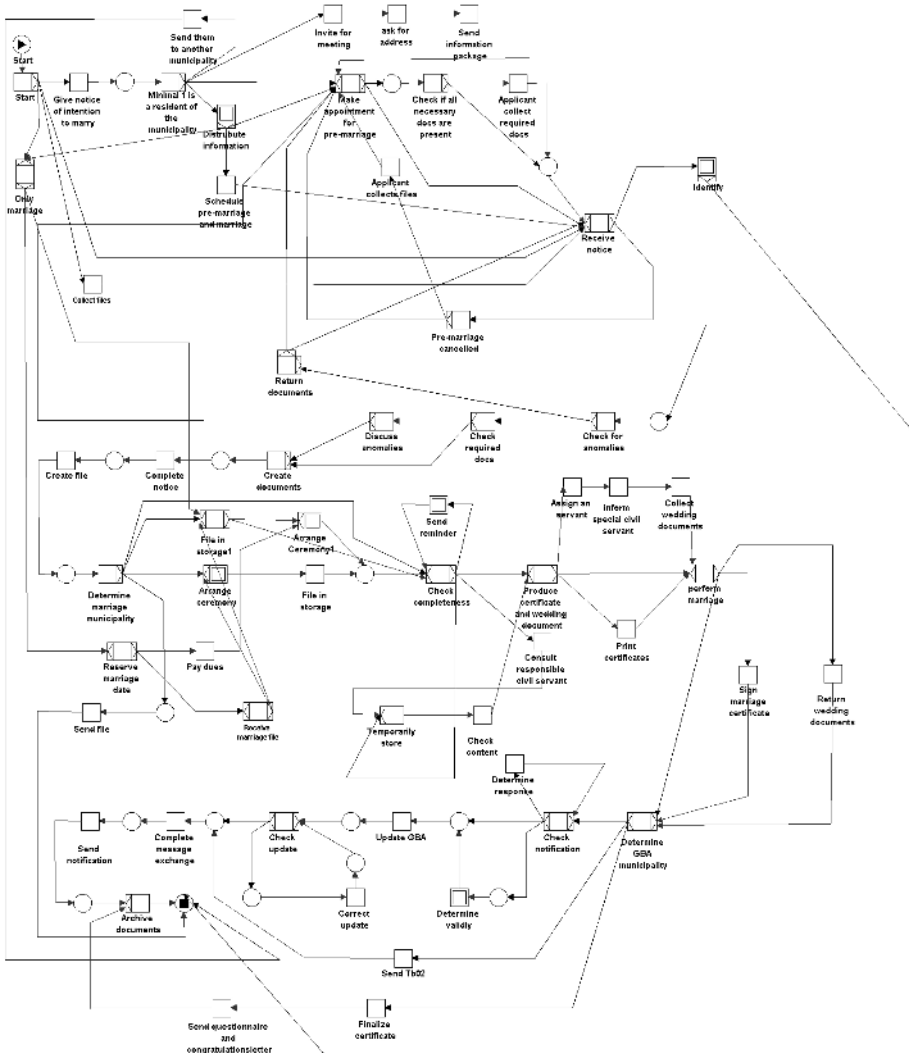


Figure A.22: The integrated *marriage* process transformed to YAWL: The YAWL model can easily be split into the three parts of the wedding process: the pre-marriage at the top, the wedding itself in the middle, and the GBA database update at the bottom. Also note the various distinct points at which the process can complete, indicated through direct arcs to the output condition of the net.

A.4 Issuing Death Certificate

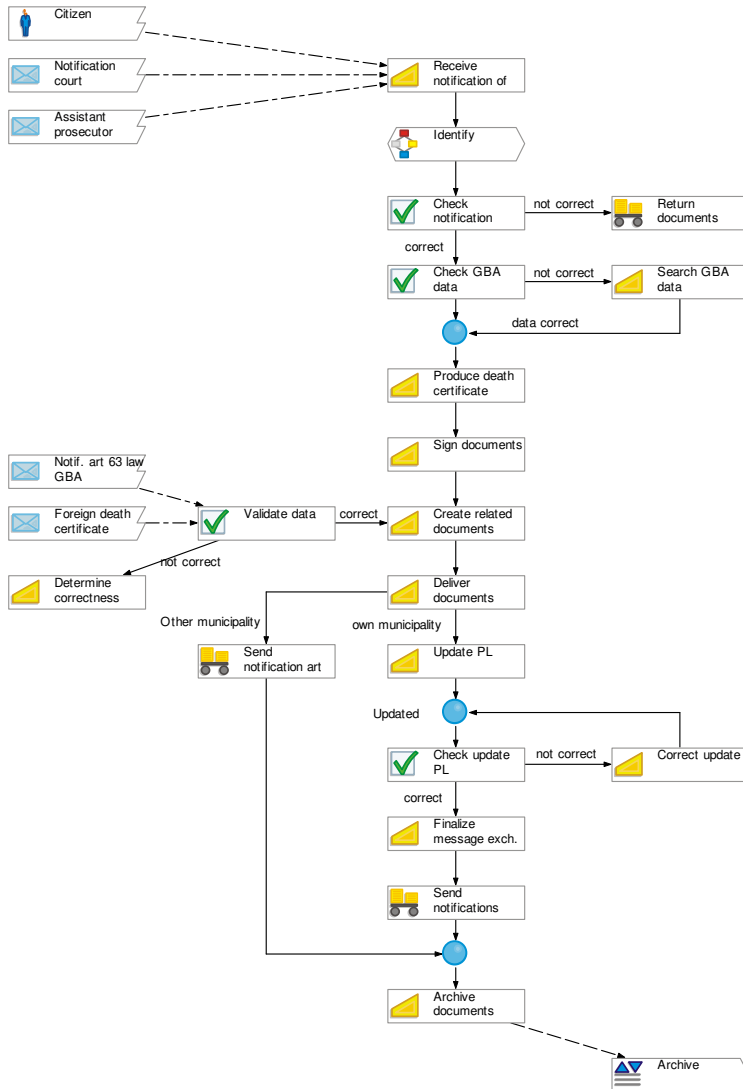


Figure A.23: Reference model of the NVVB for *issuing a death certificate*: The NVVB model suggests that municipalities first check the identity of the person that informs them over the death of another person. Then, the details of the information are checked for completeness before the details are compared with the details in the GBA database. If all data is correct, the death certificate and related documents are produced. Depending on if the deceased was living in the municipality where the process is executed, the municipality either updates its registration data or it sends the documents to the responsible municipality. Among the four suggestions of the NVVB, the process seems to be shorter and simpler than the registration of a child or a marriage, but slightly more complex than acknowledging an unborn child.

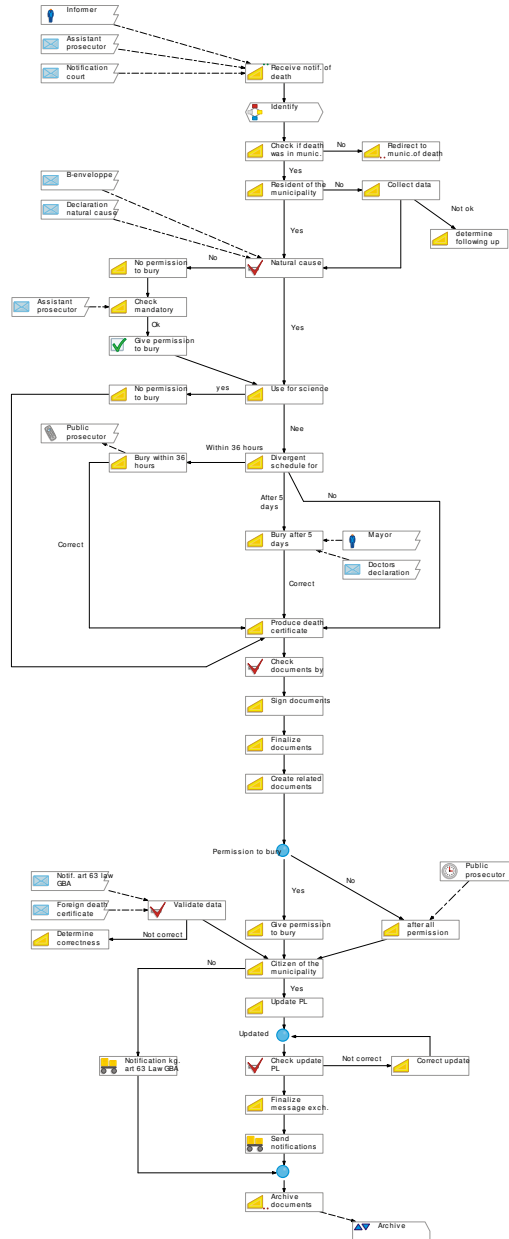


Figure A.24: Process model for *issuing a death certificate* in Municipality 1 (small): The model of Municipality 1 contains quite a number of additional steps compared to the recommendations by the NVVB. For example, it checks if the death happened in Municipality 1 and if the deceased was a resident of Municipality 1 (otherwise it first collects the permission to bury the deceased will be suspended until the investigation releases the body), if the body should be used for science, or if the body should be buried unusually quickly (in less than 36 hours) or late (after more than 5 days). The update of the deceased’s registration happens in line with the suggestions of the NVVB.

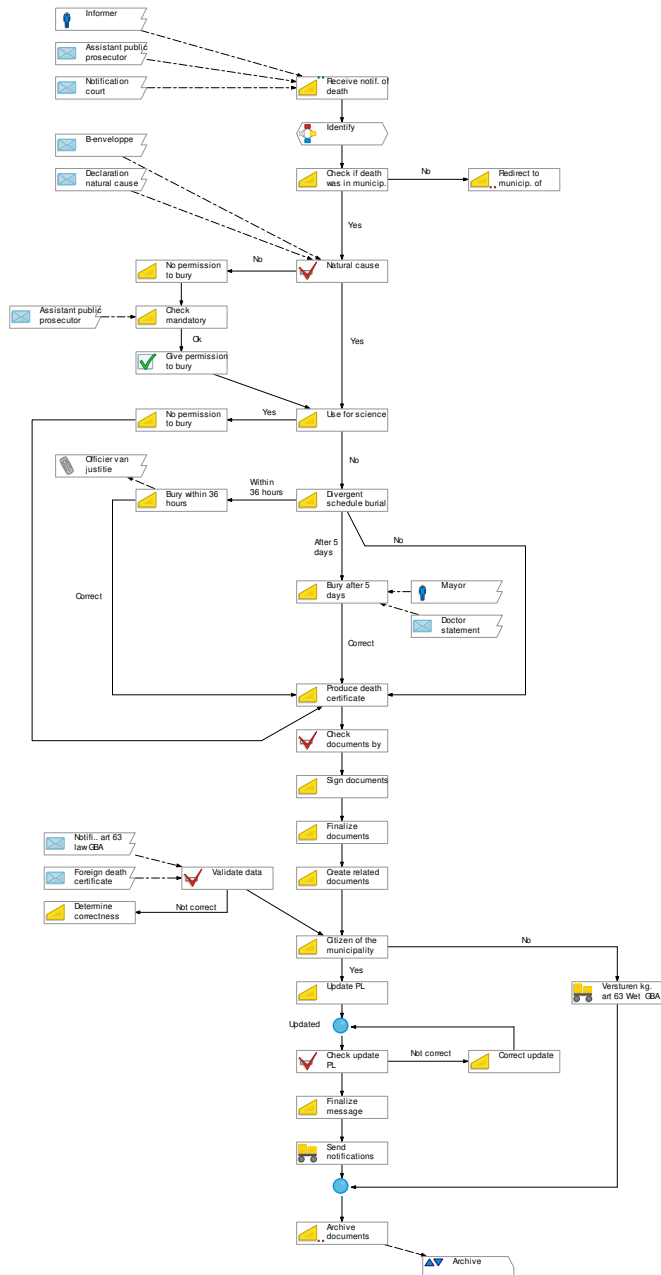


Figure A.25: Process model for *issuing a death certificate* in Municipality 2 (medium-size): The process model of Municipality 2 is almost identical to the one of Municipality 1. The only main difference is that Municipality 2 does not check in the beginning of the process if the deceased was residing in Municipality 2 — this check only happens when the registration data needs to be updated.

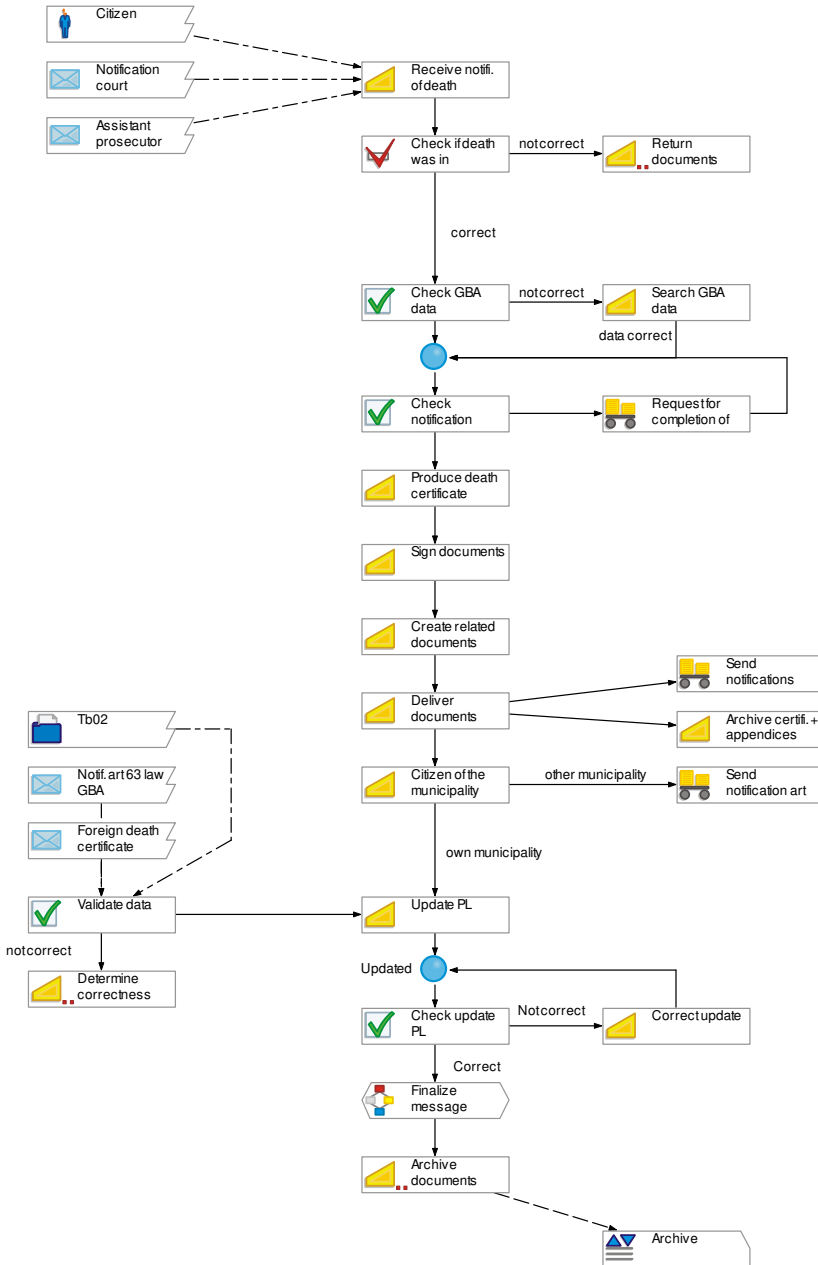


Figure A.26: Process model for *issuing a death certificate* in Municipality 3 (large): The process model of Municipality 3 is mostly line with the suggestions of the NVVB. The only differences are that Municipality 3 checks in the beginning of the process if the death happened in Municipality 3 and that the NVVB suggests checking the completeness of the documents before checking the data in the GBA database while Municipality 3 does it the other way around.

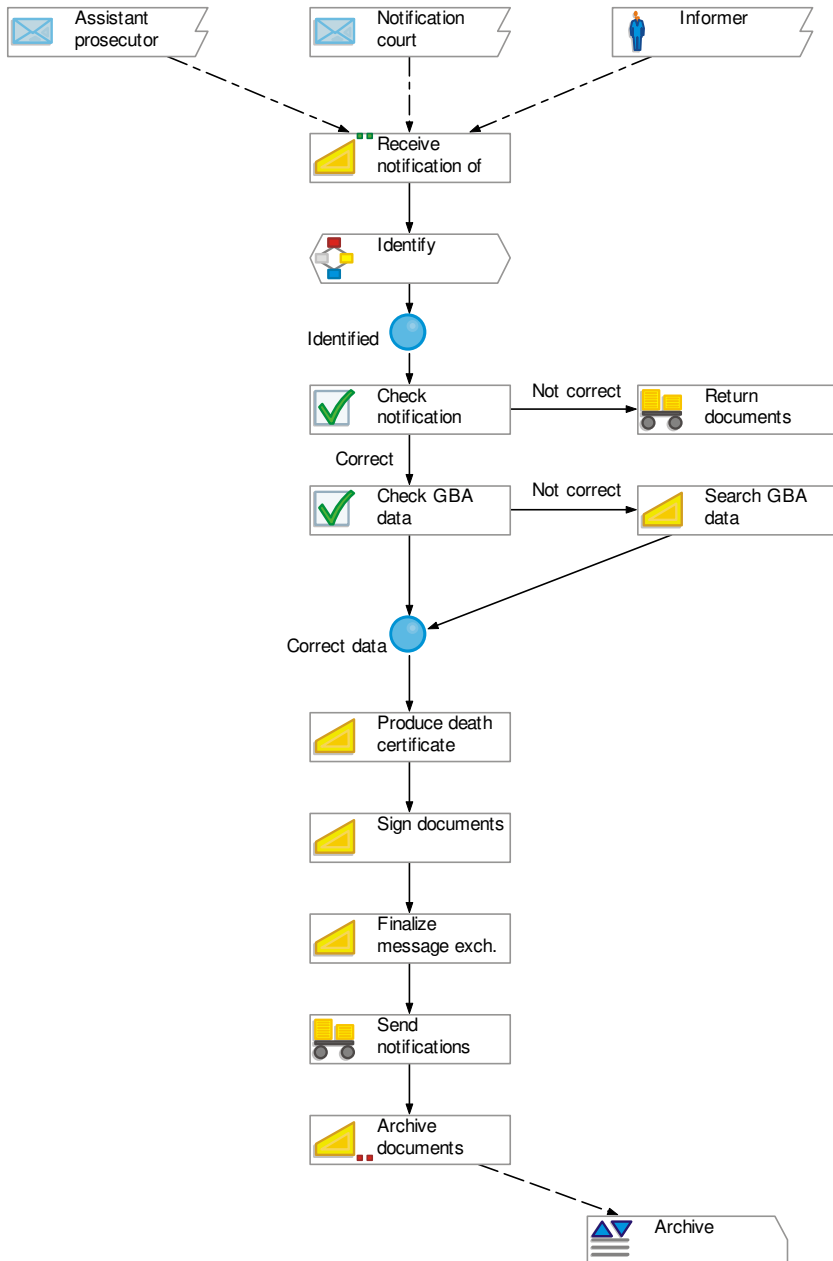


Figure A.27: Process model for *issuing a death certificate* in Municipality 4 (very large): Municipality 4's process matches in the beginning the suggestions of the NVVB. However, it lacks those parts that deal with updating the deceased's registration in the second part of the process.

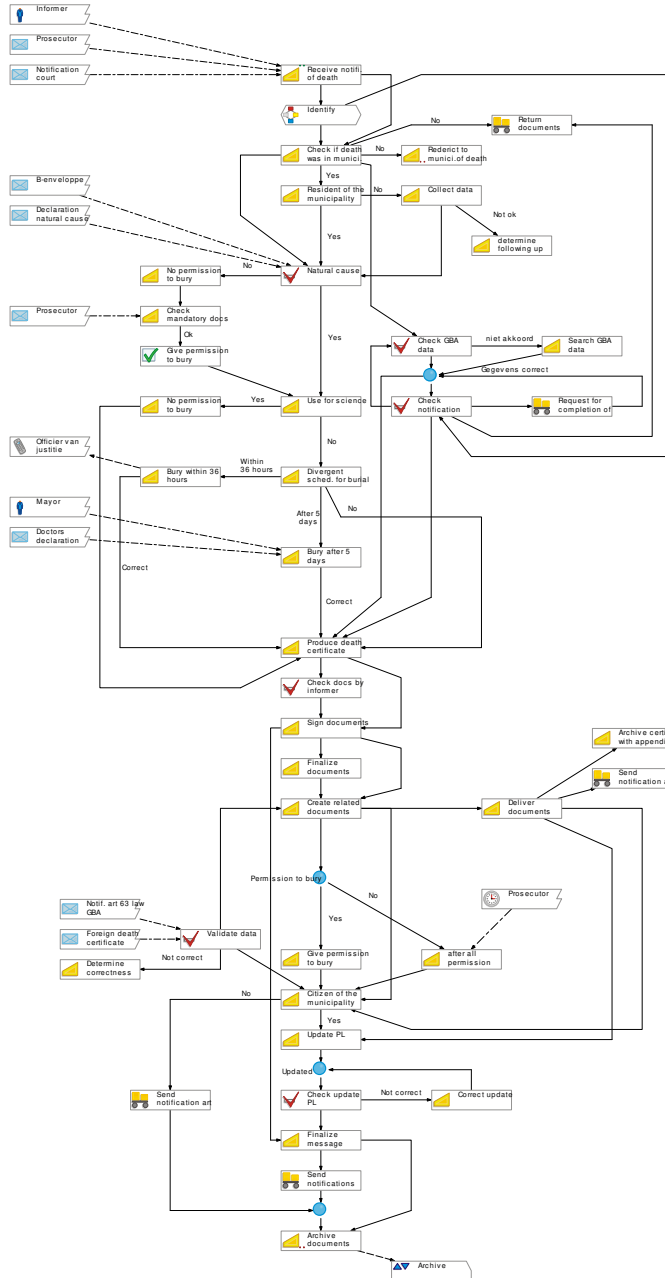


Figure A.28: Integration of the five process related variants for *issuing a death certificate*: The integrated model clearly shows that the process can be split in two parts, the phase before the documents are signed and the phase after that. In both parts a number of variations exists. Also, quite a number of arcs depict that steps are skipped in some of the models. Hence, the same behavior as indicated by these arcs could also be achieved by configuring the skipped tasks as hidden.

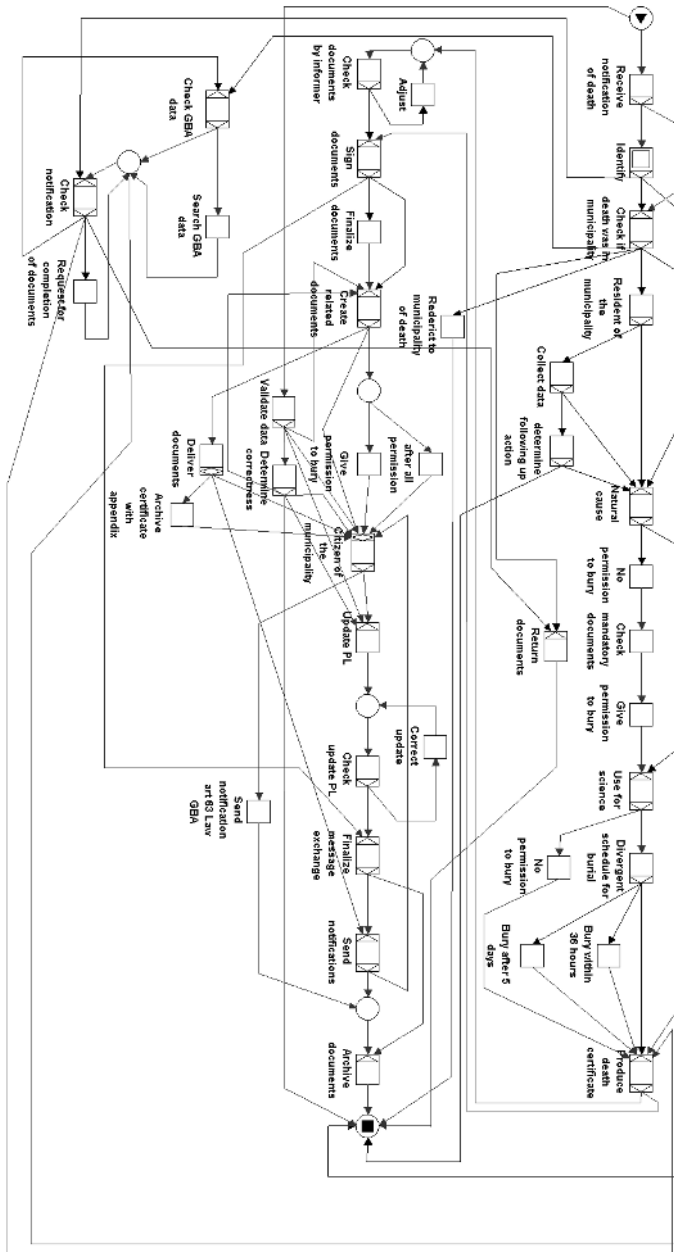


Figure A.29: The integrated process for *issuing a death certificate* transformed to YAWL: Also in the YAWL model the two phases can clearly be identified. Note the OR-split/OR-join. As is, its behavior can cause unsound behavior. To eliminate this issue, the OR-split and subsequent ports need to be restricted by configuration such that only the sound behavior remains possible.

Bibliography

- [1] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999. (cited on p. 31)
- [2] W.M.P. van der Aalst. Parallel Computation of Reachable Dead States in a Free-choice Petri Net. In A. Tentner, editor, *High Performance Computing 1998*, pages 425–432. Society of Computer Simulation, June 1998. (cited on p. 213)
- [3] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997. (cited on pp. 23, 24, 25, and 223)
- [4] W.M.P. van der Aalst. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer, 2000. (cited on p. 1)
- [5] W.M.P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J.M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer, 2001. (cited on pp. 48, 59, 60, and 61)
- [6] W.M.P. van der Aalst and C.W. Günther. Finding Structure in Unstructured Processes: The Case for Process Mining. In T. Basten, G. Juhas, and S. Shukla, editors, *Applications of Concurrency to System Design (ACSD 2007)*, pages 3–12, Bratislava, Slovak Republic, 2007. IEEE Computer Society. (cited on pp. 150, 151, and 156)
- [7] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002. (cited on p. 1)
- [8] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005. (cited on pp. 37, 39, and 40)
- [9] W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume

- 1806 of *Lecture Notes in Computer Science*, 2000. Springer. (cited on p. 1)
- [10] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003. (cited on p. 194)
- [11] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1): 5–51, 2003. (cited on pp. 26, 27, 28, 35, 63, 81, 103, 104, and 194)
- [12] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003. (cited on p. 1)
- [13] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004. (cited on p. 98)
- [14] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004. (cited on p. 195)
- [15] W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer, 2007. (cited on pp. 151, 153, 156, 194, and 197)
- [16] W.M.P. van der Aalst, M. Dumas, F. Gottschalk, A.H.M. ter Hofstede, M. La Rosa, and J. Mendling. Correctness-Preserving Configuration of Business Process Models. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE 2008)*, volume 4961 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2008. (cited on p. 12)
- [17] W.M.P. van der Aalst, M. Dumas, F. Gottschalk, A.H.M. ter Hofstede, M. La Rosa, and J. Mendling. Correctness-Preserving Configuration of Business Process Models. *Formal Aspects of Computing*, 2009. doi: 10.1007/s00165-009-0112-0. (forthcoming). (cited on p. 12)
- [18] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-step Approach to Balance between Underfitting and Overfitting. *Software and Systems*

- Modeling*, 2009. doi: 10.1007/s10270-008-0106-z. (forthcoming). (cited on p. 195)
- [19] P.A. Abdulla, S.P. Iyer, and A. Nylm. SAT-Solving the Coverability Problem for Petri Nets. *Formal Methods in System Design*, 24(1):25–43, 2004. (cited on p. 224)
- [20] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Implementing Dynamic Flexibility in Workflows using Worklets. BPM Center Report BPM-06-06, BPMcenter.org, 2006. (cited on p. 39)
- [21] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In P. Bunemann and S. Jajodia, editors, *ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM, 1993. (cited on p. 233)
- [22] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *International Conference on Extending Database Technology (EDBT'98)*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 1998. (cited on p. 195)
- [23] L. Algermissen, P. Delfmann, and B. Niehaves. Experiences in Process-oriented Reorganisation through Reference Modelling in Public Administrations — The Case Study Regio@KomM. In Bartmann D., Rajola F., Kallinikos J., Avison D., Winter R., Ein-Dor P., Becker J., Bodendorf F., and Weinhardt C., editors, *European Conference on Information Systems (ECIS)*, pages 1434–1445, 2005. (cited on p. 139)
- [24] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C.K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. OASIS Standard, April 2007. URL <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>. (cited on p. 44)
- [25] A.K. Alves de Medeiros. *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, 2006. (cited on pp. 156, 194, and 195)
- [26] A.K. Alves de Medeiros, C. Pedrinaci, W.M.P. van der Aalst, J. Domingue, M. Song, A. Rozinat, B. Norton, and L. Cabral. An Outlook on Semantic Business Process Mining and Monitoring. In Z. Tari R. Meersman and P. Herrero, editors, *On the Move to Meaningful Internet Systems (OTM Workshops, Part II, International Workshop on Semantic Web and Web Semantics, SWWS 2007)*, volume 4806 of *Lecture Notes in Computer Science*, 2007. (cited on p. 149)
- [27] A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: An Experimental Evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007. (cited on pp. 150, 151, and 156)
- [28] S. Arbaoui and F. Oquendo. Reuse Sensitive Process Models: Are Process Elements Software Assets Too? In D.E. Perry, editor, *10th International*

- Software Process Workshop (ISPW '96)*, pages 21–24. IEEE Computer Society, 1996. (cited on p. 4)
- [29] D.J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006. (cited on p. 4)
- [30] D. Avrilionis and P.-Y. Cunin. Process Model Reuse Support — The OP-SIS Approach. In D.E. Perry, editor, *10th International Software Process Workshop (ISPW '96)*, pages 25–28. IEEE Computer Society, 1996. (cited on p. 4)
- [31] F. Bachmann and L. Bass. Managing Variability in Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 26(3):126–132, 2001. (cited on p. 62)
- [32] T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001. (cited on pp. 46 and 49)
- [33] J. Becker, P. Delfmann, A. Dreiling, R. Knackstedt, and D. Kuropka. Configurative Process Modeling – Outlining an Approach to increased Business Process Model Usability. In M. Khosrow-Pour, editor, *Innovations Through Information Technology: Information Resources Management Association International Conference (IRMA)*. IGI Publishing, 2004. (cited on pp. 6, 61, 62, 120, and 232)
- [34] J. Becker, P. Delfmann, and R. Knackstedt. Adaptive Reference Modelling: Integrating Configurative and Generic Adaptation Techniques for Information Models. In J. Becker and P. Delfmann, editors, *Reference Modeling. Efficient Information Systems Design Through Reuse of Information Models*, pages 27–58. Springer, 2007. (cited on pp. 6, 10, 61, 62, 101, 102, 120, and 232)
- [35] T.J. Biggerstaff and A.J. Perlis, editors. *Software Reusability: Vol. 2, Applications and Experience*. ACM, 1989. (cited on p. 4)
- [36] T.J. Biggerstaff and A.J. Perlis, editors. *Software Reusability: Vol. 1, Concepts and Models*. ACM, 1989. (cited on p. 4)
- [37] M.-J. Blin, J. Wainer, and C. Bauzer Medeiros. A Reuse-Oriented Workflow Definition Language. *International Journal of Cooperative Information Systems*, 12(1):1–36, 2003. (cited on p. 6)
- [38] J. vom; Brocke. *Referenzmodellierung: Gestaltung und Verteilung von Konstruktionsprozessen*. PhD thesis, Westfälische Wilhelms-Universität Münster, 2003. (cited on p. 6)
- [39] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In J. Bézivin, J.-M. Favre, and B. Rumpe, editors, *International Workshop on Global Integrated Model Management (GaMMA '06)*, pages 5–12. ACM, 2006. (cited on pp. 196 and 197)
- [40] P. Cabena, P. Hasjanian, R. Stadler, J. Verhees, and A. Zanasi. *Discovering Data Mining: From Concept to Implementation*. Prentice-Hall, 1998. (cited on p. 149)

- [41] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999. (cited on p. 222)
- [42] I. Classen, H. Weber, and Y. Han. Towards Evolutionary and Adaptive Workflow Systems—infrastructure Support Based on Higher-Order Object Nets and CORBA. In Z. Milosevic, editor, *International Enterprise Distributed Object Computing Conference (EDOC '97)*, pages 300–308. IEEE Computer Society, 1997. ISBN 0-8186-8031-8. (cited on p. 10)
- [43] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on p. 62)
- [44] J.E. Cook and A.L. Wolf. Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176, 1999. (cited on pp. 156 and 197)
- [45] J.E. Cook, Z. Du, C. Liu, and A.L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, 53(3):297–319, 2004. (cited on p. 195)
- [46] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri Nets from State-Based Models. In R. Rudell and R.A. Rutenbar, editors, *IEEE/ACM International Conference on Computer-aided Design (ICCAD '95)*, pages 164–171. IEEE Computer Society, 1995. (cited on p. 197)
- [47] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997. (cited on p. 197)
- [48] T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997. (cited on pp. 2, 4, 139, and 150)
- [49] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Robert Glück and Michael Lowry, editors, *International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676/2005 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005. (cited on pp. 10 and 102)
- [50] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. (cited on p. 120)
- [51] M. Daneva. ERP Requirements Engineering Practice: Lessons Learned. *IEEE Software*, 21(2):26–33, 2004. (cited on pp. 2 and 4)
- [52] T.H. Davenport. Putting the Enterprise into the Enterprise System. *Harvard Business Review*, 76(4):121–131, 1998. (cited on p. 2)
- [53] J. Desel and T. Erwin. Modeling, Simulation and Analysis of Business Processes. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors,

- Business Process Management, Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes In Computer Science*, pages 129–141. Springer, 2000. ISBN 3-540-67454-3. (cited on p. 1)
- [54] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995. (cited on pp. 25 and 211)
- [55] B.F. van Dongen. *Process Mining and Verification*. PhD thesis, Eindhoven University of Technology, 2007. (cited on p. 194)
- [56] B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Process Mining: Building Instance Graphs. In P. Atzeni, W. Chu, H. Lu, S. Zhou, and T.W. Ling, editors, *International Conference on Conceptual Modeling (ER 2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2004. (cited on pp. 153, 155, 156, 158, 185, 186, 188, 194, and 198)
- [57] B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Mining: Aggregating Instance Graphs into EPCs and Petri Nets. In D. Marinescu, editor, *International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PNCWB 2005)*, pages 35–58, Miami, FL, 2005. Florida International University. (cited on pp. 32, 151, 153, 155, 156, 158, 171, 185, 186, 188, 194, 195, and 198)
- [58] B.F. van Dongen, M.H. Jansen-Vullers, H.M.W. Verbeek, and W.M.P. van der Aalst. Verification of the SAP Reference Models Using EPC Reduction, State-space Analysis, and Invariants. *Computers in Industry*, 58(6):578–601, 2007. (cited on pp. 6 and 234)
- [59] A. Dreiling, M. Rosemann, W.M.P. van der Aalst, W. Sadiq, and S. Khan. Model-Driven Process Configuration of Enterprise Systems. In O.K. Ferstl, E.J. Sinz, S. Eckert, and T. Isselhorst, editors, *Wirtschaftsinformatik 2005. eEconomy, eGovernment, eSociety*, pages 687–706. Physica-Verlag, 2005. (cited on pp. 61, 62, and 63)
- [60] A. Dreiling, M. Rosemann, W.M.P. van der Aalst, and W. Sadiq. From Conceptual Process Models to Running Workflows: A Holistic Approach for the Configuration of Enterprise Systems. *Decision Support Systems*, 45(2):189–207, 2008. (cited on p. 10)
- [61] A. Ebersbach, M. Glaser, R. Heigl, and A. Warta. *Wiki: Web Collaboration*. Springer, 2008. (cited on p. 235)
- [62] E. Ellmer, D. Merkl, G. Quirchmayr, and A. M. Tjoa. Process Model Reuse to Promote Organizational Learning in Software Development. In *Conference on Computer Software and Applications (COMPSAC '96)*, pages 21–26. IEEE Computer Society, 1996. (cited on p. 4)
- [63] E.A. Emerson and J.Y. Halpern. “Sometimes” and “Not Never” Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986. (cited on p. 222)

- [64] G. Faustmann. Configuration for Adaptation – A Human-centered Approach to Flexible Workflow Enactment. *Computer Supported Cooperative Work (CSCW)*, 9(3–4):413–434, 2000. (cited on p. 10)
- [65] John Favaro. What Price Reusability?: A Case Study. *Ada Letters*, XI(3):115–124, 1991. (cited on p. 4)
- [66] P. Fettke and P. Loos. Classification of Reference Models — a Methodology and its Application. *Information Systems and e-Business Management*, 1(1):35–53, 2003. (cited on p. 4)
- [67] P. Fettke, P. Loos, and J. Zwicker. Business Process Reference Models: Survey and Classification. In C. Bussler and A. Haller, editors, *Business Process Management Workshops, Workshop on Business Process Reference Models (BPRM 2005)*, volume 3812 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2006. (cited on pp. 4 and 139)
- [68] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O’ Reilly, 2004. (cited on p. 4)
- [69] P. Freeman, editor. *Tutorial, Software Reusability*. IEEE Computer Society, 1987. (cited on p. 4)
- [70] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995. (cited on p. 1)
- [71] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996. (cited on pp. 21 and 46)
- [72] F. Gottschalk, M. Rosemann, and W.M.P. van der Aalst. My Own Process: Providing Dedicated Views on EPCs. In M. Nüttgens and F. Rump, editors, *Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2005)*, volume 167 of *CEUR Workshop Proceedings*, pages 156–175, 2005. (cited on p. 223)
- [73] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Configurable Process Models – A Foundational Approach. In J. Becker and P. Delfmann, editors, *Reference Modeling. Efficient Information Systems Design Through Reuse of Information Models*, pages 59–78. Springer, 2007. (cited on p. 12)
- [74] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. SAP WebFlow Made Configurable: Unifying Workflow Templates into a Configurable Model. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 262–270. Springer, 2007. (cited on p. 12)
- [75] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and H.M.W. Verbeek. Protos2CPN: Using Colored Petri Nets for Configuring and Testing Business Processes. *International Journal on Software Tools for Tech-*

- nology Transfer (STTT)*, 10(1):95–110, 2007. (cited on pp. 12, 33, 185, and 233)
- [76] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Mining Reference Process Models and their Configurations. In R. Meersman, Z. Tari, and P. Herrero, editors, *On the Move to Meaningful Internet Systems (OTM Workshops, Workshop on Enterprise Integration, Interoperability and Networking, EI2N 2008)*, volume 5333 of *Lecture Notes in Computer Science*, pages 263–272. Springer, 2008. (cited on p. 12)
- [77] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Merging Event-driven Process Chains. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems (OTM, Part I, International Conference on Cooperative Information Systems, CoopIS 2008)*, volume 5331 of *Lecture Notes in Computer Science*, pages 418–426. Springer, 2008. (cited on p. 12)
- [78] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa. Configurable Workflow Models. *International Journal of Cooperative Information Systems (IJCIS)*, 17(2):177–221, 2008. (cited on pp. 12 and 98)
- [79] F. Gottschalk, T.A.C. Wagemakers, M.H. Jansen-Vullers, W.M.P. van der Aalst, and M. La Rosa. Configurable Process Models: Experiences from a Municipality Case Study. In P. van Eck and J.Gordijn, editors, *International Conference on Advanced Information Systems Engineering (CAiSE 09)*, volume 5565 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2009. (cited on p. 12)
- [80] J. Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981. (cited on pp. 219 and 225)
- [81] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Mining Expressive Process Models by Clustering Workflow Traces. In H. Dai, R. Srikant, and C. Zhang, editors, *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2004)*, volume 3056 of *Lecture Notes in Computer Science*, pages 52–62. Springer, 2004. (cited on p. 195)
- [82] C.W. Günther. *Process Mining in Unstructured Environments*. PhD thesis, Eindhoven University of Technology, 2009. (cited on p. 194)
- [83] C.W. Günther and W.M.P. van der Aalst. A Generic Import Framework for Process Event Logs. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, Workshop on Business Process Intelligence (BPI 2006)*, volume 4103, pages 81–92. Springer, 2006. (cited on pp. 146 and 185)
- [84] C.W. Günther and W.M.P. van der Aalst. Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in*

- Computer Science*, pages 328–343. Springer, 2007. (cited on pp. 150, 151, 156, and 195)
- [85] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computer Surveys (CSUR)*, 15(4):287–317, 1983. (cited on pp. 219 and 225)
- [86] T. D. Han, S. Purao, and V.C. Storey. A Methodology for Building a Repository of Object-Oriented Design Fragments. In *International Conference on Conceptual Modeling (ER '99)*, pages 203–217. Springer, 1999. (cited on pp. 6 and 10)
- [87] Y. Han, A. Sheth, and C. Bussler. A Taxonomy of Adaptive Workflow Management. In *Workshop of the 1998 ACM Conference on Computer Supported Cooperative Work*, Seattle, WA, 1998. (cited on pp. 10 and 233)
- [88] K.M. van Hee, A. Serebrenik, N. Sidorova, and M. Voorhoeve. Soundness of Resource-Constrained Workflow Nets. In G. Ciardo and P. Darondeau, editors, *International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 250–267, 2005. (cited on pp. 222 and 223)
- [89] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006. (cited on p. 224)
- [90] J. Herbst and D. Karagiannis. Workflow Mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004. (cited on p. 195)
- [91] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004. (cited on p. 11)
- [92] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2005. (cited on p. 44)
- [93] Y. Huang, H. Wang, W. Zhao, and J. Zhu. Analyzing Data Dependence Based on Workflow Net. In Y. Shi, G.D. van Albada, J. Dongarra, and P.M.A. Sloot, editors, *International Conference on Computational Science (ICCS '07, Part III)*, volume 4489 of *Lecture Notes In Computer Science*, pages 257–264. Springer, 2007. (cited on p. 224)
- [94] IDS Scheer AG. ARIS Platform - Product Brochure, 2008. URL <http://www.ids-scheer.com/set/6473/Product%20Brochure%202008-07.pdf>. [accessed 07-07-2009]. (cited on p. 32)
- [95] M.L. Jaccheri, G.P. Picco, and P. Lago. Eliciting Software Process Models with the E^3 Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4):368–410, 1998. (cited on p. 4)
- [96] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997. (cited on p. 62)

- [97] M.H. Jansen-Vullers, W.M.P. van der Aalst, and M. Rosemann. Mining Configurable Enterprise Information Systems. *Data and Knowledge Engineering*, 56(3):195 – 244, 2006. (cited on pp. 197 and 198)
- [98] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3–4):213–254, 2007. (cited on pp. 28 and 33)
- [99] P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work (CSCW)*, 9(3):269–292, 2000. (cited on pp. 10 and 233)
- [100] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, USA, 1990. (cited on p. 121)
- [101] A. Kaplan. *The Conduct of Inquiry: Methodology for Behavioral Science*. Chandler Publications in Anthropology and Sociology. Chandler Pub. Co., 1964. (cited on p. 11)
- [102] M. Karow, D. Pfeiffer, and M. Räckers. Empirical-Based Construction of Reference Models in Public Administrations. In M. Bichler, T. Hess, H. Krcmar, U. Lechner, F. Matthes, A. Picot, B. Speitkamp, and P. Wolf, editors, *Multikonferenz Wirtschaftsinformatik 2008. Referenzmodellierung*, pages 1613–1624. GITO-Verlag, 2008. (cited on p. 139)
- [103] G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik 89, University of Saarland, Saarbrücken, Germany, 1992. (in German). (cited on p. 28)
- [104] K. Kennedy and J.R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2002. (cited on p. 224)
- [105] B. Kiepuszewski, A. H. M. Ter Hofstede, and W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. *Acta Informatica*, 39:143–209, 2002. (cited on p. 213)
- [106] H. Kilov. Generic Information Modeling Concepts: A Reusable Component Library. In J. Bézivin and B. Meyer, editors, *International Conference on Technology of Object-oriented Languages and Systems (TOOLS 4)*, pages 187–201. Prentice-Hall, 1991. (cited on p. 6)
- [107] E. Kindler. On the Semantics of EPCs: Resolving the Vicious Circle. *Data and Knowledge Engineering*, 56(1):23–40, 2006. (cited on p. 31)
- [108] A. Koschmider and E. Blanchard. User Assistance for Business Process Model Decomposition. In *IEEE International Conference on Research Challenges in Information Science*, pages 445–454, 2007. (cited on p. 6)

- [109] C.W. Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992. (cited on p. 4)
- [110] M. La Rosa. *Managing Variability in Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, Brisbane, 2009. (cited on p. 119)
- [111] M. La Rosa, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Questionnaire-based Variability Modeling for System Configuration. *Software and Systems Modeling*, 8(2):251–274, 2008. (cited on pp. 105, 106, 109, and 120)
- [112] M. La Rosa, M. Dumas, A.H.M. ter Hofstede, J. Mendling, and F. Gottschalk. Beyond Control-Flow: Extending Business Process Configuration to Roles and Objects. In Q. Li, S. Spaccapietra, E. Yu, and A. Olivé, editors, *International Conference on Conceptual Modeling (ER '08)*, volume 5231 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008. (cited on pp. 216, 218, 221, and 223)
- [113] M. La Rosa, F. Gottschalk, M. Dumas, and W.M.P. van der Aalst. Linking Domain Models and Process Models for Reference Model Configuration. In A. ter Hofstede, B. Benatallah, and H.-Y. Paik, editors, *Business Process Management Workshops, Reference Model Workshop (RefMod 2007)*, volume 4928 of *Lecture Notes in Computer Science*, pages 417–430. Springer, 2008. (cited on pp. 12, 105, and 106)
- [114] M. La Rosa, A.H.M. ter Hofstede, M. Rosemann, and K. Shortland. Bringing Process to Post Production. In *International Conference on "Creating Value: Between Commerce and Commons"*, Brisbane, Australia, 2008. Queensland University of Technology. (cited on p. 139)
- [115] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In M. Dumas and R. Heckel, editors, *International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2008. (cited on p. 44)
- [116] R.J. Malak, Jr. and C.J.J. Paredis. Foundations of Validating Reusable Behavioral Models in Engineering Design Problems. In R.G. Ingalls, M.D. Rossetti, J.S. Smith, and B.A. Peters, editors, *Winter Simulation Conference (WSC '04)*, pages 420–428. IEEE Computer Society, 2004. (cited on p. 4)
- [117] Salvatore T. March and Gerald F. Smith. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4): 251–266, 1995. (cited on p. 11)
- [118] J. Mendling and C. Simon. Business Process Design by View Integration. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, Workshop on Business Process Design (BPD 2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 2006. (cited on p. 196)
- [119] J. Mendling, H.M.W. Verbeek, B.F. van Dongen, W.M.P. van der Aalst, and G. Neumann. Detection and Prediction of Errors in EPCs of the SAP

- Reference Model. *Data and Knowledge Engineering*, 64(1):312–329, 2008. (cited on pp. 150 and 234)
- [120] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. (cited on p. 21)
- [121] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In R.C. Smith, editor, *ACM/IEEE Design Automation Conference*, pages 52–57. ACM, 1990. (cited on pp. 59, 71, 114, 115, and 206)
- [122] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. (cited on p. 25)
- [123] A. Oberweis. Person-to-Application Processes: Workflow Management. In M.Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede, editors, *Process-Aware Information Systems*, pages 21–36. Wiley & Sons, 2005. (cited on p. 1)
- [124] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(2–3):162–198, 2007. (cited on p. 44)
- [125] Pallas Athena BV. *Protos User Manual*. Plasmolen, The Netherlands, 2004. (cited on p. 34)
- [126] M.H. Pesic, M. Schonenberg, N. Sidorova, and W.M.P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems (OTM, International Conference on Cooperative Information Systems, CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 77–94, 2007. (cited on p. 224)
- [127] J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981. (cited on p. 25)
- [128] S.S. Pinter and M. Golani. Discovering workflow models from activities’ lifespans. *Computers in Industry*, 53(3):283–296, 2004. (cited on p. 195)
- [129] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering — Foundations, Principles and Techniques*. Springer, 2005. (cited on p. 120)
- [130] R. S. Pressman. *Software Engineering: A Practitioner’s Approach*. Higher Education. Mc Graw Hill, 6th edition, 2005. (cited on p. 121)
- [131] O.O. Prisecaru. Resource workflow nets: an approach to workflow modelling and analysis. *Enterprise Information Systems*, 2(2):101–120, 2008. (cited on pp. 222 and 223)
- [132] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske. Variability Mechanisms for Process Models. PESOA-Report TR 17/2005, Process Family Engineering in Service-Oriented Applications (PESOA), June, 2005. (cited on pp. 10, 61, 62, and 102)

- [133] D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann, 1999. (cited on p. 149)
- [134] I. Raedts, M. Petković, Y.S. Usenko, J.M. van der Werf, J.F. Groote, and L. Somers. Transformation of BPMN Models for Behaviour Analysis. In J.C. Augusto, J. Barjis, and U. Ultes-Nitsche, editors, *International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS '07)*, pages 126–137. INSTICC Press, 2007. (cited on p. 35)
- [135] E. S. Raymond. The CML2 Language. In *International Python conference*, 2001. (cited on p. 120)
- [136] M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998. (cited on p. 224)
- [137] H.A. Reijers, R.S. Mans, and R.A. van der Toorn. Improved model management with aggregated business process models. *Data and Knowledge Engineering*, 68(2):221–243, 2009. (cited on pp. 101 and 102)
- [138] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, 1998. Springer. (cited on p. 25)
- [139] A. Rickayzen, J. Dart, C. Brennecke, and M. Schneider. *Practical Workflow for SAP – Effective Business Processes using SAP’s WebFlow Engine*. Galileo Press, 2002. (cited on pp. 2, 4, and 42)
- [140] S. Rinderle, M. Reichert, and P. Dadam. Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems (OTM, International Conference on Cooperative Information Systems, CoopIS 2004)*, volume 3290, pages 101–120, 2004. (cited on pp. 10 and 233)
- [141] R. Roberts. The Rise and Fall of the Public Law Litigation Model: Implications for Public Management. *Public Administration and Management*, 13(1):51–106, 2008. (cited on p. 123)
- [142] M. Rosemann. Preparation of Process Modeling. In J. Becker, M. Kugeler, and M. Rosemann, editors, *Process Management: A Guide for the Design of Business Processes*, pages 41–78. Springer, 2003. (cited on p. 1)
- [143] M. Rosemann and W.M.P. van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, 2007. (cited on pp. 10, 30, 98, 99, 100, and 119)
- [144] A. Rozinat and W.M.P. van der Aalst. Conformance Checking of Processes based on Monitoring Real Behavior. *Information Systems*, 33(1):64–95, 2008. (cited on pp. 8, 10, 134, 178, 191, and 197)
- [145] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Busler et al., editor, *Business Process Management Workshops, Workshop on Business Process Intelligence (BPI 2005)*, volume 3812 of *Lecture Notes*

- in Computer Science*, pages 163–176. Springer, 2005. (cited on pp. 8, 10, and 197)
- [146] A. Rozinat, R.S. Mans, M.S. Song, and W.M.P. van der Aalst. Discovering colored Petri nets from event logs. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(1):57–74, 2008. (cited on pp. 185 and 233)
- [147] A. Rozinat, A.K. Alves de Medeiros, C.W. Günther, A. J. M. M. Weijters, and W.M.P. van der Aalst. The Need for a Process Mining Evaluation Framework in Research and Practice. In A. ter Hofstede, B. Benatalah, and H.-Y. Paik, editors, *Business Process Management Workshops, Workshop on Business Process Intelligence (BPI 2007)*, volume 4928 of *Lecture Notes in Computer Science*, pages 84–89. Springer, 2008. (cited on p. 156)
- [148] N. Russell. *Foundations of Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, 2007. (cited on p. 28)
- [149] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Resource Patterns. BETA Working Paper Series WP 127, Eindhoven University of Technology, 2004. (cited on p. 28)
- [150] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2005. (cited on p. 28)
- [151] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org, 2006. (cited on p. 28)
- [152] M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering*, 11(3):174–193, 2006. (cited on pp. 195 and 196)
- [153] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. A Relationship-Driven Framework for Model Merging. In *International Workshop on Modeling in Software Engineering (MISE '07)*, pages 2–8. IEEE Computer Society, 2007. (cited on p. 196)
- [154] S.W. Sadiq, M.E. Orlowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Information Systems*, 30(5): 349–378, 2005. (cited on p. 224)
- [155] SAP AG. SAP History: From Start-Up Software Vendor to Global Market Leader, 2008. URL <http://www.sap.com/company/history.epx>. [accessed 27-10-2008]. (cited on pp. 4 and 35)
- [156] SAP AG. Graphic: Approve Travel Request, 2006. URL http://help.sap.com/saphelp_erp2005vp/helpdata/en/04/928b1846f311d189470000e829fbbd/frameset.htm. [accessed 07-07-2009]. (cited on p. 5)

- [157] SAP AG. Automatically Approve Travel Requests, 2006. URL http://help.sap.com/saphelp_erp2005vp/helpdata/en/f5/4fe23cab43ba5be10000000a114084/frameset.htm. [accessed 07-07-2009]. (cited on p. 5)
- [158] K. Sarshar and P. Loos. Comparing the Control-Flow of EPC and Petri Net from the End-User Perspective. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 434–439. Springer, 2005. (cited on p. 32)
- [159] A.-W. Scheer. *Architecture of Integrated Information Systems : Foundations of Enterprise Modelling*. Springer, 1992. (cited on p. 1)
- [160] A.-W. Scheer. ARIS Toolset: A Software Product is Born. *Information Systems*, 19(8):607–624, 1994. (cited on p. 32)
- [161] A.-W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer, 1994. (cited on pp. 4 and 139)
- [162] A.-W. Scheer and F. Habermann. Enterprise resource planning: making ERP a success. *Communications of the ACM*, 43(4):57–61, 2000. (cited on p. 2)
- [163] A.-W. Scheer, O. Thomas, and O. Adam. Process Modeling Using Event-Driven Process Chains. In M.Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede, editors, *Process-Aware Information Systems*, chapter 6, pages 119–145. Wiley & Sons, 2005. (cited on pp. 1 and 32)
- [164] A.W. Scheer. *ARIS - Business Process Frameworks*. Springer, 3rd edition, 1999. (cited on p. 32)
- [165] G. Schimm. Mining Exact Models of Concurrent Workflows. *Computers in Industry*, 53(3):265–281, 2004. (cited on p. 195)
- [166] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In M. Glinz and R. Lutz, editors, *IEEE International Conference on Requirements Engineering (RE '06)*, pages 139 – 148. IEEE Computer Society, 2006. (cited on p. 121)
- [167] E. Schwartz. Who’s in Charge, Anyway? Software Should Work for Us, Not the Other Way Around, March 2004. URL <http://www.infoworld.com/d/developer-world/whos-in-charge-anyway-989?page=0,0>. [accessed 07-07-2009]. (cited on p. 2)
- [168] S. Seidel, M. Rosemann, A.H.M. ter Hofstede, and L. Bradford. Developing a Business Process Reference Model for the Screen Business - A Design Science Research Case Study. In S. Spencer and A. Jenkins, editors, *Australasian Conference on Information Systems (ACIS 2006)*. Australasian Association for Information Systems, 2006. (cited on p. 139)
- [169] D. Seo and P. Loucopoulos. Formalisation of Data and Process Model Reuse Using Hierarchic Data Types. In *International Conference on Advanced Information Systems Engineering (CAiSE '94)*, pages 256–268. Springer, 1995. (cited on p. 4)

- [170] P. Soffer, B. Golany, and D. Dori. ERP Modeling: A Comprehensive Approach. *Information Systems*, 28(6):673–690, 2003. (cited on p. 120)
- [171] S. Stephens. The Supply Chain Council and the Supply Chain Operations Reference Model. *Supply Chain Management — An International Journal*, 1(1):9–13, 2001. (cited on p. 139)
- [172] T. Stoesser. Don't Share If You Don't Want To, April 2009. URL http://communities.softwareag.com/ecosystem/communities/alignspace/2009/dont_share_if_you_dont_want_to.html. [accessed 06-05-2009]. (cited on p. 235)
- [173] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. In F. van der Linden, editor, *International Workshop on Software Architectures for Product Families (IW-SAPF-3)*, volume 1951 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2000. (cited on p. 62)
- [174] M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005. (cited on p. 62)
- [175] S. Tam, W.B. Lee, W.W.C. Chung, and E.L.Y. Nam. Design of a Reconfigurable Workflow System for Rapid Product Development. *Business Process Management Journal*, 9(1):33–45, 2003. (cited on p. 10)
- [176] O. Thomas, B. Hermes, and P. Loos. Towards a Reference Process Model for Event Management. In A. ter Hofstede, B. Benatallah, and H.-Y. Paik, editors, *Business Process Management Workshops, Reference Model Workshop (RefMod 2007)*, volume 4928 of *Lecture Notes in Computer Science*, pages 443–454. Springer, 2008. (cited on p. 139)
- [177] W. Tracz. *Software Reuse: Emerging Technology*. IEEE Computer Society, 1988. (cited on p. 4)
- [178] N. Trčka, W.M.P. van der Aalst, and N. Sidorova. Analyzing Control-Flow and Data-Flow in Workflow Processes in a Unified Way. Computer Science Report 08-31, Eindhoven University of Technology, 2008. (cited on p. 224)
- [179] N. Trčka, W.M.P. van der Aalst, and N. Sidorova. Data-Flow Anti-Patterns: Discovering Dataflow Errors in Workflows. In P. van Eck and J. Gordijn, editors, *International Conference on Advanced Information Systems (CAiSE '09)*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009. (cited on pp. 216, 217, 218, 219, 220, 221, and 224)
- [180] S. Uchitel and M. Chechik. Merging Partial Behavioural Models. *SIGSOFT Software Engineering Notes*, 29(6):43–52, 2004. (cited on p. 197)
- [181] H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL Processes using Petri Nets. In D. Marinescu, editor, *International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PNCWB 2005)*, pages 59–78, Miami, FL, 2005. Florida International University. (cited on p. 44)

- [182] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001. (cited on pp. 25, 34, 214, 223, 224, and 225)
- [183] H.M.W. Verbeek, M. van Hattem, H.A. Reijers, and W. de Munk. Protos 7.0: Simulation Made Accessible. In G. Ciardo and P. Darondeau, editors, *International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 465–474, Miami, USA, 2005. Springer. (cited on p. 32)
- [184] A. Vinter Ratzter, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In W.M.P. van der Aalst and E. Best, editors, *International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer, 2003. (cited on pp. 28 and 33)
- [185] T.A.C. Wagemakers. Configurable workflow models: A case study. Master’s thesis, Eindhoven University of Technology, 2009. (cited on p. 140)
- [186] A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integral Computer-Aided Engineering*, 10(2):151–162, 2003. (cited on pp. 150, 151, 156, and 195)
- [187] L. Wells. Monitoring a CP-net, March 2006. URL http://wiki.daimi.au.dk/cpntools-help/monitoring_a_cp-net.wiki. [accessed 07-07-2009]. (cited on p. 33)
- [188] L. Wen, J. Wang, and J. Sun. Detecting Implicit Dependencies Between Tasks from Event Logs. In X. Zhou, J. Li, H.T. Shen, M. Kitsuregawa, and Y. Zhang, editors, *Asia-Pacific Web Conference on Frontiers of WWW Research and Development (APWeb 2006)*, volume 3841 of *Lecture Notes in Computer Science*, pages 591–603. Springer, 2006. (cited on p. 195)
- [189] J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. Process Discovery Using Integer Linear Programming. In K.M. van Hee and R. Valk, editors, *International Conference on Applications and Theory of Petri Nets (PETRI NETS ’08)*, volume 5062 of *Lecture Notes in Computer Science*, pages 368–387. Springer, 2008. (cited on p. 195)
- [190] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007. (cited on p. 1)
- [191] S.A. White and D. Miers. *BPMN Modeling and Reference Guide*. Future Strategies, 2008. (cited on p. 34)
- [192] S.A. White et al. Business Process Modeling Notation (BPMN), Version 1.2. OMG specification, 2008. URL <http://www.omg.org/spec/BPMN/1.2/>. [accessed 07-07-2009]. (cited on p. 34)
- [193] S. Zhang, C. Zhang, and Q. Yang. Data Preparation for Data Mining. *Applied Artificial Intelligence*, 17(5):375–381, 2003. (cited on p. 149)

Index

- absence of livelocks, 24
- abstract class, 6
- abstract from behavior, 228
- abstract view, 195
- abstraction, 47, 49
- academia, 15
- accept travel request, 29, 175
- access data, 216, 219, 221
- accommodation, 156
- accommodation quote, 22, 55
- ACID, 225
- acknowledging an unborn child, 125, 127–129, 185, 188, 191
- activate event
 - SAP WebFlow, 71, 72
- activate link
 - BPEL, 43
 - SAP WebFlow, 68, 69
- active behavior, 159
- activity
 - BPEL, 42, 72, 74, 75
 - BPMN, 34
 - Protos, 32
 - SAP WebFlow, 40, 66
- activity diagram, 62, 102
- acyclic path, 204, 205
- adapt model, 45, 61
- adapt template manually, 5
- adaptation mechanism, 61, 62
- adaptation option, 61
- adaptation practices, 61
- adaptation technique, 6, 62
- adaptive workflow system, 10
- add behavior, 10, 45, 46, 48, 61, 63, 151, 220, 228, 232
- add content, 6, 45
- add functionality, 46
- add information, 45
- add property, 46
- add task, 216, 217
- add transition, 60
- adding a delay, 26
- additional behavior, 46, 178, 197, 199
- additional choices, 156
- additional functionality, 46
- address verification, 90
- ADEPTflex, 224
- administration, 19, 22, 34, 55, 56, 123, 137
- administration process, 123
- administrator, 22
- advanced notation, 21
- advanced pattern, 27
- advanced process modeling language, 15, 28, 52, 103
- age verification, 90
- aggregated EPC, 101
- aggregating log events, 148
- aggregation, 6, 232
- align system description, 195
- align variants, 198
- aligning log files, 198
- AlignSpace, 235
- allow all variants, 59
- allow behavior, 45, 65
- allow outflow, 58
- allow port, 54, 55, 59, 63, 68
 - BPEL, 72, 74
 - YAWL, 78–80, 84–86, 88–90, 92–95, 129
- allow state changes, 54
- allow step, 68
 - SAP WebFlow, 71
- allow task, 105

- allow transition
 - workflow net, 55, 58, 180, 181, 183
- alternative execution, 1
- alternative implementation, 62
- alternative path, 216, 217
- alternative task, 217
- alternative to transition, 180
- amend configuration, 178
- analysis plug-in
 - ProM, 194
- analysis technique, 25
- analyzing complex systems, 28
- \wedge connector
 - EPC, 29, 32, 162, 169, 171, 196
- \wedge join type
 - function graph, 161, 164, 167, 169
- \wedge split type
 - function graph, 160, 161, 163, 166, 168, 169, 171
- AND-join, 27, 29, 32, 34, 36, 43, 53, 55, 74, 76, 77, 81, 94, 96, 158, 198, 214, 215
- AND-split, 27, 29, 32, 34, 36, 53, 55, 77, 78, 92, 93, 96, 101, 102, 158, 198, 214, 215
- anonymous data, 148
- anti-pattern, 216, 224
- appropriate, 10
- appropriateness, 197
- approval notification, 34
- approve requisition, 22
- approve travel request, 19, 66, 70, 71, 172
- arbitrary transitions, 48
- arc
 - EPC, 30, 100, 101
 - function graph, 159, 162–164
 - Petri net, 21, **21**, 191
 - Protos, 32
 - UML, 102
 - YAWL, 36, 76–78
- ARIS, 28, 32
- ask for update, 56
- ask question, 108
- assign resource
 - SAP WebFlow, 72
- assign type to arc, 159, 161–163, 165–168, 171
- assistant, 22
- association rule, 233
- atomic expression, 58
- atomic formula, 16, 17
- atomic requirement
 - YAWL, 84, 85, **85**, 86, 87, 90, **90**
- atomic statement, 71
- atomic task, 36, 194
 - BPMN, 34
- atomicity, 225
- audit trail entry, 146
- authorization, 216
- automated mining, 149
- automated process execution, 10
- automatic approval, 68, 69, 71
- automatic configuration, 208
- automatic travel request approval, 5
- automatic trip approval, 5
- automotive industry, 138
- avoid configuration, 51
- avoid transition, 199
- back office, 26, 27
- balanced level of detail, 148
- bank, 138
- base case, 213
- basic process model, 52, 59–61, 63, 68, 79, 87, 98, 105, 124, 128, 129, 143–145, 156–158, 172, 178–180, 183, 188, 189, 191, 197, 199, 201, 212, 216, 217, 219–222, 225, 229, 230, 232–234
- behavior, 46, 178
 - EPC, 170, 171, 174
 - function graph, 163, 169, 171
 - YAWL, 93
- behavior of subclass, 47
- behavioral block, 27
- behavioral equivalent, 46
- behavioral impact, 46
- behavioral pattern, 148
- behavioral point of view, 174
- behavioral requirements, 165
- behavioral science, 11

- benefit, 124
- best-practice, 125, 136, 139, 143, 229, 235
- birth certificate, 125
- block
 - BPEL, 72, 75
 - SAP WebFlow, 66–71, 76
- block behavior, 65, 98, 101, 102
- block implementation, 62
- block inflow port, 54
- block outflow port, 54
- block path
 - EPC, 100
- block port, 54, 58, 59, 63, 68, 103
 - BPEL, 72, 74, 75
 - Protos, 191
 - SAP WebFlow, 72
 - YAWL, 78–81, 84, 86, 88, 90, 92–95, 97, 129, 132, 214
- block process branch, 63
- block process part, 63
- block step
 - SAP WebFlow, 68, 69, 71
- block structure, 40, 42
 - BPEL, 72, 74, 103
 - SAP WebFlow, 65, 66
- block task, 216, 217, 220, 222
 - YAWL, 215
- block transition
 - LTS, 47, 49–51
 - workflow net, 55, 56, 58, 60, 180, 181, 183, 203, 206, 208
- blocking, 47, 49, 52, 63
- book travel, 84
- book trip, 19, 51, 106
- booking engine, 42
- booking workflow, 36
- boolean condition, 40
- boolean constraint, 230
- boolean expression, 74, 204, 227
- boolean variable, 106, 109, 204
- BPEL, 12, 13, 16, 35, 42–44, 55, 64, 65, 72, 74–76, 84, 88, 102, 103, 211, 229, 231
 - configurable, 72
 - BPEL standard, 44
- BPM, 1, 3, 4, 12, 32, 135, 137
- BPM|one, 32, 137
- BPMN, 12, 16, 28, 34, 35, 62, 102, 161, 231
 - BPMN specification, 34
- branch control-flow, 30, 66
- branching behavior, 194
- branching bisimulation, 46
- branching control-flow, 194
- branching patterns, 198
- building a process model, 152
- building block, 6, 25
- building configurable process models, 143, 144
- business application, 41
- business process, 1, 3–5, 15, 19, 20, 22, 23, 27, 28, 51, 62, 158, 198
- business process modeling language, 15, 16, 25, 28
- business requirement, 232
- business transaction event, 41
- business trip, 19
- busy place, 26
- bypass, 153, 180, 181, 183
- C-, 66–68, 74–76, 84, 87–92, 96–104, 106, 111, 117–119, 121, 203, 214, 215, 225, 229, 230
- cancel booking, 36, 79
- cancel branch
 - SAP WebFlow, 71
- cancellation
 - BPMN, 34
- cancellation of behavior
 - SAP WebFlow, 42
 - YAWL, 94
- cancellation pattern, 36, 103
- cancellation port
 - YAWL, 80, **80**, 86
- cancellation region, 36
 - YAWL, 38, 80, 82, 85, 92
- capabilities of process mining algorithms, 156
- cartesian product of sets, **17**
- case, 22
 - SAP WebFlow, 68–71

- YAWL, 77
- case identifier, 22, 27
- case study, 123, 125, 128, 133, 134, 139, 140, 183, 185, 188, 189, 191, 230, 234
- case token, 26
- cash, 105
- cash payment, 36, 78
- change
 - request for, 22
- change of document
 - SAP WebFlow, 41
- change of properties, 19, 21
- change of state, 20
- change process model, 10, 233
- change travel request, 56
- characteristics of mining algorithm, 194
- checking conformance, 197
- choice, 20, 29, 150, 153, 156
 - BPEL, 42
 - Protos, 33
- choice between behavior, 196
- circle, 21, 32
- civil affairs department, 124
- claim resource, 222
- class, 46
- class of events, 151
- clause, 17
- cleaning log file, 198
- cleanup algorithm
 - YAWL, 91, 92, 96, 97, 214, 215
- clear model, 158
- clearing center, 29, 172
- coinciding post-sets, 25
- colored Petri net, 26–28, 33, 185
- combination of properties, 21
- combinations of process parts, 156
- combine configurations
 - YAWL, 87, 92
- combine process definitions, 3
- combine requirements
 - YAWL, 90
- combine variants, 198
- combined configuration
 - YAWL, 83, **83**
- combined log file, 198
- combined model, 129, 153
- combined set of traces, 156
- complete configuration
 - YAWL, 87
- commercial tool, 16
- common subclass, 59
- compact model, 158
- compact representation, 21
- company size, 178
- compare behavior, 46, 47, 197
- compare configuration approaches, 45
- compare event streams, 197
- compare models, 61
- comparing events, 150
- comparison of process mining algorithms, 156
- compensation
 - BPMN, 34
- compensation management, 4
- competitive advantage, 235
- competitor, 148
- compiler, 224
- complete configuration, 199
 - YAWL, 83, **83**, 87, 90–92
- complete firing sequence, 179, 180
- complete task, 26, 55, 63, 194
- complete task execution, 53
- complete workflow, 70
- completeness, 10, 137
- completion condition
 - SAP WebFlow, 41
- completion of case, 24
- completion of process, 23, 24
- complex system, 195
- complexity, 206
- composite task, 39
 - BPMN, 34
 - YAWL, 36, 38, 39, 88, 90
- compromise, 25, 211
- concatenate constraints, 71, 206
- concatenate log files, 151
- concatenate sequences, **18**
- concurrency, 150
- condition
 - BPEL, 43, 73

- colored Petri nets, 27
- implicit (YAWL), 38
- SAP WebFlow, 40, 66
- YAWL, 36, 37, 76–78, 80, 84–87, 92–96
- confidential, 148
- configurable block
 - SAP WebFlow, 71
- configurable connector
 - EPC, 99, 100
- configurable EWF-net, **87**
- configurable function
 - EPC, 99
- configurable LTS, **52**
- configurable port, 55, 58, 61, 65, 76
- configurable process model, 3, 11, 45, 52, **52**, 59, 61, 63, 101, 143, 145, 150, 153, 157, 158, 171, 172, 178, 198, 199, 206, 227, 228, 233
- configurable process models, 148
- configurable workflow net, **59**, 65, 214
- configurable workflow specification
 - YAWL, 88, 90, 91, **91**, 97
- configurable workflow system, 10
- configuration, 6, 178, 180, 181, 183, 197, 199
 - LTS, 49, **49**, 50
 - workflow net, 55, 56
 - YAWL, **81**
- configuration choice, 156
 - EPC, 100
- configuration constraint, 58, **58**, 59, 63, 72, 76, 138, 204, 206, 208, 214, 215, 218–221, 223, 224, 228, 230, 232
 - BPEL, 74
 - YAWL, 84, 90, 91
- configuration correctness, 140
- configuration decision, 49, 50, 52, 58, 70, 74, 76, 81, 84, 87, 88, 91, 98, 99, 105, 124, 132, 134, 138, 193, 204, 214, 224, 225, 229, 233
- configuration dependency
 - YAWL, 84
- configuration extension, 64, 66, 103
- configuration framework, 139, 215
- configuration freedom, 136
- configuration function
 - YAWL, 81, 83
- configuration mechanism, 6, 62
- configuration of log, **181**
- configuration opportunity, 45, 62, 63
- configuration option, 10, 11, 63, 72, 197
- configuration pattern, 10, 63
- configuration recommendation, 232
- configuration requirement
 - BPEL, 74
 - EPC, 101
 - YAWL, 86, 87
- configuration restriction
 - YAWL, 84
- configuration space, 63, 107
- configuration step, 208
- configuration value, 56, 71
- configuration variant, 198
- configure port, 55, 59
- configure process model, 61
- configure transition
 - LTS, 51
 - workflow net, 55
- configured
 - LTS, 49, 50, **50**, 51
 - workflow net, **56**, 57, 59, 203, 205, 215
- conform to behavior, 153
- conformance checker, 191, 197
- conjunction, 16, 205, 206
- conjunction of clauses, 17
- conjunction of literals, 17
- conjunctive normal form, 17, **17**, 206
- connection
 - Protos, 32
- connector
 - EPC, 28, 31, 99–101, 162
- connector type
 - EPC, 29, 169
- consistency, 157, 225
- constraint, 63, 201, 206, 208, 229
 - SAP WebFlow, 66, 71

- constraint violation, 222
- construct basic process model, 230
- consultancy firm, 2, 135, 138, 139, 235
- consultant, 123, 136, 138
- consulting firm, 2
- consume token, 26
 - function graph, 161, 164
- context menu, 76
- contribute to process, 23, 24
- control configuration space, 62
- control link
 - BPEL, 42, 72–74
- control-flow, 28, 36, 66, 72, 88, 102–104, 127, 129, 134, 140, 149, 163, 164, 175, 194, 215, 223, 230, 231
- control-flow correctness, 222
- control-flow pattern, 61, 63
- conversion of process models, 158
- conversion plug-in
 - ProM, 194
- conversion tools, 175
- copies of form, 23
- core process, 2
- corporate success, 2
- correct process model, 59
- correctness, 201
 - workflow net, 59
- couple of places, 25
- cover behavior, 178, 197
- coverability
 - Petri net, 224
- CPN Tools, 33
- create data, 216–221, 225
- create instance, 37
 - YAWL, 80, 81
- create process model, 151
- creation of bypass, 153
- creation of document
 - SAP WebFlow, 41
- creative addition, 62
- creative domain, 140
- creative freedom, 6
- credit card, 78, 79, 84, 105, 106, 108
- credit card payment, 36
- critical path, 208
- CTL, 216, 221, 222
- customer interaction, 136
- customer satisfaction, 232
- customer support, 2
- cycle, 27, 109, 204
 - firing in, 24
- daily practice, 125
- dashed arrow, 108, 172
- data, 149
 - colored Petri nets, 26
 - Protos, 33
 - SAP WebFlow, 40, 41
- data availability, 215–217
- data collection, 148
- data configurability, 216
- data configuration, 218
- data consistency, 224
- data dependency, 224
- data evaluation
 - BPMN, 34
- data existence, 219
- data hazard, 224
- data mining, 143, 172
- data perspective, 221
- data processing, 1, 225
- data requirements, 28
- data source, 148
- data value, 26
- data-flow, 129, 134, 141, 216, 219, 222–224, 231
- database, 148
- database access, 4
- database theory, 225
- database transaction, 225
- de-select configuration parameter, 62
- de-selected behavior, 62
- de-synchronize behavior, 196
- deactivate link
 - SAP WebFlow, 68, 69
- dead transitions, 25
- deadlock, 20, 24, 51, 70, 71, 75, 84, 201, 203, 211, 213, 216, 230
- debug a software, 148
- decision making, 8, 99, 153
- decision node, 10

- UML, 102
- decrease number of instances
 - YAWL, 80, 83, 85, 92
- default configuration, 59, 61, 63, 106, 229
 - SAP WebFlow, 71, 72
 - YAWL, 87, 88, 91
- default decision, 134
- default value, 106
- define execution order, 8
- defining inheritance relations, 46
- delete arc
 - UML, 102
- delete data, 216, 217, 220, 221
- dependency, 63, 108, 153, 194, 215, 216
- dependent fact, 108
- depth-first search, 96
- derive individual process, 178
- derive subclass, 48
- derive superclass, 49
- deselect parameters, 62
- deselect undesired options, 2
- deselecting element types, 101
- deselecting elements, 101
- deselecting functions
 - EPC, 101
- design decision, 219
- design pattern, 4, 6
- design phase, 233
- design science, 11
- desirable property
 - Petri net, 25
- desired behavior, 10, 45, 58
- desired configuration, 228
- desired information, 194
- desired property, 172
- detecting configuration, 197
- detecting inheritance relations, 46
- deviate from solution, 199
- deviating event trace, 150
- deviating execution, 194
- deviating processes, 196
- diagnosis phase, 233
- difference between multi-sets, **18**
- different views, 196
- directed arc, 159
- directed edge, 18
- directed path, 23, 204
- disable feature, 106
- disallow behavior, 197
- discard event trace, 178
- disconnect path, 203
- discover a model, 145
- discriminating question, 108
- disjunction, 16, 206
- disjunction of clauses, 17
- disjunction of literals, 17
- disjunctive normal form, 17, **17**
- distinguish behavior, 47, 48
- distinguish multiple cases, 27
- distribution channel, 107
- do not enter sign, 78, 80
- document process, 124, 150, 157
- documents
 - Protos, 33
 - SAP WebFlow, 42
- domain configuration, 108, 215
- domain configuration model, 109
- domain constraint, 52, 105, 107, 108, 135
- domain context, 106
- domain expert, 135, 215, 227, 229, 230
- domain fact, 106–109, 121, 132, 134, 135, 137, 138, 215
- domain feature, 106
- domain knowledge, 105, 140
- domain of function, **17**
- domain variability, 105, 106, 109, 229
- domain-related question, 229
- domestic travel, 22, 55
- drop travel request, 19, 22, 56, 58
- dummy invoke activity
 - BPEL, 74, 75
- dummy net
 - YAWL, 89, 97
- durability, 225
- dynamic behavior, 63, 204
- dynamic instance creation
 - YAWL, 37, 80, 92
- edge of graph, 18, 20
- eEPC, 32

- effective process execution, 2
- effects of transitions, 48
- efficiency, 223
- element, 17
- element of multi-set, **18**
- element of sequence, 18, **18**
- element selection, 62
- eliminate activity, 102
- eliminate behavior, 178, 228
- eliminate element, 7, 18, 62
- eliminate inconsistent data, 220
- eliminate modeling efforts, 6
- eliminate task, 218, 219, 221, 225
- eliminate undesired behavior, 45
- employee, 22, 34, 55, 56, 222
- empty sequence, **18**
- empty set, **17**
- enable feature, 106
- enable task
 - YAWL, 76–78, 80
- enabled link
 - BPEL, 43
- enabled task, 26
 - YAWL, 36, 76
- enabled transition, **24**, 179
 - Petri net, 23
- enabling rule
 - Petri net, **23**
- enact process variant, 59
- enactment of business process, 15
- encapsulation, 47, 49, 228
- end event, 31, 174
 - BPMN, 34
- end user, 140
- enforce behavior, 11
- enforce usage, 62
- enhance models, 233
- enterprise system, 4, 10, 35, 40, 72, 139
- EPC, 12, 16, 28–32, 34, 40, 52, 98–102, 104, 111, 117–119, 121, 139, 158, 159, 161–165, 169–176, 185, 188, 189, 196, 197, 211, 231
 - extended, 32
 - well-formed, 30, **31**
- EPC from function graph, **170**
- equivalence, 17, 46
- equivalennce, 46
- equivalent models, 188
- equivalent state, 46
- ERP, 2, 3
- error, 148, 234
 - reference model, 150
- error handling
 - BPMN, 34
- essential task, 51, 58, 201, 230
- evaluate requirement
 - YAWL, 86, 90
- event
 - EPC, 28, 52, 100
 - SAP WebFlow, 40, 41, 67–69, 71, 72
- event creator
 - SAP WebFlow, 41, 69
- event identifier, 178
- event management, 139
- event name, 149, 151
 - EPC, 174, 175
- event port
 - SAP WebFlow, 67–69
- event stream, 197
- event trace, 149, 150, 156, 178–180, 194, 196
- event type
 - BPMN, 34
 - log file, 146
- Event-driven Process Chain (EPC), 28, **30**
- EWF-net, 36–39, 76–78, 80, 81, 83–85, 87–94, 96, 97, 111
- exception
 - SAP WebFlow, 41
- exclude arc
 - UML, 102
- exclude node
 - UML, 102
- exclusive choice, 27, 63, 103, 107
- exclusive disjunction, 16
- executability, 201, 214, 222, 223
- executable process model, 129, 135, 150
- execute function

- EPC, 52
- execute task, 26, 50, 53–55, 63
 - YAWL, 94
- execute transition, 20, 47–50
 - Petri net, 52
- execute workflow
 - YAWL, 98
- executed behavior, 150
- execution errors, 216
- execution log, 8
- execution outcome, 20
- execution path, 217, 221
- execution semantics, 15
- execution time, 26, 34, 149
- execution traces, 150
- expanded workflow net, 183
- expert interview, 140, 230
- export plug-in
 - ProM, 194
- expressiveness, 25, 211
- extended EPC, 32
- Extended Workflow net (EWF-net),
37
- external effect, 50
- extracting information, 194
- extreme case, 172

- facilitate reuse, 234
- fade out element, 62
- failed execution, 194
- failed replay, 180
- false*, 16
- feasibility, 124, 135, 139, 223
- feasible configuration, 199
- feature diagrams, 120
- feedback link, 215
- filter a sequence, 18, **18**
- filter plug-in
 - ProM, 194
- filtering irrelevant content, 143
- final condition
 - YAWL, 86
- final element, 151
- final function, 169
 - function graph, 165
- final marking, 179
 - workflow net, 24, **24**, 25
- final place, 213
 - workflow net, 24
- final state
 - LTS, 20, 51
- financial system, 41
- find errors, 148
- find superclass, 48
- fire transition, 179
 - Petri net, 23–25, 27
 - workflow net, 55, 60
- firing rule
 - Petri net, **23**
- firing sequence, 179
- first-class model, 109
- first-order logic, 71
- fitness, 8, 156, 194, 197
- flow
 - BPEL, 42, 72
- flow of case, 54
- flow relation
 - Petri net, **21**
 - workflow net, 56
 - YAWL, 37, 92, 93, 95, 96
- Flower, 137
- focus group interview, 123, 135
- fork
 - SAP WebFlow, 40, 66, 68, 71
- formal definition, 14
 - log file, 149
- formal foundation of patterns, 28
- formal language, 15, 16
- formal process definition, 15, 19
- formal semantics
 - BPMN, 35
 - workflow nets, 19
- formal specification, 1, 58
- formal verification, 25
- forward case, 55
- free-choice, 225
 - Petri net, 25, **25**, 225
 - workflow net, 211–214, 224
- free-choice nets, 231
- frequency profile, 197
- function, 17, **17**
 - EPC, 28, 99–101, 161, 163, 164

- function graph, 159, 161, 162
- function graph, 158, 159, **159**, 161–172, 174, 175
- future research, 231
- future work, 227

- garbage, 221
- gateway
 - BPMN, 34
- general requirement
 - YAWL, 85, 86
- general status message
 - SAP WebFlow, 41
- generalization, 156, 158, 172, 198
- generalized soundness, 214
- generate process model, 148, 151, 153, 157, 194, 197, 198
- generating frequency profile, 197
- generating integrated model, 196
- generic adaptation, 6, 62, 232
- generic task, 39
- get married, 125
- global policy, 138
- goal of configuration, 45
- good configuration, 144
- good process model, 225
- good-quality content, 234
- graph, 18, **18**, 21
- graph notation, 19, 158
- graph structure, 102
- graph-based constructs, 42
- graphical manner, 194
- guarantee correctness, 3
- guarantee soundness, 231
- guard, 27
- guideline, 139
 - EPC, 101
- guiding principle, 45

- harmonize naming, 127
- headquarter policy, 136
- hidden, 48, 78, 180, 181, 183, 194, 203
- hide behavior, 65, 98, 101, 102
- hide inflow port, 54
- hide outflow port, 55
- hide port, 54, 59, 63, 68, 103
 - BPEL, 72, 74, 75
 - YAWL, 78–80, 85, 86, 88–90, 94, 95, 97, 129, 132
- hide process part, 63
- hide step, 68
 - SAP WebFlow, 69, 71
- hide task, 55, 58, 62, 100, 217–220, 223
 - EPC, 101
- hide transition
 - LTS, 49–51
 - workflow net, 55, 56, 60, 208
- hiding, 49, 50, 52, 63
- hierarchy, 36
 - YAWL, 76
- hierarchy configuration
 - YAWL, **90**
- hierarchy port
 - YAWL, 88, **88**, 89–91
- high-quality information, 234
- high-quality process model, 1
- hotel, 36, 73, 78, 106, 107
- HR process, 2, 136
- human resources, 137

- IBM, 2, 139
- identical behavior, 46–48, 127, 172
- identifier, 46, 106, 179
- identify behavior, 47
- identify configuration, 197
- identify used tasks, 178
- IDS Scheer, 28
- implementation project, 4, 235
- implication, 16
- implicit condition, **38**
 - YAWL, 38
- implicit place, 33
- implicit status, 33
- import plug-in
 - ProM, 194
- improve process, 150
- In place, 26, 27
- in-house business process, 32
- include arc
 - UML, 102
- include function
 - EPC, 99

- include node
 - UML, 102
- incoming arc, 26, 55, 102, 161, 163, 165, 171, 203
 - EPC, 99
- incoming control link
 - BPEL, 43, 72, 74
- incoming path, 53, 175
- incoming process branch, 27, 53
- incoming process path, 53
- incomplete configuration
 - YAWL, 83, 87
- incomplete log, 150
- inconsistency, 5
- inconsistent data, 219, **219**, 220, 221, 224, 225
- inconsistent level of detail, 148
- incorrect process model, 58, 199
- increase number of instances
 - YAWL, 80, 83, 85, 92
- individual application, 10
- individual model, 133
- individual need, 4
- individual needs, 45
- individual requirement, 2, 4
- individual variant, 127
- induction hypothesis, 213
- industrial enterprise, 139
- industrial standard, 2
- inflow, 86
 - YAWL, 94
- inflow port, 53–55, 58, 63, 68, 88
- informal language, 58
- informal process model, 2
- information society, 1
- information system, 1, 12, 15, 145, 194
- inhabitant, 125
- inherit behavior, 46
- inheritance concept, 4, 45, 46, 50, 63
- inheritance of dynamic behavior, 45
- inheritance perspective, 60
- inheritance relation, 46, 47, 49, 63, 228
- inhibit behavior, 63, 227, 228
- inhibit configuration, 51
- inhibit functionality, 47
- inhibit process flow, 54
- inhibit task, 51
- inhouse process, 125
- initial element, 151
- initial function, 169, 174
 - function graph, 165
- initial marking, 179
 - workflow net, 24, **24**, 25
- initial state
 - LTS, 20, 51
- initial transition, 181
- innovation, 2
- innovative process, 143
- input arc
 - SAP WebFlow, 66
- input condition
 - YAWL, 36, 37, 77, 86, 88, 92, 96, 214
- input data, 156
- input log file, 145, 156
- input model, 171, 172
- input node, 19, 31
- input of activity, 15
- input path, 52
- input place, 151
 - CPN, 33
 - workflow net, 214
- input port
 - BPEL, 72, 74, 75
 - Protos, 191
 - SAP WebFlow, 67, 68, 70
 - YAWL, 77, **77**, 78–82, 84–86, 94, 95
- input process, 60
- input screen, 124, 129, 133, 137
- instance
 - YAWL, 37, 80, 81, 86
- instance creation
 - YAWL, 83
- instantiation, 6, 232
- instruction pipeline, 224
- integer programming problem, 198
- integrate behavior, 59, 198
- integrate process variants, 143, 144, 152, 198
- integrated process model, 2, 129, 132–134, 151, 175, 178, 188, 198

- integrating behavior, 199
- integrating process variations, 140
- interactive approach, 157
- interface, 6
- interface separation, 62
- interleaved parallel routing, 63
- intermediate event
 - BPMN, 34
- intermediate format, 158
- intermediate state, 42
- internal effect, 50
- internal place, 204
- international, 148
- international journal, 12
- international travel, 22, 208
- internet shop, 79, 80, 96, 98
- intersection of sets, **17**
- interview partner, 135, 137, 140
- invalid configuration, 72
- inverse direction, 63
- invisible task, 150, 153, 178, 194
- invocic verification, 2
- invoke activity
 - BPEL, 42, 72–74
- IP address, 145
- irrelevant element type, 101
- isolation, 22, 225
- issues of configurable notations, 45
- issuing death certificate, 125, 260–266
- IT, 15, 35, 143, 145, 146
- IT system, 35, 145, 146
- iteration over arcs, 175
- ITIL, 139

- join behavior
 - YAWL, 38, 76, 94, 95
- join condition
 - BPEL, 43
 - SAP WebFlow, 68, 71
- join connector
 - EPC, 31, 164, 169, 171, 175
- join control-flow
 - SAP WebFlow, 66
- join node
 - UML, 102
- join pattern, 52

- join type
 - function graph, 159, 161, 162, 164, 165, 167, 169, 171, 175
- joining fork
 - SAP WebFlow, 40

- key conference, 12

- label
 - LTS, 20
 - Petri net, 21
- Labeled Transition System (LTS), **20**
- lack of consistency, 157
- late data deletion, 221
- law, 123, 148, 233
- least common multiple, 48, 52, 59, 60
- leave task, 53
- legislation, 123
- legislative body, 125
- length of sequence, 18, **18**
- level of abstraction, 66, 67
- level of decision making, 99
- level of detail, 1
 - log file, 148
- library, 173
- life cycle, 233
- limit configuration space, 58, 63
- limit process behavior, 10
- linear time complexity, 175
- linear to the number of functions, 175
- link
 - BPEL, 43
 - SAP WebFlow, 67–69
- link port
 - BPEL, 73
- linkage of data
 - SAP WebFlow, 42
- linkage of events to workflows
 - SAP WebFlow, 42
- linkage of steps to workflows
 - SAP WebFlow, 42
- linked event
 - SAP WebFlow, 41
- linked operation
 - BPEL, 42
- linking configurations, 137

- literal, 17
- literature study, 61
- livelock, 24, 75, 230
- local regulation, 136
- local requirement, 138
- log, 143
- log entry, 148
- log event, 150, 151, 153, 156, 178, 179, 188, 194, 198
- log event identifier, 149
- log file, 143–146, 148, 149, **149**, 150–153, 156, 157, 172, 175, 178–181, 183, 185, 188, 189, 191, 194, 197–199, 230, 232, 234
- log format, 183
- log level, 197
- log replay, 179, 181, 197
 - advanced, 197
- logical expression, 84
- logical operator, 16, **16**, 17, 84, 85
- logical term, 62
- logistics, 4
- loop, 27, 28
 - BPEL, 75
- lost update problem, 225
- LTL, 216, 221, 224, 231
- LTS, 16, 19–21, 25, 31, 33, 35, 40, 42, 44, 46–52, 55, 57, 65, 73, 74, 78, 80, 100, 101, 103
- LTS from BPEL, 44
- LTS from SAP WebFlow, 42
- LTS from workflow net, **25**

- machine learning, 233
- maintainability, 137
- maintenance change, 233
- maintenance contract, 32
- making reservation, 175
- manager, 1
- mandatory fact, 106
- manual adaptation, 2, 51, 232
- manual approval, 70
- manual effort, 157
- manual update, 156
- map configuration options, 63
- map connector
 - EPC, 99
- map domain variation, 105
- map event identifiers, 178
- map event to task, 150
- map log events, 149, 198
- map workflow net to LTS, 25, **25**
- Mapper, 117
- mapping, 114, 121, 129, 134, 215, 229
 - YAWL, 36
- mapping functions
 - EPC, 172, 173
- mapping names, 143
- mapping task
 - YAWL, 89
- mark condition
 - YAWL, 96
- mark place
 - workflow net, 55
- mark visited element
 - YAWL, 96
- market leader, 32
- marketing, 2
- marking, 213
 - colored Petri net, 27
 - Petri net, 23, **23**, 25, 27, 52
 - workflow net, 24, 55, 179, 180
- marking arcs, 197
 - function graph, 159–161, 163, 165–169
- marking nodes, 197
- marking places
 - Petri net, 55, 176
- marking properties, 53
- marriage, 125
- matching event names, 149
- matching identifiers, 173
- material management, 4
- mathematical notation, 16
- maximal number of instances
 - YAWL, 80, 92
- maximal possible behavior, 59
- meaningless process, 201
- measure variations, 138
- merge algorithm, 158, 161, 171, 172, 174, 188, 189, 198, 230, 234
- merge function graphs, 165, 175

- merge of behavior, 197
- merge of function graphs, 158
- merge process models, 144, 157, 158, 161, 165, 174, 175, 194, 196, 198, 227, 230
- merge system descriptions, 195
- merging algorithm, 172
- merging EPCs, 172
- merging function graphs, 167, **167**
- message
 - BPMN, 34
- meta level, 101
- metrics, 197
- Microsoft, 28
- minimal common multiple, 61
- minimal number of instances
 - YAWL, 80, 92
- minimal regions, 197
- minimal subclass, 48
- mining a basic process model, 145
- mining algorithm, 144, 146, 151, **151**, 172
- mining approach, 183
- mining process model, 172, 198
- missing data, 90, 203, 216, **216**, 217, 219, 221, 222, 224, 225, 231
- model behavior, 46, 101, 159
- model checker, 222
- model completeness, 230
- model designer, 234
- model level, 197
- model merging, 172, 178, 198
- model provider, 233
- model quality, 144
- model transformation, 58
- model user, 45, 138, 178, 199, 233
- model variant, 59, 232
- modeling construct, 16, 28
- modeling experience, 140
- modeling guideline, 85
- modeling language, 52, 58
- modeling purpose, 16
- multi-choice, 27, 36, 63, 103
- multi-phase miner, 153, 155, 156, 158, 172, 185, 188, 189, 194, 198
- multi-set, 18, **18**
- multiple condition
 - SAP WebFlow, 40
- multiple data deletion, 221
- multiple elements, 18
- multiple inheritance, 48
- multiple instance task, 37
 - YAWL, 81
- multiple instances, 103
 - YAWL, 80, 81
- multiple processor cores, 224
- multiple variants, 197
- multiplicity
 - YAWL, 38, 82, 85
- municipality, 32, 123, 125, 127, 129, 132–137, 139, 140, 183, 191, 229, 234
- municipality process, 124
- mutually exclusive behavior, 63
- MXML, 145–147, 149, 185
- N-out-of-M-join, 81
- name clash, 39
- naming convention, 143
- natural language, 58, 121
- natural numbers, 18
- natural science, 11
- natural-language questionnaire, 227, 229
- necessary condition, 224
- necessary information, 105
- necessary requirement, 225, 231
 - YAWL, 85
- negation, 16, 17
- Netherlands, 28
- never deleted data, **220**, 221
- new requirements, 137
- new state, 53
- new version, 233
- newborn child, 125
- node, 18, 159
 - graph, 18, 20, 188
- node type, 21, 52, 65
- noise, 144, 150, 156, 183, 194
- non-blocked transition, 204
- non-core process, 2
- non-desired behavior, 228

- non-initial function
 - function graph, 161
- non-local choice, 156
- non-local dependency, 153
- non-observable effect, 50
- non-required tasks, 179
- not deleted on time, **221**
- not invented here syndrome, 4
- NP, 58, 206
- NP-complete, 58, 206
- null classes, 62
- number of elements, **17**
- number of instances
 - YAWL, 92
- number of visits, 197
- NVVB, 125, 127, 139, 144, 188, 191, 193

- object-oriented programming, 4, 6, 46
- observable behavior, 54
- observable effect, 50
- observe adaptations, 10
- omit component, 62
- omit components, 62
- omit task, 102, 217
- on-time deletion, 221
- ontological concept, 149
- ontology, 148
- ontology class, 149, 150, 172
- open standard, 16
- open-source, 66
- operations on sequences, **18**
- optimal basic process model, 59
- optimal solution, 61
- optimal superclass, 48
- option to complete, 25, 213
- optional behavior, 62
- optional function
 - EPC, 99
- optional routing, 197
- optional task, 197
- optionality pattern, 63
- ∨ connection
 - Protos, 33
- ∨ connector
 - EPC, 29, 32, 163, 164, 169, 171, 175, 196
 - ∨ join type
 - function graph, 164, 165, 167, 169
 - ∨ relation
 - BPEL, 43
 - ∨ split type
 - function graph, 160, 161, 163, 165, 167–169, 171
- OR-join, 27, 31, 32, 34, 36, 77, 80, 81, 94, 96, 158, 161, 171, 172, 175, 198, 230
- OR-split, 27, 32, 34, 36, 53, 73, 74, 77, 78, 80, 92, 93, 96, 134, 158, 171, 172, 175, 198, 230
- Oracle, 2
- orchestrating web-services, 36
- orchestration of tasks, 1
- order dependency, 108, 109
- order of executing functions, 174
- ordering constraint, 15
- organization, 143, 150
- originator, 149
- out place, 26
- outcome of choice, 33
- outcome of task execution, 19, 26
- outflow, 86
 - YAWL, 95
- outflow port, 53–55, 63, 68, 69, 78, 88
- outgoing arc, 55, 102, 171, 203
 - EPC, 99
- outgoing control link
 - BPEL, 42, 72, 73
- outgoing link port, 74
 - BPEL, 74, 75
- outgoing path, 53, 175
- output arc
 - SAP WebFlow, 67
- output block
 - SAP WebFlow, 68
- output condition
 - YAWL, 36, 37, 88, 92, 96, 214
- output node, 19, 31
- output of activity, 15
- output place, 151
 - CPN, 33

- workflow net, 214
- output port
 - BPEL, 72
 - Protos, 191
 - SAP WebFlow, 67, 68
 - YAWL, 78, **78**, 79–82, 85, 86, 92, 93, 132
- outsourcing company, 2
- over-approximate behavior, 153, 171, 172, 197–199
- over-approximating algorithm, 198
- over-generalized model, 172
- overall process, 21, 196
- overall state, 15
- overview of process mining algorithms, 156
- overwrite data, 219

- Pallas Athena, 32, 135, 137
- parallel execution, 1, 27
- parallel process branch, 219
- parallel processing, 224
 - SAP WebFlow, 40
- parallel split, 27, 63, 103
- parameterize decision node, 102
- parametrization, 62, 102
- partial configuration
 - YAWL, 91
- partial dependency, 107
- partial function, **18**
- partial matches, 197
- participant, 34
- path, 101, 102, 203, 204, 208, 214
 - EPC, 162
 - Petri net, 24
 - YAWL, 91, 96, 97, 214
- path , 204
- path in graph, 19, **19**
- pay travel, 84
- pay trip, 19
- payment, 51
- payment authorization, 2
- payment method, 36, 80, 84, 105
- payment workflow, 36
- perceived result, 50
- performance characteristics, 232
- performance indicator, 232
- personal data, 148
- personal information, 148
- personal time management, 4
- Petri net, 19, 21, **21**, 22, 23, 25–27, 32, 36, 38, 52, 56, 65, 103, 151, 158, 161, 171, 172, 175, 176, 191, 197, 201, 211, 214
 - from BPEL, 43
 - from BPMN, 35
 - from Protos, 33
- PinkRocade Local Government, 135, 137
- place
 - implicit, 33
 - Petri net, 21, **21**, 23, 24, 191
 - Protos, 32
 - workflow net, 24, 56
- placeholder, 6, 62
- pool
 - BPMN, 34
- pool of resources, 26
- population, 125
- port
 - BPEL, 74, 75
 - Protos, 191
 - SAP WebFlow, 66, 67
 - YAWL, 76–81, 83–86, 88, 92, 93, 135
- port concept, 73
- port configuration, 58, 61, 132
 - YAWL, 79
- possible future, 46
- post production, 139
- post-condition
 - YAWL, 36, 77, 78
- post-set, 19, **19**, 25, 204, 208, 212
- powerset, **17**
- practical model, 62
- practical need, 10, 135
- practical usefulness, 135
- practical-oriented process modeling language, 55
- practitioners, 15
- pre-adjustment of process models, 157
- pre-condition

- YAWL, 36, 76, 78
- pre-processing, 145, 148, 149, 151, 156, 157, 178, 198
- pre-processing effort, 183
- pre-set, 18, **19**, 204, 208, 212, 213
- preceding event
 - EPC, 28
- preceding function
 - EPC, 174, 175
- preceding place
 - Petri net, 23
 - workflow net, 55
- predicate
 - YAWL, 93
- pregnant partner, 125
- preliminaries, 16
- prepare log file, 143
- prepare travel form, 55
- preserve behavior, 7, 51, 161, 168, 171, 172, 198, 230
- preserve correctness, 201, 203, 211, 229
- preserve executability, 224
- preserve situation, 51
- preserve soundness, 211, 213, 230
- prevent task execution, 51, 217
- primary process, 1
- primitive activity
 - BPEL, 42
- privacy right, 148
- process adaptation, 137, 227
- process analysis technique, 194
- process analysts, 150
- process aware information system, 146
- process behavior, 8, 19, 27, 50, 143, 150, 159, 162, 178, 194, 195
- process branch, 27, 216, 225
- process completion, 23, 51, 221
- process configuration practice, 14
- Process Configurator, 117, 132
- process constraint, 87, 101, 106, 114, 215
- process correctness constraint, 205, **205**, 208, 211, 214, 215, 222
- process data, 93
- process definition tool
 - BPEL, 76
- process design, 233
- process designer, 4–6, 101, 135, 188
- process documentation, 4
- process domain, 51, 105
- process enactment, 233
- process environment, 51, 233
- process execution, 8, 19, 20, 23, 52, 68, 74, 105, 153, 159, 165, 219–221, 233
- process fact, 109, 121, 135, 215
- process flow, 6, 15, 54, 62, 134, 150, 165, 215
 - EPC, 100
 - YAWL, 36, 76
- process implementation, 143, 157, 178
- Process Individualizer, 118, 132
- process instance, 10, 23, 146, 150, 151, 153, 188, 189, 233
- process landscape, 1
- process mining, 143–145, 148, 150, 156, 157, 172, 175, 178, 183, 188, 194, 198, 230, 234
- process mining algorithm, 145, 150, 151, **151**, 152, 153, 156, 194, 198
- process mining framework, 153, 194
- process mining plug-in
 - ProM, 194
- process model, 1, 2, 4, 8, 15, 19, 25, 28, 45, 48–52, 54, 56, 59–63, 144, 153, 156–159, 161, 172, 194, 195, 197–199
- process model construction, 8
- process model design, 233
- process model library, 6
- process model life cycle, 231, 233
- process model reuse, 6, 10, 11, 124, 227, 231
- process model variability, 105
- process modeling costs, 4
- process modeling experience, 2
- process modeling language, 11, 15, 55, 59, 66
- process modeling notation, 25, 59, 63, 64, 158
- process modeling skills, 11

- process owner, 125
- process part, 63, 153, 178
- process participant, 34, 35
- process performance, 232, 233
- process property
 - Protos, 32
- process repository, 235
- process start, 23
- process state, 19
- process template, 2, 4
- process travel request, 51
- process variability, 109
- process variant, 3, 59, 60, 63, 133, 134, 143, 156, 157, 172, 178, 198, 199, 230
- processing time, 33
- <ProcessInstance>, 149
- processor, 224
- produce data, 217
- production process, 2, 136
- programmer, 148
- programming-like constructs, 42
- prohibit execution, 59
- ProM, 153, 156, 158, 172, 173, 175, 185, 188, 191, 194, 197
- ProM import framework, 146, 185
- ProM plug-in, 172, 175, 194
- proper completion, 25, 213
- proper subset, **17**
- property dialogue
 - Protos, 32
- proposition, 16, 17
- propositional formula, 16, 17, 87
- propositional letter, 16, 225
- propositional logic, 16, 58, 87, 107, 205, 221, 224, 225
- propositional statement, 16
- propositional variable, 205
- Protos, 16, 28, 32, 125, 127, 129, 133–135, 137, 146, 158, 175, 183, 185, 188, 189, 191, 193
- Protos2CPN, 33, 185
- providing variations, 198
- proving assumptions, 15
- pruning of dependencies, 194
- public administration, 139
- Quaestio, 115, 132
- quality of log file, 149
- quality of model, 156
- quantifier, 84, 85, 87
- question, 105, 106, 108, 109
- question order, 108
- questionnaire, 105, 106, 113–115, 121, 123, 124, 129, 132–138, 140, 215, 229, 230
- Questionnaire Designer, 115
- questionnaire model, 129
- race condition, 219
- range of function, **17**
- Rational Process Library, 139
- raw log file, 149
- re-configurable workflow system, 10
- re-connect arc, 62
- re-create data, 216, 221
- re-direct process flow, 6
- re-evaluate constraint, 208
- re-file travel request, 19
- re-name events
 - EPC, 174
 - log file, 150
- re-playing EPC, 31
- re-use of models, 148
- re-use of token, 26
- reach state, 50, 51
- reachable node
 - workflow net, 23
- read data, 216–221, 225
- read-after-write, 224
- real-life environment, 135
- real-world log file, 145
- realistic model, 156
- receive activity
 - BPEL, 42
- recommendation
 - EPC, 101
- record executed behavior, 145
- record executed work, 198
- record modifications, 232
- record variation, 235
- reduce formula, 206
- reduced form, 208

- reduction card, 36, 73, 74, 78, 79, 90, 106–108
- redundant data, 217
- reference guide, 14
- reference model, 2–4, 6, 123, 125, 127, 135, 136, 138–140, 151, 191, 229, 233–235
 - classification, 139
- reference model provider, 233
- reference modeling practice, 6
- region-based approach, 197
- registering a newborn, 125
- registration process, 123, 229
- reinvent the wheel, 3
- remove activity
 - UML, 102
- remove behavior, 45, 228, 232
- remove condition
 - YAWL, 96
- remove flow
 - YAWL, 92, 93, 96
- remove log events, 178
- remove node
 - YAWL, 96
- remove place
 - workflow net, 57
- remove task, 216
 - YAWL, 96, 97
- remove token, 211
 - Petri net, 23
 - YAWL, 36
- remove transition, 50, 51, 56, 203
 - workflow net, 57
- replace transition, 50, 51
- replay, 179–181, 183, 191, 193, 197, 199
- reply activity
 - BPEL, 42
- request change, 22, 29, 175
- request quote, 175
- required data, 216
- requirement, 61
 - municipality, 123
 - YAWL, 86, 87, 97
- requirements engineering, 4
- research approach, 11
- research contribution, 11
- research goal, 11
- research question, 11
- reservation, 172, 175
- resource, 15, 26–28, 203, 222
 - Protos, 33
 - SAP WebFlow, 42, 71
- resource assignment, 222
- resource availability, 215, 216, 222
- resource configurability, 216
- resource constraint, 222
- resource involvement, 138
- resource pattern, 28
- resource perspective, 223
- resource requirement, 222, 223
- resource-constrained process model, 222
- resource-constrained workflow net, 222
- restrict behavior, 7, 8, 45, 49, 59, 62, 63, 102, 230, 231
 - YAWL, 98
- restrict choices, 62
- restrict configuration opportunities, 52, 63
- restrict dynamic instance creation, 81
- restrict routing, 102
- restrict state changes, 54
- restrict using port, 54
- reuse potential, 10
- reuse process definition, 3
- reusing established artifacts, 4
- reusing process models, 3
- reusing software code, 4
- risk for error, 1
- roll-back
 - BPMN, 34
- root node, 39
- routing construct
 - SAP WebFlow, 40
- run-time, 78, 93, 98, 101, 102, 150, 153
- run-time decision, 8, 39, 127, 134
- running task, 36
- salary payment, 2
- sales and distribution, 4
- SAP, 2, 4, 28, 35, 40, 71, 72, 139, 234
- SAP installation, 40

- SAP reference model, 150
- SAP WebFlow, 16, 35, 40, 44, 55, 64–72, 74, 76, 84, 88, 103, 229
 - configurable, 66
- SAT, 58, 59, 71, 114, 115, 117, 206
- SAT solver, 59
- satisfiable formula, 206
- SBDD, 59, 115, 206
- schema
 - BPEL, 76
- SCOR, 139
- screen business, 139
- screenshot, 115
- secretary, 22, 55, 156
- select desired options, 2
- selection criteria for resources, 28
- semantic constraint, 58
- semantic correctness, 87, 88, 211, 224, 225
- semantic validity, 58
- semantically conflicting behavior, 88
- semantically equivalent, 198
- semantics, 15, 53
 - BPEL, 75
 - EPC, 31, 101
 - function graph, 161, 165
 - OR-join, 31
 - Petri net, 65
 - UML, 102
 - workflow net, 55, 203
 - YAWL, 66, 77, 92, 96
- sequence, 17, 18, **18**
 - BPEL, 42, 72, 75
 - EPC, 99, 100
- service-oriented architecture, 36
- set, 17, **17**
- set fact, 108
- set parameters, 62
- silent function, 165
- silent step, 50
- silent task, 46, 153
 - YAWL, 94, 95
- silent transition, 20, 46, 50, 51, 57, 153, 156, 175, 179–181, 183, 191, 193, 203
 - Petri net, **21**
 - workflow net, 56
- simple dependency, 108
- simple merge, 27, 63, 103
- simplified adaptation, 140
- simulation
 - Protos, 33
 - workflow net, 24
- simulation engine, 33, 183
- sink place, 203, 204, 214
 - Petri net, 23, **23**
 - workflow net, 24
- size of multi-set, **18**
- skip activity
 - BPEL, 75
- skip behavior, 89, 94
 - YAWL, 94
- skip block
 - SAP WebFlow, 68
- skip component, 62
- skip function
 - EPC, 99
- skip task, 50, 54, 62, 153, 194
 - EPC, 101
 - YAWL, 80
- skip transition, 153, 199
- social community, 235
- Software AG, 235
- software configuration management, 120
- software development, 4, 46
- software engineering, 4, 6
- software library, 4, 6
- Software Product Line Engineering, 120
- software provider, 123, 136, 137
- software reuse, 4
- software vendor, 2
- solver, 206
- sound process model, 227, 230
- soundness, 19, 213, 214, 222–225
 - process model, 156, 225
 - Protos, 34
 - workflow net, 24, **24**, 211, 213
 - YAWL, 84, 85
- soundness analysis, 214
- soundness criteria, 70

- soundness preservation, 225
- source place, 55, 203, 204, 214
 - Petri net, 23, **23**
 - workflow net, 24
- specific requirement
 - YAWL, 85
- split behavior
 - YAWL, 38, 77, 93, 94
- split connector, 163, 169, 171, 174
- split control-flow, 31
- split type, 175
 - function graph, 159, 160, 162, 164, 165, 167, 170, 171
- splitting fork
 - SAP WebFlow, 40
- staff member, 26
- staged configuration, 206, 214, 225
- stakeholder, 15, 124, 125, 129, 132, 135–138, 140
- standard notation, 28, 34, 36
- standard solution, 2, 4, 137, 229
- start event, 31, 174
 - BPMN, 34
- start from scratch, 4
- start task, 27, 194
- start transition, 26, 27
- state, 21
 - BPEL, 73
 - change of, 32
 - LTS, 20, 25
 - Petri net, 23
- state based model, 194
- state change, 20, 21, 26, 52–54, 102, 153
 - YAWL, 78
- state machine, 62
- state of process, 15, 19
- state of system, 196
- state properties, 20
- state space, 21, 31, 42
- state-based model, 197
- state-transition system, 196
- statement, 16
- static instance creation
 - YAWL, 37, 83, 85, 92
- status
 - Protos, 32
- steer execution, 229
- steer process configuration, 137, 230
- step
 - SAP WebFlow, 40, 66, 68, 71, 72
- storage space, 221, 232
- strict dependency, 108
- strongly lost data, 219, **219**, 221
- strongly redundant data, 217, **217**, 218, 219
- structurally correct, 211
- structurally incorrect, 201
- structured activity
 - BPEL, 42
- student status verification, 90
- sub-block
 - BPEL, 42, 74, 75
 - SAP WebFlow, 66, 68–71
- sub-sequence, **18**
- subclass, 46–49, 52, 59–61, 211
- subsequent path, 27, 53, 54
- subsequent place, 55
- subsequent state, 50, 54
- subsequent task, 55, 217
- subsequent transition, 50
- subset, **17**
- subset of process branches, 27
- subset of traces, 156
- succeeding event, 29
- succeeding function
 - EPC, 174, 175
- succeeding place, 23
- successful termination, 20
- sufficient condition, 224
- sufficient requirement, 211, 214, 225
- sum of multi-sets, **18**
- superclass, 46–49, 59
- superfluous elements, 172
- support process, 1
- swimlane
 - BPMN, 34
- switch
 - BPEL, 42
- synchronization, 27, 63, 103
- synchronization behavior, 162, 194
- synchronization patterns, 198

- synchronize behavior, 77, 196, 230
- synchronize process branches, 53
- synchronizing merge, 27, 36, 63, 103
- Synergia, 115, 121
- syntactic correctness, 87, 211, 215, 225
- syntactic validity, 58
- syntactically correct, 59, 63, 87, 205, 206, 208, 211
- syntactically incorrect, 214
- syntactically invalid, 58
- syntactically motivated, 58
- syntactically valid, 63, 85
- syntactically wrong, 208
- syntax
 - workflow net, 201, 203
- synthesis, 197
- synthesis of state-based models, 197
- synthesizing models, 194
- system architecture, 62
- system behavior, 4
- system configuration, 233
- system design, 62
- system state, 52
- system vendor, 4
- systems of constraints, 58

- tailor behavior, 10
- target function, 161, 167
- target market, 178
- task, 28
 - BPEL, 42
 - Protos, 32, 33
 - SAP WebFlow, 40, 66
 - workflow net, 55
 - YAWL, 36, 37, 76–80, 83–86, 88–90, 92–96
- task completion
 - SAP WebFlow, 40
 - YAWL, 81, 88
- task description, 134
- task execution, 21, 146, 218, 222
 - YAWL, 76, 78, 80
- task implementation
 - YAWL, 90
- task mapping
 - YAWL, 88, 90, 91
- task properties
 - YAWL, 38
- τ task
 - YAWL, 89
- τ transition, 20, 21, 56, 57, 62
 - LTS, 100
- technical constraint, 51
- template
 - SAP WebFlow, 68, 69, 71, 72
- template repository, 4
- template variant, 5
- temporal logic, 216, 221, 222, 225
- temporal operator, 221
- terminating event, 41
- threshold
 - YAWL, 37, 38, 81, 83, 85, 92
- time, 26
 - colored Petri nets, 26
- time complexity
 - merge algorithm, 175
- time-stamp, 26
- timer
 - BPMN, 34
- token, 36, 211
 - colored Petri net, 26, 27
 - function graph, 160, 161, 163, 165
 - Petri net, 23, 103
 - workflow net, 55
 - YAWL, 76–78, 80, 85, 86
- tool support, 139, 140
- trace, 179, 180, 198
- train ticket, 36, 74, 78, 79, 106, 107
- transaction management, 148
- transform EPC to function graph, 175
- transform function graph to EPC, 174, 175
- transformation algorithm, 66, 158, 171
 - SAP WebFlow, 72
 - YAWL, 76, 83, 91, 92, 98
- transformation rule
 - BPEL, 75, 76
- transition
 - LTS, 20, 100
 - Petri net, 21, **21**, 27, 65, 191
 - workflow net, 55
- transition configuration, 56

- transition firing, 180, 181, 211
- transition label
 - LTS, 20, 50
 - Petri net, 21, **21**
- transition system, 47, 197
- transition valuation, 206
- translate EPC to Petri net, 158, 171
- translate EPC to Protos, 158
- translate Petri net to EPC, 158, 172
- translate Protos to EPC, 158
- travel agency, 78, 97, 105
- travel approval, 2, 68, 71, 72, 215
- travel approval process, 55, 172, 181
 - BPMN, 34
 - EPC, 29
 - LTS, 19, 20
 - Petri net, 22
 - SAP WebFlow, 40, 41
- travel booking, 98, 105, 106
- travel booking process
 - BPEL, 42, 43
 - YAWL, 37
- travel form, 34, 55, 146, 148, 156, 172, 175
- travel plan approval, 5
- travel request, 2, 5, 19, 22, 51, 58, 67–69, 72, 216
- tree, 175
- tree-like hierarchy
 - YAWL, 38
- tree-like structure
 - YAWL, 36
- trigegr task
 - YAWL, 80
- trigger path, 53–55
- trigger task, 52–55, 63
 - YAWL, 78
- trip approval, 5
- true*, 16
- truth value, 16, 208, 225
- twice deleted data, **221**
- two-out-of-three join
 - SAP WebFlow, 41
- UML, 62, 102, 161
- uncompleted execution, 194
- unconnected net, 58
- undefined data, 216
- undesired behavior, 10, 45, 48, 75
- union of sets, **17**
- unique event, 174
- unique final function, 174
- University of Saarland, 28
- unset variable, 206, 208
- user decision
 - SAP WebFlow, 40
- user interface, 4, 72
- user-decision
 - SAP WebFlow, 66
- valid configuration, 52, 58, **58**, 59, 62
 - EPC, 101
 - YAWL, 87, **87**, 91, 92
- valid process model, 63
- validate model, 124
- valuating place, 204
- valuating transition, 204
- valuation, 205, 206, 208
- variability mechanism, 61
- variable, 204, 206
- varying behavior, 196
- verification, 25, 108, 225
- verification complexity, 25, 211
- verification tool, 34, 231
- viable configuration, 204
- visible tasks, 178
- visible transition, 179–181, 183
- Visio, 28
- wait for event
 - BPMN, 34
 - SAP WebFlow, 41, 69
- waiting time, 34
- weakly lost data, 219, **219**
- weakly redundant data, **218**, 219
- web service, 42
- well-formed EPC, 30, **31**, 171
- well-formed EPC from function graph, **171**
- well-formed net
 - YAWL, 85
- while

- BPEL, 75
- wiki, 235
- Woflan, 34, 214, 224, 225
- workaround, 136, 138
- workflow behavior, 27
- workflow block
 - SAP WebFlow, 67, 68
- workflow builder, 5
- workflow definition, 4, 89
 - YAWL, 98
- workflow engine, 2, 3, 11, 66, 91, 104, 129, 132, 137, 229
 - BPEL, 76
 - SAP WebFlow, 40, 72
 - YAWL, 98
- workflow environment, 128
- workflow language, 15, 16, 25, 35, 64, 65
- workflow log, 146
- workflow management system, 1, 28, 35
- workflow net, 16, 19, 23, **23**, 24, 25, 55, 58, 65, 151, 179–181, 183, 201, 203, 208, 211, 213, 214, 231
 - from EPC, 32
- workflow net configuration, 55, **55**
- workflow pattern, 15, 16, 26–28, 35, 52, 63, 65, 66, 76, 103, 134, 194
- workflow specification, 36, 38, **39**, 144
 - YAWL, 88–91, 97
- workflow specification constraint
 - YAWL, 91, **91**
- workflow support, 2
- workflow system, 1, 4, 10, 66
- workflow template, 5, 68, 139
- <WorkflowModelElement>, 146, 149
- worklet service architecture, 39
- write data, 216, 219–221, 225
- write-after-read, 224
- write-after-write, 224
- WS-BPEL, 42

- XML, 42, 98, 115, 117, 118
- XML schema
 - YAWL, 98

- XOR* connection
 - Protos, 33
- XOR* connector
 - EPC, 29, 32, 162, 169, 171, 174, 175, 196
- XOR* gateway
 - BPMN, 34
- XOR* join type
 - function graph, 161, 164, 165, 167, 169
- XOR* split type
 - function graph, 160, 161, 163, 165, 167–169, 171
- XOR-join, 27, 29, 32, 34, 36, 53, 76, 77, 94, 158, 191, 198
- XOR-split, 27, 29, 32, 34, 36, 53, 77, 78, 92, 93, 158, 191, 198

- YAWL, 12, 13, 16, 35, 36, 39, 40, 42, 44, 55, 64–66, 76, 80, 84, 88, 91, 98, 101–103, 106, 112, 114, 117–119, 121–124, 128, 129, 132–134, 158, 161, 175, 203, 214, 215, 225, 229, 230
- YAWL editor, 98
- YAWL system, 76, 98

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

BPEL Business Process Execution Language.

BPM Business Process Management.

BPMN Business Process Modeling Notation.

C- a configurable variant of the subsequent process modeling language.

CPN Colored Petri Nets.

CTL Computation Tree Logic.

EPC Event-driven Process Chain.

ERP Enterprise Resource Planning.

EWF-net Extended Workflow net.

IP Internet Protocol.

IT Information Technology.

ITIL IT Infrastructure Library.

log log file.

LTL Linear Temporal Logic.

LTS Labeled Transition System.

MXML Mining XML.

NP Nondeterministic Polynomial Time.

NVVB Nederlandse Vereniging Voor Burgerzaken.

SAT Boolean Satisfiability Problem.

SBDD Shared Binary Decision Diagram.

SCOR Supply Chain Operations Reference.

UML Unified Modeling Language.

WS-BPEL Web Services Business Process Execution Language.

XML Extensible Markup Language.

YAWL Yet Another Workflow Language.

Symbols

Configuration	\mathcal{C}	(Def. 3.1/3.4), also configured elements 49, 50, 52, 55–59, 71, 81–84, 86–88, 90, 92–96, 178, 181, 204–206, 212–214, 312
	<i>allow</i>	allowed element 55, 57, 58, 71, 82, 84–86, 90, 94–96, 109, 111–113, 178, 181
	<i>block</i>	blocked element 49, 50, 52, 55, 56, 58, 71, 82, 85, 86, 90, 92–94, 109, 111–113, 178, 181, 212
	\mathbb{C}_{WF}	all configurations of WF 55, 56, 58, 59, 82, 83, 87, 88, 90, 91
	<i>def</i>	default configuration 59, 87, 88, 91, 92
	<i>hide</i>	hidden element 49, 50, 52, 55, 57, 71, 82, 85, 86, 90, 94–96, 109, 111, 112, 178, 181
	MC	mapping between domain facts and process facts 111, 113, 114
	PC	process constraint 58, 59, 87, 88, 91, 92, 110, 111, 113, 114
	PCC	process correctness constraint 205, 206, 208, 212, 214
	<i>valid</i>	valid configuration(s) 58, 59

CTL*

$E x$	there exists at least one path from the current state where x holds 222
$F x$	x has to hold eventually, i.e. somewhere on the subsequent path (finally) 222
$x U y$	x has to hold in the subsequent path until at some point y holds 222

C-YAWL

$CEWF$	configurable EWF-net 92
\oplus	combines two configurations 83, 90
$\bigoplus_{X \in Q^\diamond} \mathcal{C}^X$	combination of all configurations \mathcal{C}^X such that X is an EWF-net contained in the set of EWF-nets Q^\diamond 90, 91
cs	subset of conditions 78, 86, 93–95
dyn	dynamic creation 82, 86, 92
in	input 77, 81–86, 90, 92, 94–96, 109, 111–113
max	maximum 82, 86
min	minimum 82, 86
out	output 78, 81–86, 90, 92–96, 112, 113
$ports$	ports of an EWF net 77, 78, 80–83, 85, 86, 88–90, 93–95

	<i>req</i>	requirement(s) 85, 87, 90, 111
	<i>thres</i>	threshold 82, 86
EPC	<i>EPC</i>	(Def. 2.21) 30, 162, 170, 171
	<i>A</i>	set of arcs 30, 31, 162, 163, 170, 171
	<i>E</i>	set of events 30, 31, 162, 163, 170, 171
	<i>F</i>	set of functions 30, 31, 162, 170, 171
	<i>m</i>	mapping of connectors to connector types 30, 31, 162, 163, 170, 171
	<i>X</i>	set of connectors 30, 31, 162, 163, 170, 171
Function graph	<i>FG</i>	(Def. 7.3) 159, 162, 163, 165, 167, 170
	<i>A</i>	set of arcs 159, 160, 162, 163, 167, 170
	<i>F</i>	set of functions 159, 160, 167, 170
	<i>l</i>	mapping of arcs to join/split types 159, 160, 162, 163, 167, 170
Graph	<i>G</i>	(Def. 2.10) 18, 19
	<i>E</i>	set of edges 18, 19
	<i>E*</i>	all paths over <i>E</i> 19, 23, 162, 163
	<i>N</i>	set of nodes 18, 19, 171

	$n^{\bullet G}$	output nodes of node n in G (post-set) 19, 23–25, 30, 31, 78, 159, 167, 170, 171, 204, 207, 212, 213
	$\bullet n^G$	input nodes of node n in G (pre-set) 19, 23, 24, 30, 31, 77, 159, 167, 170, 171, 204, 207, 212–214
General	\wedge	AND-split / AND-join type 29, 30, 32, 38, 77, 78, 94, 99, 100, 159–164, 166–171, 196, 286
	<i>join</i>	join behavior 30, 31, 37, 38, 77, 78, 80, 81, 83, 85, 87, 92, 95, 96, 159, 160, 162, 163, 167, 170
	\mathbb{N}	natural numbers 18, 23, 38, 82, 85, 213
	\vee	OR-split / OR-join type 29, 30, 32, 33, 38, 43, 77, 78, 99, 159–165, 167–172, 175, 177, 196, 299
	<i>split</i>	split behavior 30, 31, 37, 38, 77, 78, 80, 81, 83, 85, 87, 92, 94–96, 159, 160, 162, 163, 167, 170
	<i>XOR</i>	XOR-split / XOR-join type 29, 30, 32–34, 38, 42, 77, 78, 85, 94, 99, 100, 159–165, 167–171, 174, 175, 196, 308
Log file	<i>LOG</i>	(Def. 7.1) 149–151, 181
	α	mining algorithm 151
	I	set of log event identifiers 149
	<i>events</i> (σ)	events in the trace σ 149, 181
	Γ	set of all log files 149–151

	<i>prep</i>	preprocessing/preprocessed 149–151
	<i>raw</i>	raw log file 149, 150
	θ	event trace 149, 179–181
LTS	<i>LTS</i>	(Def. 2.13) 20, 25, 49, 50, 52
	<i>CPM</i>	configurable process model 52
	<i>CS</i>	set of possible configurations 52
	<i>L</i>	set of transition labels 20, 21, 25, 49, 50, 52
	<i>S</i>	set of states 20, 21, 25, 49, 50, 52
	<i>S_F</i>	set of final states 20, 21, 25, 49, 50, 52
	<i>S_I</i>	set of initial states 20, 21, 25, 49, 50, 52
	<i>T</i>	set of transitions 20, 21, 25, 49, 50, 52
	τ	label for silent transitions 20, 21, 46, 50, 51, 306
Petri net	<i>PN</i>	(Def. 2.14) 21, 23–25, 204, 205
	<i>A</i>	set of arcs 21, 23–25, 55–59, 178, 180, 181, 204, 205, 212, 213
	<i>exp</i>	elements of a workflow net that is expanded with silent transitions as alternatives to vis- ible transitions 181

hid	set of hidden elements 180, 181
l	function assigning labels to transitions 21, 23–25, 55–59, 178, 180, 181, 204, 205, 212, 213
L	set of labels 21, 23–25, 55–59, 178, 180, 181, 204, 205, 212, 213
M	marking of places 23–25, 212–214
$\mathbb{M}(PN)$	set of all markings of PN 23, 24
M_I	initial marking 24, 25, 179, 212, 213
$M \xrightarrow{\sigma}_{PN} M'$	firing the transition sequence σ from marking M in PN leads to marking M' 24, 179, 212, 213
$M \xrightarrow{t}_{PN} M'$	firing the transition t from marking M in PN leads to marking M' (Def. 2.8) 24, 25
M_O	final marking 24, 25, 179, 212, 213
$M[t\rangle$	t is enabled at M 24, 25
$PN[M\rangle$	set of markings reachable from M in PN 24, 25
P	set of places 21, 23–25, 55–59, 178, 180, 181, 204–207, 212, 213
Φ	all firing sequences of the net 24, 181
Φ^{IO}	all firing sequences of a workflow net that lead from M_I to M_O 179, 181

	p_I	input place 23, 24, 57, 201, 203–205, 208, 212–215
	p_O	output place 23–25, 57, 201, 203–205, 208, 212–215
	ψ	acyclic path of the net 204–207
	Ψ	set of all acyclic paths of the net 204, 205
	ψ_I	acyclic input paths of the net 205–207
	ψ_O	acyclic output paths of the net 205–207
	T	set of transitions 21, 23–25, 55–59, 178, 180, 181, 204–207, 212, 213
	τ	label for silent transitions 21, 56, 57, 94–96, 100, 180, 181, 201, 203, 306
	V	transitions that are fired when replaying a log file 181
	WF	workflow net 24, 25, 55, 56, 58–61, 151, 178–181, 205, 206, 212–214
	Δ	set of all workflow nets 23, 151
Set	S	(Def. 2.4) 17, 18
	$dom(f)$	domain of a (partial) function f 17, 18, 38, 39, 49, 50, 80, 82–84, 88, 90, 92
	$\langle \rangle$	empty sequence 18
	\emptyset	empty set 17, 19, 21, 23, 25, 31, 38, 39, 78, 90, 92, 167, 170

f	(partial) function 17, 18
$\pi_S(\sigma)$	filters σ for elements of S 18, 180, 181
\mathbb{P}	powerset of S 17, 38, 39, 77, 78, 149
$rng(f)$	range of a (partial) function f 17
σ	sequence (Def. 2.8) 18, 24, 179–181, 212
S^*	all sequences over S 18, 149
$\mathbb{B}(S)$	all multi-sets over S 18, 149
Z	multi-set 18

YAWL

<i>dynamic</i>	dynamic instance creation 38, 92
<i>EFW</i>	EFW-net 37–39, 77, 78, 80–83, 85–88, 90–93, 95, 96, 111
<i>EWFS</i>	set of EFW-nets 39, 90
<i>ext</i>	elements extending the original net 38
F	set of flows (arcs) 37, 38, 77, 78, 80, 81, 83, 85, 87, 92–96
∞	infinite 38, 82, 85, 112
\mathbf{i}	input condition 37, 38, 77, 78, 80, 81, 83–87, 91, 92, 96, 97, 109, 112, 214, 215

K	set of conditions 37–39, 77, 78, 80, 81, 83, 85–87, 92–96
map	maps composite tasks onto sets of EWF-nets 39, 88–91, 112, 113
$nofi$	number of a task’s instances that are started 37, 38, 77, 78, 80–87, 90, 92, 95, 96, 112
\circ	output condition 37, 38, 77, 78, 80, 81, 83, 85–87, 91, 92, 96, 97, 112, 113, 214, 215
π	selector for the number of instances parameter 38, 82, 83, 86, 92
Q°	set of EWF-nets 39, 88–91, 312
Q	partitions Q° into sets of EWF-nets 39, 88–91
rem	cancelation region of task 37, 38, 77, 78, 80–85, 87, 90, 92, 95, 96, 112
$SPEC$	workflow specification 89–91
$static$	static instance creation 38, 92
T	set of tasks 37–39, 77, 78, 80, 81, 83, 85–87, 92–96
T°	set of all tasks 39, 88–91
top	top level net of workflow specification 39, 88–91

Summary

Configurable Process Models

Business process models and workflow systems aim at guaranteeing efficient and reliable executions of business processes. For this, they require detailed specifications of the individual steps that need to be taken during the process execution. As defining such process definitions from scratch requires good knowledge of both the domain of the process as well as the process modeling technique, process modeling is often cumbersome and prone to errors that inhibit sound process executions.

For that reason, many providers of process modeling software as well as process consultants offer so-called reference process models or template repositories. These repositories provide established process specifications for both general as well as industry specific processes, from travel approvals to invoice verifications. Still, even for such common processes, execution variations exist among organizations. Hence, organizations have to adapt the templates manually to individual requirements. The process modeling skills required for these adaptations are equal to the skills required for modeling from scratch.

This PhD thesis suggests reducing the need for manual process model adaptations by integrating variations among different process executions into a single process model — a configurable process model. In the configurable model, process configuration allows inhibiting executing any undesired tasks. In this way, a process model providing exactly the behavior desired by an organization can be derived without manual process modeling efforts.

For defining configurable process models, the thesis first provides a formal specification of process model configuration by analyzing how behavior is added to process models and defining process configuration as the inverse. In this way, blocking and hiding of behavior, originally defined as concepts to discover inheritance relations among dynamic behavior, are identified as the two techniques to restrict behavior of configurable process models.

The practical feasibility of the suggested approach is demonstrated through depicting how process configuration can be added to the process modeling languages of existing workflow systems, namely SAP WebFlow, BPEL, and YAWL. An actual implementation is provided for YAWL. It even allows configuring the process model through natural language questions by mapping the various con-

figuration options to pre-defined answers in a questionnaire. The framework has been tested in a case study where configurable process models were developed for registration processes executed in municipalities like the registration of a newborn child or a marriage. The developed models thus allow deriving a configured, individual process variant by simply answering a questionnaire on desired and undesired options of the process. The resulting model variant is directly executable in the YAWL system environment. The models and the overall approach have afterwards also been evaluated through interviews with various stakeholders in the process configuration lifecycle.

Two main challenges arise for defining and using configurable process models: On the one hand, the creation of a process model integrating various process variants obviously requires far more work than building a process model covering only one of these variants. For that reason, the thesis suggests a set of process mining techniques, which can help in the construction of a configurable process model, e.g. by merging individual process models. On the other hand, process configuration allows restricting the process behavior depicted in the configurable process model further than desired, up to the point that the process is not executable anymore. This is addressed through discussing a range of constraints on the process model configuration that are able to guarantee sound processes.

Samenvatting

Configureerbare Proces Modellen

Procesmodellen en workflow systemen richten zich op het efficiënt en betrouwbaar uitvoeren van bedrijfsprocessen. Om dit mogelijk te maken is het noodzakelijk dat er gedetailleerde specificaties zijn van de individuele stappen die worden genomen tijdens de uitvoering van het proces. Aangezien het definiëren van dergelijke procesdefinities vanaf nul een goede kennis vereist van zowel het domein van het proces als van de procesmodelleringstechniek, is het procesmodelleren vaak omslachtig en gevoelig voor fouten, wat een correcte uitvoering van het proces in de weg staat.

Om deze reden bieden leveranciers van procesmodellingssoftware en procesconsultants referentiemodellen en voorbeeldsjablonen aan om snel tot concrete procesmodellen te komen. Hierbij gaat het vaak om erkende processpecificaties voor zowel algemene als industrie-specifieke processen, variërend van reisvergunningen tot factuurcontroles. Toch is het zo, zelfs voor veelvuldig voorkomende processen, dat de uitvoering van deze processen verschilt van organisatie tot organisatie. Daarom is het nodig dat bedrijven dergelijke referentiemodellen en voorbeeldsjablonen handmatig aanpassen op basis van hun individuele behoeften. De vaardigheden met betrekking tot procesmodelleren die nodig zijn om de processen aan te passen zijn hetzelfde als de vaardigheden die nodig zijn om de modellen vanaf nul te ontwikkelen.

In dit proefschrift is beschreven dat de noodzaak voor handmatige aanpassingen kan worden gereduceerd door het integreren van variaties van verschillende uitvoeringen van een proces in één enkel procesmodel — een configureerbaar procesmodel. In het configureerbare procesmodel zorgt het configureren van processen ervoor dat ongewenste taken niet kunnen worden uitgevoerd. Op deze manier is het mogelijk dat een procesmodel exact het gewenste gedrag vertoont zonder dat dit handmatige modelleren van het proces vereist.

Om configureerbare procesmodellen te kunnen opstellen wordt in dit proefschrift eerst een formele specificatie van een procesmodelconfiguratie gegeven. Dit wordt gedaan door te analyseren hoe gedrag wordt toegevoegd aan het procesmodel; proces configuratie wordt gedefinieerd als de inverse hiervan. Op deze manier zijn de technieken ‘blokkeren’ (blocking) en ‘verbergen’ (hiding) gedefinieerd, oorspronkelijk gedefinieerd als concepten om overervingsrelaties te

ontdekken, nu geïdentificeerd als twee technieken om het gedrag van een configureerbaar procesmodel te beperken.

De praktische haalbaarheid van de voorgestelde aanpak wordt gedemonstreerd door het toepassen van procesconfiguratie op procesmodelleertalen van bestaande workflow systemen, namelijk SAP WebFlow, BPEL, and YAWL. Voor YAWL is bovendien een implementatie van de concepten gerealiseerd. Het is zelfs mogelijk om het procesmodel te configureren door het beantwoorden van vragen die zijn gesteld in natuurlijke taal. Dit is gerealiseerd door een vertaling van de verschillende configuratie opties naar voorgedefinieerde antwoorden in de vragenlijst. Het totale raamwerk is getest in een case study waarin configureerbare procesmodellen zijn ontwikkeld voor administratieve processen binnen gemeentes, zoals de aangifte van een geboorte of een huwelijk. De ontwikkelde modellen maken het mogelijk om een geconfigureerde individuele procesvariant af te leiden, die rechtstreeks uitvoerbaar is in de YAWL omgeving, enkel door het beantwoorden van vragen in een vragenlijst over de wenselijkheid van bepaalde mogelijke uitvoeringen van het proces. De modellen en de aanpak als geheel zijn achteraf geëvalueerd met verschillende belanghebbenden in de levenscyclus van procesconfiguratie.

Uit de case study zijn twee belangrijke aandachtspunten naar voren gekomen. Ten eerste, het opstellen van procesmodellen waarin verschillende varianten zijn geïntegreerd vereist veel meer inspanning dan het opstellen van een procesmodel dat slechts één variant behelst. Om deze reden wordt in dit proefschrift een aantal process mining technieken aangedragen, die kunnen helpen bij het construeren van een configureerbaar procesmodel, bijvoorbeeld bij het samenvoegen van verschillende individuele procesmodellen. Ten tweede, procesconfiguratie maakt het mogelijk om het gedrag van het proces meer te beperken dan wenselijk is, tot aan het punt dat het proces niet meer uitvoerbaar is. In dit proefschrift wordt hiermee rekening gehouden door het bespreken van een verzameling randvoorwaarden ten aanzien van procesmodel configuratie, die er voor zorgen dat evidente fouten worden vermeden.

Acknowledgements

Yes, sometimes it seemed to be a bloody long way from completing my graduation project at Queensland University of Technology (QUT) in Brisbane, Australia, to completing this PhD thesis at Eindhoven University of Technology (TU/e) in the Netherlands. More often, however, it was a very enjoyable experience, especially because it came with the opportunity to meet and work with many great people. Thus, all these people contributed in the one or the other way to the thesis. I want to take this opportunity to thank everyone who helped me completing this challenge! A few people to which I am especially grateful, I would like to name explicitly in the following.

First and foremost, I am very thankful for the support I got from my supervisor Wil van der Aalst. Wil had the incredible skill to always provide me with the confidence that I am on the right track while at the same time never stopping in thinking ahead and demanding more. His suggestions and skills of formalizing ideas helped me a lot in achieving the solutions presented in this thesis. Wil's feedback not only contributed to raising the quality of the publications on which the thesis is based, but it also helped me in assessing research quality myself. Thanks, Wil!

In the same way, I am very grateful to my co-supervisor Monique Jansen-Vullers. Monique was a big support in putting Wil's extensive feedback in the right context as well as in learning to judge when I had done 'enough'. Thus, for me she was the well-needed complement to Wil. Also, she translated the thesis's summary for me into Dutch. Thanks, Monique!

In 2004, during my graduation project, Michael Rosemann and Alexander Dreiling not only encouraged me to start a PhD project, but they also recommended me that if I would like to do a PhD then I should apply for the position in Eindhoven. Looking back at the opportunities and supervision I got during the last four years, I am very glad they did so!

Thanks to invitations by Marlon Dumas and Arthur ter Hofstede, I was able to return to QUT several times. Not only I very much appreciated the feedback I received from Marlon and Arthur, but they also introduced me to Marcello La Rosa with whom I started a very fruitful collaboration. Marcello developed the basis of the Synergia toolset and the concepts for questionnaire models used in the thesis. Furthermore, 'endless' discussions with Marcello influenced or resulted in many of the presented ideas — often continuing after-hours in an humorous way.

Chapter 6's municipality case study is the result of a great collaboration with Teun Wagemakers. Therefore, I would like to thank Teun for testing the ideas developed in the earlier chapters as the case study's process designer and thus for giving me extensive feedback on the developed tools and techniques. Furthermore, I am very grateful to the NVVB for providing their reference models for the case study, as well as to Pallas Athena, PinkRocade Local Government, and the municipalities involved in the case study for their input and opinions. Also, I appreciated the discussions I had with Paul Eertink, Leon Gerrits, Kurt Jensen, Agnes Koschmider, Marteen Leurs, Klaas-Pieter Majoor, Steffen Mazanek, Jan Mendling, Michael zur Mühlen, Marijn Nagelkerke, Mor Peleg, Manfred Reichert, Barbara Weber, and Lisa Wells on various subjects of the thesis a lot.

During the four years of my PhD studies, I enjoyed working in the IS group of TU/e very much. Especially, I would like to thank Anke Hutzschenreuter with whom I shared a very pleasant working atmosphere in our 'German' office, characterized by the readiness to help each other and discuss research-related and non-related issues in both a productive as well as in a not so serious manner. Moreover, I am very grateful to Anne Rozinat who started her PhD project almost at the same time as I. Besides having to deal with similar issues at similar times, I always appreciated Anne's well-thought-out opinions a lot.

I guess, never I would have managed to cope with all the administrative regulations and issues if I would not have had the support of Ada Rijnberg, Anemarie van der Aa, Geertje Kramer, and Ineke Withagen for which I am very thankful. Also, thanks to Alex Nort, Ana Karla Alves de Medeiros, Boudewijn van Dongen, Christian Günther, Eric Verbeek, Hajo Reijers, Irene Vanderfeesten, Jana Samalikova, Jochem Vonk, Maja Pešić, Mariska Netjes, Minseok Song, Nataliya Mulyar, Olivia Oanea, Nick Russell, Paul Grefen, Peter van den Brand, Remco Dijkman, Ricardo Sequel Perez, Ronny Mans, Samuil Angelov, Sven Till, Ting Wang, Ton Weijters, and all the other current and former members of the IS group(s) in Eindhoven for the great time during meetings, lunch, chatting, etc.

To focus on a single subject over four years — as it is needed when writing a PhD thesis — would not have been possible for me if I would not have had friends that made me focussing on other issues in my spare-time. Therefore, I am very glad for the good times I had with Bill (also thanks for the editing suggestions!), Bjorn, Caro, Carolina, Carsten, Daniel, Guy, Ilona, Ioana (also thanks for the support in getting the thesis printed!), Jakob, Jana, Jeff, Marjan, Monica, Oliver, Serge, Sonja, Steffi, Vero, Volker, and many, many others in Eindhoven and elsewhere. Thank you all! Deeply grateful I am to Anneli, Christian, and Christina. They endured me when I was in a bad mood as well as they celebrated with me when I was in a good mood — all three in their own special ways. Thus, although there were phases when we lived on four different continents, we communicated almost daily and it thus always felt as they would be just around the corner. Thanks a lot, Anneli! Thanks a lot, Christian! Thanks a lot, Christina!

Last but for sure not least I am very thankful for the support and feedback I

received from my brother Christian and my parents. Christian, you are always a good advisor whose critical feedback I appreciate a lot. Thanks for your help in designing the cover of this thesis. Mutti, Papa, I know you are the two who are the proudest of all of this PhD thesis — but without your endless support and love I would have never been able to achieve this. Thank you so much!

Curriculum Vitae

Florian Gottschalk was born in Göttingen, Germany, on September 7, 1979. At the age of five, he moved to the region of Hanover, Germany, where he lived until completing secondary education. From 2000 to 2005 he studied at the Technical University of Clausthal, Germany, majoring in Information Systems. Before his graduation Florian also joined ICA GmbH (IBM Global Services), Hanover, Germany, for a part-time job as web programmer. Furthermore, he worked as a student assistant at the Technical University of Clausthal and freelance in several web-design projects. For his final graduation project on “Rethinking Enterprise System Configuration: Towards an Integration of Structural and Process Configuration” Florian visited Queensland University of Technology (QUT), Brisbane, Australia, joining a research collaboration between QUT and SAP Research. From 2005 to 2009 Florian was employed as a researcher at the Technical University of Eindhoven (TU/e), The Netherlands. While working on his dissertation in the information systems group of TU/e, the Business Process Management group of QUT invited him for research visits to Brisbane in January/February and September 2007. Florian completed his doctoral studies on “Configurable Process Models” in summer 2009. Today (2009), he is working as a consultant for managing SAP implementation projects by Basycon Unternehmensberatung GmbH, Munich, Germany.

configurable process models

Process model configuration restricts the behavior that is depicted in business process models. This allows that templates can contain all tasks that are potentially relevant for the execution of a process in a single process model. For example, a model for travel approvals can contain tasks for both a complex and strict process variant as well as for a quick and easy variant. By process configuration, individuals can restrict these variation options to particularly desired behavior.

This PhD thesis provides a formal foundation of process model configuration. It analyzes how behavior is added to process models and defines process configuration as the inverse. The approach's practical feasibility is demonstrated based on process modeling languages of existing workflow systems, namely SAP WebFlow, BPEL, and YAWL. An implementation for YAWL even allows configuring process models through natural language questions. A case study in which configurable process models were developed for registration processes executed in municipalities demonstrates the above. The main challenges for using configurable process models are that (1) the creation of such models requires far more work than building a model covering only one process variant, and that (2) process configuration allows restricting the process behavior more than desired. These issues are addressed by suggesting a set of process mining techniques, which help constructing configurable process models, as well as by discussing constraints on process model configuration that are able to guarantee sound processes.