university of groningen

umcg

University of Groningen

Business Process Variability

Groefsema, Heerko

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2016

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Groefsema, H. (2016). *Business Process Variability: a study into process management and verification*. [Thesis fully internal (DIV), University of Groningen]. Rijksuniversiteit Groningen.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 05-10-2026

Business Process Variability

A Study into Process Management and Verification

Heerko Groefsema

This research was supported by the University of Groningen and the Netherlands Organization for Scientific Research (NWO) under project number 638.001.207 whithin the scope of the Jacquard program.

Published by: University of Groningen
Groningen, The Netherlands

Printed by: NetzoDruk Groningen B.V.
Groningen, The Netherlands

ISBN: 978-90-367-9237-0 (book)
978-90-367-9236-3 (e-book)

© 2016, Heerko Groefsema

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system of any nature, or transmitted in any form or by any means, electronic, mechanical, now known or hereafter invented, including photocopying or recording, without prior written permission of the author.

rijksuniversiteit
groningen

Business Process Variability

A Study into Process Management and Verification

Proefschrift

ter verkrijging van de graad van doctor aan de
Rijksuniversiteit Groningen
op gezag van de
rector magnificus prof. dr. E. Sterken
en volgens besluit van het College voor Promoties.

De openbare verdediging zal plaatsvinden op

vrijdag 23 december 2016 om 12.45 uur

door

Heerko Groefsema

geboren op 16 augustus 1983
te Groningen

Promotor
Prof. dr. ir. M. Aiello

Beoordelingscommissie
Prof. dr. ir. W.M.P. van der Aalst
Prof. dr. W.J.A.M. van den Heuvel
Prof. dr. ir. J.C. Wortmann

Contents

Acknowledgements **xiii**

1 Introduction **1**

1.1 Variability and Formal Verification 4

1.2 Problem Statement 9

1.3 Methodology 11

1.4 Contents 13

1.5 Related Publications 14

2 Background **17**

2.1 Business Process Modeling and Management 17

2.2 Business Process Formalization 21

2.3 Formal Verification 27

3 State of the Art **37**

3.1 Business Process Soundness 37

3.2 Business Process Compliance 39

3.3 Business Process Variability 41
3.4 Discussion 44

4 Case Study Description and Formalization **47**

4.1 Case 1: Telecommunications Customer Support 48
4.2 Case 2: Local Dutch e-Government 53
4.3 Case 3: Bouncer Registration 61
4.4 Discussion 64

5 Verification Requirements **67**

5.1 Model Requirements 67
5.2 Specification Requirements 69
5.3 Evolutionary Requirements 77
5.4 Discussion 79

6 Business Process Verification **83**

6.1 Verifiable Model 83
6.2 Specification Semantics 87
6.3 Specification Interpretation 88
6.4 Verification over Groups and Roles 89
6.5 Verification over Conditions 90
6.6 Inheritance of Specification Sets 92
6.7 Model Reduction 95
6.8 Discussion 96

7 Verification Specifications **101**

7.1 Visualization 102

7.2 Soundness Specifications 104

7.3 Preventive Compliance Specifications 105

7.4 Variability Specifications 107

7.5 Discussion 110

8 Automated Specification Assembly **113**

8.1 Prime Event Structures 114

8.2 Prefix Unfoldings 115

8.3 Execution and Elementary Loop Identification 118

8.4 Compound event structures 121

8.5 Specification Assembly 122

8.6 Discussion 124

9 Implementation **127**

9.1 Features 127

9.2 Extensibility 132

10 Evaluation **139**

10.1 Expressive Power 139

10.2 Performance Evaluation 143

10.3 Requirements Analysis 146

10.4 Case Study (continued) 149

11 Conclusion **159**

11.1 Summary 160

11.2 Contributions 162

11.3 Results 163

11.4 Limitations 167

11.5 Implications & Future Work 170

Appendices **173**

A Business Process Model & Notation 173

B BPMN BPD CPN Formalization 178

Abbreviations **187**

English Summary **191**

Nederlandse Samenvatting **195**

Bibliography **199**

Acknowledgements

Finally. The end result of a long period of study and research is starting to take shape. What a journey it has been. A journey of persistence, continuous iteration, and will. A journey on an international level while remaining in one place. A journey where I had to balance my strength between work and leisure every day. A journey which I could never have begun or continued without the support of others. In my case this is even more true than other cases. I would like to reflect on those people that have supported and helped me throughout this period.

Firstly, I would like to express my gratitude to my advisor, Prof. dr. ir. Marco Aiello, for his continuing support throughout my Ph.D. study, for his guidance, his knowledge, his unending patience, and for his positive opinion towards my work, which was always much more positive than my own.

I would like to thank the members of my assessment committee, Prof. dr. ir. Hans Wortmann for his willingness to include me in the SaS-LeG project, Prof. dr. Willem-Jan van den Heuvel for his positive feedback, and most notably Prof. dr. ir. Wil van der Aalst for his generous amount of valuable insights and feedback.

I would like to thank Prof. dr. Serge Daan, the former dean, Prof. dr. Henk Broer, the former scientific director, Prof. dr. ir. Paris Avgeriou, Hans van der Aa from the home ventilation centre of the UMCG, Evelyn Haandrikman and Hedwig Witteveen from USG Restart, Lourens Boomsma, and Janieta de Jong for their support during and after my application process and their continuing confidence.

I would like to thank Prof. dr. Wim Hesselink and Prof. dr. Gerard Renardel de Lavalette for their guidance into methods of formal verification which made this work possible.

I would like to thank my trusty co-authors, Nick van Beest and Pavel Bulanov for their close and seamless cooperation, even while lately residing in entirely different time zones, and Doina Bucur for her edged insights. Without their discussion and input I could not have achieved the content presented here.

For her continuing care, I would like to thank Ineke Schelhaas. Without her initial support and care, I would never have had the opportunity to start my Ph.D. studies.

Elie El-Khoury, Mahir Can Doganay, and Ilche Georgievski, my different roommates over the years, I would like to thank for their support, care, and many fun discussions, being it about games, football, or TV shows. I would also like to thank Eirini Kaldeli, Viktoriya Degeler, Tuan Anh Nguyen, Ehsan Ullah Warriach, Frank Blaauw, Fatimah Alsaif, Ang Sha, and Laura Fiorini for those days they would help out when the regular roommates were not available. Without all of your willingness to help and care, I could not have achieved this milestone.

I would like to thank the other members of the distributed systems group, Alexander Lazovik, Andrea Pagani, Ando Emerencia, Faris Nizamic, Brian Setz, Azkario Rizky Pratama, Talko Dijkhuis, and administrative staff member, Esmee Elshof, for their input, encouragement, and fun discussions.

Finally, I would like to thank my parents and brother for all those years of care, support, and encouragement throughout my Ph.D. studies and life in general.

– Heerko Groefsema

CHAPTER 1

Introduction

Computer science is no more about computers than astronomy is about telescopes.

– Edsger W. Dijkstra

Business processes are collaborations between *actors*, i.e. someone or something that performs an activity or task. In a business process, each actor fulfills one or more *activities* with the aim of ultimately achieving a specific, value-added goal driven by the outside world (Ko, 2009). Take, for example, your typical parcel delivery service. When you send someone a package, you first drop it off at your local service point. Trucks pick up all packages from the service point and deliver it to a distribution hub where the packages are sorted by destination. Your package is then moved to the distribution hub of its destination, planned into a delivery route, and delivered at its destination. This, seemingly simple, business process is an immense collaboration of a large number of actors, including the service point employee, the truck driver, the automated sorting systems, and the delivery man. Its goal is delivering your package. It is value-added because this process is offered to you in exchange for a relatively small shipping fee, as well as the value of receiving a parcel. The process is driven by you, or your wish to have your package delivered at its destination without actually going there yourself. Note, however, that each step described in this, seemingly simple, business process can be broken down into a multitude of smaller activities or tasks, which can be described by a business process of its own.

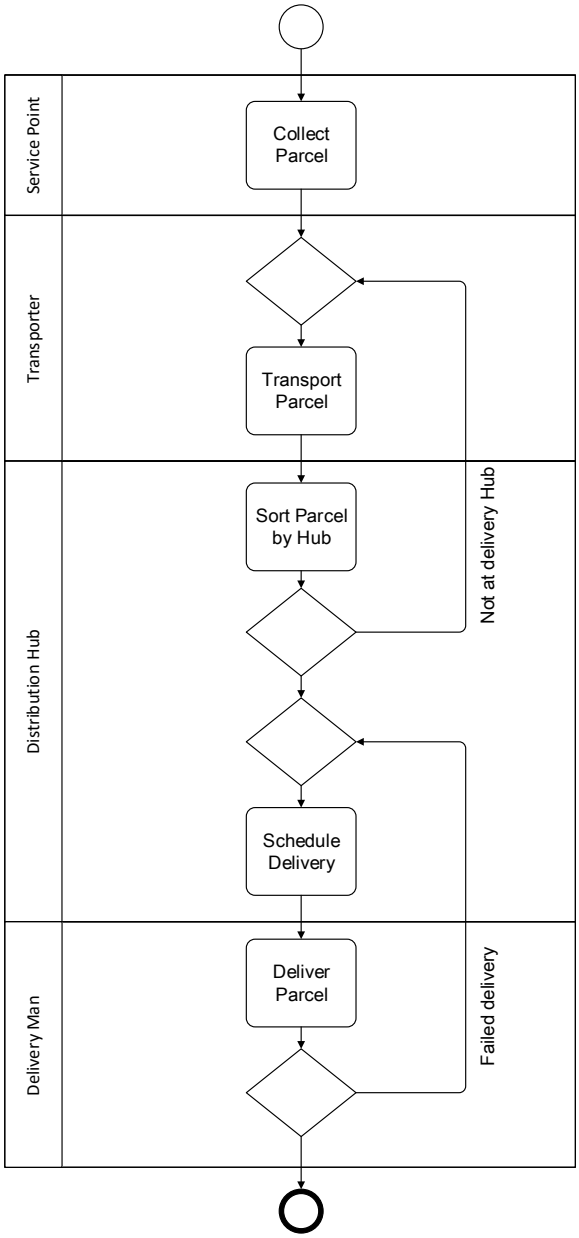


Figure 1.1: Generalized parcel delivery process.

Business processes are typically represented by *business process models*. Business process models are then directed graphs where nodes represent *activities*, *events*, or *gateways*. The edges of the graph define the order among the nodes. Gateways diverge or converge multiple edges. Different gateways diverge and converge edges by alternate sets of rules. *Exclusive gateways* activate a single activity based on conditions on outward edges after an activity on an inward edge completed, *parallel gateways* activate all activities on outward edges after all activities on inward edges completed, and *inclusive gateways* activate some activities based on conditions on outward edges after the activated activities on inward edges completed.

Figure 1.1, for example, depicts the parcel delivery process example discussed earlier as a business process model. In this model, activities are represented by rounded boxes, events by circles, and gateways by diamonds. Actors are represented by rectangular areas within which activities, events, and gateways are modeled. An activity that is modeled within an area of an actor, is performed by that actor. The business process begins at the circular start event modeled at the top of the figure, moves down through the activities and decision points performed by the different actors, and finally terminates at the thick bordered circular end event modeled at the bottom of the figure.

Business processes can be automated using business process or workflow management systems. An automated business process, or *workflow* (van der Aalst et al., 2003b), is modeled as a consecutive set of activities or tasks with decision points allowing for different outcomes. Different tasks can be assigned to one or more actors. A workflow can be either fully automated, or require human input. Most importantly, however, workflows can be monitored and managed.

The emergence of Service-Oriented Architectures and standards such as Web Services has accelerated the trend and opened a wide range of automation and integration possibilities (Papazoglou and van den Heuvel, 2007). As a result, business processes are increasingly being represented and designed as *service compositions*. Instead of modeling business processes as local and rigid sets of consecutive activities, service compositions define business processes as collections of loosely-coupled services that represent the business flow of the business process. Correspondingly, each activity is implemented as an independent, self-contained, and well-defined modular service.

Business process management is a field which aims of increasing productivity and performance of companies by managing their business processes. Management

of business processes can be useful at many different levels. For example, to increase productivity by streamlining the process, to avoid issues caused by faulty or erroneous process design and enactment, or to continually ensure compliance of processes to rules and regulations.

Originally designed to support local, user-specific, rigid, and repetitive units of work, business process management must adapt to support loosely-coupled processes in agile service oriented environments with many different users that each have customization and personalization requirements. The need to adapt processes to instances and changes becomes concrete with the notion of *variability*, which first emerged in software engineering. In software engineering, variability refers to the possibility of changes in software products and models (Sinnema et al., 2006). In the context of business process management, variability indicates that parts of a business process remain variable, or not fully defined, in order to support different versions of the same process depending on the intended use or execution context.

As the field of business process management continues to manage an increasing number of rapidly evolving customized business processes in agile service oriented environments, the evolution of each business process must continue to always behave in a correct manner and remain compliant with the laws, regulations, and internal business requirements. To manage the correct behavior of quickly evolving business processes, and the definition of a wide variety of similar business processes, we evaluate the application of formal verification techniques as a possible solution for the pre-runtime analysis of the correct behavior and compliant design of business processes within process families. Specifically, we focus on *design-time* solutions using *existing* and *well-supported formal verification techniques*.

1.1 Variability and Formal Verification

Formal verification entails proving or disproving the correctness of a system model with respect to a formal specification using formal methods of mathematics. When employing formal verification, a *system model* – often represented by a labeled transition system – is verified against a *formal specification* in the form of a set of logic formulas. One approach towards formal verification is *model checking*. When model checking, a system model is automatically, systematically, and exhaustively explored while each explored state is verified for compliance with the formal specification. Business process verification is the act of determining whether a business process model complies with a set of formal correctness properties. For-

mal verification of business process models is of interest to a number of application areas, including checking for basic business process correctness (i.e. business process *soundness* (van der Aalst, 1997)), business process *compliance* (Groefsema and van Beest, 2015; Groefsema et al., 2016), and business process *variability* (Aiello et al., 2010; Groefsema and Bucur, 2013). Although business process variability may not seem to relate to formal verification directly, there has been a trend of defining business processes using declarative techniques to support flexible process definitions (van der Aalst and Pesic, 2006; Pesic and van der Aalst, 2006). Then, when applying this to design-time process families, business process variability becomes the problem of verifying whether a business process is a legal member of a process family. This, in turn, introduces new challenges to the formal verification of business processes.

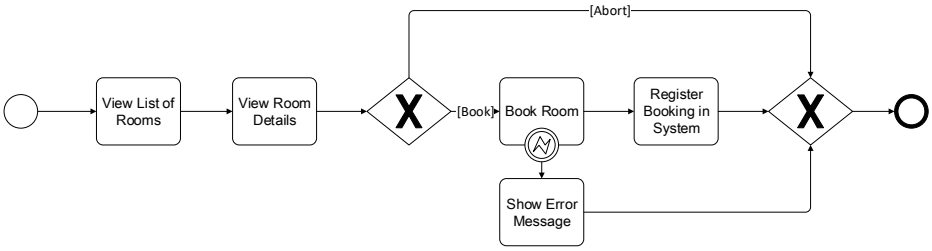


Figure 1.2: Room booking example

To explain each application area, we shall consider a deceptively simple workflow to book a meeting room in your average office building. Figure 1.2 illustrates the process. To book a meeting room, the user first views a list of available rooms, selects a room, and views the details of that room. The user then decides to book the room, or abort to restart the process. If the user decides to book the selected room, the system attempts to book the room and registers it in the system. In case the attempt fails, because someone else booked the room while the user was inspecting the details, an error event is triggered and an error message is displayed.

1.1.1 Soundness

Business process correctnesss verification entails the verification of basic properties such as *reachability* and *termination*. Reachability of a business activity requires an execution path to exist leading to that activity starting from the initial activities. A termination property requires that all possible execution traces reach a final state. Business process *soundness*, a property originally proposed in the area of Petri Net

verification (van der Aalst, 1997), is known as the combination of these two properties plus a third: the absence of related running activities at process termination (i.e., *proper completion*). Avoiding the deployment of erroneous processes that do not conform with these properties is obviously advantageous, as erroneously designed business processes may lead to failed executions or execution errors, and, ultimately, disgruntled customers or employees (Bi and Zhao, 2004).

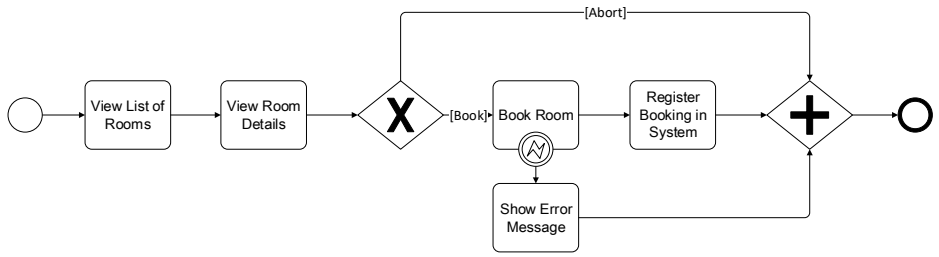


Figure 1.3: Erroneous business process example

For example, the room booking process illustrated in Figure 1.3 is erroneous. It is erroneous because the depicted process fails to satisfy the termination property. The termination property requires all execution traces to terminate. The depicted process, however, will never terminate because the merging gateway expects all incoming branches to finishing execution before activating the end event, while only one of these branches will ever be activated. This simple, but easy to make, design error can be easily caught through formal verification techniques.

1.1.2 Compliance

Business process compliance aims to confirm that a business process adheres to a set of rules imposed on that process. Rules can, for example, be imposed upon a process by international regulations, national law, or internal business rules. Where soundness verification aims at the verification of a limited set of requirements to verify reachability, termination, and possibly proper completion (van der Aalst, 2000) – compliance verification requires verification of a broad set of specifications.

Take, for example, international banking. In international banking, every transaction must be checked for possible international sanctions against the persons, countries, companies, and banks, involved in the entire transaction. If a bank fails to do so, or can not demonstrate the checks were indeed performed correctly, it may face serious financial penalties.

Existing techniques perform compliance verification at different stages of the business process lifecycle, during process design, enactment of its composition, or diagnosis. Monitoring techniques are deployed during process enactment, utilizing the runtime trace of a service composition to check if a model is executing correctly. Auditing techniques are deployed during the diagnosis phase and adopt, for example, process mining to verify if a service composition has been executed correctly.

Naturally, monitoring and auditing techniques are *after the fact* techniques, meaning that issues will only ever be detected after they already have occurred. As a result, expensive rollbacks or compensating actions are required to undo any erroneous execution before the application of sanctions. In case of auditing techniques, the damage has been done, and only a full rollback can be attempted. In case of monitoring techniques, compensating actions can not guarantee correct behavior unless they are taken immediately. When compensating actions will be taken at a later stage of the business process, monitoring techniques enter an incompliant state where they are unable to know or guarantee whether such a compensating action will actually be taken. To overcome such undesirable occurrences, we focus on preventative approaches (Elgammal et al., 2010b). Preventative approaches are design-time, and aim to prevent issues from ever occurring or prove that compensating actions will always be taken when encountering issues. Preventative compliance verification can be state-based or event-based. That is, compliance can be verified by finding illegal states within the process, or by finding a series of illegal steps leading up to such a state.

For example, our room booking example could feature the requirement that when a user chooses to book a viewed room, it must always result in a booked room. Obviously, this specification fails for the example process illustrated in Figure 1.2 since there may be a booking error which does not result in a booked room.

1.1.3 Variability

Business Process Management is evolving rapidly due to emerging mass customization and personalization trends, the need for adaptation to varying business and execution contexts, and the wider availability of service-based infrastructures. Where business process management originally supported local, user-specific, rigid, and repetitive units of work, now it is required to support loosely-coupled processes in agile service oriented environments and many different users with many different requirements. Variability is an abstraction and management method that addresses a number of the related issues.

In the domain of software engineering, variability refers to the possibility of changes in software products and models (Sinnema et al., 2006). When introduced to the domain of business process management, it indicates that parts of a business process remain either open to change, or not fully defined, in order to support several versions of the same process depending on the intended use or execution context (Aiello et al., 2010). Currently, when multiple similar business processes are required, they either exist as one large process definition using intricate branching descriptions or in multiple separate process definitions. This makes readability and maintainability a major problem in case of processes with intricate branching routes, or creates redundancy issues in case of multiple separate process definitions (Sun et al., 2010; Aiello et al., 2010).

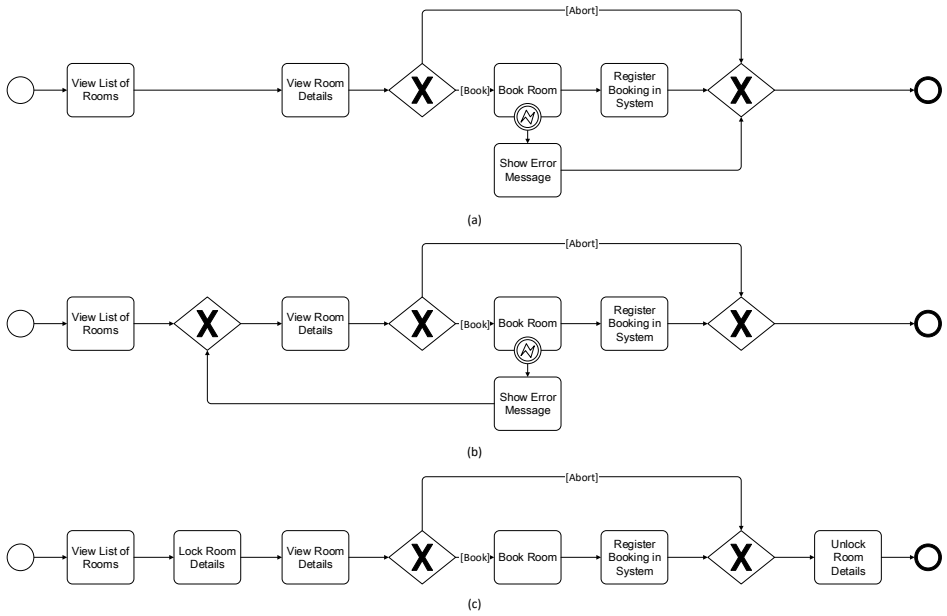


Figure 1.4: Variability example

By introducing variability to the area of business process management, support is introduced for both re-usability and flexibility, ameliorating the readability, maintainability, and redundancy issues. Multiple similar but different process instances, called *variants*, may be based upon a single re-usable process by applying several changes as allowed by the variability, and may then possibly be adapted at runtime due to this same flexible nature.

The room booking business process, for example, can be modeled in many different ways. Figure 1.4 illustrates three possible variants. The first process model, variant (a), equals the process illustrated in Figure 1.2. Variant (b) changes the process slightly by returning the user to the room details after it failed to book that room. Finally, variant (c) avoids any booking errors by denying other users booking access while a room is being booked. Note, however, that this final variant would pass the example compliance specification given in the previous section.

Variability can be introduced to the area of business process management using imperative or declarative approaches (Schonenberg et al., 2008; Aiello et al., 2010). Where imperative approaches exactly specify possible change, declarative approaches constrain the process behavior, allowing any change within those constraints. When mapping these to design- and run-time, we notice four possible directions when applying variability to business process management. Most research currently focuses on the areas of imperative/design-time and declarative/run-time (Groefsema et al., 2011, 2012). Declarative approaches offer a number of advantages. Since declarative approaches constrain the allowed behavior, instead of specifying it directly, the approach inherently allows quickly more variability. In addition, since imperative approaches exactly specify all possible change, they require all possible change to be known in advance. Declarative approaches do not require such knowledge. Therefore, we focus on the approach where variability is offered as a declarative extension of pre-runtime, preventative, event-based compliance verification. Using this approach, a business process is part of a family of business processes if it is compliant with a set of specifications belonging to that business process family. A business process of that family is applicable to change as long as the changed process remains compliant with this set of specifications.

1.2 Problem Statement

Now that the application areas have been set, we identify the open challenges. Balko et al. (2009) present a set of open research challenges in the field of business process extensibility and/or variability. Of this set we highlight the five challenges which relate to the problems discussed in this document:

- **Reference process conformance:** The ability to verify whether a process extension/variant conforms to, or complies with, a reference process.
- **Reference process patchability:** The ability to patch, update, or change the reference process such that all change is automatically propagated to every extension/variant that is based on that reference process.

- **Extension mining:** The ability to automatically detect manual ad-hoc deviations from the reference process and automatically derive extensions.
- **Stacked extensions:** The ability to define parent-child relations between both different reference processes and/or different extensions.
- **Design-time usability:** The ability to design reference processes and extensions/variants with support of toolsets.

In this document, we focus on the challenges of reference process conformance, stacked extensions, and design-time usability. Although the challenges of reference process patchability and extension mining are not discussed directly, they are related to the artifacts and techniques presented throughout the document. These challenges are related in such a way that they can be solved when combining those artifacts and techniques with techniques outside of the scope of this document, such as versioning and ad-hoc deviation detection.

Formal verification entails proving or disproving whether a *system model* conforms to a *formal specification*. When applying this to the challenges, we notice a clear similarity in the challenge to verify whether a *process variant* conforms to a *reference process*. In other words, by defining a reference process as a formal specification, and a process variant as a system model, it must be possible to formally prove conformance through model checking. At the same time, we also notice similarities between this approach and business process compliance verification. Therefore, to provide support for all five challenges, we propose business process variability as an extension of design-time, preventative, business process compliance verification using model checking. Our goal is to assess the feasibility of this approach by proposing models and techniques for it. Consequently, we arrive at the following main research question:

> *To which extent can formal verification through model checking be used to support verification of business processes variability as an extension of design-time, preventative, business process compliance?*

The research question specifies the consolidation of a number of research areas, namely those of business process variability, preventative compliance, and formal verification through model checking, and then asks to which extent this merger is feasible. To answer this complex question, we specify a number of sub-questions:

1. *Which goals for design-time business process verification can be identified?*

To fully understand the research question, we must first identify the goals of business process verification. As soon as the goals are known, we can also identify the requirements for verification.

2. *What system model adequately represents the business process for variability verification?*

   Formal verification through model checking verifies a system model against a formal specification. To be able to verify reference process conformance, we must therefore identify which system model accurately describes a business process. With an accurate and full system model we will not be forced to make certain concessions at later stages of our research.

3. *In which manner can the system model be reduced without relevant information loss?*

   Although a full and accurate system model is required, it does pose issues when its state space becomes too large to verify through model checking. To keep the system model verifiable through model checking, we must first keep the system model from expanding too quickly, and secondly allow reductions of the state space without losing any required information.

4. *What can be verified using well-supported specification languages?*

   To support reference process conformance, patchability, and stacking, we must first identify what and how much of a reference process can be described through a combination of the well-supported specification languages and a full and accurate system model.

5. *In which way can specifications be obtained automatically?*

   To support both the automatic procurement of specifications for reference process conformance verification, and the automated detection of manual ad-hoc deviations, we must device a way to automatically obtain specifications.

6. *For which business processes is the resulting system model verifiable?*

   Finally, to answer the feasibility of the approach proposed in the research question, we must identify to which extent business processes are verifiable.

1.3 Methodology

The research is triggered by both the known challenges described in (Balko et al., 2009), as well as observations in practice. In addition, parallels between the known challenges and well-supported techniques can be drawn. Knowing that there clearly

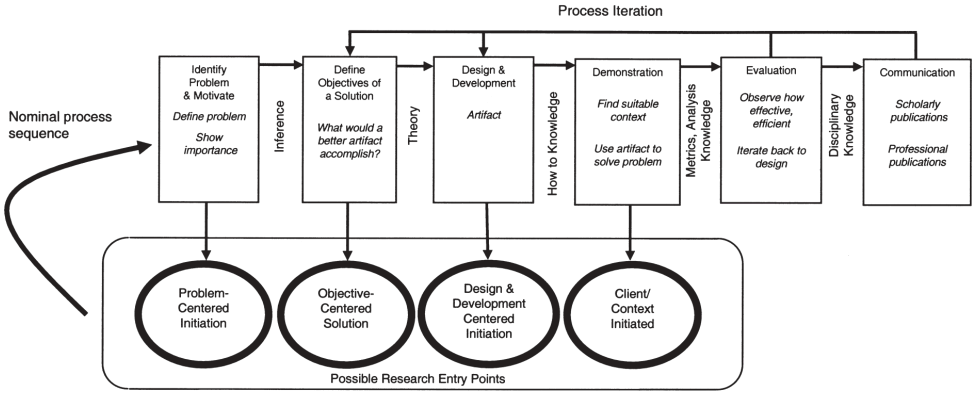


Figure 1.5: The design science research methodology. Source: (Peffers et al., 2007).

is a lack of artifacts that address the issues, we propose the design of new artifacts that bridge the gap between the challenges and well-supported techniques.

The design science research methodology as described by Peffers et al. (2007) provides a clear overview of the steps required for designing new artifacts. The methodology is illustrated in Figure 1.5. Since, in our case, the design of new artifacts is triggered by known challenges and observations in practice and simultaneously aims to evaluate the feasibility of well-supported techniques towards its solution, the methodology features both a problem centered initiation where the problem is first identified and motivated, and an objective centered solution where the objectives of a solution are defined. After these initial steps, the artifacts are designed and developed, demonstrated as a solution, and evaluated towards its applicability. And, finally, the results are communicated.

The research presented in this document follows the design science research methodology as depicted in Figure 1.6. The research initiation is discussed in Chapters 4 and 5, where we first identify the issues observed in practice through a set of three case studies, before defining the objectives in a detailed requirements analysis. The design of the artifacts is discussed in Chapters 6, 7, and 8, where we define both the models and specifications needed to support business process variability as an extension of preventative compliance verification. The development of the artifacts is presented in Chapter 9. Finally, the artifacts are demonstrated using the presented case studies and evaluated for expressive power, performance, and requirements satisfaction in Chapter 10.

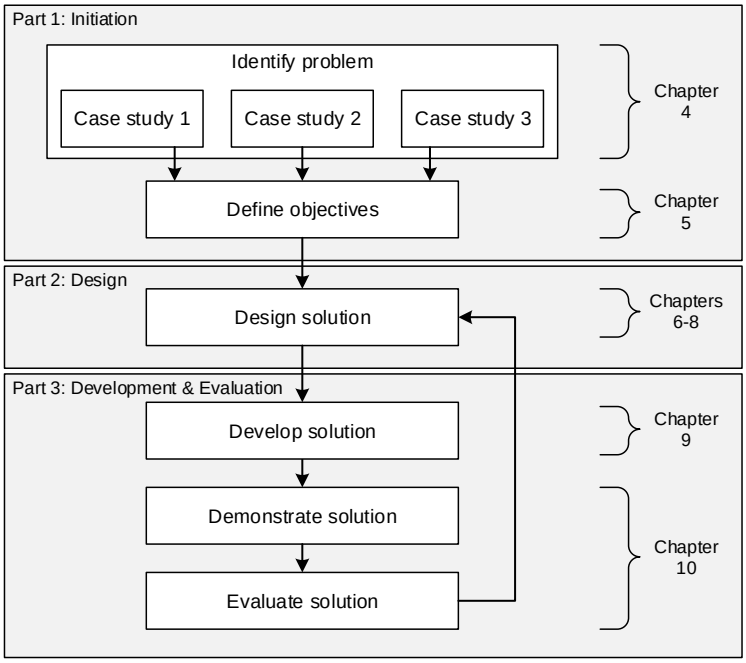


Figure 1.6: Research method in relation to the presented text.

1.4 Contents

The remainder of the document is structured as follows. Chapter 2, details the required background in the areas of business process management, business process formalization, and formal verification. Chapter 3, discusses the state of the art. Chapter 4 presents three case studies towards business process compliance and variability. The first case study details a customer support process resulting from a compliance study at an Australian telecommunications provider. The second case discusses a variability study throughout a number of Dutch municipalities. And, the third case, features a collaborative business process. Finally, Chapter 5 lists the requirements towards business process verification.

The second part of the document details the design of the artifacts. In Chapter 6 a novel mapping of business process models to a system model is presented. The resulting model allows the verification of preventative compliance and variability using well-known temporal logics and model checking techniques while providing full insight into parallel executing branches and the local next activity invocation. Furthermore, the mapping causes limited state explosion, and allows for signifi-

cant further model reduction. Next, Chapter 7 matches the set of requirements to specifications applicable to the presented model. Finally, Chapter 8 presents an approach to apply these specifications and automatically obtain reference processes. At the same time, the approach is also capable of incorporating ad-hoc runtime deviations of business processes.

The third part of the document features the implementation, demonstration, and evaluation of the presented artifacts. Chapter 9 presents the resulting tool chain. The tool features business process modeling abilities, saving and loading, automated generation of the system model required for verification, automated verification using one of multiple model checkers, and transparent visual and textual feedback of the generated models and verification results. Chapter 10 then proceeds with the evaluation and demonstration of the presented artifacts, by first evaluating the expressive power of the proposed system model, then evaluating performance of the generation of different complexities of system models, performing a requirements analysis, and finally demonstrating the applicability of the proposed artifacts on the relevant case studies.

Finally, Chapter 11 concludes the presented work by presenting a detailed discussion of the research while evaluating each of the presented research questions.

1.5 Related Publications

The work presented in this document has been realized in collaboration with a number of other researchers. In particular, Marco Aiello, Nick van Beest, Pavel Bulanov, Luciano García-Bañuelos, and Doina Bucur.

The basis of the research is primarily described in (Aiello et al., 2010) and (Groefsema and Bucur, 2013), while the development is described in (Groefsema et al., 2011), (Groefsema et al., 2012), (Groefsema and van Beest, 2015), (Groefsema et al., 2016) and (van Beest et al., 2016). The implementation is supported by (Groefsema et al., 2011b), (Groefsema and van Beest, 2015), and (Groefsema et al., 2016). And, finally, the evaluation is described in (Groefsema and van Beest, 2015), (Groefsema et al., 2016), and (van Beest et al., 2016).

N.R.T.P. van Beest, H. Groefsema, L. García-Bañuelos, and M. Aiello. Variability in business processes: automatically obtaining a generic specification. In preparation, 2016.

H. Groefsema, N.R.T.P. van Beest, and M. Aiello. A formal model for compliance verification of service compositions. *IEEE Transactions on Services Computing*, 2016. To appear.

H. Groefsema and N.R.T.P. van Beest. Design-time compliance of service compositions in dynamic service environments. In *Int. Conf. on Service Oriented Computing & Applications*, pages 108–115, 2015.

H. Groefsema and D. Bucur. A survey of formal business process verification: From soundness to variability. In *International Symposium on Business Modeling and Software Design*, pages 198–203, 2013.

H. Groefsema, P. Bulanov, and M. Aiello. Imperative versus declarative process variability: Why choose? Technical Report JBI 2011-12-6, University of Groningen, dec 2012.

H. Groefsema, P. Bulanov, and M. Aiello. Declarative enhancement framework for business processes. In *Int. Conf. Service-Oriented Computing (ICSOC)*, pages 495–504, 2011a.

H. Groefsema, P. Bulanov, and M. Aiello. Business process variability: A tool for declarative template design. In *International Conference on Service-Oriented Computing - Demo Track*, pages 241–242, 2011b.

M. Aiello, P. Bulanov, and H. Groefsema. Requirements and tools for variability management. In *IEEE Workshop on Requirement Engineering for Services at IEEE COMPSAC*, 2010.

CHAPTER 2

Background

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on eachother, like a wall of mini stones.

– Donald Knuth

Business process management already is a cross-disciplinary field in itself. It includes paradigms from, among others, economics, organization management theory, computer science, mathematics, philosophy, and even linguistics (Ko, 2009). The work described herein is positioned at the intersection of three fields of research, specifically those of business process management, business process formalization, and formal verification. As a direct result, the work presented here requires the necessary background from all three fields of research.

2.1 Business Process Modeling and Management

Business process management (BPM) is a field of operations management which focuses on the business processes within a company. By managing business processes BPM aims to increase the productivity and performance of a company.

2.1.1 Business Process Modeling

Business processes (BP) are collaborations between actors, each fulfilling roles to perform tasks or activities, with the aim to achieve a specific value-added goal driven by the outside world (Ko, 2009). In this regard, a BP consists of a collection of tasks, or structured activities performed in a specific order by actors fulfilling roles. When roles and actors are spread over multiple entities, the business process is called a collaborative business processes (CBP), or simply, a collaboration (Ko, 2009). BP can be automated using workflow management systems (WfMS) or, more modern, business process management systems (BPMS). Hence, an automated business process is referred to as a workflow (van der Aalst et al., 2003b).

To summarize, informally, a process is a tuple $P = (SA, F)$, where:

- SA is a finite set of *structured activities*,
- F defines the *control flow* over SA, i.e., its order.

Business processes are represented by BP models. BP models are defined through several specification techniques, most notably, through imperative and declarative specification of models.

Imperative Specification

The imperative specification technique is the most common form of specification when modeling BP. Imperative specifications are intuitive due to their focus on how a task is performed. In its most basic form, an imperatively defined BP model is a directed graph. Structured activities are represented by vertices of the graph. The edges of the graph define the order among the structured activities. Gateways are introduced to diverge or converge multiple edges. Different gateways diverge and converge edges by different sets of rules (e.g. activating either one or all tasks on outward edges after either one or all tasks on inward edges completed). Although gateways are, just like structured activities, nodes of the graph, they do not represent units of work and are part of the control flow instead.

Summarizing, an imperative process model is a triple $P_I = (SA, G, F)$, where:

- SA is a finite set of *structured activities*,
- $G = G_a \cup G_o \cup G_x$ is a set of *gateways*, consisting of and, or, and xor gateways, respectively,
- $F = F_t \cup F_g$ is a set of *edges*, where:

- $F_t : (SA \backslash \{\otimes\}) \rightarrow SA$ is a finite set of edges which assign a next state for each structured activity,
- $F_g : G \rightarrow 2^{SA}$ is a finite set of edges which assign a nonempty set of next states for each gateway.

Declarative Specification
The declarative specification technique of BP is gaining in popularity within the scientific community. This fact can mainly be attributed to its highly flexible nature. Instead of focusing on how a task is performed, like imperatively specified processes, it focuses on what tasks are performed (Schonenberg et al., 2008). In its most basic form, a declaratively defined BP model consist of a set of tasks and a set of constraints (e.g., temporal logic formulas) enforcing some possible ordering on the set of tasks.

Summarizing, a declarative process model is a tuple $P_D = (SA, F)$, where:

- SA is a finite set of *structured activities*,
- F is a finite set of control flow *constraints*.

2.1.2 Standards

BP are supported by a variety of *standards*. These standards can be categorized into two broad groups: *modeling standards* and *programming standards*. Modeling standards include formalizations towards the notation of BP in the form of models. Programming standards include formalizations aimed at the enactment and serialization of workflows. The most notable modeling standards consist of the Unified Modeling Language (UML) Activity Diagrams (OMG, 2015) and the Business Process Model and Notation (BPMN) standard's Business Process Diagram (BPD) (OMG, 2011). The most used enactment standard is the Web Service Business Process Execution Language (WS-BPEL) (Arkin et al., 2007).

In this thesis the BPMN BPD is used to model all BP. BPMN BPD are imperatively specified BP models and consist of a graph conceived from flow objects and connecting objects annotated with data, artifacts, and swim lanes. Flow objects include a myriad of tasks (e.g. activities, sub-processes, or transactions), events (e.g. start events, end events, or intermediate throwing and catching events), and gateways (e.g. exclusive, inclusive, parallel, or event-based gateways). Connecting objects include sequence flows, conditional flows, and default flows. BPD can be annotated with data objects and data stores representing the flow and storage of information

or artifacts such as groups. Finally, pools represent participants and can be subdivided into lanes to embody roles. Elements within a pool or lane are performed by the role attached to that pool or lane. Enclosed within Appendix A a comprehensive list of BPMN BPD elements and their function can be found. Although BPD are at the basis of all techniques presented within this document, note that all the represented techniques can easily be applied to other modeling and programming standards such as the UML Activity Diagram or WS-BPEL processes.

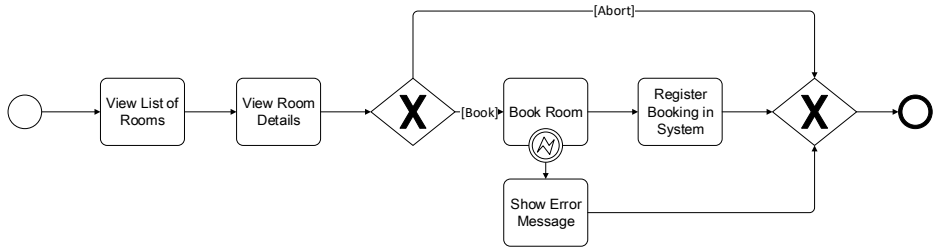


Figure 2.1: BPMN BPD example

Figure 2.1 illustrates a BPMN BPD describing a process for the booking of meeting rooms. The illustrated BPD includes tasks, gateways, events, and sequence flows (represented by rounded rectangles, diamonds, circles, and arrows respectively). When the process is initiated, the user reviews a list of rooms. Upon selecting a room, the room details are shown. The user then decides to either book this room or exit the process (upon which the user can simply restart it to select another room). If the user decides to continue, the system attempts to book the room. If successful, the user is notified that the room has been booked and the process is terminated. If an error occurs during the booking process, due to, for example, someone booking the room in the meantime, the user is notified and the process is terminated.

2.1.3 Management

A business process is a collaboration with the aim to achieve a specific value-added goal driven by external need. BPM manages and optimizes the business processes of a company with the aim to significantly increase the productivity and performance of that company. Figure 2.2 illustrates the process. The four phases of the BPM life-cycle consist of (van der Aalst et al., 2003b):

- **Process Design.** The business process is (re-)designed.
- **System Configuration.** The enactment system is configured to execute the designed business process.

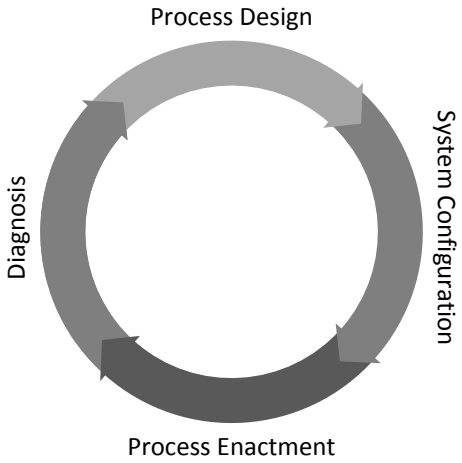


Figure 2.2: The BPM Life-cycle. Source: (van der Aalst et al., 2003b)

- **Process Enactment.** The implemented business process is executed.
- **Diagnosis.** The business process is monitored, simulated, or mined in order to identify and solve issues or find optimization possibilities.

BPM is considered the second evolution of workflow management (WfM) (van der Aalst et al., 2003b). Where WfM originally focused on the design, configuration, and enactment of workflows, BPM introduces the additional concept of diagnosis. During diagnosis, business processes are monitored, simulated, or otherwise observed in order to identify and solve issues or to identify opportunities for process optimization. In this document, we will primarily focus on the design and diagnosis phases of the BPM life-cycle to support automated design-time variability and compliance verification of business process models.

2.2 Business Process Formalization

Business process modeling techniques are often informal, that is, they often lack formally defined semantics (van der Aalst et al., 2003b). As a result, these modeling techniques are hardly suitable for any formal analysis. Models defined using such an informal business process modeling technique must, therefore, be formalized before formal analysis can contribute adequate results.

2.2.1 Petri Nets

Place/transition nets, or Petri nets, are mathematical models for the description of distributed systems (Petri, 1966). Petri nets are directed bigraphs with nodes consisting of *places* and *transitions*. Transitions within Petri nets represent events while places represent conditions. *Arcs* form weighted directed edges between place and transition pairs. Places may contain *tokens*. A distribution of tokens over places is called a *marking*. Unlike most business process modeling techniques, Petri nets do possess a formally defined semantics while offering a graphical notation. A Petri net is defined as follows (Petri, 1966; Reisig and Rozenberg, 1998):

Definition 2.2.1 (Net)**.** *A net is a triple* $N = (P, T, A)$*, where:*

- P *is a finite set of* places*,*
- T *is a finite set of* transitions*, such that* $P \cap T = \emptyset$*,*
- $A \subseteq P \times T \cup T \times P$ *is a finite set of* arcs*.*

Definition 2.2.2 (Petri Net)**.** *A Petri net is a triple* $PN = (N, M, W)$*, where:*

- $N = (P, T, A)$ *is a net (Definition 2.2.1),*
- $M : P \to Z$ *is a place multiset, the* marking*, where* Z *is a countable set,*
- $W : A \to Z$ *is an arc multiset, the* weight *of the arc.*

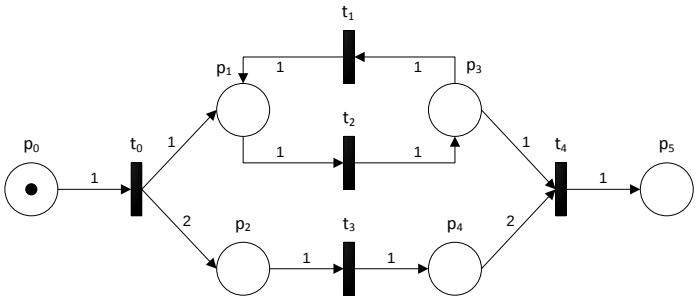


Figure 2.3: Petri net example

Figure 2.3 illustrates a Petri net in its graphical notation where bars represent transitions, circles represent places, and dots at circles represent the distribution of tokens over places. Places with an arc that run to (from) a transition are called the *input (output) places* of that transition. For a transition t, we write $\bullet t$ and $t\bullet$ for the set of in- and output places, respectively. A transition may *occur* when it is *enabled*, i.e., when all of its input places contain more or an equal amount of tokens than

the weight at the connecting arc. When a transition occurs it consumes tokens at its input places and produces tokens at its output places. The number of tokens consumed/produced at a place corresponds to the *weight* of the connecting arc. For example, transition t_0 of Figure 2.3 is enabled because its input place p_0 contains a token. It may then occur, consuming the token at p_0 and producing one token at p_1 and two tokens at p_2.

2.2.2 Petri Net analysis

One analysis tool used with Petri nets is the reachability graph (RG). A RG is a transition system obtained through the firing rule. Starting from the initial marking M_0, states are created for each encountered marking while enabled binding elements occur to generate new markings. The RG of a Petri net is defined as follows (Huber et al., 1986):

Definition 2.2.3 (Reachability Graph)**.** *The reachability graph of a Petri net with markings $M_0, ..., M_n$ is a rooted directed graph $G = (V, E, v_0)$, where:*

- $V = \{M_0, ..., M_n\}$ *is the set of* vertices*,*
- $v_0 = M_0$ *is the* root node*,*
- $E = \{(M_i, t, M_j) \mid M_i \in V \wedge M_i \xrightarrow{t} M_j\}$ *is the set of* edges*, where each edge represents the firing of a transition t at a marking M_i such that a marking M_j is produced.*

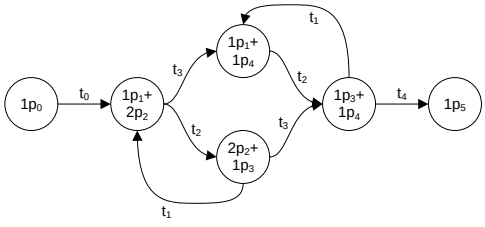


Figure 2.4: Reachability graph example

Nodes of the RG represent the different possible markings of the Petri net, i.e., the distribution of tokens over places. The initial distribution of tokens, or initial marking forms the root node. Edges represent the occurrence of transitions and the

related changes in the distribution of tokens over places. Figure 2.4 illustrates the RG of the Petri net and initial marking depicted in Figure 2.3.

2.2.3 Colored Petri Nets

Colored Petri nets (CPN) extend the normal class of Petri nets with the ability to attach information, called *colors*, to tokens. The color of a token can be inspected and modified by occurring transitions. A CPN is defined as follows (Jensen, 1981):

Definition 2.2.4 (Colored Petri Net)**.** *A Colored Petri Net is a 9-tuple* $CPN = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$*, where:*

- Σ *is a finite set of non-empty types, called* color sets*,*
- P *is a finite set of* places*,*
- T *is a finite set of* transitions*,*
- A *is a finite set of* arcs *such that* $P \cap T = P \cap A = T \cap A = \emptyset$*,*
- N_f *is a* node function *defined from* A *over* $P \times T \cup T \times P$*,*
- C_f *is a* color function *defined from* P *into* Σ*,*
- G_f *is a* guard function *defined from* T *into expressions such that* $\forall t \in T : [Type(G_f(t)) = Bool \wedge Type(Var(G_f(t))) \subseteq \Sigma]$*,*
- E_f *is an* arc expression function *defined from* A *into expressions such that* $\forall a \in A : [Type(E_f(a)) = C_f(p(a))_{MS} \wedge Type(Var(E_f(a))) \subseteq \Sigma]$ *where* $p(a)$ *is the place of* $N_f(a)$*,*
- M_0*, the* initial marking*, is a function defined on* P*, such that* $M(p) \in [C(p) \to \mathbb{N}]_f$ *for all* $p \in P$*.*

The CPN state, often referred to as the *marking of CPN*, is a function M defined on P, such that $M(p) \in [C_f(p) \to \mathbb{N}]_f$ for all $p \in P$. Let p be a place and t a transition. Elements of $C_f(p)$ are called *colors*. p is an *input place* (*output place*) for t iff $(p,t) \in N_f$ ($(t,p) \in N_f$) (Jensen, 1981). Every CPN is paired with an initial marking M_0. Transitions of a CPN may *occur* in order to change the marking of the CPN per the *firing rule* (Jensen, 1981). Places containing tokens in a marking enable possible binding elements (t, b), consisting of a transition t and a binding b of variables of t. A binding element is enabled if and only if enough tokens of the correct color are present at the input places of transition t and its guard evaluates true. More formally, iff $\forall p \in P : E_f(p,t)\langle b \rangle \leqslant M(p)$. An enabled binding element may

occur, changing the marking, by removing tokens from the input places of t and adding tokens to the output places of t as dictated by the arc evaluation function. Then, a multiset Y of binding elements (t, b), or a step, is enabled iff $\forall p \in P : \sum_{(t,b)\in Y} E_f(p,t)\langle b\rangle \leqslant M(p)$, or if the sum of the binding elements is enabled. The occurrence of a step Y at a marking M_i produces a new marking M_j as denoted by $M_i \xrightarrow{Y} M_j$. All possible states of a CPN can be obtained from the initial marking through the firing rule.

2.2.4 Workflow Nets

Petri nets are a popular method used for the formalization and analysis of business process models. Petri nets were first introduced as an analysis tool for business processes through the application of Workflow nets (WF-net) (van der Aalst, 1997). A WF-net is a Petri net where transitions represent activities and the control flow of the WF-net is represented by the distribution of tokens at places, i.e., its marking. A WF-net is defined as follows (van der Aalst, 1997):

Definition 2.2.5 (Workflow net)**.** *A net $N = (P, T, A)$ (Definition 2.2.1) is a workflow net iff:*

- *$i \in P$ is a* source *place with $\forall t \in T : (t, i) \notin A$,*
- *$o \in P$ is a* sink *place with $\forall t \in T : (o, t) \notin A$,*
- *if we add a transition t^* to T and the arcs (t^*, i) and (o, t^*) to A, then the resulting net is strongly connected, meaning that for every pair of nodes $x \in P \cup T$ and $y \in P \cup T$ there exists a directed path from x to y.*

Further formalization was introduced by adding support for Or-joins and cancelation regions (Wynn et al., 2009), and a basic translation from processes described using the BPMN standard to WF-nets was introduced in (Dijkman et al., 2008).

Figure 2.5 illustrates the process depicted in Figure 2.1 for the booking of meeting rooms as a WF-net. The start and end events are represented by the source and sink places respectively, activities are represented by transitions, and places are added to determine the control flow. Additionally, a *silent transition*, i.e., a transition without an attached action (generally represented by a black transition), is required in order to support the option of aborting the booking of the selected room. Silent transitions offer a generic solution to support the behavior of different gates when representing BP using Petri nets (Dijkman et al., 2008).

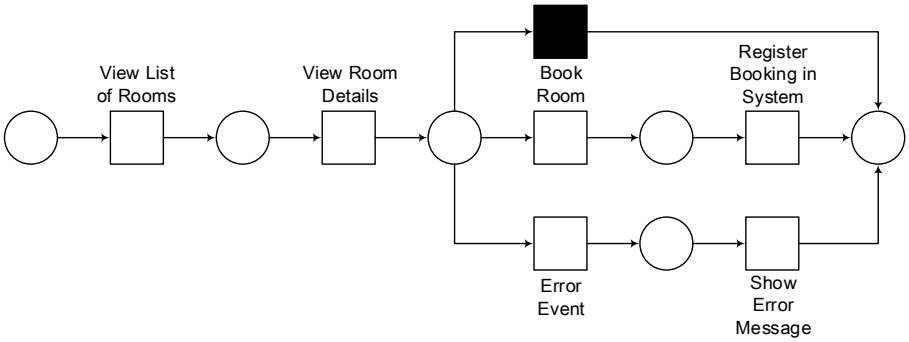


Figure 2.5: WF-net example

2.2.5 Workflow Patterns

The workflow pattern initiative aims to provide a conceptual basis for process technology (van der Aalst et al., 2003a). It offers an exhaustive list of patterns that should be supported by BP modeling and enactment techniques. The list of patterns is primarily used to select the most suitable BP modeling specifications or enactment systems for a specific task based on their pattern support, but can also be used as a reference for the definition of new, or updated versions of, such modeling specifications or enactment systems. In addition, they can be used as a basis for the development of tools.

The set of workflow patterns consists of five broad categories, specifically those regarding data, resource, exception handling, and, most notably, the control flow of BPs. Control flow patterns include those patterns related to the control flow dependencies between tasks (van der Aalst et al., 2003a). That is, it includes patterns describing the sequence, parallelism, choice, and synchronization of tasks within BPs. These patterns are described in an imperative way and represented by CPN models. Initially, the workflow patterns contained 20 patterns which describe the control flow of BP (van der Aalst et al., 2003a). The list of control flow patterns was later revisited and extended to over 40 patterns of which several list one or more alternatives (Russell et al., 2007).

Throughout this document we use the control flow patterns to formalize BP as CPN through the pattern mapping presented in Appendix B. In this way, the patterns provide a well-supported, specification independent, and formal foundation for the presented research and toolset.

2.3 Formal Verification

Validation and verification are procedures used to investigate whether a software or hardware product fulfills its intended purpose. Validation investigates if the specified product fulfills the needs of the user, that is, it tries to answer the question if the correct product is being made. Verification, on the other hand, investigates if the product conforms to its specifications – or, in other words, whether the product is being made correctly. When applying formal methods of mathematics to verification, the procedure is called formal verification. Formal verification entails proving or disproving the correctness of a system model with respect to a formal specification using formal methods of mathematics. When employing formal verification, a system model – often represented by a labeled transition system – is verified against a formal specification in the form of a set of logic formulas. One approach towards formal verification is model checking. When model checking a system model is automatically, systematically, and exhaustively explored while each explored state is verified to be compliant with the formal specification. Next, the system models and fomal specifications used for the formal verification of business process models in the remainder of the text are introduced.

2.3.1 System Models

During formal verification, a system model – often a labeled transition system (LTS) – is verified against specifications of interest. An LTS is a directed graph where nodes represent different states of the system and edges represent state transitions. Two labeling functions may exist over an LTS; a labeling function over nodes maps states with those properties that hold at that state, while a labeling function over edges maps state transitions with the actions which cause the state change. When considering model checking, the resulting model is sometimes called a state graph. Other, more specific, system models used in the domain of model checking are Kripke structures (Clarke et al., 1999) and Büchi automata (Büchi, 1962). A Kripke structure is an LTS with a labeling function over its nodes. Kripke structures are often used to interpret temporal logics. A Kripke structure is defined as follows (Clarke et al., 1999).

Definition 2.3.1 (Kripke structure)**.** *Let AP be a set of* atomic propositions*. A Kripke structure K over AP is a quadruple $K = (S, S_0, R, L)$, where:*

- *S is a finite set of* states,
- *$S_0 \subseteq S$ is a set of* initial states,

- $R \subseteq S \times S$ *is a* transition relation *such that it is* left-total*, meaning that for each* $s \in S$ *there exists a state* $s' \in S$ *such that* $(s, s') \in R$*,*
- $L : S \to 2^{AP}$ *is a* labeling function *with the set of atomic propositions that are true in that state.*

Büchi automata, on the other hand, are a form of automaton with a labeling function over its edges. Additionally, Büchi automata define an acceptance function which only accepts those runs of the automaton which visits one of a set of accepting states infinitely often. In the domain of model checking, Büchi automata are used to represent linear temporal logics. A non-deterministic Büchi automaton is defined as follows (Büchi, 1962).

Definition 2.3.2 (Non-deterministic Büchi automaton)**.** *Let* Σ *be a finite* alphabet *and* Σ^ω *the infinite set of* words *over* Σ*. A Büchi automaton over* Σ *is a quadruple* $A = (Q, I, \Delta, F)$*, where:*

- Q *is a finite set of* states*,*
- $I \subseteq Q$ *is a set of* initial states*,*
- $\Delta \subseteq Q \times \Sigma \times Q$ *is a* transition relation*,*
- $F \subseteq Q$ *is a set of* accepting states*.*

A run of a Büchi automaton is an infinite word $\omega = \alpha_1\alpha_2...$ as an infinite sequence of states $\pi = q_0q_1q_2...$ such that $q_0 \in I$ and $(q_i, \alpha_{i+1}, q_{i+1}) \in \Delta$ for $i \leqslant 0$. A run of a Büchi automaton is said to be accepting iff $\text{inf}(\pi) \cap F \neq \emptyset$ where $\text{inf}(\pi) = \{q \mid q$ occurs infinitely often in $\pi\}$. Kripke structures can be converted to Büchi automata through the following definition.

Definition 2.3.3. *Let* AP *be a set of atomic propositions,* Σ *a finite alphabet, and* $K = (S, S_0, R, L)$ *a Kripke structure over* AP*. The Büchi automaton* $A = (Q, I, \Delta, F)$ *over* Σ *of the Kripke structure* K *is:*

- $Q = S \cup \{q_0\}$ *is the set of Kripke structure states with an additional initial state* q_0*,*
- $I = \{q_0\}$ *is the initial state* q_0*,*
- $\Sigma = 2^{AP}$ *is the set of words obtained from the atomic propositions,*
- $(q, v, q') \in \Delta$ *if* $(s, s') \in R$ *and* $v = L(q')$ *is the set of transitions obtained from the Kripke structure relations and labeling function,*
- $(q_0, v, q') \in \Delta$ *if* $q' \in S_0$ *and* $v = L(q')$ *is the set of transitions from the initial state* q_0 *to the initial states obtained from the Kripke structure initial states,*
- $F = S \cup \{q_0\}$ *is the set of accepting states.*

2.3.2 Formal Specifications

Kripke structures are used to interpret temporal logics. Temporal logics are formalisms that are able to reason about the temporal succession of states within system models. Often used with formal verification, temporal logics can specify events over sequences of states or states in tree-like structures. Linear-time temporal logics specify properties (e.g., the universality of a certain state property, and the order of states) over states occurring on process execution paths. Branching-time temporal logics extends this set of temporal operators with path quantifiers, such that formulas can specify properties over branching executions (i.e. computation trees). In this way, linear-time temporal logics treat time as if each moment only has one distinct future, while branching-time temporal logics allow time to split into multiple possible futures. The most notable temporal logics include Linear-time Temporal Logic (LTL) (Pnueli, 1977), Computation Tree Logic (CTL) (Emerson and Halpern, 1982), and their superset Computation Tree Logic* (CTL*) (Clarke et al., 1999).

Linear-time Temporal Logic

LTL specifies temporal operators over sequences of states known as paths (Pnueli, 1977). A path $\pi = s_0 s_1 s_2 ...$, defined on a Kripke structure, is an infinite sequence of states such that $(s_i, s_{i+1}) \in R$ for $i \leqslant 0$.

Definition 2.3.4 (LTL syntax)**.** *The language of well-formed LTL formulas is generated by the following grammar, assuming $p \in AP$:*

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid$$
$$X\phi \mid G\phi \mid F\phi \mid \phi \; U \; \phi$$

LTL is equipped with four *temporal operators*:

- $X\phi$ Nexttime: ϕ has to hold at the next state.
- $G\phi$ Globally: ϕ has to hold at all states of the subsequent path.
- $F\phi$ Future: ϕ has to hold at the current or a future state.
- $\phi \; U \; \phi'$ Until: ϕ has to hold until ϕ', which holds at a future state or the current state itself.

Definition 2.3.5 (Semantics of LTL)**.** *$\pi, s_i \models \phi$ means that the formula ϕ holds for the path $\pi = s_0 s_1 s_2 ...$ at state s_i. The relation $\models$ is defined inductively as follows:*

$$\begin{array}{lll}
\pi, s_i \models \top & \textit{iff} & \pi, s_i \not\models \bot \\
\pi, s_i \models p & \textit{iff} & p \in L(s_i) \\
\pi, s_i \models \neg\phi & \textit{iff} & \pi, s_i \not\models \phi \\
\pi, s_i \models \phi \vee \phi' & \textit{iff} & \pi, s_i \models \phi \vee \pi, s_i \models \phi' \\
\pi, s_i \models X \; \phi & \textit{iff} & \pi, s_{i+1} \models \phi \\
\pi, s_i \models \phi \; U \; \phi' & \textit{iff} & \exists m : (m \geqslant 0 \wedge \pi, s_{i+m} \models \phi' \wedge \forall n : (0 \leqslant n < m : \pi, s_{i+n} \models \phi))
\end{array}$$

Further LTL operators can be obtained through the following equivalences:
$F\phi \equiv true\ U\ \phi$
$G\phi \equiv \neg F \neg \phi$

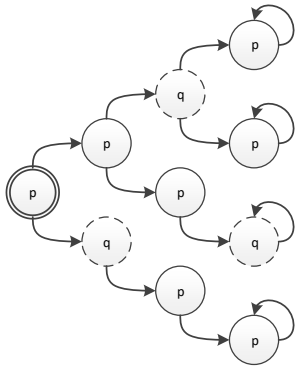


Figure 2.6: Linear-time Temporal Logics example

For example, Figure 2.6 depicts a Kripke structure where we evaluate the LTL formula Fq. Note that when applying LTL to Kripke structures, a formula must hold for all possible paths from every initial node in order to evaluate true. The formula Fq holds at the model since every path from the double bordered initial node includes a future state where q holds, i.e. the dashed states.

Linear-time Temporal Logics with Past-time Modalities
Linear-time Temporal Logics with Past-time Modalities (PLTL) introduces past-time operators to LTL (Markey, 2003). Adding past-time operators to LTL does not increase its expressiveness, but does make the logic exponentially more succinct.

Definition 2.3.6 (PLTL syntax)**.** *The language of well-formed PLTL formulas is generated by the following grammar, assuming $p \in AP$:*

$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid$
$\quad X\phi \mid G\phi \mid F\phi \mid \phi\ U\ \phi \mid$
$\quad Y\phi \mid H\phi \mid O\phi \mid \phi\ S\ \phi$

PLTL introduces four new *temporal operators*:

- Yϕ Previous-time: ϕ has to hold at the previous state.
- Hϕ Historically: ϕ has to hold at all states of the preceding path.
- Oϕ Once: ϕ has to hold at the current or a preceding state.
- ϕ S ϕ' Since: ϕ has to hold since a point where ϕ' holds, which holds at a past state or the current state itself.

Definition 2.3.7 (Semantics of PLTL)**.** *PLTL inherits all semantics from LTL.* $\pi, s_i \models \phi$ *means that the formula* ϕ *holds for the path* $\pi = s_0 s_1 s_2 ...$ *at state* s_i*. The relation* $\models$ *is defined inductively as follows:*

$\pi, s_i \models Y\ \phi$ *iff* $i \geqslant 1 \wedge \pi, s_{i-1} \models \phi$

$\pi, s_i \models \phi\ S\ \phi'$ *iff* $\exists m : (0 \geqslant m \leqslant i \wedge \pi, s_{i-m} \models \phi' \wedge \forall n : (m < n \leqslant i : \pi, s_n \models \phi))$

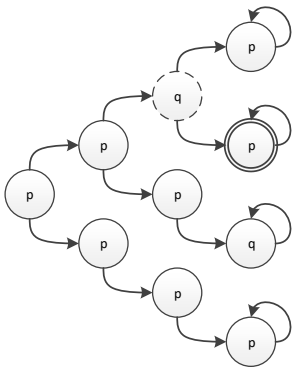


Figure 2.7: Linear-time Temporal Logics with Past-time Modalities example

Further PLTL operators can be obtained through the following equivalences:
$O\phi \equiv true\ S\ \phi$
$H\phi \equiv \neg O \neg \phi$

Figure 2.7, for example, depicts a model on which the PLTL formula Hq is evaluated. The formula Hq holds at the depicted model since all paths from the initial node include a historical, previous, state where q holds.

Computational Tree Logic*

The branching-time temporal logic CTL* augments LTL with two operators over paths to specify whether some or all branches possess properties starting at the current state (Clarke et al., 1999). CTL* defines the following operators over paths:

- Aψ All: ψ holds on all paths flowing from the current state.
- Eψ Exists: ψ holds on at least one path flowing from the current state.

Definition 2.3.8 (CTL* syntax)**.** *The language of well-formed CTL* formulas is generated by the following grammar, assuming $p \in AP$:*

$\Phi ::= \top \mid \bot \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid \Phi \Rightarrow \Phi \mid \Phi \Leftrightarrow \Phi \mid E\phi \mid A\phi$
$\phi ::= \Phi \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid X\phi \mid G\phi \mid F\phi \mid [\phi \, U \, \phi]$

Definition 2.3.9 (Semantics of CTL*)**.** *CTL* inherits the semantics of its operators from LTL, excluding $\top$, $\bot$, and p. $\pi \models \phi$ means that the formula ϕ holds for the path $\pi = s_0 s_1 s_2 ...$ at state s_0. Additionallly, $M, s_i \models \phi$ means that the formula ϕ holds at state s_i of the model M. When the model M is understood, $s_i \models \phi$ is written instead. The relation $\models$ is defined inductively as follows:*

$\pi \models \Phi$ *iff* $s_0 \models \Phi$

$s_i \models \top$ *iff* $s_i \not\models \bot$
$s_i \models p$ *iff* $p \in L(s_i)$
$s_i \models \neg\Phi$ *iff* $s_i \not\models \Phi$
$s_i \models \Phi \vee \Phi'$ *iff* $s_i \models \Phi \vee s_i \models \Phi'$
$s_i \models E\,\phi$ *iff* $\exists \pi : \pi = s_0 ... \mid s_0 = s_i \wedge \pi \models \phi$

Further CTL* operators can be obtained through the following equivalences:
$F\phi \equiv true\ U\ \phi$
$G\phi \equiv \neg F \neg \phi$
$A\phi \equiv \neg E \neg \phi$

For example, one can evaluate the CTL* formula $EFGq$ on the model depicted in Figure 2.8. The formula clearly holds at the model since there exists a path from the initial node which includes a future state where q holds indefinitely.

Computational Tree Logic

The branching-time temporal logic CTL is a subset of CTL* (Emerson and Halpern, 1982). Instead of allowing arbitrary combinations of temporal operators and operators over paths, CTL pairs each temporal operator with an operator over paths.

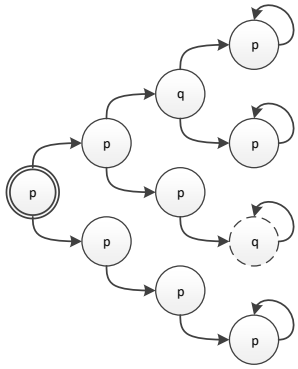


Figure 2.8: Computation Tree Logic* example

Definition 2.3.10 (CTL syntax)**.** *The language of well-formed CTL formulas is generated by the following grammar, assuming $p \in AP$:*

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid$$
$$AX\phi \mid EX\phi \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi \mid A[\phi \, U \, \phi] \mid E[\phi \, U \, \phi]$$

Definition 2.3.11 (Semantics of CTL)**.** *$M, s_i \models \phi$ means that the formula ϕ holds at state s_i of the model M. When the model M is understood, $s_i \models \phi$ is written instead. The relation $\models$ is defined inductively as follows:*

$$\begin{array}{lll}
s_i \models \top & \textit{iff} & s_i \not\models \bot \\
s_i \models p & \textit{iff} & p \in L(s_i) \\
s_i \models \neg\phi & \textit{iff} & s_i \not\models \phi \\
s_i \models \phi \vee \phi' & \textit{iff} & s_i \models \phi \vee s_i \models \phi' \\
s_i \models EX\ \phi & \textit{iff} & \exists(s_i, s_{i+1}) \in R \mid s_{i+1} \models \phi) \\
s_i \models EG\ \phi & \textit{iff} & \exists \pi = s_i, s_{i+1}, s_{i+2}, ... \mid \forall n : (n \geqslant 0 \wedge s_{i+n} \models \phi) \\
s_i \models E[\phi\ U\ \phi'] & \textit{iff} & \exists \pi = s_i, s_{i+1}, s_{i+2}, ... \mid \\
 & & \exists m : (m \geqslant 0 \wedge s_{i+m} \models \phi' \wedge \forall n : (0 \leqslant n < m : s_{i+n} \models \phi))
\end{array}$$

Further CTL operators can be obtained through the following equivalences:
$EF\phi \equiv E[true\ U\ \phi]$
$AF\phi \equiv \neg EG\neg\phi$
$AX\phi \equiv \neg EX\neg\phi$
$AG\phi \equiv \neg EF\neg\phi$
$A[\phi\ U\ \phi'] \equiv \neg(E[\neg\phi'\ U\ \neg(\phi \vee \phi')] \vee EG\neg\phi')$

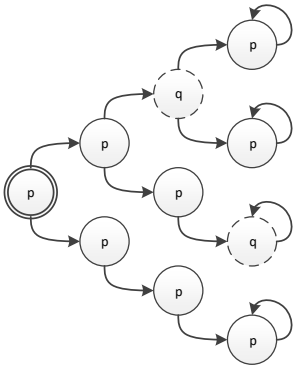


Figure 2.9: Computation Tree Logic example

For example, Figure 2.9 depicts a Kripke structure upon which the CTL formula EFq is evaluated. The formula holds at the model since there exists a path from the initial node which includes a future state where q holds. Actually, two such paths exist, as indicated by the dashed states, although the formula only requires one.

CHAPTER 3

State of the Art

You've got to want to be in this incredible feedback loop where you get the world-class people to tell you what you're doing wrong.

– Bill Gates

The area of BP verification has been the focus of a large amount of research. Existing research can be placed within three broad categories: (1) the verification of the correctness of a BP, known as *BP soundness*, (2) the verification of adherence of a BP to a set of rules, known as *BP compliance*, and (3), the verification of adherence of a BP to a flexible or adaptable specification, known as *BP variability*.

3.1 Business Process Soundness

Business process soundness is known as the combination of three properties: (1) the BP must complete (i.e., termination), (2) the absence of unreachable activities, and (3) the absence of other running activities upon termination (i.e., proper completion) (van der Aalst, 1998; Wynn et al., 2009). Soundness verification aims at verifying these three basic properties. A more relaxed notion of soundness exists as well. Known as weak soundness, it relaxes the requirement to complete in such a way that, when started, it must be possible to complete a BP merely in some cases. In (Trčka et al., 2009), three completion patterns are defined: mandatory (i.e., all paths must complete), optional (i.e., from every state there exists a path which com-

pletes), and possible completion (i.e, there exists a path which completes). These diminished version of soundness are introduced in order to verify soundness over BP with infinite state spaces (Wynn et al., 2009).

van der Aalst (1998) first introduced soundness to the field of BPM when translating BP into workflow nets. Later, (Wynn et al., 2009) perfected the approach by allowing Or-joins and cancelation regions. Finally, (van der Aalst et al., 2011) provides an overview of the different notions of soundness and shows that, although all notions are decidable for workflow nets, they become undecidable for most extensions of workflow nets. Petri nets have since been used by many in the field as intermediate formalisms in order to formalize BP. For example, (van Dongen et al., 2007), use it to formalize Event-driven Process Chains (EPC) before verifying soundness using state space analysis. Similarly, Masalagiu et al. (2009) appliy Petri nets as formalization of BPMN BPD when verifying soundness. And, Corradini et al. (2015) apply a class of Petri net to unfold business compositions for analysis and verification.

Another popular method towards soundness verification is the direct implementation of the process into the input language of a model checker. The described process is then internally converted into a labeled transition system by the model checker. Work that take this approach include those of (Karamanolis et al., 2000; Koehler et al., 2002; Nakajima, 2006) and (Masalagiu et al., 2009). However, since input languages are often designed to describe software programs, the approach is prone to requiring extra states. As a result, without carefully mapping of the process into the input language, unnecessary states are quickly introduced. For example, in (Nakajima, 2002), it is reported that the intermediate mapping causes a simple process of five activities and four transitions to be mapped to 201 states and 586 transitions in the internal state machine of the model checker. Similarly, Kherbouche et al. (2013) reports mapping a process consisting of four activities to over 115000 states using the same model checker.

Since many verification approaches only support acyclic BP, Choi and Zhao (2005) propose decomposing cyclic BP into non-cyclic subgraphs in order to detect deadlocks within feedback loops.

Other approaches towards soundness verification include propositional logic based verification (Bi and Zhao, 2004). By transforming the control flow of BP to propositional logic, deadlocks, reachability, proper completion, and infinite cycles can be asserted. Weber et al. (2010) proposes semantic annotations to verify soundness

through pre-and post-condition verification for each task. Finally, Ma et al. (2008) propose the use of π-calculus to encode BP for soundness verification.

3.2 Business Process Compliance

Compliance verification aims to prove or disprove whether a BP adheres to a set of rules that has been imposed on it through, for example, law, regulation, or business requirements. Where soundness verification aims at the verification of a limited set of requirements to verify reachability, termination, and possibly proper completion (van der Aalst, 2000) – compliance verification requires the verification of a broad set of specifications.

Existing techniques perform compliance verification at different stages of the BP lifecycle, during process design, enactment, or diagnosis (Figure 2.2). Monitoring techniques are deployed during process enactment to utilize the runtime trace of a BP to check whether a model is executing correctly. Existing monitoring techniques include, for example, (Chesani et al., 2009). Auditing techniques are deployed during the process diagnosis phase and adopt, for example, process mining techniques to verify if a BP has been executed correctly. Notable auditing techniques include those presented in (Ghose and Koliadis, 2007; Ly et al., 2011), and (Schunselaar et al., 2012a).

Naturally, monitoring and auditing techniques are *after the fact* techniques, meaning that issues will only ever be detected after they already have occurred. As a result, rollbacks are required in order to undo any erroneous execution before the application of possible sanctions. Hence, where possible, a preventative approach is preferred. Preventative approaches are deployed during design-time, aiming to prevent issues from ever occurring.

Existing preventative approaches include both formal and informal ones. Formal approaches utilize both a formal representation of the used model and a formal specification. Informal approaches are those that lack either a formal representation of the used model, a formal specification, or both. For example, an approach directly verifying CTL based specifications on a BP, without the proper support for different branching options through gateways, is considered to be informal. Informal techniques include (Awad et al., 2008) due to its incomplete reduction rules, and (Pulvermueller et al., 2010) due to the direct application of temporal logic upon the process model without taking into account different types of gates.

In (Elgammal et al., 2010a, 2014) well-known temporal logics are evaluated for suitability of preventative BP compliance verification. Although such an evaluation is extremely important for the selection of temporal logics for compliance verification, it is important to evaluate them with respect to the used model.

Other approaches introduce new or newly extended formal specifications. For example, Governatori et al. (2006), and Goedertier and Vanthienen (2006) both introduce deontics logics to formulate compliance specifications, Bulanov et al. (2011) propose Temporal Process Logics (TPL), a modal propositional logic that is able to reason about possible process executions, Gerede and Su (2007) propose a CTL based language, while Deutsch et al. (2009) proposes a first-order extension of LTL to verify all possible process executions of artifact-centric systems. Finally, D'Aprile et al. (2011) propose an extension of LTL to verify compliance based on answer set programming. However, by introducing new or extended logics the power of known and accomplished logics as well as their supporting model checkers can not be exploited.

In order to simplify the challenge of formal preventative compliance verification, techniques often limit their application to acyclic models, i.e., models that do not include arbitrary cycles or loops. Acyclic compliance techniques include, for example, (Ghose and Koliadis, 2007; Weber et al., 2008; Favre and Hagen, 2010; Montali et al., 2010). Arbitrary cycles are indeed a problematic, but very powerful, feature of BPM which can not simply be overlooked.

Further approaches encode service composition in such a way that a large amount of overhead is included within the state space of the model. This effect can often be traced to the decision of directly encoding service compositions into the modeling language of a model checker without careful analysis of the effect of the encoding on the internal state machine of the model checker. Approaches of this kind include, (Janssen et al., 1998; Latvala and Heljanko, 2000; Eshuis and Wieringa, 2004; Anderson et al., 2005; Fisteus et al., 2005; Bianculli et al., 2007), and (Kheldoun et al., 2015). For example, (Kheldoun et al., 2015) produces 247 states for a high level Petri net consisting of ten transitions and thirteen places, without parallelism.

Formal preventative compliance verification is achieved by obtaining a formal model (e.g. Kripke structure, Definition 2.3.1) from the BP model. Parallel branching constructs are then supported by interleaving concurrently executing branches. Some approaches, however, disregard parallel information entirely. Such approaches include, for example, (Feja et al., 2009).

Other approaches do interleave parallel branches correctly, but interleave to such an extent that concurrent executions are linearized entirely, parallel information is lost, and duplicate states, with accompanying state explosion, are introduced. Such approaches include (Foster et al., 2003; Fu et al., 2004), and (Liu et al., 2007). Parallelism is an important aspect of BPM, therefore information towards possible parallel execution can be of particular importance to compliance verification.

Elgammal et al. (2012) propose an integrated approach using both design- and runtime techniques. The approach builds upon CRL, a logic grounded in LTL, and guarded automata introduced in (Fu et al., 2004) for preventative design-time compliance verification, and Xpath expressions for runtime compliance verification.

In (Latvala and Heljanko, 2000), a translation from Petri nets to Kripke structures is proposed. By introducing intermediate states to the Kripke structure for each transition, the approach is able to define fairness conditions concerning the firing of transitions.

Finally, Esparza (1993) proposes an approach towards Petri net verification based on net unfolding. However, the approach bases its verification on the marking of the net (i.e., tokens at places). Instead, we are interested in the firing of transitions. Although the enabling of transitions can be obtained from the marking of a net, it introduces the same issue as verification over a reachability graph; a transition may be enabled without ever actually occurring. As a result, a stronger sense of transition enabling is required.

3.3 Business Process Variability

In software engineering, variability refers to the possibility of changes in software products and models (Sinnema et al., 2006). In the context of BPM, *variability* indicates that parts of a BP remain variable in order to support different versions of the same BP depending on the intended use or execution context. BP variability is closely related to design-time process *adaptability* and runtime process *flexibility*, which both support process change. Existing approaches that offer process change can be subdivided into those allowing change within imperatively specified BP and those allowing change through underspecification by using a declarative approach (van der Aalst and Jablonski, 2000; Marin et al., 2013; Schonenberg et al., 2008).

When offering design-time change within imperatively specified BP, many existing approaches apply principles directly from variability as used within software prod-

uct lines – such as feature modeling and variation points (Sinnema et al., 2006). A software product line consists of a family of closely related software products with a single generic implementation. Differences between each product within the product line are described using feature models. When creating a variant, the relevant features are selected and integrated at, so called, variation points within the generic implementation. When applying this principle to BP adaptability, the generic implementation is provided by an imperatively specified BP with included variation points. Approaches using these principles include (van der Aalst et al., 2005; Schnieders and Puhlmann, 2006; Chang and Kim, 2007; Rosemann and van der Aalst, 2007; van Eijndhoven et al., 2008; Gottschalk et al., 2008, 2009; Razavian and Khosravi, 2008; Sun and Aiello, 2008; Hadaytullah et al., 2009; La Rosa, 2009) and (Nguyen et al., 2011). On the other hand, Hallerbach et al. (2008) employs the same principles to offer change during process enactment. Of course, a single generic model incorporating all possible variations can contain configurations which lead to unsound processes. To address this problem, van der Aalst et al. (2012) propose a verification approach which is able to characterize all feasible configurations at design-time, while in (van der Aalst et al., 2010), the authors propose configurations which are able to maintain correctness.

In order to obtain generic models, (Schunselaar et al., 2012c, 2014; La Rosa et al., 2010, 2013), and (Bulanov et al., 2011) propose process merger. When applying this approach, variants are merged into a single generic model using a number of different techniques such as a new temporal process logic TPL, or CoSeNets. Likewise, Buijs et al. (2013) propose and evaluate four merging techniques to describe a family of BP variants using configurable process models.

Alternatively, van der Aalst and Basten (2002) utilize principles from object-oriented programming languages to define process inheritance. Inheritance is a mechanism which allows a subclass to inherit features from a superclass. When applied to BP, inhertance defines a bisimilarity relation over two process models. A process model is a subclass of another process model when the subclass and superclass are bisimilar under certain conditions. Milani et al. (2016), on the other hand, propose a decomposition based method using sub-processes which decides which parts should be modeled together, and which should not.

Design-time change within imperatively specified BP, however, require all possible features to be modeled in advance. As a result, all features must be known in advance. In addition, some features may have relations with other features (e.g., be exclusive or prerequired) which must be modeled. Declarative process specifica-

tions, on the other hand, do not require this knowledge to be modeled in advance. Therefore, we focus on declarative process specifications for generic templates of process families and and the automatic merging of variants into process family templates with varying degrees of variability.

Declarative process specifications offer change naturally through underspecification. Anything not specifically specified is subject to possible change. Numerous declarative approaches exist, of which most focus on supporting change during process enactment. Existing approaches consist of both formal and informal approaches. Informal approaches are those that lack either a formal representation of the used model, a formal specification, or both. Informal approaches include the work of Sadiq et al. (2005) which propose an algorithmic approach, and Pascalau et al. (2011) which extend the compliance work in (Awad et al., 2008) which suffers from informal reduction rules. Other approaches offer change by specifying pre-and post-conditions for structured activities. Any change is allowed, as long as pre-conditions are met and post-conditions can be met. Such approaches include (Rychkova et al., 2008) and (Dadam and Reichert, 2009).

Existing formal approaches employ temporal logics to define the control flow of BP during process enactment. Any change is allowed, as long as the temporal logic specifications are not violated. Approaches include (Pesic and van der Aalst, 2006; van der Aalst and Pesic, 2006; Demeyer et al., 2010; Hildebrandt and Mukkamala, 2011). Maggi et al. (2011) extend upon (Pesic and van der Aalst, 2006) in order to support change pre-runtime, but report verification issues when encountering arbitrary cycles. Finally, Schunselaar et al. (2012b) extend upon (Pesic and van der Aalst, 2006) with configurable inclusion of activities and specifications.

Finally, De Giacomo et al. (2015) extend the well-known imperative BPMN BPD specification with declarative flow control in order to develop a truly declarative BP specification. However, the approach does not consider parallel behavior or runtime consequences of the notation. For example, insertable tasks can be repeated any number of times, or simply avoided all together. In other words, tasks are either entirely optional, or required without any option for change. Having a required task which can be included in different places of the BP is simply not possible, and considering parallel support would only increase these issues.

Although declarative process specifications overcome the difficulties of imperative variability – which require knowledge of all change in advance – their abstract nature does introduce design difficulties which the intuitive imperative specification

does not face. In this document, we alleviate these issues by applying declarative specifications over imperative designs. In other words, we verify compliance of imperative specifications to declaratively specified process families.

3.4 Discussion

When analyzing the state of the art, we conclude that the verification of BP soundness has been perfected through the application of Petri nets. In fact, Petri nets have been found to be a popular tool when formalizing BP.

Secondly, current BP compliance approaches aim largely at after-the-fact auditing or the runtime monitoring of BP. Although certainly important, these techniques can never prevent issues of non-compliance, and can, as a best case scenario, only lead to rollbacks of work that has already been performed. Existing preventive approaches, on the other hand, either limit model behavior, severely impacting the powerful features supported by BP modeling languages, or suffer from limited support from known and accomplished model checkers because of state space requirements or the application of new, or newly extended, logics. However, while specifications have been extended to support the powerful features of BP modeling languages, little research has been devoted to exploring their support through translation of the model itself.

Thirdly, BP variability is most often supported at design-time using an imperative approach. Variability, however, has never been seen as an extension of preventive BP compliance verification. Even though the required specification rules show remarkable similarities. Although some declarative approaches exist, they are either naive, or are aimed at providing BP flexibility for linear runtime executions.

Finally, when model checking is involved, verification is often aimed at specific modeling languages and it involves a direct translation into the input language of the model checker. In these cases, the approach focuses mainly on the correct specification of the BP using that modeling language, i.e., soundness with respect to the modeling language, and not BP compliance verification or variability. Because of this focus, a large amount of states, unimportant to BP compliance verification or variability, are introduced.

Within the remainder of this thesis, we explore the support of BP verification and variability through the application of model checking. We provide means for the support of formal verification of powerful BP modeling language features through

the combined power of existing specification languages and proper model translation using Petri nets as a formalization step. In addition, we provide design-time variability mechanisms as an extension to preventive compliance verification.

CHAPTER 4

Case Study Description and Formalization

The rise of Google, the rise of Facebook, the rise of Apple, I think are proof that there is a place for computer science as something that solves problems that people face every day.

– Eric Schmidt

We present three case studies to serve as complex real world examples of BP verification. The first case entails a customer support process resulting from a compliance study at an Australian telecommunications provider which must comply to the Telecommunications Consumer Protections (TCP) code of conduct. The second case consists of a variability study throughout a number of Dutch municipalities which all are required by law to offer the same service to its residents, but tailored to local needs. Finally, the third case consists of a collaborative BP taken from (Corradini et al., 2015). We select these three cases to demonstrate the complexities of compliance verification, business process variability, and verification under collaborative concurrency. Each case is discussed and subsequently formalized by translating it to CPN according to the translation process described in Appendix B. To improve readability of the resulting CPN, the weight and color of the depicted arcs have been omitted. Unless stated otherwise, all arcs carry the weight $1`c$.

4.1 Case 1: Telecommunications Customer Support

To illustrate the complexities of preventative compliance verification, a real life case-study, concerning customer support at an Australian telecommunications company, is presented. The customer support process is depicted in Figure 4.1.

The process starts when a complaint from a customer is received. The complaint is registered in the system and the customer is called back immediately or later, depending on the urgency of the complaint. If no further contact can be established with the customer, the complaint is closed in the system and, in case the complaint concerns a Telecommunications Industry Ombudsman (TIO) complaint, the complaint is reported.

If contact is established with the customer, the issue is confirmed and the complaint is recorded. In case of a billing dispute, the credit management is suspended. If the issue can be easily resolved, the customer is informed of an offer to resolve the issue. The customer can accept or decline the offer. In case the customer does not accept the offer, a new offer can be provided if available, or the customer can escalate the issue.

If the complaint is more complicated to resolve, the customer is advised about the timeframe required to resolve the issue and possible available times of the customer are discussed. Subsequently, non-technical issues are investigated and technical issues are forwarded to Level 2 (L2) support. When a solution is available, it is presented to the customer to be accepted or declined. If there is a possible delay, the customer is notified.

Using the conversion provided in Appendix B, the customer support BPMN BPD depicted in Figure 4.1 is translated into a CPN. The resulting CPN is depicted graphically in Figure 4.2.

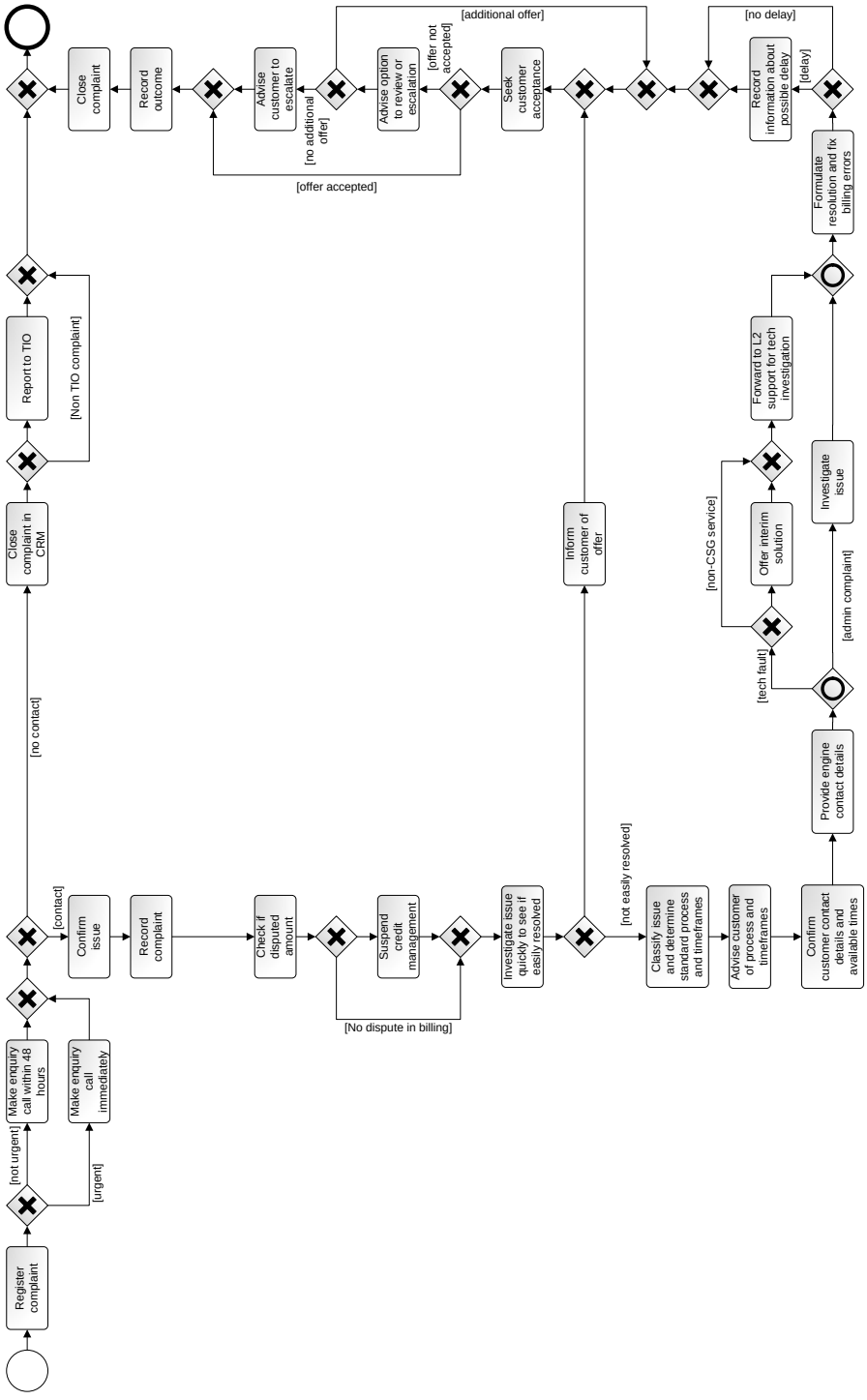
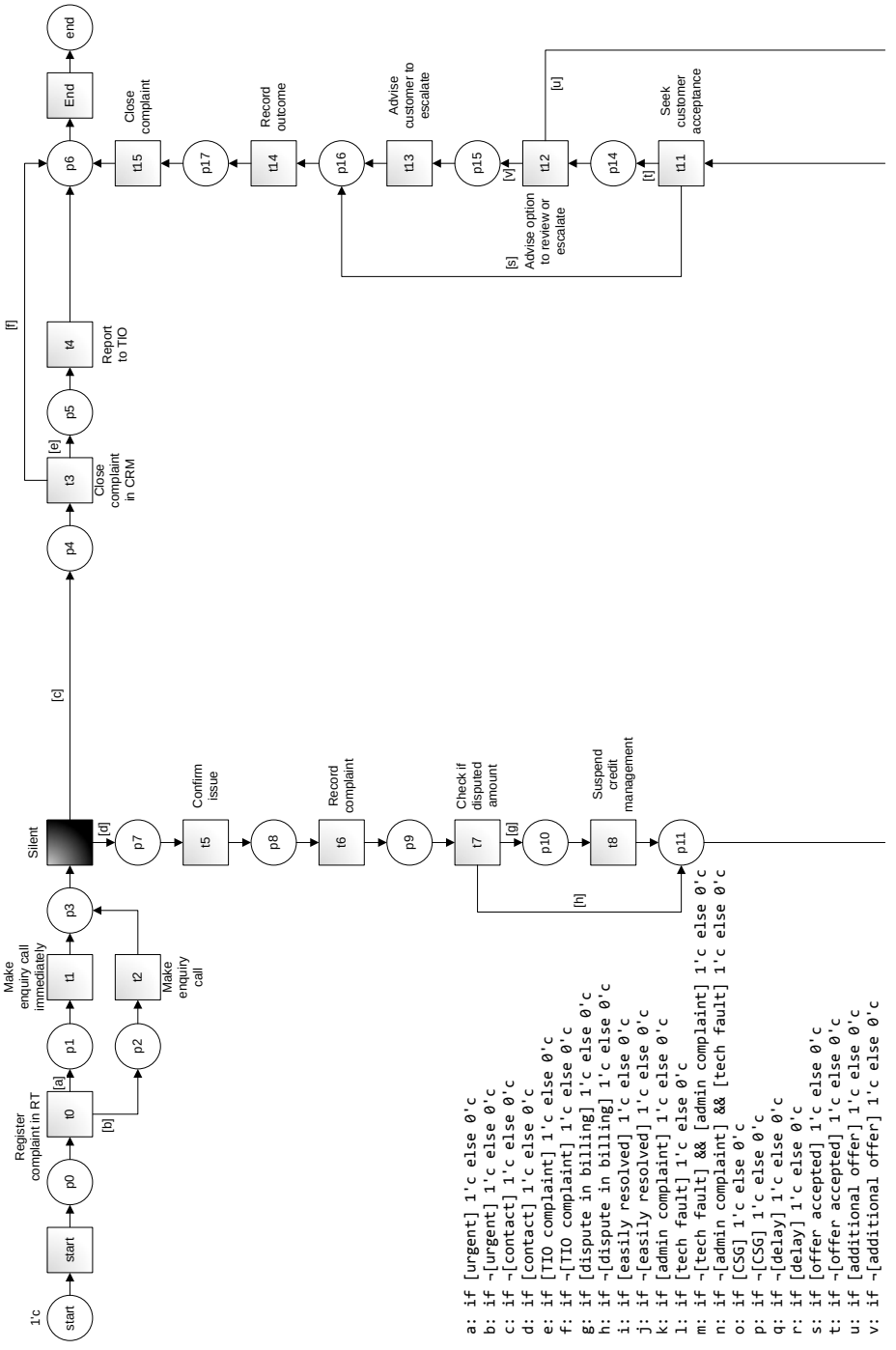


Figure 4.1: The customer support process.


1`c
start
start
p0
Register complaint in RT
t0
[a]
[b]
p1
p2
Make enquiry call immediately
t1
t2
Make enquiry call
p3
Silent
[c]
[d]
p4
t3
Close complaint in CRM
[e]
p5
t4
Report to TIO
[f]
p6
t15
Close complaint
p17
t14
Record outcome
p16
t13
Advise customer to escalate
p15
[v]
t12
[s]
Advise option to review or escalate
p14
[t]
t11
Seek customer acceptance
[u]
end
End
p7
t5
Confirm issue
p8
t6
Record complaint
p9
t7
Check if disputed amount
[g]
p10
t8
Suspend credit management
p11
[h]
a: if [urgent] 1`c else 0`c
b: if ¬[urgent] 1`c else 0`c
c: if ¬[contact] 1`c else 0`c
d: if [contact] 1`c else 0`c
e: if [TIO complaint] 1`c else 0`c
f: if ¬[TIO complaint] 1`c else 0`c
g: if [dispute in billing] 1`c else 0`c
h: if ¬[dispute in billing] 1`c else 0`c
i: if [easily resolved] 1`c else 0`c
j: if ¬[easily resolved] 1`c else 0`c
k: if [admin complaint] 1`c else 0`c
l: if [tech fault] 1`c else 0`c
m: if ¬[tech fault] && [admin complaint] 1`c else 0`c
n: if ¬[admin complaint] && [tech fault] 1`c else 0`c
o: if [CSG] 1`c else 0`c
p: if ¬[CSG] 1`c else 0`c
q: if ¬[delay] 1`c else 0`c
r: if [delay] 1`c else 0`c
s: if [offer accepted] 1`c else 0`c
t: if ¬[offer accepted] 1`c else 0`c
u: if [additional offer] 1`c else 0`c
v: if ¬[additional offer] 1`c else 0`c

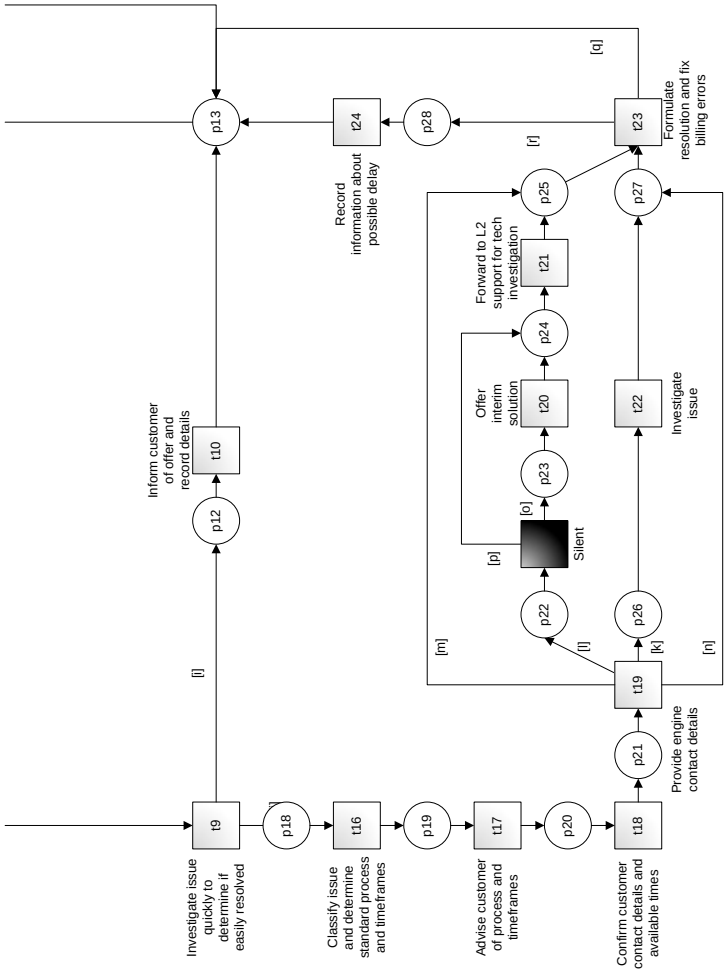


Figure 4.2: The complaint process as CPN.

To ensure good service and fair outcomes for all consumers of telecommunications products in Australia, all service providers whom supply telecommunications products to customers in Australia are required to comply to the Telecommunications Consumer Protections (TCP) code of conduct. The code is registered by the Australian Communications and Media Authority (ACMA), which has appropriate powers of enforcement. As a result, the customer support process as described above has to comply with a number of rules in order to meet the code of conduct. A number of those rules are enumerated below and used later to evaluate our methodology. Rules eight and nine are not part of the TCP code, but can be inferred to verify a number of control-flow requirements.

Table 4.1: TCP Compliance Rules.

#	**Compliance Rule**
1.	Resolutions to complaints should always be checked for acceptance with the customer, unless there is no contact with the customer.
2.	Offers are either accepted or the customer is advised to escalate.
3.	A complaint that is confirmed is recorded immediately.
4.	Once a complaint has been confirmed, its outcome is always recorded.
5.	Once a complaint has been confirmed, possible delays are recorded and communicated to the customer.
6.	All issues are covered prior to formulating a resolution.
7.	Escalated complaint are immediately recorded.
8.	When both technical and non-technical issues are involved in a complaint, they must be solved in parallel.
9.	After the complaint category is determined, a resolution must always be provided to the customer.

4.2 Case 2: Local Dutch e-Government

The Netherlands consists of 418 municipalities which all differ greatly. Because of this, each municipality is allowed to operate independently according to their local requirements. However, all the municipalities have to provide the same services and execute the same laws. An example of such a law which is heavily subjected to local needs is the WMO (Wet maatschappelijke ondersteuning, Social Support Act, 2006), a law providing needing citizens with support ranging from wheelchairs, help at home to home accessibility improvement.

Figures 4.3 through 4.8 illustrate the main process flows of the WMO process observed at three distinct municipalities, and their CPN versions obtained by applying the conversion process provided in Appendix B. The illustrated processes were obtained through interviews with different municipalities located in the Northern area of the Netherlands (van Beest et al., 2010, 2012). Municipalities interviewed ranged in size, population, income, and differed in being urban or rural.

Figure 4.3 depicts the simplest variant of the three WMO processes. The process starts with an application procedure which determines if the request made by the citizen falls under the WMO law. If this is not the case, the citizen is advised by the municipality employee towards his next steps. When the request made by the citizen does fall under the WMO law, the application is accepted, and a decision whether to approve the requested provision is made based upon the intake, possible requests of medical advice, and a possible home visit. The request is then either approved or rejected. In case of a positive decision, the requested provision is either arranged directly for the citizen, or a personal budget is assigned in case of personal care. In case of a negative decision, the citizen can appeal the decision. If the appeal is found to have merit, the decision is revised and the process is renewed. Figure 4.4 illustrates the same process as a CPN.

Figure 4.5 depicts a second variant of the WMO process. The main difference of this variant, with respect to the variant illustrated in Figure 4.3, is the option to approve the requested provision in part. When this option is taken, further medical advice may be acquired. When the medical advice declares that the citizen qualifies for the requested provision in full, the decision to approve in part is revised. Another difference is included after the reverse decision option in case of an appeal. Instead of renewing the process, this variant only revisits the approve or reject options. Finally, an activity is included to provide the citizen with information in case a personal budget is assigned. The CPN version of this variant is depicted in Figure 4.6.

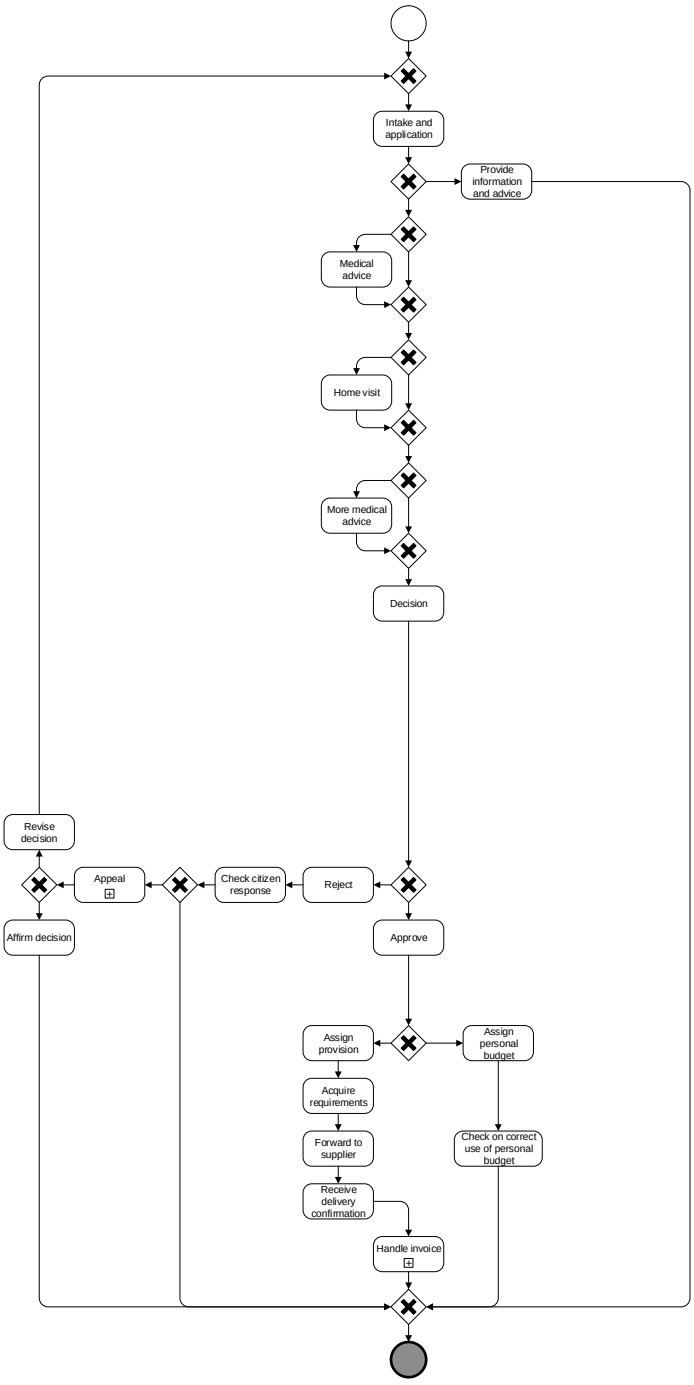


Figure 4.3: WMO Provision Request Process of Municipality A.

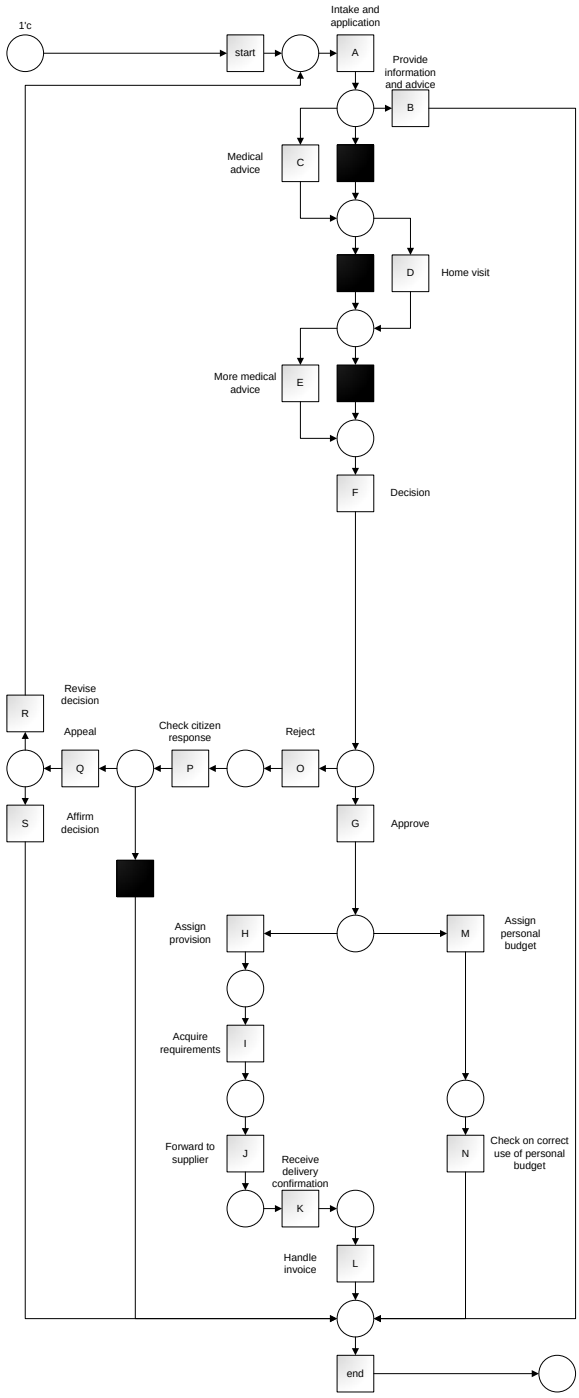


Figure 4.4: WMO Provision Request CPN of Municipality A.

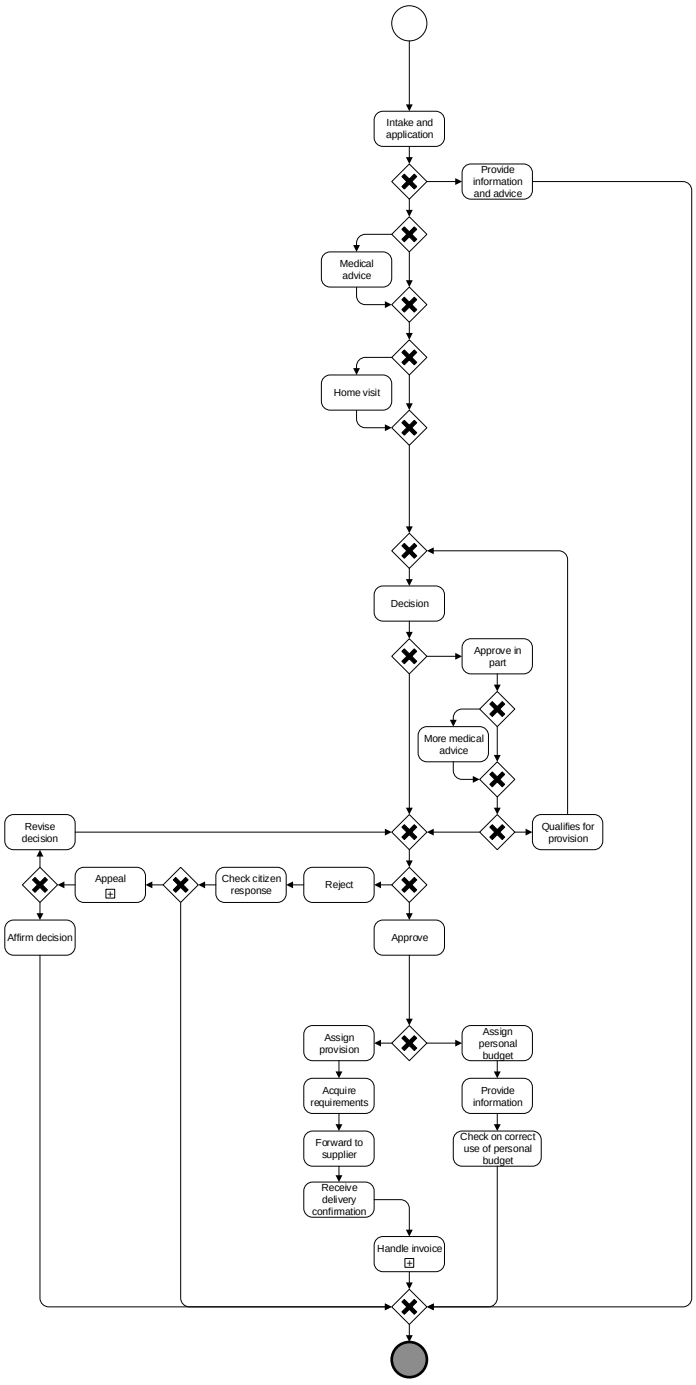


Figure 4.5: WMO Provision Request Process of Municipality B.

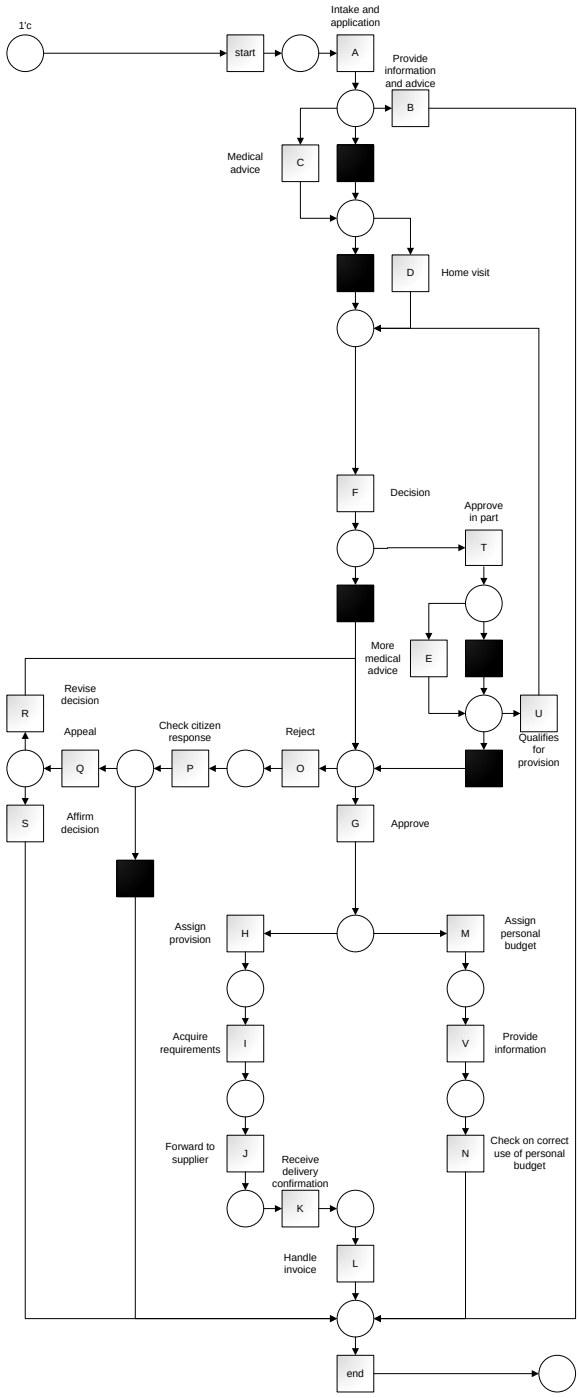


Figure 4.6: WMO Provision Request CPN of Municipality B.

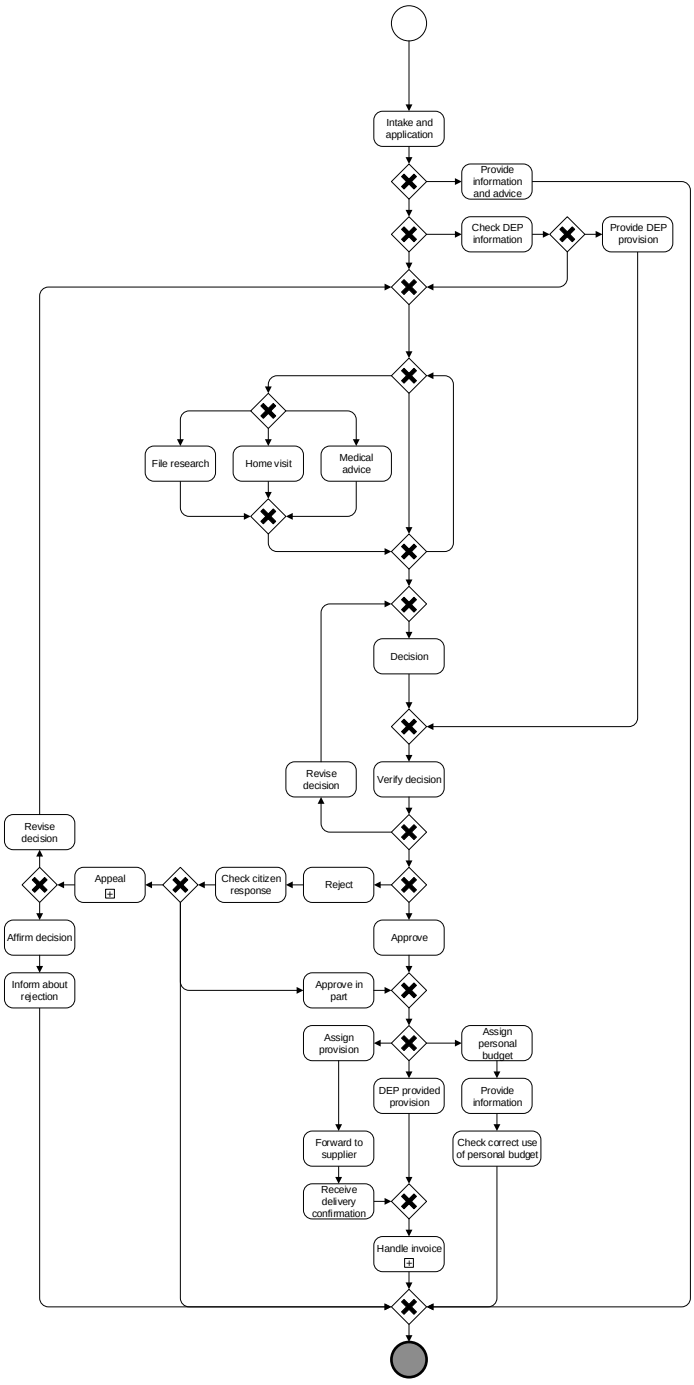


Figure 4.7: WMO Provision Request Process of Municipality C.

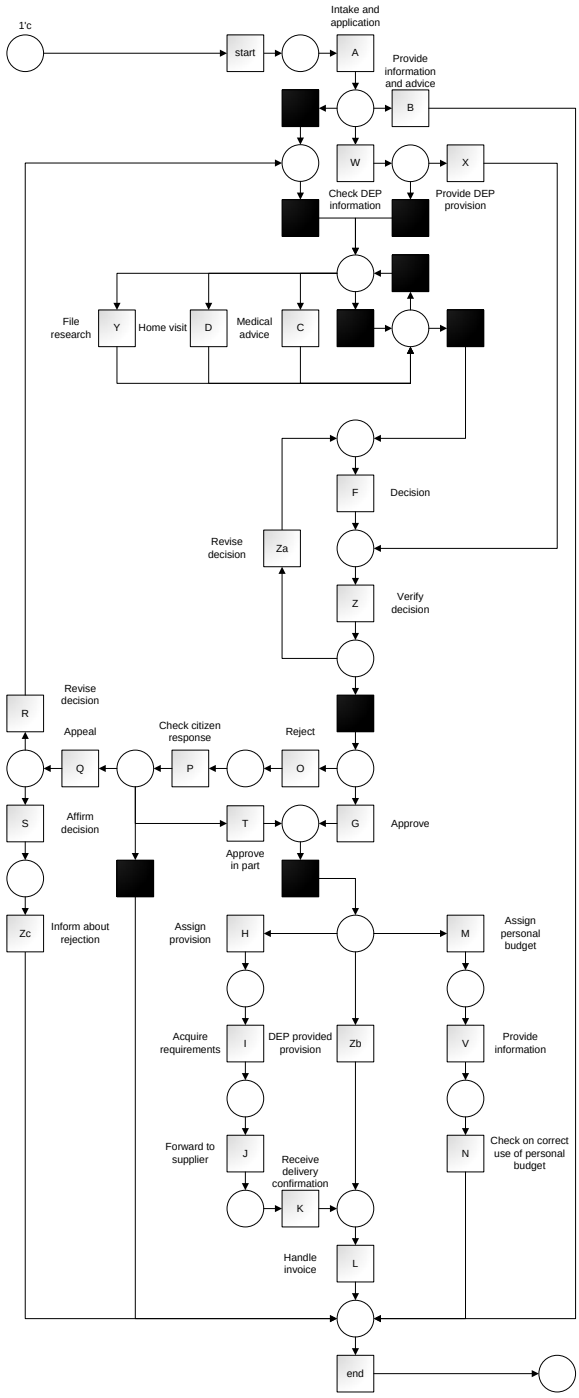


Figure 4.8: WMO Provision Request CPN of Municipality C.

A third variant of the WMO process is illustrated in Figure 4.7 and demonstrates a large amount of diversity. The CPN version of this variant is depicted in Figure 4.8. In this variant, a new branch is introduced for Directly Executable Provisions (DEP). In case of DEP, the requested provision is directly provided without taking the long decision making procedure. The information gathering process before making a decision also includes an option to do file research. In addition, each activity during this process can be performed an arbitrary number of times and in any order. All decisions in this variant are verified as well, and can be revised before the official approval or rejection. Similar to the second variant, an option to approve in part is included. In this case, however, the option is included after the rejection of the initially requested provision. Similar to the second variant, the citizen is also provided with information in case a personal budget is assigned. In addition, a rejection is explained after a failed appeal by a citizen. And finally, the activity of acquiring requirements of any assigned provision is handled by the supplier without involvement of the municipality.

When analyzing the three respective BP illustrated in Figures 4.3 through 4.8, we can clearly see the same generic process flow in all three versions of the WMO process. Beginning at the intake and application, the generic process flow continues through decision, approval, and finally the assignment of either the requested provision or a personal budget. Similar generic process flows can be seen at the alternate information and advice path and the reject/appeal path. Variations appear at the information gathering stage of the process, where the file research, home visit, and medical advice activities contribute to understanding the background of the citizens making the WMO provision request. Other variations appear through additional paths for approval in part, simple provision requests (i.e., DEP requests), and decision verification. And, finally, the simplest variations appear through the inclusion of additional informative activities in existing paths.

4.3 Case 3: Bouncer Registration

Collaborative BP (CBP) are BP where actors and roles are spread over multiple entities. For example, Figure 4.9 depicts a BPMN CBP featuring the national registration of bouncers originally published by Corradini et al. (2015). Bouncers are employed by nightclubs and other venues to control crowds.

The registration process features three roles: (A) the requester that applies for registration, (B) the prefecture handling the request, and (C) the police which perform a background check of the requester. Each role performs its assigned activities in order to arrive at the decision to authorize or inhibit the registration of the requester as a bouncer. They collaborate by each performing activities in separate BP, while communicating through messages depicted as message flows in the CBP of Figure 4.9. First, the requester makes a registration request. The prefecture receives the request, initiates the procedure, and proceeds to request a profile check from the police. The prefecture communicates the relevant laws with the requester, before proceeding to check the documents. If needed, the prefecture will request further documentation from the requester, before analyzing the documents. Meanwhile, the police will check for any impediment and report them to the prefecture. The prefecture then takes a decision based on the documents and police report, and either provides authorization or inhibition of registration of the requester as a bouncer. (Corradini et al., 2015)

Note that the original process published by Corradini et al. (2015) contained a cycle where the prefecture could repeatedly request further documentation from the requester. However, since the approach presented by Corradini et al. (2015) removes cycles by decoupling any backward looping flow during the Petri net conversion, we remove the loop in order to arrive at a fair comparison of the approaches. Using the conversion provided in Appendix B, the CBP of the bouncer registration is translated to a CPN. Figure 4.10 depicts the resulting CPN graphically.

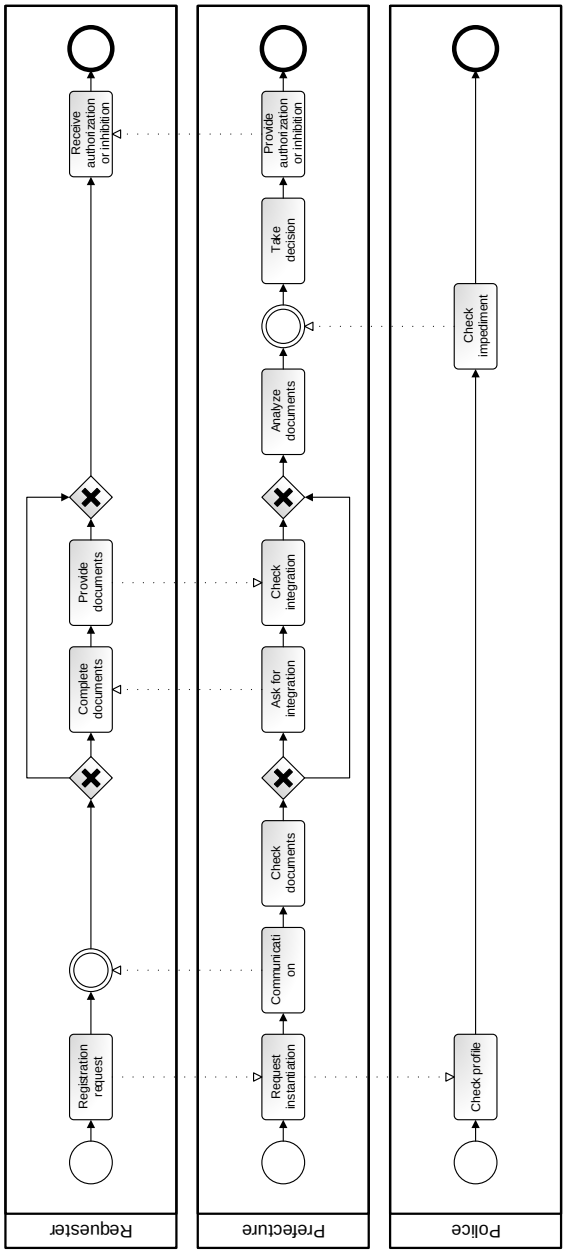


Figure 4.9: Bouncer Registration BPMN Model (Corradini et al., 2015).

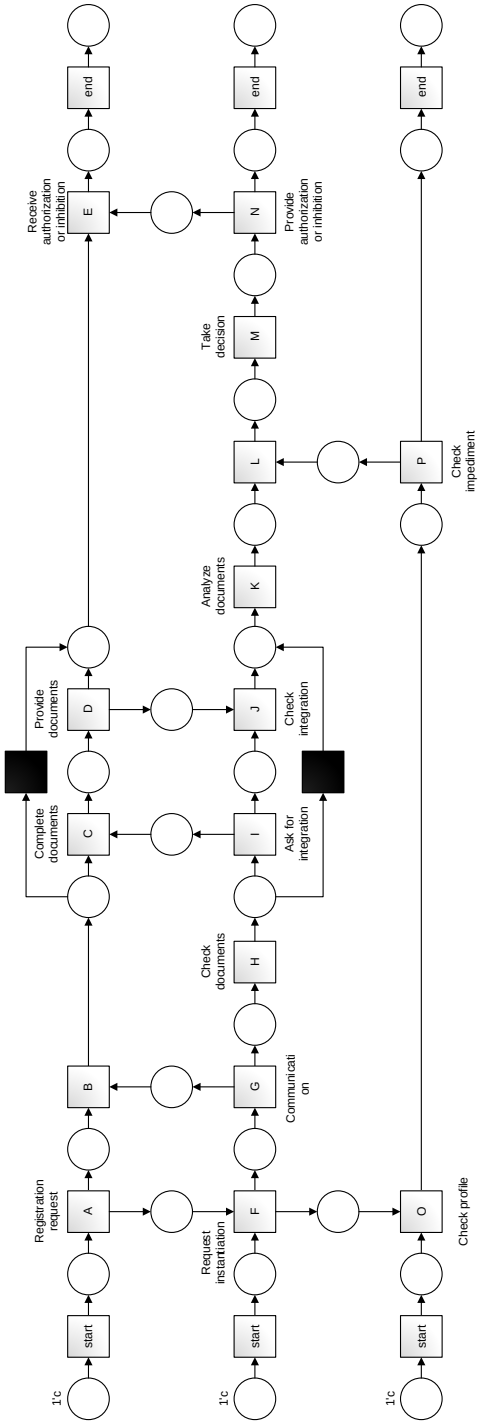


Figure 4.10: Bouncer Registration CPN.

4.4 Discussion

The presented case studies illustrate several scenarios found in real life. Although simplified for illustrative purposes, they each represent the general process flow found at business or government. We select these three cases because they each illustrate different sets of complications concerning design-time BP verification.

The first case, telecommunications customer support, represents a compliance case where a large amount of conditional paths motivate different outcomes. The second case, local Dutch e-government, represents a variability case where local needs drive very similar, yet individually tailored, decision making. Finally, the last case, bouncer registration, represents a highly concurrent case where multiple entities work in parallel to complete a single task.

Even though the cases illustrate different issues, they also demonstrate certain similarities. For example, each case describes BP with many different complex pathways, and consist mainly of the simplest constructs. Most complexity, therefore, can be traced to the many exclusive conditions contained within each BP. The many conditions, however, often prompt unstructured BP design (i.e., the use of multiple end events, incorrect forking and merging, etc).

Little concurrency is included when considering the individual business processes described by each case. Even when concurrency could be easily included by inclusive branching, the paths are described through intricate paths with exclusive choices within loops instead (e.g. the file research, home visit, and medical advice activities of Figure 4.7). Whether this is the result of inexperienced design or deliberate choice is unknown, but the fact remains that the process flow would have had increased efficiency when concurrency had been introduced.

Due to the simple constructs used to describe each BP, the formalization of the BP using CPN is straightforward. Even though several BP required a number of additional (silent) transitions to describe the control flow, their complexity is maintained at a minimum. More complex constructs can be described, but at the cost of increasing the complexity of the resulting CPN. And, in turn, increase the complexity of any subsequent verification.

CHAPTER 5

Verification Requirements

Premature optimization is the root of all evil in programming.

– Donald Knuth

As formal verification of BP at design-time includes several different goals, requirements vary greatly. For example, where some soundness verification frameworks focus solely on reachability of structured activities and process termination, other frameworks aim to verify the correctness of a specification. At the same time, some compliance verification frameworks aim to verify compliance conditions over the events of BP, whie others aim to verify conditions over the state of the BP. Additionally, with the convergence of BPM and service-orientation of recent years, requirements related to service-orientation have made their way into BP verification. With each of these aims, a different set of requirements is applicable on both the model and specifications used for verification. We describe the requirements based upon the verification of the process described by the BP itself and not an individual specification or implementation.

5.1 Model Requirements

Requirements on the model describe the various features which the verifiable model must be able to describe. Without support, BP which include these features can not be verified, or only be verified in part. BP offer a very powerful range of ex-

pressive features such as structured and unstructured cycles, exclusive, inclusive, and parallel branching, and different event handling mechanisms. Existing frameworks, however, often neglect to fully support many of the more powerful features possible within a model in order to simplify the task of verification. We assume support for the basic features and iterate through the problematic requirements.

Requirement 5.1.1 (Unsound Processes). *Verification frameworks must be able to verify business processes that are not sound.*

Existing verification frameworks often limit support to structured processes only. Structured processes are processes which, for example, only include matching gates with matching number of forking and joining branches. Although structured processes allow for easier verification and should always be used when possible, many in use processes are described in an unstructured manner. Verification frameworks must be able to support both structured and unstructured models.

Requirement 5.1.2 (Parallel Branching). *Verification frameworks must be able to verify parallel behavior within business processes, including inclusive branching.*

Verification frameworks must be able to express different branching constructs within its model in such a way that parallel behavior remains visible during verification. Although some frameworks do claim to support parallel behavior (Foster et al., 2003; Liu et al., 2007), the behavior is often described in such a way that it is not visible to the verification process.

Requirement 5.1.3 (Arbitrary Cycles). *Verification frameworks must be able to verify business processes that include arbitrary cycles.*

Verification frameworks must be able to express its specifications in the presence of arbitrary cycles without them severely affecting the outcome of those specifications due to the possibility of infinitely looping. For example, a scheduling specification stating a response from a structured activity may return true after evaluation of a path without an arbitrary cycle. However, after an arbitrary cycle is included within the same path between the effect and response structured activities, the same specification will always be evaluated as being false. This behavior is caused by the possibility of infinite looping within the arbitrary cycle. Indeed, when the arbitrary cycle is executed an infinite many times after the effect structured activity, the further ahead responding structured activity is never actually reached. A natural and correct conclusion when verifying software programs or electronic circuits – but, in the case of the verification of BP, this is highly unwanted behavior.

Unlike certain software processes or electronic circuits, BP are, at most, long living and will always terminate. Arbitrary cycles, therefore, will never execute infinitely many times. BP will either terminate correctly, or terminate with an error causing a possible rollback.

Requirement 5.1.4 (Intermediary Events). *Verification frameworks must be able to verify business processes that include intermediary events.*

Verification frameworks must be able to express inline events within its model. Events include simple message flows to error catching events with complex compensation handlers attached.

5.2 Specification Requirements

Formal verification of business process models is of interest to a number of different applications, including checking for basic process correctness, business compliance, and process variability. Basic process correctness, or soundness as it is sometimes called, aims at verifying the basic properties of BP, including reachability and absence of deadlocks. Compliance verification aims to prove whether a process complies with a set of business rules, laws, or regulations. Variability extends this notion by allowing parts of a BP to remain variable, or not fully defined, in order to support different versions. Different versions are then verified against its specification in order to verify whether the changed version remains within the context of its specification and business rules.

5.2.1 Soundness

The first goal of BP verification consists of verifying basic properties such as reachability and termination. Reachability of a business activity requires an execution path to exist leading to that activity starting from the initial activities. A termination property requires that all possible execution traces terminate. Business process soundness, a property originally proposed in the area of Petri Net verification, is known as the combination of these two properties plus a third: the absence of related running activities at process termination (i.e., proper completion) (van der Aalst, 1997). Avoiding the deployment of erroneous processes that do not conform with these properties is obviously advantageous: *"[erroneously] designed workflow models can result in failed workflow processes, execution errors, and disgruntled customers and employees"* (Bi and Zhao, 2004).

Requirement 5.2.1 (Reachability). *Every structured activity included within a BP must be reachable trough the execution of a sequence of structured activities.*

Requirement 5.2.2 (Termination). *All possible fair execution traces within a BP always eventually terminate (i.e., no deadlocks).*

Requirement 5.2.3 (Proper Completion). *Upon BP termination no structured activities within a BP remain in an executing state.*

5.2.2 Compliance

The second goal of BP verification consists of verifying the compliance of a BP against a set of norms, laws, and/or regulations. BP compliance entails the verification of (A) the occurrence of structured activities and their effects, (B) the order of execution of structured activities and their effects, (C) resource allocation (e.g. which roles and users perform which structured activities), and (D) time related constraints, while (E) under condition of possible data values. Design-time compliance verification aims at preemptive verification of BP, and therefore is limited to the verification of structured activity occurrence, order, and role allocation. Although time related conditions could be verified based upon estimations of time, true verification of compliance towards time related constraints can, naturally, only be accomplished by after the fact process mining or runtime monitoring techniques. Similarly, user information towards resource allocation verification is only available during or after BP enactment. At design-time, user information can be extracted from the different assigned roles at most.

Requirement 5.2.4 (Occurrence). Occurrence *specifications force the execution of structured activities within the BP. Compliance frameworks must be able to express the following occurrence specifications over elements of the BP:*

(i) Absence*: The structured activity, or its effect, does not occur.*

(ii) Universality*: The structured activity, or its effect, holds.*

(iii) Existence*: The structured activity occurs, or its effect holds, in the future.*

(iv) Bounded Existence*: The structured activity occurs, or its effect holds, a limited number of times.*

Requirement 5.2.5 (Ordering). Ordering *specifications force the order of execution of structured activities within the BP. Compliance frameworks must be able to express the following ordering specifications over elements of the BP:*

(i) Precedence*: The structured activity occurs, or its effect holds, before (the effect of) another structured activity.*

(ii) Response*: The structured activity occurs, or its effect holds, after (the effect of) another structured activity.*

The compliance requirements towards the occurrence and ordering of structured activities are adapted from the property specification patterns of finite state verification (Dwyer et al., 1999) and (Elgammal et al., 2010a, 2014). The patterns towards occurrence include absence, universality, existence, and bounded existence. These patterns describe that some state or event does not hold, holds throughout, hold eventually, or holds a specific number of times, respectively. The ordering specifications include precedence and response patterns. These patterns describe that some state or event occurs before or after another state or event. In the case of BP compliance verification, the evaluated state or event is represented by the structured activities of the BP (event) or their effects (state).

Occurrence and ordering specifications are applied using scopes. Scopes define regions of the system where a specification must hold, and include either globally, before (the effect of) another structured activity, after (the effect of) another structured activity, between (the effect of) two structured activities, or after and until (the effect of) two structured activities. In addition, the ordering specifications can be extended to support precedence and response chains. In this case, a number of ordered (effects of) structured activities is preceded by, or responding to, the triggering event or state.

Requirement 5.2.6 (Resource Specifications). Resource *specifications force the execution of structured activities within the process to be linked to a certain resource. Compliance frameworks must be able to express the following resource specifications over elements of the BP:*

(i) Always performed by role*: The structured activity is always performed by the given role.*

(ii) Performed by role*: The structured activity is (sometimes) performed by the given role.*

(iii) Never performed by role*: The structured activity is never performed by the given role.*

Resource specifications require structured activities to be performed by certain roles during enactment. In this way, structured activities can be required to be performed

by users with different tasks assigned to them. For example, an employee processes an order, while a manger performs a last check.

Requirement 5.2.7 (Data Conditions). *All specifications must be verifiable under* data conditions*. Compliance frameworks must be able to express the following data conditions over specifications:*

(i) Holds always*: The specification holds under all data values.*

(ii) Holds when*: The specification holds under a given set of data values (e.g.* $x > 10$*).*

Data Conditions allow specifications to be applied to only those execution paths under which these data conditions hold. Take, for example, a specification stating that if an order over a certain sum is placed, a downpayment is required. This specification can only be supported when data conditions are supported.

5.2.3 Variability

The third goal of BP verification, variability, builds upon compliance. In the context of BPM, variability indicates that parts of a business process remain variable, or not fully defined, in order to support different versions of the same process depending on the intended use or execution context (Aiello et al., 2010). Variability aims to support different versions of the same process. This includes support of process families at design-time, when a new process variant can be derived from a generic process, and process flexibility or adaptability at runtime, where a generic process can be adapted. BP variability can be specified in two different ways. The first, imperative variability, employs the use of *variation points* to provide different options at specific points in a process. The second, declarative variability, uses specifications like those of compliance to specify how each version of a process should behave. Explicit variability management consists in the ability to enumerate the possible variations. The *expressive power* requirements provide an indication of what must be possible to express for a variation in a process.

Imperative Variability

Imperative structural adaptation consists of atomic operations which, when executed in a specific predefined sequence, rearrange a BP to form a specific variant. Multiple predefined sequences of atomic operations may exist, which may or may not be compatible with each other, to form different variants. Imperative variability must support the following requirements to enable the full set of BP changes:

Requirement 5.2.8 (Atomic Structural Adaptation). *Imperative variability frameworks must be able to express the following self-explanatory atomic structural adaptions to the BP:*

(i) Insert process fragment.

(ii) Delete process fragment.

(iii) Move process fragment.

(iv) Replace process fragment.

(v) Swap process fragments.

(vi) Copy process fragment.

(vii) Extract sub-process.

(viii) Inline sub-process.

(ix) Embed process fragment in loop.

(x) Parallelize process fragments.

(xi) Embed process fragment in conditional branch.

(xii) Add control dependency.

(xiii) Remove control dependency.

(xiv) Update condition.

Extended from (Weber et al., 2008), atomic structural adaptions form the basis of any imperative variability framework. Each atomic structural adaptation enables essential operations which together can completely rearrange a BP into one of its variants. An imperative variability framework should be able to group sets of atomic structural adaptions on specific process fragments which, when executed in a specific order, allow a variation at a specific point within the BP. A number of selected variations then form a specific variant.

Requirement 5.2.9 (Atomic Resource Adaptation). *Imperative variability frameworks must be able to express the following self-explanatory atomic resource adaptions to the BP:*

(i) Assign process fragment to role.

(ii) Retract process fragment from role.

Where atomic structural adaptation allows modification of the execution paths of the BP, atomic resource adaptation allows an imperative variability framework to

change the role assigned to process fragments. Instead of allowing change to how things are done, atomic resource adaptation allows change to who executes the BP.

Requirement 5.2.10 (Variation Relation). *Variability frameworks must be able to express the dependencies between different imperative structural and/or resource changes.*

Multiple predefined sequences of atomic operations may exist to form different variations at specific points within the BP. Different variations, however, may or may not be compatible with each other, or require a specific order in which they must be applied. Dependencies such as these are called *variation relations* (Sinnema et al., 2006). Imperative variability frameworks must be able to specify variation relations in order to be able to support the full range of possible variants.

A variability management framework should allow the process designer to express the above imperative structural adaptation requirements.

Declarative Variability

Declarative specifications consists of rules expressing variations by declaring the borders which limit the possible process modifications. Unlike atomic structural changes which indicate imperatively what can vary, a declarative specification limits the borders of changes explicitly. Initially, all possible modifications are allowed within the BP. As specifications are included, modification is being limited. The more specifications are included, the more the BP is being defined, and the less modification is allowed. Declarative variability frameworks must support the following requirements:

Requirement 5.2.11 (Inclusion Specifications). *Declarative variability frameworks must be able to express the following declarative inclusion specifications over elements of the BP:*

(i) Include*: The structured activity must be included in the variant.*

(ii) Prerequisite*: The structured activity must be included in the variant if some other structured activity is included.*

(iii) Substitution*: The structured activity must be included in the variant if some other structured activity is not included.*

(iv) Corequisite*: The structured activity must be included in the variant if an other structured activity is included as well, and vice versa.*

(v) Causal selection*: The structured activity may only be included in the variant if some other structured activity is included.*

Extended from (Sadiq et al., 2005) and (Lu et al., 2009), *inclusion* specifications force the inclusion of structured activities within the variant. That is, under a certain condition, they require a structured activity to be present in some execution path of the process. Inclusion specifications are of particular importance to variability. Instead of enforcing the execution of a structured activity, as is often the case with compliance verification, inclusion specifications only enforce the inclusion of structured activities without requiring their execution. For example, the substitution specification (Requirement 5.2.11.iii) forces the inclusion of a structured activity within the variant if some other structured activity is not included. In this way, at least one of the structured activities is included.

Requirement 5.2.12 (Exclusion Specifications). *Declarative variability frameworks must be able to express the following declarative exclusion specifications over elements of the BP:*

(i) Exclude*: The structured activity may not be included within the variant.*

(ii) Exclusion*: The structured activity may not be included in the variant if some other structured activity is included.*

(iii) Admittance*: The structured activity may not be included in the variant if some other structured activity is not included.*

(iv) Exclusive choice*: The structured activity may be included in the variant if some other structured activity is not included, and vice versa.*

Extended from (Dwyer et al., 1999; Sadiq et al., 2005; Lu et al., 2009) and (Elgammal et al., 2010a, 2014), *exclusion* specifications force the exclusion of structured activities within the variant. That is, under a certain condition, they require a structured activity to be absent from all paths of the process. For example, Requirement 5.2.12.ii species that a structured activity may not be included in the variant if another is included. In other words, the one structured activity excludes the other.

Requirement 5.2.13 (Execution Specifications). Execution *specifications force the execution of structured activities within the variant. Declarative variability frameworks must be able to express the following declarative execution specifications over elements of the BP:*

(i) Execute*: The structured activity must be included in all execution paths of the variant.*

(ii) Requirement*: The structured activity must be included in all execution paths of the variant when an other structured activity is included.*

(iii) Replacement*: The structured activity must be included in all execution paths of the variant if some other structured activity is not included.*

(iv) Backup*: The structured activity must be included in all execution paths of the variant if some other structured activity is not included in all execution paths.*

(v) Causal execution*: The structured activity must be included in all execution paths of the variant when an other structured activity is included, or not included at all.*

Extended from (Dwyer et al., 1999) and (Elgammal et al., 2010a, 2014), *Execution* specifications pose conditions on the presence of structured activities in all execution paths of the variant. That is, they force the execution of the structured activity during process enactment. For example, the backup specification (Requirement 5.2.13.iv) requires a structured activity to be always executed if an other structured activity is not always executed. In this way, the specification enforces some form of safety as one of the structured activities is guaranteed to be performed.

Requirement 5.2.14 (Option Specifications). Option *specifications force alternate paths around structured activities within the variant. Declarative variability frameworks must be able to express the following declarative specifications over options in the BP:*

(i) Option*: The structured activity may not be included in all execution paths of the variant.*

(ii) Avoidance*: The structured activity may not be included in all execution paths of the variant when an other structured activity is included.*

Option specifications force alternate execution paths around structured activities. For example, Requirement 5.2.14.i forces that a structured activity is not always executed. That is, the execution of the structured activity is optional.

Requirement 5.2.15 (Scheduling Specifications). Scheduling *specifications force a temporal relation among the execution of structured activities within the process. Declarative variability frameworks must be able to express the following declarative scheduling specifications over elements of the BP:*

(i) Response*: The structured activity is eventually followed by an other structured activity in every execution path.*

(ii) Exists response*: The structured activity is eventually followed by an other structured activity in some execution path.*

(iii) Immediate response*: The structured activity is immediately followed by an other structured activity in every execution path.*

(iv) Exists immediate response*: The structured activity is immediately followed by an other structured activity in some execution path.*

(v) No response*: The structured activity is not followed by an other structured activity in every execution path.*

(vi) Exists no response*: The structured activity is not followed by an other structured activity in some execution path.*

(vii) No immediate response*: The structured activity is not immediately followed by an other structured activity in every execution path.*

(viii) Exists no immediate response*: The structured activity is not immediately followed by an other structured activity in some execution path.*

(ix) Coexecution*: The structured activity is eventually followed by an other structured activity in every execution path, or vise versa.*

(x) Cooccurrence*: The structured activity is eventually followed by an other structured activity in some execution path, or vise versa.*

(xi) Parallel execution*: The structured activities are executed in parallel.*

(xii) Exclusive execution*: The structured activities are never executed both in every execution path.*

Extended from (Dwyer et al., 1999; Pesic and van der Aalst, 2006) and (Elgammal et al., 2010a, 2014), *scheduling* specifications enforce a specific ordering between structured activities. Although many of the listed scheduling specifications are similar to compliance or enactment specifications and enforce the order of execution in *every* execution path, variability allows the same to be specified over *some* execution paths. Furthermore, variability can require structured activities to both be included, but be performed in exclusive paths, or be performed in parallel (Requirements 5.2.15.xi and 5.2.15.xii).

A variability management framework should allow the process designer to express the above declarative specifications.

5.3 Evolutionary Requirements

From the compliance and variability management point of view, a set of changes made to rules is an *evolution* if such changes are permanent. This translates into the following requirements.

Requirement 5.3.1 (Business Process Design). *BP must always be designed using the latest set of rules.*

Updates to rules or reference processes may occur randomly over long periods of time, or in high sequence. With every update, variants should always be based upon the latest rules.

Requirement 5.3.2 (Business Process Evolution). *BP must be redesigned when they are non-compliant with the latest set of rules.*

BP must be updated when changes to the reference process or rules have been made. For example, consider a change made to a reference process where an option must be formally considered for each case by law. This change must then be propagated to all variants in such a way that this structured activity is enforced and must be included.

Requirement 5.3.3 (Evolution of Process Instances). *Running process instances must be redesigned when they are non-compliant with the latest set of rules.*

Running instances should be updated when changes to the reference process or rules (for example through law) have been made. All changes to the reference process or rules should be propagated to all running instances of the BP. Consider again the case where a structured activity must be formally considered for each case. Now also assume that there exists a running instance of this variant. Such running instances should also be adopted in such a way that it includes the structured activity if possible.

Requirement 5.3.4 (Soundness). *Verification frameworks must guarantee continued soundness of a BP from one version to another.*

Process soundness enforces the correctness of a BP. Process soundness includes reachability, termination, and proper completion requirements (i.e. no structured activities within the BP remain in an executing state upon termination). Naturally, a BP must remain sound after process redesign.

Requirement 5.3.5 (Compliance). *Verification frameworks must guarantee continued compliance of a BP after evolution.*

Process compliance is the requirement of a BP to comply to a set of norms, laws, and/or regulations. Variants must remain compliant even after redesign. Verification frameworks, therefore, must incorporate mechanisms to ensure compliance of BP before deployment.

5.4 Discussion

When evaluating the requirements for BP against the state of the art described in Chapter 3, we immediately notice the lack of model support. Where the specification requirements are often extensively compiled, requirements regarding the model – upon which the specifications are to be interpreted – are often neglected or even omitted. The effect is twofold.

First, specification support is often implied but upon further review not actually supported consistently throughout the model. One large offender is the proposed use of the next operator, X, for the verification of an immediate response. The issue originates from the often discarded notion of stuttering. Stuttering, within models, occurs when two or more systems are modeled concurrently. A single step in the state of one of the systems may take multiple steps in the concurrent model as the other systems may take several steps in the interim. The steps that one system makes, in the concurrent model, are therefore being stuttered as the other concurrently executing systems are allowed to take steps in between. Normally, this issue is avoided when modeling concurrent systems by simply seeing the next operator, X, as a *global* next, i.e., a next step between all concurrently executing systems. And, when there is no concurrency, it can simply be used as the normal *local* next, within the one system. Business processes, however, can inherently be both non-concurrent and concurrent within a single model due to the multiple parallel branching mechanisms available to the modeler. As a result, the next operator, X, returns inconsistent results between concurrent and non-concurrent regions of a BP. Returning global results in the first, and seemingly local results in the other, often causing discrepancies within the interpretation of the results. Other offenders include reductions on the model which cause lack of support of multiple operators, or combinations of operators, without formally addressing the implications on the specifications and the use of these operators.

Second, requirements on the model are easily neglected in favor of simplistic, and often faulty, model reductions. For example, the stuttering issue with the next operator is often solved by not allowing parallel branching, or by simply interpreting parallel branching as exclusive branching. Although this solves the issue of the state explosion caused by interleaving concurrent branches, it can hardly claim to support fully the class of models described by BP standards. And, in the second case, may even return false results as only one of the parallel branches counts as being executed. Another common example entails the decoupling, or unwinding, of arbitrary cycles while claiming full loop support. Naturally, any point of decou-

pling, even after unwinding the cycle a set number of times, causes incorrect or inconsistent returns of formal specifications.

When comparing the requirements for declarative specifications to the state of the art, we immediately notice that execution and non-exist scheduling specifications are favored over all others. This preference can be traced back to the verification of linear execution traces for either compliance verification or the specification of highly flexible BP. For design-time verification of variability, however, additional specifications are required to describe behavior over and between single paths.

Next, we define models and specifications solving many of these issues in a formal and correct manner.

CHAPTER 6

Business Process Verification

Controlling complexity is the essence of computer programming.

– Brian Kernighan

Formal verification entails the evaluation of a specification on a model, e.g., the evaluation of a temporal logic formula on a transition system such as a Kripke structure (Definition 2.3.1). In order to obtain a Kripke structure from a formalized BP, in the form of a CPN (Definition 2.2.4), the CPN is processed in a way similar to that of the reachability graph (Definition 2.2.3). However, where the reachability graph fails to maintain essential BP information regarding parallel and forking behavior, this model must maintain all BP information while keeping the required state space to a minimum (Groefsema and van Beest, 2015; Groefsema et al., 2016).

6.1 Verifiable Model

When describing business processes, transitions in CPN relate directly to activities. A common technique for converting CPN into transition systems entails the inclusion of transitions as states in the transition system upon their occurrence. While traversing the CPN from its initial marking M_0, while they occur, transitions are continuously added as states. A major drawback of this technique occurs when transitions are encountered multiple times during, for example, the interleaving of parallel paths. In such cases, the approach causes the inclusion of multiple copies

of the same state. Due to this, an enormous amount of duplicate states is created. Instead, we define a verifiable model which only includes states for each marking and each set of binding elements that are not just enabled, but will occur at that marking. This set of parallel enabled binding elements at a marking is formalized in Definition 6.1.1.

Definition 6.1.1 (Parallel Enabled Binding Elements)**.** *The sets of all possible parallel enabled binding elements $Y_{par}(M)$ at a marking M is obtained through the following steps:*

- $Y(M) = \{(t,b) \mid \forall p \in P : E_f(p,t)\langle b\rangle \leqslant M(p)\}$ *is the set of binding elements enabled at a marking M.*
- $Y_{sim}(M) = \{Y \mid Y \in \mathcal{P}(Y(M)) \wedge \forall p \in P : \sum_{(t,b)\in Y} E_f(p,t)\langle b\rangle \leqslant M(p)\}$ *are the sets of possible binding elements Y that may occur simultaneous at marking M, with $\mathcal{P}(Y(M))$ being the power set of enabled binding elements.*
- $Y_{par}(M) = \{Y \mid Y \in Y_{sim}(M) \wedge \forall Y' \in Y_{sim}(M) : Y \not\subset Y' \wedge Y \neq \emptyset\}$ *are the largest sets of parallel enabled binding elements Y obtained from $Y_{sim}(M)$.*

In order to obtain a verifiable system model from the markings of a CPN, we first specify what the places containing tokens in a marking represent. Places containing tokens at a marking M allow binding elements (t,b) to be enabled. A binding element is enabled iff $\forall p \in P : E_f(p,t)\langle b\rangle \leqslant M(p)$. Then, a multiset Y of binding elements (t,b), or a step, is enabled iff $\forall p \in P : \sum_{(t,b)\in Y} E_f(p,t)\langle b\rangle \leqslant M(p)$, or if the sum of the binding elements is enabled. The occurrence of a step Y at a marking M_i produces a new marking M_j as denoted by $M_i \xrightarrow{Y} M_j$. The enabled powerset $Y_{sim}(M)$ of the enabled binding elements represents the set of possible parallel occurrences. Then, by taking those sets which are no subset of other sets of the enabled powerset, we find the different sets $Y_{par}(M)$ of binding elements that may occur at that marking. Once a binding element of such a set occurs, the other elements of that set will occur in the future. This set, $Y_{par}(M)$, is used in upcoming definitions to determine the different labelings when multiple sets of binding occurrences can occur simultaneously at a marking while advancing between states.

Using these conventions, we convert a colored Petri net *CPN* into a Kripke structure K by creating states at each marking M_i for each set of binding elements that can occur concurrently at a marking M_i, and then having each binding element occur individually to find possible next states. Although binding elements could occur simultaneous, allowing these would only provide for additional relations, creating shorter paths between existing states when interleaving. Even though CPN could theoretically reach an infinite number of markings, the use of the sound and safe

workflow patterns restrict the CPN in such a way that it always produces a number of markings that is finite. The verifiable system model of a business process model, called the transition graph, is formalized in Definition 6.1.2.

Definition 6.1.2 (Transition Graph)**.** *Let AP be a set of atomic propositions. The transition graph of a CPN with markings $M_0, ..., M_n$ is a Kripke structure $K = (S, S_0, R, L)$ over AP, with:*

- $AP = \{M_0, ..., M_n\} \cup \{(t, b) \in Y \mid Y \in Y_{par}(M_0) \cup ... \cup Y_{par}(M_n)\}$,
- $S = \{s_i^Y \mid Y \in Y_{par}(M_i) : 0 \leqslant i \leqslant n\}$,
- $S_0 = \{s_0^Y \mid Y \in Y_{par}(M_0)\}$,
- $L(s_i^Y) = \{M_i\} \cup Y$,
- $R = \{(s_i, s_j) \mid (t, b) \in L(s_i) \wedge M_i \in L(s_i) \wedge M_j \in L(s_j) \wedge M_i \xrightarrow{(t,b)} M_j\}$. [1]

Definition 6.1.2 introduces a novel conversion from the marking of the CPN to a model including the occurrence of binding elements. That is, the model is developed from both state and event (i.e. occurrence) information. When a state is labeled with a binding element, it can be interpreted as that binding element currently occurring. Binding elements, however, can be found as occurring over multiple states. A binding element has only occurred (i.e. finished occurring) when it is occurring at one state and not occurring at the next state. Binding elements occur concurrently during interleaving of parallel branches. In such cases, states are labeled with multiple binding elements. Although the transition graph is a graph containing states with labels over the markings $M_0, ..., M_n$ and steps (t, b), only the steps (t, b) are used as verification propositions. The reason is twofold, first, it allows for greater model reduction later on, and second, the verification purposes listed in Chapter 5 only require (t, b). When b is understood, we simply write t (such as with business process models translated from workflow patterns where only one functional binding $b = \langle c \rangle$ is used).

Figure 6.1 depicts the transition graph resulting from this conversion process on the CPN of the customer support process presented in Section 4.1. Although bindings are abstracted away in the examples to increase readability, they formally do exist in the models. As such, any information included within the model within complex

[1] Although Definition 6.1.2 uses elements from the definition itself to define R (i.e. the labeling function L), this is merely done to produce a more concise and readable definition.

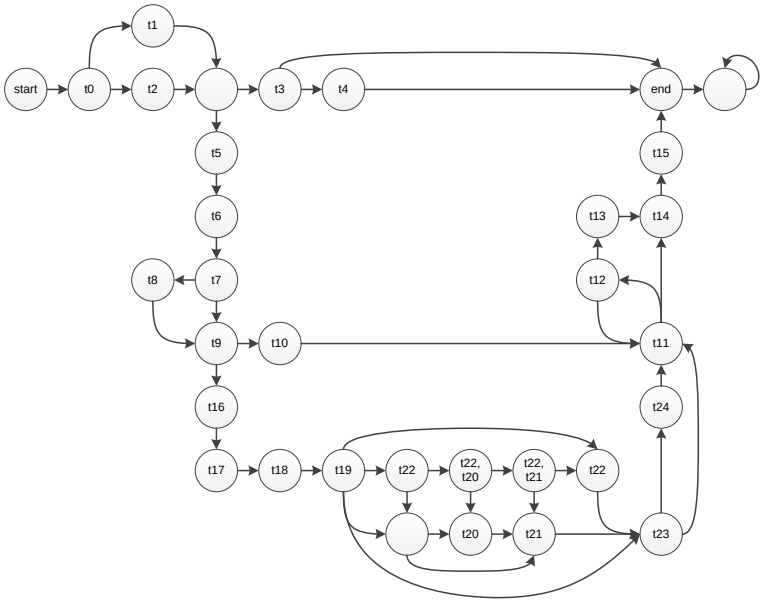


Figure 6.1: The customer support process from Section 4.1 as a Kripke structure.

bindings, such as data, is maintained within the transition graph (though, possibly, at the cost of model size and performance).

Even though states are labeled with markings $M_0, ..., M_n$, these should not be used as propositions when verifying by means of the transition graph. The markings are only included in the transition graph to obtain a correct model (i.e. to detect the difference between a marking where a step (t, b) is enabled without additional tokens at places and a similar marking with additional tokens unrelated to (t, b)). When verifying over markings, using the well-known reachability graph is preferred. The reachability graph can equally be obtained from the transition graph.

Definition 6.1.3 (Reachability Graph of a Transition Graph)**.** *Let AP be a set of atomic propositions. The reachability graph of the Transition Graph $K = (S, S_0, R, L)$ over AP is a rooted directed graph $G = (V, E, v_0)$, with:*

- $V = \{M_i \mid s_i \in S : M_i \in L(s_i)\}$ *is the set of vertices,*
- $v_0 = M_0 \mid s_0 \in S_0 : M_0 \in L(s_0)$ *is the root node,*
- $E = \{(M_i, (t, b), M_j) \mid (s_i, s_j) \in R \wedge M_i \in L(s_i) \wedge M_j \in L(s_j) \wedge (t, b) = L(s_i) \setminus L(s_j) \setminus M_i\}$ *is the set of edges.*

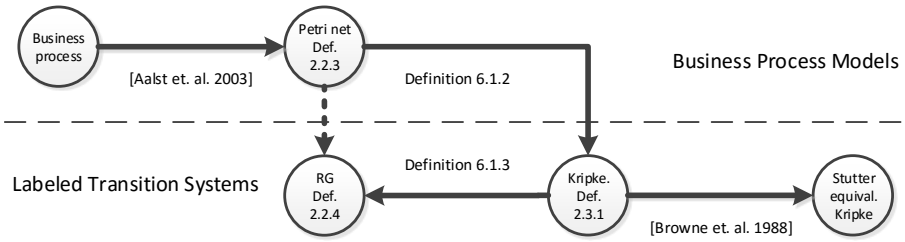


Figure 6.2: Model Conversion.

Definition 6.1.3 completes the cycle of model conversions as depicted in Figure 6.2. Together with earlier definitions, Definition 6.1.3 allows a CPN to be transformed into a transition graph, which then can be transformed into a reachability graph.

6.2 Specification Semantics

The semantics of CTL can be defined upon the possible executions of a CPN. An occurrence path of a CPN is a sequence of sets of enabled transitions that can occur concurrently $\pi = y_1, y_2, ...$ with $y_i \in Y_{par}(M_i)$ and $M_i \xrightarrow{(t_i,b_i)\in y_i} M_{i+1}$ for $i > 0$. Here, $y_i \in Y_{par}(M_i)$ (Definition 6.1.1) are versions of the marking M_i where different sets of binding elements are enabled (i.e. those that can occur simultaneously). A binding element (t, b) is occurring at y_i iff $(t, b) \in y_i$. The semantics of CTL on the possible executions of a colored Petri net is defined using the minimal set of CTL operators $\{\neg, \vee, EX, EG, EU\}$:

Definition 6.2.1 (CTL semantics on Reachability Graphs)**.**
$G, y_i \models \phi$ means that the formula ϕ holds at $y_i \in Y_{par}(M_i)$ of marking M_i of the reachability graph G. When the model G is understood, $y_i \models \phi$ is written instead. The steps (t, b) form the propositions of the language of CTL. When b is understood, t is written only. The relation $\models$ is defined inductively as follows:

$$
\begin{array}{lll}
y_i \models (t,b) & \textit{iff} & (t,b) \in y_i \\
y_i \models \neg\phi & \textit{iff} & y_i \not\models \phi \\
y_i \models \phi \vee \phi' & \textit{iff} & y_i \models \phi \textit{ or } y_i \models \phi' \\
y_i \models EX\ \phi & \textit{iff} & \exists \pi = y_i, y_{i+1}, ... \mid y_{i+1} \models \phi \\
y_i \models EG\ \phi & \textit{iff} & \exists \pi = y_i, y_{i+1}, y_{i+2}, ... \mid \forall n : (n \geqslant 0 \wedge y_{i+n} \models \phi) \\
y_i \models E[\phi\ U\ \phi'] & \textit{iff} & \exists \pi = y_i, y_{i+1}, y_{i+2}, ... \mid \\
 & & \exists m : (m \geqslant 0 \wedge y_{i+m} \models \phi' \wedge \forall n : (0 \leqslant n < m : y_{i+n} \models \phi))
\end{array}
$$

6.3 Specification Interpretation

Let Φ be a set of formulas. $G, y_i \models \Phi$ iff for each $\phi \in \Phi$ it holds that $G, y_i \models \phi$. Then, a set of formulas Φ is consistent if $\Phi \not\vdash \bot$.

Lemma 6.3.1 (Lindenbaum's Lemma)**.** *For each consistent set Φ, there is a maximally consistent set Φ' such that $\Phi \subseteq \Phi'$. In other words, every consistent set Φ can be extended to a maximally consistent set.*

Lemma 6.3.2 (Truth Lemma)**.** *For any CTL formula ϕ: $G, y_i \models \phi$ iff $\phi \in y_i$.*

Proof. The proof is by structural induction on ϕ. If ϕ consists of a propositional letter (t, b) then by Definition 6.2.1. If ϕ is in the form $\phi_1 \vee \phi_2$ then by Definition 6.2.1 $y_i \models \phi_1$ or $y_i \models \phi_2$, and $\phi_1 \in y_i$ or $\phi_2 \in y_i$. If ϕ is of the form $\neg\phi$, then $y_i \not\models \phi$, and $\phi \notin y_i$. If ϕ is of the form $EX\phi$, then $\exists \pi = y_i, y_{i+1}, ... \mid y_{i+1} \models \phi$, and thus $\phi \in y_{i+1}$. If ϕ is of the form $EG\phi$, then $\exists \pi = y_i, y_{i+1}, y_{i+2}... \mid \forall n : (n \geqslant 0 \wedge y_{i+n} \models \phi)$, and thus $\forall n : (n \geqslant 0 \wedge \phi \in y_{i+n})$. The case where ϕ is of the form $E[\phi_1 \; U \; \phi_2]$ follows similarly. □

Theorem 6.3.1. *Every maximally consistent set Φ has a model, i.e., there is a model G and state y_i such that for all $\phi \in \Phi$, $G, y_i \models \phi$.*

Proof. Suppose that Φ is a consistent set. By the Lindenbaum's Lemma, there is a maximally consistent set Φ' such that $\Phi \subseteq \Phi'$. Then, by the Truth Lemma, for each $\phi \in \Phi'$, we have $G, \Phi' \models \phi$. Then, every formula in Φ is true at Φ' in the graph. □

Theorem 6.3.2. *If $\Phi \models \phi$ then $\Phi \vdash \phi$.*

Proof. Ad absurdum, suppose that $\Phi \not\vdash \phi$. Then, $\Phi \cup \{\neg\phi\}$ is consistent. By the above theorem, there is a model of $\Phi \cup \{\neg\phi\}$. Hence, $\Phi \not\models \phi$ □

Verification of a formula ϕ on the possible executions of a CPN proves that ϕ does or does not hold at a certain point of its execution. More specifically, a formula ϕ may or may not hold at a version $y_i \in Y_{par}(M_i)$ of marking M_i (Definition 6.1.1). When a step (t, b) holds at y_i, that step is occurring. When a formula ϕ holds at all versions $y_i \in Y_{par}(M_i)$ of a marking M_i, it can be written that $M_i \models \phi$.

Using the definitions above, CTL specifications can be used to verify BP. BP can be translated into CPN, which in turn can be simulated to obtain a transition graph

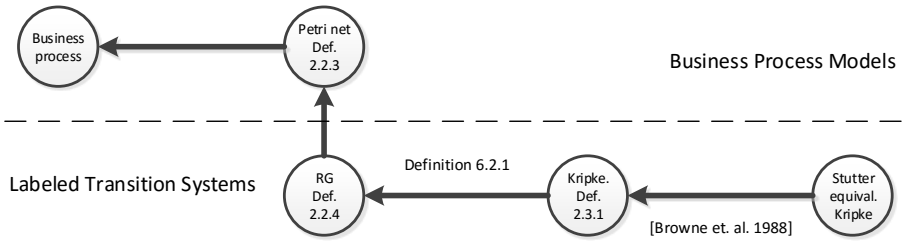


Figure 6.3: Specification Interpretation.

upon which the branching time temporal logic CTL can be interpreted (Figure 6.3). This interpretation can then be understood upon the possible executions of the CPN as expressed by its reachability graph – and, in turn, upon the CPN and BP.

6.4 Verification over Groups and Roles

BP modeling standards feature several techniques to bind activities to roles which must perform those activities. For example, BPMN BPD's offers grouplike structures known as pools and swimlanes. Pools represent major entities within the BPD, such as organizations, while swimlanes inside those pools represent roles or functions within that organization. When modeling the BP, the normal process flow is included within the pools and lanes of the BPD. Activities within swimlanes of pools are then assigned to the role within the organization that is assigned to that swimlane. To verify over pools, lanes, and other groups, we propose an additional layer of labels within the transition graph.

Definition 6.4.1 (Labeled Colored Petri Net)**.** *A labeled colored Petri net is a quintuple* $LN = (N, G_l, R_l, L_g, L_r)$*, where:*

- $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ *is a colored Petri net (Definition 2.2.4),*
- G_l *is a set of group labels,*
- R_l *is a set of role labels,*
- $L_g : T \rightarrow G_l^2$ *is a labeling function that labels transitions with the labels of the groups it belongs to,*
- $L_r : T \rightarrow R_l^2$ *is a labeling function that labels transitions with the labels of the roles which are assigned to the transition.*

While BPD activities directly correspond to transitions of a CPN, groups within BPD do not correspond to any such an element. Instead, the transitions corresponding to the grouped elements within the BPD receive additional labels which mark them as belonging to those groups, e.g., if transition t is part of a pool with label o_0 and swimlame r_0, then $L_r(t) = \{o_0, r_0\}$.

Definition 6.4.2 (Labeled Transition Graph)**.** *Let $LN = (N, G_l, R_l, L_g, L_r)$ be a labeled colored Petri net, AP a set of atomic propositions, and $K = (S, S_0, R, L)$ the transition graph of N over AP. Then, K is a labeled transition graph, iff:*

- *$G_l \subseteq AP$ the group labels G_l are included in AP*
- *$\forall r \in R_l, \forall t \in T : (t, r) \in AP$ the role labels are included in AP,*
- *$\forall s_i \in S, \forall (t, b) \in L(s_i) : L_g(t) \subseteq L(s_i)$,*
- *$\forall s_i \in S, \forall (t, b) \in L(s_i), \forall r \in L_r(t) : (t, r) \in L(s_i)$.*

The labeled transition graph includes group labels $L_g(t)$ at each state of the transition graph where the transition t is labeled using binding elements (t, b). In other words, if a transition belonging to one or more groups is occurring at a state within the transition graph, then that state is also labeled with the group labels to which that transition belongs. Specifications can then be interpreted over those group labels. For example, the CTL specification $AG(g_1 \Rightarrow AFg_2)$ can be used to verify if the elements within group g_1 are always followed by any element of group g_2.

Similarly, role labels are included as atomic propositions for any possible pair of role and transition. All possible pairs are included to support the verification of transitions not being performed by roles. States labeled with binding element (t, b), of which t is performed by r are labeled with (t, r). For example, the CTL specification $AG((t, b) \Rightarrow (t, r))$ can be used to verify if transition t is always performed by role r (i.e., t falls in swimlane r).

For the sake of simplicity, we shall refrain from referring to the labeled versions of the colored Petri net or transition graph in the remainder of the document. Instead, when verifying over roles or otherwise grouped activities, we assume the labeled versions of these graphs are in effect.

6.5 Verification over Conditions

Although the transition graph is an extremely powerful model when used to interpret CTL specifications over possible execution paths, it lacks support for specifi-

cation interpretation over conditions on conditional branches. The markings of a CPN, $M_0, ..., M_n$, include every possible marking after the occurrence of a binding element (t, b), even if the occurrence of (t, b) can lead to different markings due to conditions on its in- and outgoing arcs. As the transition graph is directly based on the set of markings $M_0, ..., M_n$, it includes any possible state caused by conditions on the CPN its arcs. Although, at first sight, one could use a form of labeling on the states of the transition graph to represent conditions as they hold throughout paths of the CPN, this again would cause uncertainties as paths merge – or, otherwise, cause further state explosion. Instead, we propose verification of conditional specifications on transition graphs with related conditions.

Definition 6.5.1 (Conditional CTL)**.** *Let ϕ be a CTL specification and c a Boolean condition, then $[c]\phi$ is a CTL specification which must hold only under condition of c holding.*

A conditional CTL specification $[c]\phi$ is a CTL formula which must only hold under condition of c, that is, if c holds, then ϕ must hold. For $[c]\phi$, we read *when c, then also ϕ*. When $[true]\phi$, we simply write ϕ. For example, the customer complaint process (Figure 4.2) could feature a specification which requires that when contact is made, the complaint is always recorded, i.e., the conditional CTL specification $[contact]AFt6$.

Definition 6.5.2 (Conditional Transition Graph)**.** *Let K be a transition graph of a CPN with markings $M_0, ..., M_n$, and c a Boolean condition, then $K[c]$ is the transition graph which excludes those markings M_i, as states in S, and steps $M_i \xrightarrow{(t,b)} M_j$. as relations in R, which are not reachable under condition c.*

A conditional transition graph $K[c|$ is a transition graph which is limited with respect to those executions of the CPN for which the condition c holds. Then, conditional specifications $[c]\phi$ must be evaluated on the conditional transition graph which is limited to the same condition as the specification prescribes, i.e., $K[c]$. More formally, $K[c] \models [c]\phi$.

For example, Figure 6.4 illustrates the conditional transition graph $K[contact]$ of the customer support process presented in Section 4.1. The conditional CTL specification $[contact]AFt6$ can be evaluated on this structure, and evaluates to true. In contrast, the same specification ($AFt6$) would not hold on the full, non conditional, structure depicted in Figure 6.1 because, in case of no contact, the issue can not be resolved, and thus, not be recorded.

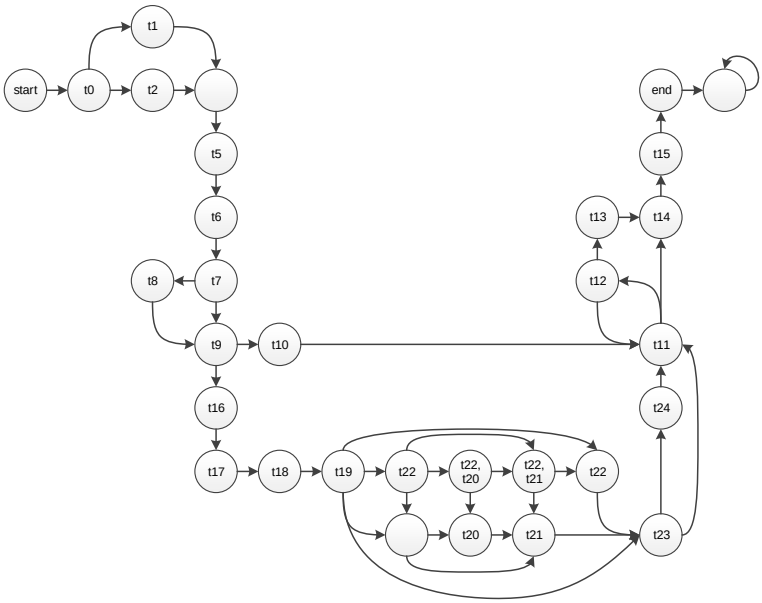


Figure 6.4: The customer support process from Section 4.1 as a conditional Kripke structure.

6.6 Inheritance of Specification Sets

Model checking verifies a system model against a specification of interest. By allowing sets of specifications to be defined, we enable the definition of BP with a high degree of allowed change. Model checking can then be used to verify whether a BP complies with a set of specifications through its transition graph.

Definition 6.6.1 (Template Process)**.** *A template process TP is a triple $TP = (N, AP, F)$, where:*

- *N is a colored Petri net $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$,*
- *$AP \subseteq T$ is a set of atomic propositions,*
- *F is a set of CTL formulas over AP which all evaluate to true on the transition graph K of N.*

Template processes consist of three parts: a CPN, a set of atomic propositions containing transitions of the CPN (i.e. a set of events and activities of the business process which the CPN describes), and a set of temporal logic formulas over the

atomic propositions. Although the definition specifies a CPN as the process type, in reality it can also feature any process specification which can be translated into a CPN similar to to the methods specified in Appendix B.

BP can be based on a template process. A CPN $N' = (P', T', C', W', M'_0)$ is based on a template process iff $\exists t \in T' : t \in AP \subseteq T$ – or, in other words, if it shares elements with the template process. Then, a CPN N', that is based on the template process TP, is compliant with TP iff all formulas $f \in F$ of TP evaluate true on the transition graph K' of N'. A CPN that is compliant with its template process is called a variant of the template process.

Definition 6.6.2 (Variant). *A CPN $N' = (\Sigma', P', T', A', N'_f, C'_f, G'_f, E'_f, M'_0)$ is a variant of template process $TP = (N, AP, F)$ iff:*

- $\exists t \in T' : t \in AP \subseteq T$,
- *All formulas $f \in F$ evaluate to true on the transition graph K' of N'.*

As variants are CPN (or business processes internally translated to CPN) which adhere to all the constraints of the template process TP, they may replace the CPN N of the template process. In such a case, the variant becomes the template process. Continuing this trend, the CPN N of a template process $TP = (N, AP, F)$ can be a variant of another template process $TP' = (N', AP', F')$. As a result, the CPN N adheres to the formulas from both templates $f \in F \cup F'$ over $AP \cup AP'$. Then, a template process TP is a subtemplate of another template process TP' when the subtemplate TP extends the set of formulas and the set of atomic propositions of TP'. More formally:

Definition 6.6.3 (Subtemplate). *A template $TP = (N, AP, F)$ is a subtemplate of $TP' = (N', AP', F')$, iff:*

- *$AP' \subseteq AP$ is in the set of atomic propositions,*
- *$F' \subseteq F$ is in the set of formulas,*
- *All formulas $f \in F$ evaluate true on the transition graph K of N.*

We say that a template process $TP = (N, AP, F)$ is a *child* of another template process TP' if all atomic propositions $AP' \subseteq AP$ and formulas $F' \subseteq F$ of the template process TP' are in TP and all formulas evaluate to true. Dually, TP' is the *parent*

of ST. Two template processes are siblings when they share a parent. Note that subtemplates may also inherit from multiple parents. Although, the subtemplate is defined as containing the formulas and atomic propositions of its template process at initialization, in practice the sets should remain separate until verification. The result is a directional graph of template processes where changes at the top of the graph are automatically propagated down through a myriad of children to the actual verification of variants.

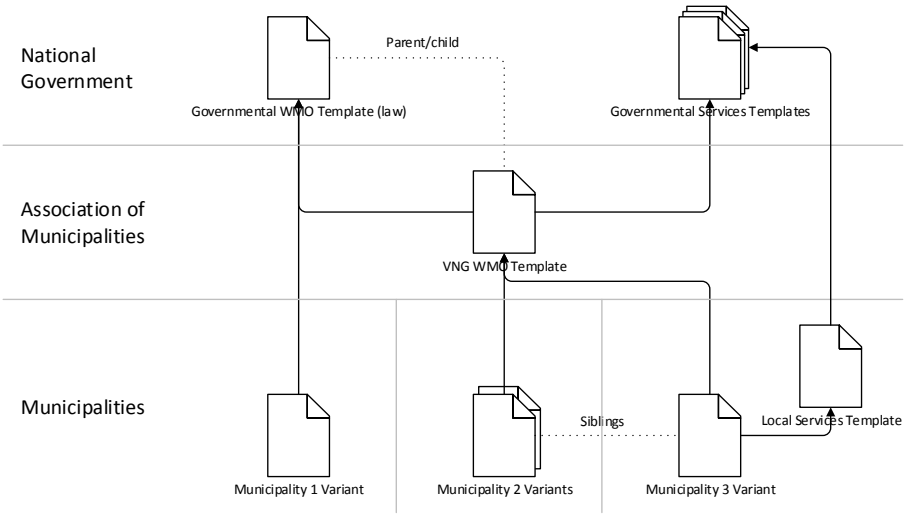


Figure 6.5: Templates and Variants

Figure 6.5 illustrates how templates, subtemplates, and variants could be employed in the domain described by the case study. In this example, the national government publishes a template describing the WMO law, as well as a set of templates describing the use of governmental services (Top left in the Figure). A cooperation of municipalities then bundles these resources and refines the process as a subtemplate which each municipality then may inherit to implement one or more variants for local deployment (center of the figure). As soon as one of the templates is updated, each variant could automatically issue a re-verification routine before a new instance of the process is executed, ensuring continued adherence after updates.

6.7 Model Reduction

The transition graph can be reduced before the model is verified by model checking. As model checking techniques verify models with given specifications in an exhaustive fashion, any reduction of the model benefits performance.

Model reduction can be achieved through the removal of unused atomic propositions and model equivalence under the absence of the nexttime operator, otherwise known as equivalence with respect to stuttering (Browne et al., 1988). Equivalence with respect to stuttering is a useful notion when considering concurrent systems– or, in our case, concurrently executing branches. In such cases it may be dangerous to evaluate the nexttime operator, X, as it refers to the next global state (i.e. the typical interleaved execution of concurrent programs or branches) and not the next local state (i.e. the execution of one such program or branch) (Lamport, 1983). Instead, when one considers the nexttime operator, one actually means to describe that something occurs before other local occurrences– which, in turn, can be specified easily using the other operators. For example, the until operator can be used on the transition graph to specify the next local occurrence (e.g. $AG(t20 \Rightarrow A[t20\ U\ t21])$ can be used to specify that $t20$ is followed by $t21$ in every execution branch of the compliant process depicted in Figure 6.1). To offer a parallel interleaving safe evaluation of the nexttime operator, any nexttime operator is parsed into an until operator. As an additional benefit, without having to evaluate the nexttime operator, the model can be significantly reduced before verification through the notion of equivalence with respect to stuttering.

A finite Kripke structure K can be uniquely identified by a single CTL formula F_K (Browne et al., 1988). As a result, F_K can be used to evaluate the equivalence of other Kripke structures K' to K. When considering F_K without nexttime operators, the equivalence of K' can be evaluated with respect to stuttering. Two Kripke structures K and K' are equivalent with respect to stuttering if all paths from the initial states $s_0 \in S_0$ of K are stutter equivalent with the paths from the initial states $s'_0 \in S'_0$ of K' and vice versa. Two paths are stutter equivalent, as denoted by $\pi \sim_{st} \pi'$, if both paths can be partitioned into blocks of states $\pi = k_0, k_1, ...$ and $\pi' = k'_0, k'_1, ..$ such that $\forall s \in k_i, \forall s' \in k'_i : L(s) = L(s')$ for $i \geqslant 0$ (Groote and Vaandrager, 1990).

To reduce the model, first those atomic propositions not used by specifications, with the exception of those relating to events, are removed. Then, the atomic propositions related to markings are removed from the labels of all states and the set AP

such that $M_i \notin AP$ and $\forall s \in S : M_i \notin L(s)$ for $0 \leqslant i \leqslant n$. Finally, a stutter equivalent model with respect to the used atomic propositions is obtained. Although the removed labels are needed during the conversion process to ensure unique states to be generated, they can be removed at this point because they are not used by specifications or because specifications should only be expressed using bindings on activities or events of the business process (i.e. transitions) and not its progression information (i.e. marking).

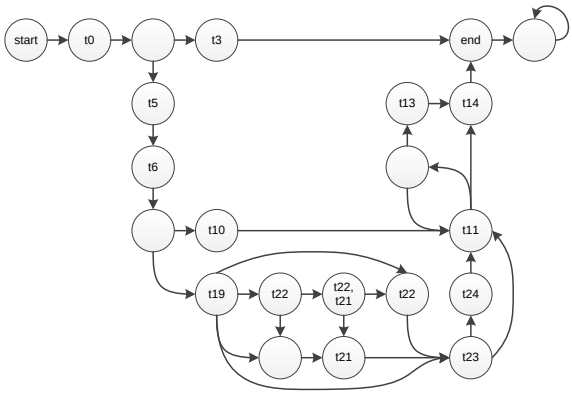


Figure 6.6: The customer support process from Section 4.1 as a reduced Kripke structure w.r.t the atomic propositions required by the compliance specifications.

Figure 6.6 depicts the stutter equivalent model of the Kripke structure depicted in Figure 6.1 after the removal of the unused atomic propositions. Note that several unlabeled states remain. These can not be removed, as it would affect the evaluation of formulas (e.g. $AG(t19 \Rightarrow A[(t19 \vee t21 \vee t22)\ U\ t23])$ would incorrectly evaluate true).

6.8 Discussion

The introduction of the transition graph presents a novel model for the formal verification of BP. Where other models capture events (i.e. occurrences of binding elements in CPN) as atomic actions, we see them as continuous actions over subsequent states. In doing so, we capture parallel execution information, as well as the local next, i.e., the next event in individual branches, be they exclusive or parallel. Using the branching-time temporal logic CTL, we can specify temporal relations, verify them on the transition graph, and interpret them on the possible executions of the CPN.

An additional layer of labeling on both the CPN as well as the transition graph provides verification options over previously unsupported BPMN BPD groups. Both groups of structured activities and grouplike structures, such as pools and swimlanes, can be used for verification. The implications are twofold. Firstly, by verifying over groups, specifications can make statements over multiple elements simultaneously. For example, the CTL specification $AG(p \Rightarrow AFg)$, where g is a group containing q and r, specifies that p is always eventually followed by an element enclosed within g – i.e., q or r. Secondly, we can verify whether an element is enclosed within a group. In doing so, it is possible for a CTL specification to verify whether a structured activity is performed by a certain role. For example, the CTL specification $AG((p \wedge s) \vee \neg p)$ verifies whether p is always contained within the swimlane s – and, thus, is performed by the role of s.

Conditions are an important feature of BP to provide flow control of branching constructs. Although the specification logic CTL is a branching-time temporal logic, it is unable to interpret over such conditions. However, by allowing specifications to be paired with conditions, and allowing those specifications to be verified on transition graphs generated under those conditions, we enable an additional layer of expressiveness without demanding additional functionality from models and specifications, or their supporting model checkers.

Inheritance is provided through a parent-child relation over specifications. In doing so, we provide declarative design-time variability support as an extension of compliance verification. By specifying a BP using declarative compliance-like specifications within a template, and then allowing variants to be based upon such a template, we can employ formal verification and model checking to verify whether the variant BP adheres to the specification rules provided within the parent template.

Formal verification using model checking techniques verifies models against specifications in an exhaustive manner. By providing model reduction, we gain on performance. Model reduction is achieved through the application of model equivalence with respect to stuttering. This is possible because of the absence of the next-time operator, which is dangerous when evaluated within concurrently executing branches. Any, by specifications unused, atomic propositions are removed from the model before the model is reduced by equivalence with respect to stuttering. The result is a model that is optimized with respect to the atomic propositions used for verification. In addition, this process makes it possible to achieve further reduction by splitting specifications into smaller sets, each with less total atomic propositions. Each set of specifications is then veried on a reduced model specific to that set.

When applied together, the presented techniques allow for design-time compliance and variability verification over BP containing parallel, local next, group, role, and condition information while using a model with extensive model reduction.

CHAPTER 7

Verification Specifications

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

– Edsger W. Dijkstra

The specification of rules over business processes is not without difficulties. Not only is the process of specifying rules difficult when using temporal logics, but also the relations between rules themselves can quickly overwhelm. In addition, we find a clear gap between traditional imperative BP design and the declarative rule based design offered by compliance and variability (Aiello et al., 2010). Where the behavior of imperatively designed BP are easily interpreted, declarative BP must be interpreted with the ability to evaluate alternative and exceptional situations.

Variability is commonly offered to imperatively designed BP through feature modeling and variation points (Sun and Aiello, 2008). Although extremely powerful and straightforward to understand when requiring low amounts of variability, the method quickly encounters difficulties when having to deal with the high amounts of variability offered by flexible declarative process definitions. At the same time, the declarative style of BP definition fares extremely well when dealing with highly flexible BP, but would require an amount of rules incomprehensible to any user when designing a process with low amounts of variability (Groefsema et al., 2012).

Declarative specifications are rule based specifications over (groups of) structured activities in BP. We define declarative specifications for BP soundness, compliance, and high variability. Each specification is (A) related to specific requirements described in Chapter 5, (B) defined using the temporal logic CTL, and (C) includes a visual element, where applicable.

To ease the application of specifications, other BP compliance and flexibility approaches included visual elements in the form of connectors and labels to support their application (van der Aalst and Pesic, 2006; Awad et al., 2008). We continue this trend while recognizing that – although this approach is very effective for BP compliance or BP variability when considering extremely flexible BP – it requires additional support when considering BP variability with low amounts of flexibility.

7.1 Visualization

Declarative specifications enforce temporal conditions on the presence of structured activities within possible executions of a BP. To ease design and readability of specifications, each specification can be represented both in a formal way (e.g. by a CTL formula) and in a visual way. Since existing visualization languages are either designed towards compliance verification of interleaved linear executions of BP (Awad et al., 2008), or designed to define flexibile linear interleaved BP executions (van der Aalst and Pesic, 2006; Pesic and van der Aalst, 2006), they lack support for the most loosely defined variability specification requirements (e.g., Requirement 5.2.11.ii) as well as support for different branching requirements (e.g., Requirement 5.2.15.ii), parallel requirements (e.g., Requirement 5.2.15.xi), and consistent immediate response requirements throughout the model (e.g., Requirement 5.2.15.iii). Because of this, a new visualization language is required for declarative BP variability. The visualization of the formal representation is presented through the rules described below.

Specifications consists visually of a set of *labels*. Labels are either applied directly to (groups of) structured activities or part of connectors. Labels are shapes of triangles ▶, squares ■, or diamonds ◆ that can be either open or filled:

- Triangles signify presence in some paths (open ▷) or all (filled ▶).
- Squares signify absence in some paths (open □) or all (filled ■).
- Diamonds signify presence in exclusive (open ◇) or parallel (filled ◆) paths.

For example, a structured activity that is labeled with an open triangle ▷ must be present in at least one execution path, and a group containing multiple structured

activities that is labeled with a closed triangle ▸ enforces that in every execution path one of the structured activities contained within the group must be present.

Connectors enforce temporal relations between the presence of two (groups of) structured activities in execution paths. Where labels on (groups of) structured activities represent basic constraints on the process, connectors describe complicated conditions and effects. Connectors can be uni- or bi-directional.

Uni-directional connectors specify a condition and effect relation between its source and target. Uni-directional connectors contain a filled circle ● at its source, possibly followed by a number of labels, which together specify the condition of the connector. If no further labels are specified as part of the condition, the condition describes encountering the source in an execution path. An open circle ○ specifies a relation between a structured activity and a role performing that activity instead. The effect of the connector is described by labels at its target. Connectors can either consist of a solid or dashed line. The effect of a connector with a solid line mult hold immediately, whereas the effect of a connector with a dashed line mult hold eventually. When multiple labels are specified as part of the condition, each label is paired with a corresponding effect label at its target, of which only applicable effects are applied (i.e. those corresponding to conditions that hold). For example, Figure 7.1 depicts the step by step interpretation of a connector which specifies that when x is larger than zero, and the activity p is absent from the process, the activity q must be absent as well, and when p is present, q must be too.

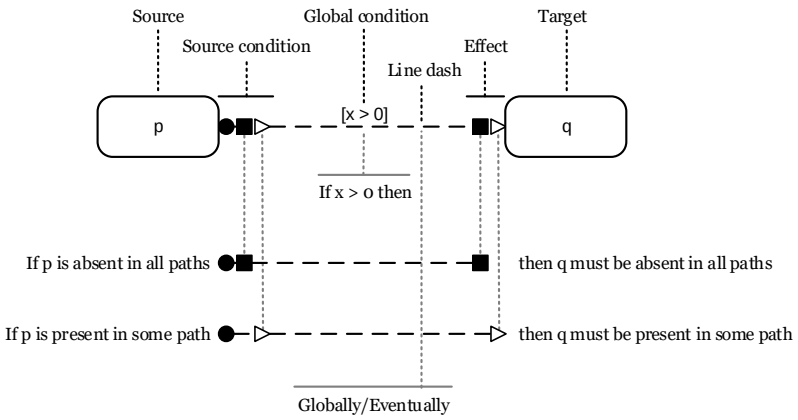


Figure 7.1: Visualization Example

Bi-directional connectors have a circle at the center of the connector with its con-

ditions and effects facing outwards both ways. Bi-directional connectors can be interpreted as two uni-directional connectors joined at its conditions, with a filled circle ● specifying that the conditions and effects on each side must both be satisfied, and an open circle ○ specifying that the conditions and effects on one of the sides must be satisfied instead.

7.2 Soundness Specifications

Soundness specifications include specifications for the verification of the correctness of BP. Specifications include those verifying reachability of states, process termination, and proper process completion.

Specifications 7.2.1 (Reachability). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The reachability of structured activities $t \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:*

Requirement	***Specification***	***Visual element***
5.2.1 Reachability	$EF\ t$	*N/A*

The reachability of individual structured activities can be verified using Specification 7.2.1. A structured activity, as represented by a transition $t \in T$ of the CPN N, is reachable iff there exists a path within the transition graph K of N where eventually t holds. When Specification 7.2.1 evaluates to false for a transition $t \in T$ then there exists no marking (i.e. distribution of tokens over places P) in the CPN N where t is enabled and would occur eventually.

Specifications 7.2.2 (Termination). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The termination of N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:*

Requirement	***Specification***	***Visual element***
5.2.2 Termination	$AF\ end$	*N/A*
5.2.2 Weak termination	$EF\ end$	*N/A*
5.2.2 Weak termination (alt.)	$\neg E[\neg end\, U\ (\neg EF\ end \wedge \neg end)]$	*N/A*

The termination of a BP, represented by a CPN N, can be verified against Specification 7.2.2. Specification 7.2.2 includes three different formulas which denote termination. The first, $AFend$ denotes whether a transition associated with an end event is always eventually encountered in every execution path. Since this formula fails when encountering loops in the Kripke structure, an additional two formulas are included. These weak termination formulas denote whether there exists an

execution path where a transition associated with an end event is eventually encountered, and whether this is the case at each reachable state[1], respectively.

Specifications 7.2.3 (Proper completion). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The proper completion of N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:*

Requirement	***Specification***	***Visual element***
5.2.3 Proper Completion	$AG(end \Rightarrow AX\,AG\,\neg end)$	*N/A*

The proper completion of a BP, represented by a CPN N, can be verified against Specification 7.2.3. Specification 7.2.3 denotes whether, after encountering a state associated with an end event, no such a state is encountered again. Whenever this formula evaluates to false, there exists an interleaving of concurrently executing paths involving the end event. In other words, there exists an execution where a structured activity is being executed concurrently to the invocation of the end event and the BP is not properly completed.

7.3 Preventive Compliance Specifications

Preventive compliance verification verifies whether a BP is compliant with a set of rules represented by compliance specifications. Preventive compliance specifications include specifications on the occurrence of structured activities and their effects, the order of execution of structured activities and their effects, and resource allocation (e.g. which roles perform which structured activities), while under condition of possible data values. Although the binding elements captured by the transition graph could capture the effects of the execution of structured activities within its binding, we have found that many such effects and related rules can be inferred from events pertaining to the BP. We, therefore, focus on the occurrence of events and related rules.

Specifications 7.3.1 (Occurrence). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The occurrence of structured activities $t \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:*

[1] This formula is equivalent to $A[EF\ end\ W\ end]$, where W is the lesser known Unless operator.

Requirement	*Specification*	*Visual element*
5.2.4.i Absence	$AG\ \neg t$	■
5.2.4.ii Universality	$AG\ t$	*N/A*
5.2.4.iii Existence	$AF\ t$	▶
5.2.4.iv Bounded existence	$\neg EF(\neg t \wedge EX(t \wedge EF(\neg t \wedge EX(t \wedge EF(\neg t \wedge EX(t))))))$	*N/A*

The compliance specifications towards the occurrence of structured activities are compiled from the property specification patterns of finite state verification (Dwyer et al., 1999). Four formulas pertaining to the occurrence of structured activities $t \in T$ are included in Specification 7.3.1 and verify the absence, universality, existence, and bounded existence of a structured activity, respectively. Although universality and bounded existence are described as being evaluated over a single $t \in T$, their evaluation would only result in useful feedback when evaluating over sets of structured activities as described in Section 6.4.

Specifications 7.3.2 (Ordering). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The ordering of structured activities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:*

Requirement	*Specification*	*Visual element*
5.2.5.i Precedence	$\neg E[\neg t_0\ U\ t_1]$	●◀– – – – – – – – – –
5.2.5.ii Response	$AG(t_0 \Rightarrow AF\ t_1)$	●– – – – – – – – – –▶

The compliance specificationd towards the ordering of structured activities are also compiled from the property specification patterns of finite state verification (Dwyer et al., 1999). Two formulas specifying the order between two structured activities $t_0, t_1 \in T$ are included in Specification 7.3.2. The first describes a precedence relation which specifies that t_0 must precede any occurrence of t_1. Similarly, the second describes a response relation specifying that any occurrence of t_0 must eventually be followed by an occurrence of t_1. Although seemingly similar, the two formulas behave entirely different around branching and merger of exclusive paths, as well as in their indirect inclusion requirements of t_0 and t_1 respectively.

Specifications 7.3.3 (Resource Specifications). *Let $LN = (N, G_l, R_l, L_g, L_r)$ be a labeled CPN. Resource specifications over structured activities $t \in T$ within N, roles $r \in R_l$, and $(t, r) \in AP$ are determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the labeled transition graph of N:*

Requirement	*Specification*	*Visual element*
5.2.6.i Always performed by role	$AG(t \Rightarrow (t, r))$	
5.2.6.ii Performed by role	$EF(t \Rightarrow (t, r))$	
5.2.6.ii Never performed by role	$AG(t \Rightarrow \neg(t, r))$	

Resource specifications include specifications on the roles that perform specific structured activities. Specification 7.3.3 includes three resource related formulas that verify whether a structured activity $t \in T$ is performed by role $r \in R_l$ either always, at least once, or never.

Specifications 7.3.4 (Data Conditions). *Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN, c a condition related to N, $K = (S, S_0, R, L)$ the transition graph of N, and $K[c] = (S, S_0, R, L)$ the conditional transition graph of N. The specification ϕ is determined by evaluating the following:*

Requirement	*Specification*	*Visual element*
5.2.7.i Holds always	$K \models \phi$	*N/A*
5.2.5.ii Holds when	$K[c] \models [c]\phi$	*N/A*

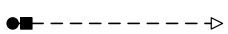
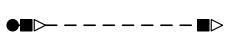
Instead of including additional formulas, Specification 7.3.4 adds an additional layer of expressive power to all other specifications by allowing specifications to be evaluated under condition of (ranges of) data values. Verification of formulas over data conditions $[c]\phi$ increases the expressive power of a formula by evaluating that formula ϕ on a Kripke structure K which only includes states that are reachable under condition of c.

7.4 Variability Specifications

Variability verification extends preventive compliance verification by defining sets of specifications to describe the core operations of a BP. As a result, variability verification allows BP to be designed and redesigned with several implementations as long as the set of specifications evaluates to true. Potentially, many different BP can be designed and implemented for local specific use from the exact same set of specifications describing the generic case. Although we focus on redesign during the process design phase (Figure 2.2), the same principles could, in theory, be applied to support BP redesign during process enactment as long as already executed and currently executing sections of the BP remain unaltered.

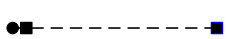
Specifications 7.4.1 (Inclusion Specifications).
Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The inclusion of structured activ-

ities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:

Requirement	***Specification***	***Visual element***
5.2.11.i Include	$EF\ t_0$	
5.2.11.ii Prerequisite	$EF\ t_0 \Rightarrow EF\ t_1$	
5.2.11.iii Substitution	$AG\ \neg t_0 \Rightarrow EF\ t_1$	
5.2.11.iv Corequisite	$(EF\ t_0 \Rightarrow EF\ t_1) \wedge$ $(EF\ t_1 \Rightarrow EF\ t_0)$	
5.2.11.v Causal selection	$(EF\ t_0 \Rightarrow EF\ t_1) \wedge$ $(AG\ \neg t_0 \Rightarrow AG\ \neg t_1)$	

Inclusion specifications enforce the inclusion of (sets of) structured activities. Specification 7.4.1 includes five selection formulas. The first formula enforces the simple inclusion of structured activities, while the other four enforce more complex inclusion relations between structured activities. Note, that inclusion does not equal existence (Specification 7.3.1). That is, the selection of a structured activity does not require its execution during process enactment and only requires its occurrence in at least one possible execution path.

Specifications 7.4.2 (Exclusion Specifications).
Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The exclusion of structured activities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:

Requirement	***Specification***	***Visual element***
5.2.12.i Exclude	$AG\ \neg t_0$	
5.2.12.ii Exclusion	$EF\ t_0 \Rightarrow AG\ \neg t_1$	
5.2.12.iii Admittance	$AG\ \neg t_0 \Rightarrow AG\ \neg t_1$	
5.2.12.iv Exclusive choice	$(EF\ t_0 \Rightarrow AG\ \neg t_1) \wedge$ $(EF\ t_1 \Rightarrow AG\ \neg t_0)$	

Exclusion specifications enforce the exclusion of structured activities. Specification 7.4.2 includes four selection formulas. The first formula enforces the simple exclusion of structured activities, while the other three enforce exclusion relations between structured activities.

Specifications 7.4.3 (Execution Specifications).
Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The execution of structured activities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:

Requirement	*Specification*	*Visual element*
5.2.13.i Execute	$AF\ t_0$	
5.2.13.ii Requirement	$EF\ t_0 \Rightarrow AF\ t_1$	
5.2.13.iii Replacement	$AG\ \neg t_0 \Rightarrow AF\ t_1$	
5.2.13.iv Backup	$EG\ \neg t_0 \Rightarrow AF\ t_1$	
5.2.13.v Causal execution	$(EF\ t_0 \Rightarrow AF\ t_1) \wedge$ $(AG\ \neg t_0 \Rightarrow AG\ \neg t_1)$	

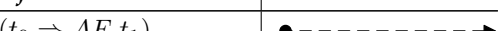
Execution specifications enforce requirements on the execution of structured activities. That is, they require that structured activities are encountered in all possible execution paths. Specification 7.4.3 includes five execution formulas. Similar to earlier specifications, the first formula enforces the simple execution of structured activities, while the other four enforce more complex execution relations between structured activities.

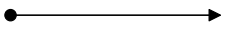
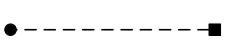
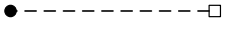
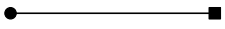
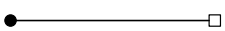
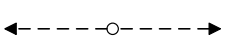
Specifications 7.4.4 (Option Specifications).
Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The optional execution of structured activities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:

Requirement	*Specification*	*Visual element*
5.2.14.i Option	$EG\ \neg t_0$	
5.2.14.ii Avoidance	$EF\ t_0 \Rightarrow EG\ \neg t_1$	

Specification 7.4.4 includes two option formulas. Similar to execution specifications, option specifications enforce requirements on the execution of structured activities. However, instead of enforcing the occurrence of structured activities in all possible execution paths, option specifications enforce that structured activities do not occur in all execution paths. For example, a structured activity with both include (Specification 7.4.1.i) and option (Specification 7.4.4.i) must occur in a path, but not all. As a result, the structured activity is truly optional to execute.

Specifications 7.4.5 (Scheduling Specifications).
Let $N = (\Sigma, P, T, A, N_f, C_f, G_f, E_f, M_0)$ be a CPN. The scheduling of structured activities $t_i \in T$ within N is determined by evaluating the following specifications on $K = (S, S_0, R, L)$ the transition graph of N:

Requirement	*Specification*	*Visual element*
5.2.15.i Response	$AG(t_0 \Rightarrow AF\ t_1)$	
5.2.15.ii Exists response	$AG(t_0 \Rightarrow EF\ t_1)$	

5.2.15.iii Immediate response	$AG(t_0 \Rightarrow A[t_0\ U\ t_1])$	
5.2.15.iv Exists immediate response	$AG(t_0 \Rightarrow E[t_0\ U\ t_1])$	
5.2.15.v No response	$AG(t_0 \Rightarrow AG\neg t_1)$	
5.2.15.vi Exists no response	$AG(t_0 \Rightarrow EG\neg t_1)$	
5.2.15.vii No immediate response	$AG(t_0 \Rightarrow \neg E[t_0\ U\ t_1])$	
5.2.15.viii Exists no imm. response	$AG(t_0 \Rightarrow \neg A[t_0\ U\ t_1])$	
5.2.15.ix Coexecution	$AG(t_0 \Rightarrow AF\ t_1) \vee$ $AG(t_1 \Rightarrow AF\ t_0)$	
5.2.15.x Cooccurence	$AG(t_0 \Rightarrow EF\ t_1) \vee$ $AG(t_1 \Rightarrow EF\ t_0)$	
5.2.15.xi Parallel execution	$EF(t_0 \wedge t_1)$	
5.2.15.xii Exclusive execution	$AG(t_0 \Rightarrow AG\neg t_1) \wedge$ $AG(t_1 \Rightarrow AG\neg t_0)$	

Scheduling specifications enforce requirements on the order of execution between structured activities. Specification 7.4.5 includes twelve scheduling formulas, which range from requiring a simple response to requiring parallel execution.

7.5 Discussion

Formal verification is a technique where a model and specification are used in concert to verify a system. The model describes the system, while the specification describes the required behavior. The required behavior is then verified on the model. However, that which is not described by the model, can never be verified by a specification. When evaluating the area of formal verification of BP models, we often see a large discrepancy between what models describe (and, thus, what behavior can be verified by a specification). In case of verification of the soundness of modeling languages, every detail is described by the model and any related specification can be verified. The models themselves, however, quickly consist of uncountable many states. In case of the verification of BP compliance and variability, all these detailed states are unnecessary. However, the information captured by models specific to compliance often capture too little information. For example, the immediate response and parallel execution specifications (Specifications 7.4.5.iii and xi) are often left unsupported when capturing concurrent behavior of executing branches.

Through the definition of the transition graph we allow a large set of event-based specifications. The transition graph supports specifications towards the verification of soundness, compliance, and variability. However, since the transition graph

is Petri net-based, and soundness verification is solved for Petri net definitions (van der Aalst, 1998; Wynn et al., 2009), we decide to perform soundness verification on the Petri net as a preprocessing step instead. The benefits are twofold. First, soundness specifications allow for limited reduction of the transition graph as every atomic proposition must be used for the verification of reachability of each structured activity. Without these specifications, unused atomic propositions can be removed, and model reduction can be applied to greater effect. Second, with a sound Petri net, the transition graph can be formed programmatically under saver conditions. This way, the BP is first verified for soundness, and then transformed to a transition graph and verified against compliance and variability specifications.

For example, by applying the specification patterns defined throughout this chapter, we can interpret the TCP compliance rules from the telecommunications customer support case (Table 4.1) to form the CTL specifications of Table 7.2.

Table 7.2: TCP Compliance Rules as CTL specifications.

#	CTL Formula	Specification
1.	`!E[!(t11 ∨ t3) U end]`	7.3.2.i
	`AF(end)`	7.4.3.i
2.	`AG(t10 ⇒ AF(t13 ∨ t14))`	7.4.5.i
3.	`AG(t5 ⇒ A[t5 U t6])`	7.4.5.iii
4.	`AG(t5 ⇒ AF(t14))`	7.4.5.i
5.	`AG(t5 ⇒ EF(t24))`	7.4.5.ii
6.	`!EF(t21 ∧ t23)`	7.4.5.xi
	`!EF(t22 ∧ t23)`	7.4.5.xi
7.	`AG(t13 ⇒ A[t13 U t14])`	7.4.5.iii
8.	`EF(t21 ∧ t22)`	7.4.5.xi
9.	`AG(t19 ⇒ EF(t23))`	7.4.5.ii

CHAPTER 8

Automated Specification Assembly

Even perfect program verification can only establish that a program meets its specification. [...] Much of the essence of building a program is in fact the debugging of the specification.

– Fred Brooks

Business processes, or process families, with large amounts of variability are easily described using the earlier presented methods, however, processes with smaller amounts of variability require a significant number of specifications to describe using the same declarative techniques. In other words, as the amount of variability decreases, the set of specifications increases dramatically when using declarative techniques. Often this set becomes so large that it is impossible to define manually.

To support both declarative definitions of BP with low amounts of variability and automated detection and incorporation of ad-hoc runtime deviations of BP, we propose a method which uses sets of source BP to automatically generate the required specifications. That is, we allow the input of a set of source BP describing the allowed variants, consolidate this set into a single structure defining the combined relations between the included structured activities, and output a set of specifications describing those variants, and any possible variant in between.

Source BP can either be previously designed imperative BP variants (McMillan and Probst, 1995; Esparza et al., 1996), or be mined from execution traces of previous executions of BP variants (van Beest et al., 2015). The result is a set of specifications applicable within a template process (Definition 6.6.1) together with one of the source BP. The proposed technique can be applied to describe either entire BP, or sections within BP that must not allow any change in the form of sub-processes.

8.1 Prime Event Structures

From the perspective of a declarative specification, the allowed control-flow is specified by the relations between activities with respect to their allowed or constrained behavior. That is, what activities are required, what activity sequences are not allowed in one run (conflict), which activities can be executed in parallel, etc. As such, so-called *behavioral relations* are required to be obtained from each input model in order to be combined into a set of common behavioral relations that together represent the generic process model.

Event structures are specifically designed to obtain and represent these relations between event occurrences in a process. Therefore, we convert each input process to an event structure, which are subsequently merged with the aim of generating a generic set of behavioral relations that together can be converted into a variability specification which describes a family of BP.

A Prime event structure (Nielsen et al., 1981) is a graph of events, where each event e represents the occurrence of a task or activity in the business process. As such, multiple occurrences of the same activity are represented by different events. Events can have the following binary behavior relations: i) *Causality* ($e < e'$) indicates that event e is a prerequisite for e'; ii) *Conflict* ($e \# e'$) implies that e and e' cannot occur in the same run; iii) *Concurrency* ($e \parallel e'$) indicates that no order can be established between e and e'. This can be defined formally as follows:

Definition 8.1.1 (Labeled Prime Event Structure (Nielsen et al., 1981))**.** *A* Labeled Prime Event Structure *over the set of event labels $\mathcal{L}$ is the tuple $\mathcal{E} = \langle E, \leqslant, \#, \lambda \rangle$ where:*

- *E is a set of events,*
- *$\lambda : E \to \mathcal{L}$ is a labeling function.*
- *$\leqslant \subseteq E \times E$ is a partial order, referred to as* causality *relation,*
- *$\# \subseteq E \times E$ is an irreflexive, symmetric* conflict *relation,*
- *$\| : E^2 \setminus (< \cup <^{-1} \cup \#)$ is the* concurrency relation, *where $<$ denotes the irreflexive causality relation.*

The *conflict relation* satisfies the principle of conflict heredity, i.e. $e\#e' \wedge e' \leqslant e'' \Rightarrow e\#e''$ for $e, e', e'' \in E$.

The sets of events that can occur together in an execution of the system underlying the event structure system represent the possible "states" in the execution context of that system and are referred to as *configurations*. A set of events $C \subseteq E$ is a configuration iff (i) C is causally closed: for each event $e \in C$, the configuration C also contains all causal predecessors of e, i.e. $\forall e' \in E, e \in C : e' \leqslant e \Rightarrow e' \in C$, and (ii) C is conflict free: C does not contain any pair of events in mutual conflict, i.e. $\forall e, e' \in C \Rightarrow \neg(e\#e')$. An event e is an extension of a configuration C, denoted $C \oplus e$, iff $C \cup \{e\}$ is also a configuration.

A *local configuration* of an event e is defined as $\lceil e \rceil \triangleq \{e' \mid e' \leqslant e\}$. The *strict causes* of an event e are defined as $\lceil e) \triangleq \lceil e \rceil \setminus \{e\}$. Two events e_1 and e_2 are in *immediate conflict*, denoted $e_1 \#_\mu e_2$, iff $e_1 \# e_2$ and they are both possible extensions of the same configurations, which can be verified by checking if $\lceil e_1) \cup \lceil e_2 \rceil$ and $\lceil e_1 \rceil \cup \lceil e_2)$ are both configurations or not.

8.2 Prefix Unfoldings

A *branching process* is an alternative class of Petri nets that explicitly represents all partially-ordered runs of the original net in a single tree-like structure (Engelfriet, 1991; Esparza, 1994). A run of a net is a partially-ordered set of events that can occur in one execution of that net. Branching processes are intimately related with prime event structures, as they explicitly represent the same set of behavioral relations, which is formally shown in the following definition.

Definition 8.2.1. *Let $N = (P, T, A)$ be a net and $x, y \in P \cup T$ two nodes in N.*

- *x and y are* causal, *written $x <_N y$, iff $(x, y) \in A^+$,*
- *x and y are in* conflict, *denoted $x \#_N y$, iff $\exists\, t, t' \in T : t \neq t' \wedge {\bullet t} \cap {\bullet t'} \neq \emptyset \wedge (t, x), (t', y) \in A^*$,*
- *x and y are* concurrent, *denoted $x \|_N y$, iff neither $x <_N y$, nor $y <_N x$, nor $x \#_N y$.*

Subsequently, a branching process can now be defined formally as follows (Esparza, 1994):

Definition 8.2.2. *Let $N = (P, T, A, M_0)$ be a Petri net. The* branching process *$\beta = (B, E, G, \rho)$ of N is the net (B, E, G), where:*

- *B is the set of conditions,*
- *E is the set of events,*
- *G denotes the flow relation of the branching process,*

and is defined by the inductive rules in Figure 8.1. The rules also define the function $\rho : B \cup E \to P \cup T$ that maps each node in β to a node in N. Given the set $X \subseteq B \cup E$, $\varrho(X)$ is a shorthand for $\{\rho(x) \mid x \in X\}$.

$$\frac{p \in M_0}{b = \langle \emptyset, p \rangle \in B \qquad \rho(b) = p}$$

$$\frac{t \in T \quad B' \subseteq B \quad B'^2 \subseteq \|_\beta \quad \varrho(B') = {\bullet t}}{e = \langle B', t \rangle \in E \qquad \rho(e) = t}$$

$$\frac{e = \langle B', t \rangle \in E \quad t{\bullet} = \{p_1, \ldots, p_n\}}{b_i = \langle t', p_i \rangle \in B \qquad \rho(b_i) = p_i}$$

Figure 8.1: Inductive construction of a branching process

$Min(\beta)$ denotes the set of minimal elements of $B \cup E$ with respect to the transitive closure of G. As such, $Min(\beta)$ corresponds to the set of places in the initial marking of N, i.e., $\varrho(Min(\beta)) = M_0$. Labels on a net N can be carried over to its branching process β by composing λ and ρ, i.e., $\lambda_\beta \triangleq \lambda_N \circ \rho$.

The behavioral relations derived from the branching process generate a prime event structure (Nielsen et al., 1981). Transitions in the branching process correspond to events in the event structure. Subsequently, we can extrapolate the notion of configuration prime event structures to branching processes. Given a branching process $\beta = (B_\beta, E_\beta, G_\beta, \lambda_\beta)$ of a marked net N, the event structure $\mathcal{E}$ of N is defined as $\mathcal{E}(N) \triangleq (E_\beta, \leqslant_\beta \cap E_\beta^2, \#_\beta \cap E_\beta^2, \lambda_\beta|_{E_\beta})$[1]. In (Armas-Cervantes et al., 2016), it is shown that silent events can be removed in a behavior-preserving manner, under the notion of visible-pomset equivalence.[2]

The branching process of a Petri net with cycles is potentially infinite. For safe nets, however a *prefix* of a branching process fully encodes the behavior of the original

[1] We use A^2 to denote the cartesian product of a set A.

[2] This holds if every sink event in the event structure is a visible event, which can be ensured by adding a dummy labeled final event to the Petri net from which the event structure is generated.

net, as shown in (McMillan and Probst, 1995). Such a prefix is referred to as the *complete prefix unfolding* of a net. We use β_m to denote a maximal branching process and β_f to denote a complete prefix unfolding of β_m.

Definition 8.2.3. *Let $\beta_m = (B_m, E_m, G_m, \rho_m)$ be the maximal, possibly infinite branching process of the net system N.*

- *A* local configuration *$\lceil e \rceil$ of an event e in a branching process is the set of events that causally precede e, i.e. $\lceil e \rceil = \{e' \in E_m \mid (e', e) \in G_m^*\}$.*
- *The* reachable marking *of a local configuration, denoted $Mark(\lceil e \rceil)$, is the set of places in N that get marked after all the transitions in $\varrho(\lceil e \rceil)$ fire.*
- *An* adequate order *$\lhd$ is a strict well-founded partial order on local configurations, such that $\lceil e \rceil \subset \lceil e' \rceil$ implies $\lceil e \rceil \lhd \lceil e' \rceil$*[3]*.*
- *An event e of a branching process is a* cutoff event *if there exists a* corresponding event *e', such that $Mark(\lceil e \rceil) = Mark(\lceil e' \rceil)$ and $\lceil e' \rceil \lhd \lceil e \rceil$. The pair cutoff/corresponding events (e, e') is referred to as a* cc-pair*.*
- *Let $E_f \subseteq E_m$ be the set of events of β_m such that $e \in E_f$ iff no event $e' <_{\beta_m} e$ is a cutoff event. A* complete prefix unfolding *β_f is the subnet of β_m having E_f as set of events.*

A cutoff-corresponding pair is called a *cc-pair*. The isomorphism on the future of the events in a cc-pair (e, f), that is $\lceil e \rceil \Uparrow$ and $\lceil f \rceil \Uparrow$, will be denoted as $\mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}$.

To represent the behavior specified by a business process model, we use the prime event structure derived from the complete prefix unfolding of the corresponding Petri net, which is referred to as the *PES prefix unfolding* of a model.

Definition 8.2.4. *Let $\beta_f = (B_f, E_f, G_f, \rho)$ be the complete prefix unfolding of the net system N, with labelling function λ_β. Let $\overline{E_f} \subseteq E_f$ be the set of events of β_f such that $e \in \overline{E_f}$ iff e is labelled (i.e. $\lambda_\beta(e) \neq \tau$), or e is cutoff or corresponding event. The* PES prefix unfolding *of N, denoted $\overline{\mathcal{E}}(N)$:*

$$\overline{\mathcal{E}}(N) \triangleq (\overline{E_f}, \leqslant_\beta \cap \overline{E_f}^2, \#_\beta \cap \overline{E_f}^2, \lambda_\beta|_{\overline{E_f}})$$

It is easy to see that the computation of a PES prefix unfolding is the same as for a regular prime event structure except that we keep track of cc-pairs, and we do not abstract away a silent event when such event is either a cutoff event or a corresponding event.

[3] Several definitions of *adequate order* exist; we use the one defined in (Esparza et al., 2002), because it has been shown to generate compact unfoldings.

8.3 Execution and Elementary Loop Identification

As a result of the PES prefix unfolding, some configurations are not explicitly represented. For these cases, we use the "shift" operation on net unfoldings introduced in (Esparza, 1994), to facilitate the exploration of configurations. Given a cc-pair (e, f), we can "shift" from one configuration to the other, as the futures of $\lceil e \rceil$ and $\lceil f \rceil$ are isomorphic. The shift operation is essentially a "step" function that allows to move from one configuration to another. This intuition is captured in the following definition:

Definition 8.3.1. *Let (e, f) be a cc-pair of the PES prefix $\overline{\mathcal{E}}$ and $\mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}$ the isomorphism from $\lceil e \rceil \Uparrow$ to $\lceil f \rceil \Uparrow$. Moreover, let C be a configuration of $\overline{\mathcal{E}}$. The (e, f)-shift of C, denoted $\mathcal{S}_{(e,f)}(C)$:*

$$\mathcal{S}_{(e,f)}(C) = \lceil f \rceil \cup \mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}(C \setminus \lceil e \rceil)$$

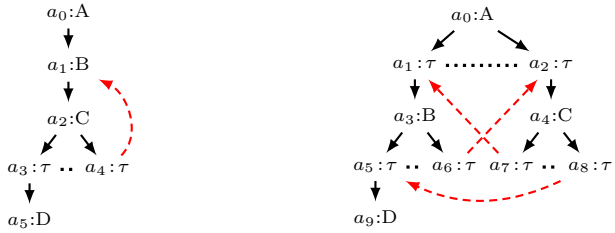
$\mathcal{S}_{(e,f)}(C)$ is a *backward shift* iff $\lceil f \rceil \subset \lceil e \rceil$, that is, the corresponding event f is included in the local configuration of the cutoff event e, otherwise $\mathcal{S}_{(e,f)}(C)$ is called a *forward shift*. Moreover, an event e is a *backward cutoff event* iff it entails a backward shift. Intuitively, a backward shift "moves back" to a configuration that has already been observed in the past of the run. Overloading the notation, we use the following variant:

$$\overline{\mathcal{S}}_e(C) = \begin{cases} C & \text{if } e \text{ is not a cutoff event} \\ \mathcal{S}_{(e,\text{corr}(e))}(C) & \text{otherwise} \end{cases}$$

The shift operation is sufficient to identify the subsequent relations of cutoff events. However, in case of cycles, eventually the same cutoff event will be visited again and this relation cannot be inferred from the existing relations. Consequently, we require a graph of configurations and configuration extensions, for which we rely on the notion of *partially ordered multisets (pomset)* (Pratt, 1986). A pomset is a directed acyclic graph where the nodes are configurations, and the edges represent direct causality relations between configurations. Each edge is labeled by an event. As a pomset represents one possible execution, it does not contain conflict relations. The behavior of a PES can be characterized by the set of pomsets it induces.

When the PES prefix captures cyclic behavior via cc-pairs, the set of induced pomsets is infinite. This can be resolved by extracting a set of elementary pomsets that collectively cover all the possible pomsets induced by a PES prefix, similar to the notion of elementary paths. As such, every cycle is unfolded such that it is traversed only once.

By successively applying configuration extensions and shift operations on the PES prefix, we obtain a so-called *expanded prefix*, which is a directed acyclic graph reflecting the configuration extension relation. Although we do not explicitly build the entire expanded prefix for the entire PES, we are implicitly using it in order to identify loop relations between events. Figure 8.2 shows two PES prefix unfoldings with cyclic behavior, one with one loop entry (a) and one with two loop entries (b). The red dashed arrows depict the shift from the cutoff events to their respective corresponding events. We will subsequently use these examples to explain the prefix expansion for identifying loop relations.



(a) Single entry cycle (b) Double entry cycle

Figure 8.2: PES prefix unfoldings of cyclic models

To identify the set of events that are part of an elementary loop, we need to obtain the full loop configuration, denoted by C_L. For the PES prefix unfolding in Figure 8.2a, it is easy to see that $C_L = \{a_1, a_2, a_4\}$, whereas for the PES prefix unfolding in Figure 8.2b, $C_L = \{a_1, a_2, a_3, a_4, a_6, a_7\}$. Note that these configurations include all cc-pairs involved in the loop as well.

For that purpose, we adapt Algorithm 1 from (García-Bañuelos et al., 2015) to compute the set of *elementary pomsets* of a PES prefix. The function FINDEPOMSETS adds one path (or branch) to the expanded prefix every time an elementary pomset is found (in lines 13 and 20). The events in the resulting expanded prefix are not explicitly represented in the PES prefix. However, considering the order obtained by the depth first traversal followed by FINDEPOMSET, the value of conf is always unique, as conf is finitely extended by a different event until a complete configuration is found or until a duplicate event occurs. Using sconf, i.e. the shifted version of conf, cycles can be identified. In line 11, we check whether the shift has already been observed. If so, each identified full loop configuration C_L (defined by n_conf \ entryCauses) is stored in cycles (line 12), which represents a complete set of full loop configurations upon completion of FINDEPOMSETS. Using the identified loop

Algorithm 1 Identification of elementary pomsets

```
1: procedure FINDEPOMSETS(conf, sconf, causes, visited, var cycles, var runs)
2:     APPEND(visited, (conf, sconf, causes))
3:     for (sconf --e--> n_sconf) do
4:         n_conf ← conf ∪ {e}
5:         if e is cutoff event then
6:             n_sconf ← S(n_sconf)
7:             n_causes ← n_sconf \{corr(e)}
8:         else
9:             n_causes ← conf
10:        end if
11:        if ∃(entryConf, n_sconf, entryCauses) ∈ visited ∧ n_sconf ∩ ||[e] = ∅ then
12:            cycles ← cycles ∪ {(n_conf \ entryCauses, entryConf)}
13:            ADDBRANCHTOEXPPREFIX(visited)
14:        else
15:            FINDEPOMSETS(n_conf, n_sconf, n_causes, visited, cycles, runs)
16:        end if
17:    end for
18:    if sconf is a maximal configuration then
19:        runs ← runs ∪ {conf}
20:        ADDBRANCHTOEXPPREFIX(visited)
21:    end if
22:    REMOVELAST(visited)
23: end procedure
```

configurations, we determine which events are in a loop relation (i.e., events e and e' for which there is a path from e to e' and a path from e' to e).[4]

First, we require the notion of the *directly follows* relation. Two events e_i and e_j are directly following if they are directly causal or if there exists a path from e_i to e_j, where any intermediate event on that path is a τ transition. Formally: $e_i <_d e_j$, iff $e_i < e_j \wedge \forall e_k \in E : e_i < e_k < e_j \implies \lambda(e_k) = \tau$. Moreover, we write $e_i \leqslant_d e_j$ iff $e_i <_d e_j \vee e_i = e_j$.

Two events $e_i, e_j \in C_L$ are in a *causal loop relation*, denoted $e_i \nleftarrow e_j$, iff $e_i <_d e_j \vee \exists e_k \in C_L : e_k$ is cutoff $\wedge\, e_i \leqslant_d e_k \wedge \text{corr}(e_k) \leqslant_d e_j$. Two events $e_i, e_j \in C_L$ are in an *inverse causal loop relation*, denoted $e_i \nrightarrow e_j$, iff $e_j \nleftarrow e_i$.

[4] The completeness and correctness of Algorithm 1 can be derived from the proofs for algorithms for identifying elementary cycles as presented in (Tiernan, 1970; Szwarcfiter and Lauer, 1976).

8.4 Compound event structures

To capture the generic behavior between multiple business processes, we merge the behavior of each input process by combining each input PES prefix unfolding into a *Compound Prime Event Structure (CPES)*. Armed with the PES prefix unfolding and causal loop relation defined above, a CPES can be formally defined as follows:

Definition 8.4.1 (Compound Prime Event Structure)**.** *Let $\mathcal{V} = \{\overline{\mathcal{E}}(N_0), ..., \overline{\mathcal{E}}(N_n)\}$ be a set of PES prefix unfoldings. A compound prime event structure over $\mathcal{V}$ is a quadruple $\mathcal{E}_\mathcal{V} = (E_\mathcal{V}, \xrightarrow{t}, \rightarrow, \overrightarrow{T})$, where:*

- $E_\mathcal{V} = \bigcup_{\overline{\mathcal{E}}(N_i) \in \mathcal{V}} E_i$ *is the combined set of events, with $\overline{\mathcal{E}}(N_i) = (E_i, \leqslant_i, \#_i, \lambda_i)$,*
- $\xrightarrow{t} = \{<_d, \leqslant, \hookleftarrow, \nleftarrow, \#, \|, \nrightarrow, \not\hookrightarrow, \geqslant, >_d\}$ *defines the possible types of relations between the set of events, where:*
  - $e_i <_d e_j$ *denotes a directly follows relation, such that $e_i < e_j \wedge \forall e_k \in E : e_i < e_k < e_j \implies \lambda(e_k) = \tau$.*
  - $e >_d e'$ *denotes an inverse directly follows relation, such that $e' <_d e$,*
  - $e \leqslant e'$ *denotes a causality relation,*
  - $e \geqslant e'$ *denotes an inverse causality relation, such that $e' \leqslant e$,*
  - $e \nleftarrow e'$ *denotes a left side undefined relation, such that for a PES prefix unfolding $\overline{\mathcal{E}}(N_i) : e' \in E_i, e \notin E_i$,*
  - $e \nrightarrow e'$ *denotes a right side undefined relation, such that for a PES prefix unfolding $\overline{\mathcal{E}}(N_i) : e \in E_i, e' \notin E_i$,*
  - $e \hookleftarrow e'$ *denotes a causal loop relation, such that for a PES prefix unfolding $\overline{\mathcal{E}}(N_i) : e, e' \in C_L \wedge (e <_d e' \vee \exists e_k \in C_L : e_k$ is cutoff $\wedge\ e \leqslant_d e_k \wedge corr(e_k) \leqslant_d e')$.*
  - $e \not\hookrightarrow e'$ *denotes an inverse causal loop relation, such that for a PES prefix unfolding $\overline{\mathcal{E}}(N_i) : e' \hookleftarrow e$.*
- $\rightarrow = E_\mathcal{V} \times \xrightarrow{t} \times E_\mathcal{V}$ *is the set of relations existing between all events $e, e' \in E_\mathcal{V}$,*
- $\overrightarrow{T}(e, e') = \{t \mid (e, t, e') \in \rightarrow\} \subseteq \xrightarrow{t}$ *is a function that returns the set of relation types which exist between two events $e, e' \in E_\mathcal{V}$.*

A CPES $\mathcal{E}_\mathcal{V}$ over a set of PES prefix unfoldings $\mathcal{V} = \{\mathcal{E}_0, ..., \mathcal{E}_n\}$ combines the set of events and relations from all PES prefix unfoldings. Since the sets of events of the PES prefix unfoldings in $\mathcal{V}$ do not necessarily fully overlap, additional relations

are introduced for cases where an event exists in one PES prefix unfolding while in the other it does not. Similarly, loop relations are included to differentiate between causal and inverse causal relations originating from a single PES prefix unfolding or multiple PES prefix unfoldings. As a result of the combination, two events $e, e' \in E_{\mathcal{V}}$ of the CPES may maintain multiple relations of different types. That is, the function $\overrightarrow{T}(e, e')$ may return any subset of $\xrightarrow{t}$, except for the empty set.

8.5 Specification Assembly

To obtain an extendible set of rules, which define a family of processes, we take the CPES and use its function $\overrightarrow{T}(e, e')$ to automatically assemble different sets of specifications that together describe the process family.

Definition 8.5.1 (Variability Specification)**.** *Let $\mathcal{E}_{\mathcal{V}} = (E_{\mathcal{V}}, \xrightarrow{t}, \rightarrow, \overrightarrow{T})$ be a combined labeled prime event structure over $\mathcal{V} = \{\mathcal{E}_0, ..., \mathcal{E}_n\}$. A variability specification over $\mathcal{E}_{\mathcal{V}}$ is a set of CTL specifications $V = V_{iresp} \cup V_{iprec} \cup V_{eiresp} \cup V_{eresp} \cup V_{conf} \cup V_{par}$, where:*

- $V_{iresp} = \{AG(e \Rightarrow A[(e \vee s)\ U\ (\bigvee E')]) \mid e \in E_{\mathcal{V}}, \forall e' \in E' \subseteq E_{\mathcal{V}} : (e, <, e') \in \rightarrow\}$ *is the set of immediate response CTL specifications,*
- $V_{iprec} = \{\neg E[\neg(\bigvee E')\ U\ e] \mid e \in E_{\mathcal{V}}, \forall e' \in E' \subseteq E_{\mathcal{V}} : (e,' <, e) \in \rightarrow\}$ *is the set of precedence CTL specifications,*
- $V_{eiresp} = \{AG(e \Rightarrow E[(e \vee s)\ U\ e']) \mid e, e' \in E_{\mathcal{V}} : \overrightarrow{T}(e, e') = \{<\} \vee \overrightarrow{T}(e, e') = \{<, \nleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \hookleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \nleftarrow, \hookleftarrow\}\}$ *is the set of exists immediate response CTL specifications,*
- $V_{eresp} = \{AG(e \Rightarrow EFe') \mid e, e' \in E_{\mathcal{V}} : \overrightarrow{T}(e, e') = \{\leqslant\} \vee \overrightarrow{T}(e, e') = \{<, \leqslant\} \vee \overrightarrow{T}(e, e') = \{\leqslant, \nleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \leqslant, \nleftarrow\}\}$ *is the set of exists response CTL specifications,*
- $V_{conf} = \{AG(e \Rightarrow AG\neg e') \mid e, e' \in E_{\mathcal{V}} : \overrightarrow{T}(e, e') = \{\#\}\}$ *is the set of exclusive execution, or conflict, CTL specifications,*
- $V_{par} = \{EF(e \wedge e') \mid e, e' \in E_{\mathcal{V}} : \overrightarrow{T}(e, e') = \{\|\} \wedge \overrightarrow{T}(e', e) = \{\|\}\}$ *is the set of parallel execution CTL specifications,*

A variability specification V is a set of CTL specifications which defines a family of related BP that offer different features and/or orders of execution. The set consists of multiple subsets. The first two sets, V_{resp} and V_{prec}, define the basic execution paths by enumerating response and precedence relations. The following two

sets, V_{eiresp} and V_{eresp}, define the mandatory paths between events. The fifth set, V_{conf}, defines the conflict relations. Although the CTL specification only defines a conflict in a single direction (i.e., from e to e'), the symmetry of the # relation is maintained within the combined prime event structure, which ensures the inclusion of the specification in the other direction. Finally, the sixth set, V_{par}, defines the parallel execution specifications, i.e., specifications which define that the events in different parallel paths are both executed. Since the parallel execution CTL specification is symmetric in itself, the concurrency relation between events must hold in both directions as well.

Although the set V_{eresp} is required to define paths with variable sections, it can be reduced heavily. In its current form, V_{eresp} contains for each event specifications to every response from every event in any path, even if this path is already described through a set of different specifications in $V_{eresp} \cup V_{ieresp}$. For example, consider the path $\pi = e_0, e_1, ..., e_n$ for which the relations $e_i < e_{i+1}$ are described in V_{eiresp} for $0 \leqslant i < n$ and $i \neq 1$. In other words, every relation is included in V_{eiresp} except $e_1 < e_2$ (e.g. because some other event can be inserted between e_1 and e_2). The set V_{eresp} would then contain rules for every $e_i \leqslant e_j$ relation with $0 \leqslant i < j \leqslant n$ in the compound prime event structure. However, in this case, the rule for the relation $e_0 \leqslant e_n$ would be redundant because it is already described by the set of rules for $e_0 < e_1 \leqslant e_2 < ... < e_n$ in $V_{eresp} \cup V_{ieresp}$. Actually, in this case, all rules in V_{eresp} are redundant except for the rule for $e_1 \leqslant e_2$. We therefore remove from the set V_{eresp} any rule that can be described by a set of relations in $V_{eresp} \cup V_{ieresp}$. This reduction, however, can only be performed after the set of rules described by $V_{eresp} \cup V_{ieresp}$ is known. More formally:

Definition 8.5.2 (V_{eresp} reduction)**.** *Let V_{eresp} be the set of exists response specifications over the combined labeled prime event structure $\mathcal{E}_\mathcal{V} = (E_\mathcal{V}, \overset{t}{\rightarrow}, \rightarrow, \overrightarrow{T})$. A reduced V_{eresp} is a set of CTL specifications $V_{eresp} = V_{eresp} \setminus \{AG(e \Rightarrow EFe') \mid (e, e') \in R_f : (\exists e'' \in E_\mathcal{V} : e \neq e' \neq e'' \wedge (e, e'') \in R \wedge (e'', e') \in R)\}$, with:*

- *$R = R_f \cup R_n$ is the set of causality relations consisting of:*
- *$R_f = \{(e, e') \mid e, e' \in E_\mathcal{V} : \overrightarrow{T}(e, e') = \{\leqslant\} \vee \overrightarrow{T}(e, e') = \{<, \leqslant\} \vee \overrightarrow{T}(e, e') = \{\leqslant, \nleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \leqslant, \nleftarrow\}\}$ the set of eventual causality relations,*
- *$R_n = \{(e, e') \mid e, e' \in E_\mathcal{V} : \overrightarrow{T}(e, e') = \{<\} \vee \overrightarrow{T}(e, e') = \{<, \nleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \hookleftarrow\} \vee \overrightarrow{T}(e, e') = \{<, \nleftarrow, \hookleftarrow\}\}$ the set of immediate causality relations.*

The resulting set of specifications, V, is considered to be the basic set of specifications required to define a coherent, yet variable, family of BP. Other, more

strict, specifications can, however, be included easily. For example, the set $V_{excl} = \{EFe \Rightarrow AG\neg e' \mid e, e' \in E_{\mathcal{V}} : \overrightarrow{T}(e, e') = \{\nrightarrow\}\}$ could be included to enforce the exclusion of unrelated events, i.e., the exclusion of events that did not appear together in any prime event structure $\mathcal{E}_i \in \mathcal{V}$. In this way, specifications can be introduced for each result of $\overrightarrow{T}(e, e')$ as subsets of $\xrightarrow{t}= \{<, \leqslant, \#, \|, \nleftarrow, \nrightarrow\}$. For example, the results of $\overrightarrow{T}(e, e') = \{\#, \nleftarrow\}$, $\overrightarrow{T}(e, e') = \{\#, \nrightarrow\}$, and $\overrightarrow{T}(e, e') = \{\#, \nleftarrow, \nrightarrow\}$ could be used to introduce additional conflict relations. However, as additional specifications are introduced, the variability specification V becomes stricter, allowing a smaller range of variants. The decision of introducing additional specifications is, therefore, left to be considered on a per case basis.

8.6 Discussion

The introduction of the compound labeled prime event structure offers novel insights into the merging of BP. Where others merge BP at the control flow level, we combine BP at the level of relations between events, or structured activities. In doing so, we allow every different set of resulting relations between events to be analyzed and acted upon differently in a generic way. The compound labeled prime event structure can be constructed from either runtime traces of BP or imperatively designed BP. Any number of BP can be merged. For example, the relational behavior of a single BP can be merged from its runtime traces, or the relational behavior of a set of related imperative BP designs can be merged.

The variability specification, in turn, builds upon the compound labeled prime event structure to provide an automated and generic method to define BP in a declarative manner. Each set of relations defined by the compound labeled prime event structure can be assigned a distinct specification which describes the defined relations. However, instead of defining specifications for each possible set, we define the minimal set of relational specifications to support a consistent yet variable declarative definition. Due to the extensible nature of the set, further specifications can then be easily introduced to define a stricter declarative definition. The method can be applied to both single BP and multiple related BP.

This generic definition of relations through CTL specifications is only possible because of the definition of the transition graph (Definition 6.1.2) and its direct support for the parallel behavior of events and the next local occurrence of events. Without support for parallel behavior, the concurrent relation could not be defined without severely limiting BP definitions in other areas (e.g., the definition of ar-

bitrary cycles). More importantly, without support for the next local occurrence of events, the next occurrence would return different results for parallel branches than non-parallel branches, and only future occurrences could be captured correctly. The relations between events, as described by prime event structures, could therefore only be defined in a very limited manner.

CHAPTER 9

Implementation

Everybody [...] should learn how to program a computer because it teaches you how to think.

– Steve Jobs

The Verification extension for Business Process Modeling Tool (VxBPM) is a BP modeling tool which provides the verification techniques presented throughout the previous chapters. The tool is written using the Java programming language and features BPMN BPD design abilities, saving and loading to XPDL format, automated pattern-based model transformation to CPN, automated generation of the Kripke structures required for verification, automated verification using one of multiple model checkers, and transparent visual and textual feedback of the generated models and verification results.

9.1 Features

The VxBPM tool offers the user three main views of the BP, providing the user with transparent feedback on the verification process. The main views include the BPMN BPD modeling view, the CPN view, and the Kripke structure view. Other, minor views, include the textual output of the main views, a list of specifications in textual form, and the raw in- and output of the called model checker.

9.1.1 BPMN BPD design

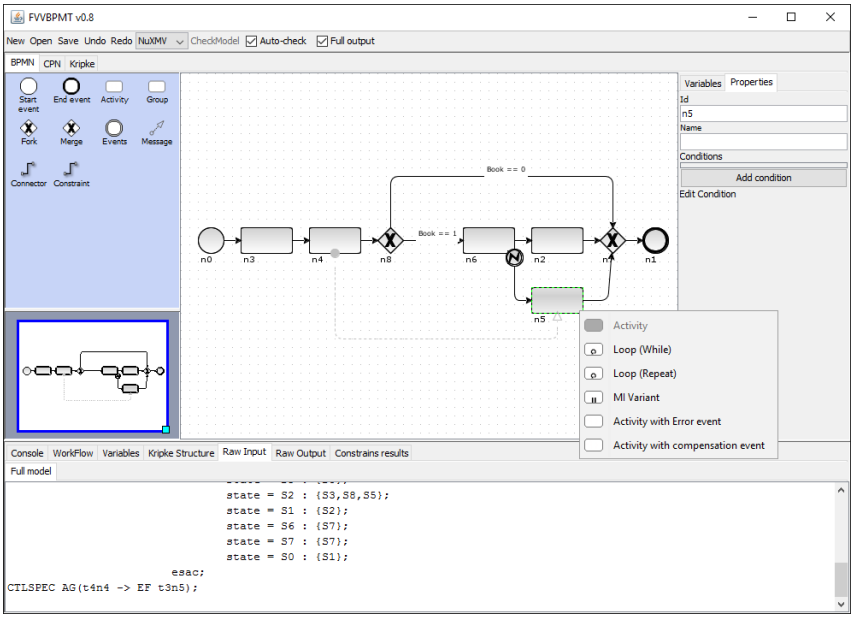


Figure 9.1: VxBPM's BPMN BPD Modeling View.

Figure 9.1 illustrates the BPMN BPD modeling view of the VxBPM tool. The modeling view consists of three areas, the modeling palette, the modeling panel, and the properties panel. The modeling palette on the left contains a list of basic BP elements which can be dragged on the central modeling panel, including the connector-based specifications introduced in Chapter 7. For example, the grey exists response arrow between n4 and n5 may highlight green or red after verification, depending on verification results. Further elements can be accessed through the context menu of each basic element. Finally, the properties of (and variables used by) each element can be changed in the properties panel to the right. Element specific specifications can be added here as well. Possible specifications include both normal specifications as well as conditional specifications relating to conditional (i.e. partial w.r.t. that condition) transition graphs as presented in Definitions 6.5.1 and 6.5.2.

The minor views can be seen at the bottom, and include console output, a list of variables and possible values for verification, textual output of the CPN and (conditional) Kripke structure(s), the raw in- and output of the selected model checker per (conditional) Kripke structure, and a list of specifications and their results per (conditional) Kripke structure.

9.1.2 Petri net Transformation

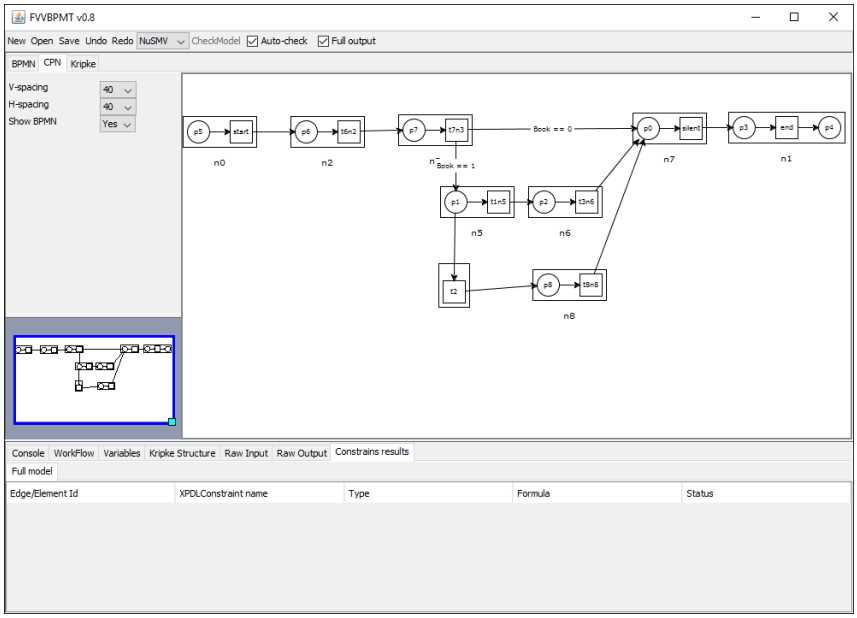


Figure 9.2: VxBPM's CPN View.

The CPN view of the VxBPM tool is depicted in Figure 9.2. This view is automatically populated by a workflow pattern-based transformation of the BPMN BPD depicted in the modeling view. Each element depicted in the BPMN BPD is directly mapped to a set of CPN elements in the CPN view. For the purpose of clarity, the bounding boxes of the mapped elements have been enabled. In addition to the pattern-based transformation, groups, pools, and swimlanes are converted using an additional labeling function over the CPN elements as presented in Definition 6.4.1. Since the view is populated automatically, only diagram related settings can be changed. As a result, manual modeling of the CPN is also not possible. For further details on the workflow pattern-based transformation, see Appendix B.

9.1.3 Kripke structure Generation

Figure 9.3 depicts the Kripke structure view of the VxBPM tool. The Kripke structure view is populated automatically through an implementation of the transition graph as presented in Definitions 6.1.2 and 6.4.2. In case of conditional specifications, a separate Kripke structure is generated for each set of different conditions and is paired with the specifications carrying that condition. Specifications that do not carry conditions are evaluated on the full model (i.e. the transition graph

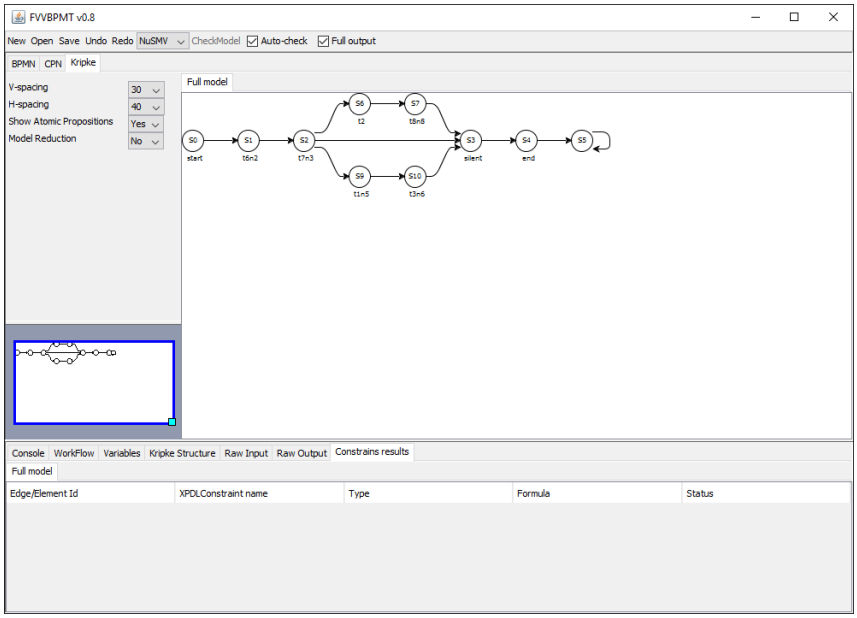


Figure 9.3: VxBPM's Kripke structure View.

that is not limited by any condition). Each of the generated Kripke structures can be inspected in this view. In addition, the user is able to specify whether reduction is applied. When reduction is applied, each Kripke structure is reduced by removing, from each state, those atomic propositions that are not evaluated by any specification paired with this specific Kripke structure and, subsequently, applying reduction through equivalence with respect to stuttering as detailed in section 6.7.

9.1.4 Model Verification

The actual verification of the generated Kripke structures is performed by one of multiple model checkers. Although the specifications can, in theory, be directly evaluated on the related Kripke structures, model checkers often do not allow a direct implementation of labeled transition systems. Model checkers verify systems defined in the modeling language of the model checker. These modeling languages range from forms of code similar to programming languages, which are then translated to transition systems, to simple transition systems themselves. To provide verification results using the presented models, the Kripke structure is automatically translated into the modeling language of the selected model checker which subsequently verifies the specifications paired with that Kripke structure.

NuSMV2 (Cimatti et al., 2002) and NuXMV (Cavada et al., 2014) are symbolic model checkers supporting both CTL and LTL based formulas. NuSMV2 is an extension to SMV, which first implemented model verification based on BDD. NuXMV extends NuSMV2 with SAT- and SMT-based algorithms. The modeling language of both NuSMV2 and NuXMV allows the user to specify a state-based labeled transition system for direct verification, which is highly preferred when implementing a Kripke structure. The conversion from a Kripke structure to the modeling language of these model checkers is explained, such that temporal logic formulas can be verified automatically.

For the purpose of implementing a Kripke structure, the modeling language of NuSMV2 and NuXMV consists of a module with four distinct blocks. The first, *VAR*, defines the states of the transition system. The second, *DEFINE*, specifies each variable and assigns them to the states in which they are true. The third, *ASSIGN*, first defines the set of initial states and then specifies next states using the *next* function. In the final block, a set of temporal logic formulas and fairness conditions is listed. In case of the implementation of a Kripke structure $M = (S, I, R, L)$ with atomic propositions AP, the *VAR* block specifies all states $s \in S$, the *DEFINE* block specifies each variable $v \in AP$ and assigns it to states $s \in S$ for which $v \in L(s)$. The *ASSIGN* block lists the initial states $s \in I$ and then specifies each relation $(s, s') \in R$ as *next(s) :=* s'. Finally, a list of temporal logic formulas and fairness conditions is included. In order to avoid issues with loops, justice fairness conditions are included. Justice conditions are considered to be true infinitely many times in fair paths. As such, infinite loops are omitted using justice conditions which state that loops may not repeat an infinite number of times.

For example, Listing 9.1 contains the NuSMV2/NuXMV code of the room booking process detailed in the previous views of Figures 9.1 through 9.3. Note that the model reduction step was not applied at the Kripke structure view, resulting in a full output of the model with all states and atomic propositions.

Listing 9.1: Model Checker Code for the Room Booking Process.

```
MODULE main
  VAR
      state:{S0,S1,S2,S3,S4,S5,S6,S7,S8,S9};
  DEFINE
      start  := ( state = S0 );
      t5n3   := ( state = S1 );
      t4n4   := ( state = S2 );
      t1n6   := ( state = S3 );
```

```
    t2     := ( state = S8 );
    t3n5   := ( state = S9 );
    t6n2   := ( state = S4 );
    silent := ( state = S5 );
    end    := ( state = S6 );
ASSIGN
  init(state) :=   {S0};
  next(state) :=
    case
      state = S0 : {S1};
      state = S1 : {S2};
      state = S2 : {S3,S8,S5};
      state = S3 : {S4};
      state = S8 : {S9};
      state = S9 : {S5};
      state = S4 : {S5};
      state = S5 : {S6};
      state = S6 : {S7};
      state = S7 : {S7};
    esac;

CTLSPEC AG(t4n4 -> EF t3n5);
```

As the models are passed to the selected model checker, results of the verification process are interpreted automatically by the VxBPM tool and displayed both in a textual and visual manner. Textually, the results for each specification is listed per (conditional) Kripke structure. Visually, each element related to a specification (i.e. the connector or (group) of activities) is colored green when evaluated true, red when evaluated false, or grey when failed to be evaluated due to an error.

9.2 Extensibility

The VxBPM tool is fully configurable and extensible in four directions: (1) support for different modeling specifications, (2) support for different or alternative pattern-based CPN transformations, (3) support for additional or alternative verification specifications and languages, and (4) support for additional model checking tools which may offer different verification features and/or specification languages.

9.2.1 Modeling Specifications

VxBPM directly supports the BPMN modeling specification (see Appendix A). Different specifications, however, can be introduced through a set of eXtensible Mark-

up Language (XML) documents and element shapes describing the input elements (*input-elements.xml*), their styles (*bpmn-style.xml*), and palette window elements (*palet-elements.xml*). For example, the XML regarding the BPMN activity element is depicted in Listings 9.2 and 9.3. Listing 9.2 describes the basic element, its palette information, shape, and the CPN workflow pattern it maps to, while Listing 9.3 describes the complex elements listed in the context menu (See Figure 9.1).

Listing 9.2: input-elements.xml for the BPMN Activity Element.

```
<inputElement id="activity">
  <name>Activity</name>
  <CPNElement>Activity</CPNElement>
  <BPMNName>activity</BPMNName>
  <width>60</width>
  <height>30</height>
  <connections maxIncoming="1" maxOutgoing="1" minIncoming="1"
      minOutgoing="1"/>
  <paletIconPath>Task.png</paletIconPath>
  <shapePath>Task.shape</shapePath>
  <styleProperties></styleProperties>
  <genId>n{x}</genId>
  <name visible="true" editable="true"></name>
</inputElement>
```

Listing 9.3: palet-elements.xml for the BPMN Activity Element.

```
<paletElement>
  <name>Activity</name>
  <paletIconPath>Task.png</paletIconPath>
  <inputElements>
    <inputElement>activity</inputElement>
    <inputElement>loopWhile</inputElement>
    <inputElement>loopRepeat</inputElement>
    <inputElement>miVariant</inputElement>
    <inputElement>activityError</inputElement>
    <inputElement>activityIntermediateEventCompensation
      </inputElement>
  </inputElements>
</paletElement>
```

9.2.2 Pattern Transformations

The modeling specification elements used in VxBPM's modeling view are directly transformed in the CPN view through a workflow pattern mapping (See Appendix B).

Although VxBPM supports most patterns directly, other alternative patterns can be supported by adding additional workflow pattern mappings as XML documents and enabling that mapping in the *input-elements.xml* document (Listing 9.2).

Listing 9.4: Workflow Pattern Mapping of the BPMN activity Element.

```
<CPNElement id="Activity">
  <incomingElements>
    <incomingElement>p1</incomingElement>
  </incomingElements>
  <outgoingElements>
    <outgoingElement>t1</outgoingElement>
  </outgoingElements>

  <incomingMessageElement>t1</incomingMessageElement>
  <outgoingMessageElement>t1</outgoingMessageElement>
  <eventElementId>p1</eventElementId>

  <places>
    <place id="p1" x="0" y="0"/>
  </places>
  <transitions>
    <transition id="t1" x="1" y="0" name="t{x}{id}"/>
  </transitions>
  <arcs>
    <arc id="a1" from="p1" to="t1"/>
  </arcs>
</CPNElement>
```

For example, Listing 9.4 contains the XML document defining the workflow pattern used to map BPMN activities to CPN elements. The document lists the places, transitions, and arcs used to describe the pattern, their relevant placement, and interactions with other patterns (e.g. in- or outward sequence or message flows).

9.2.3 Verification Specifications

As different model checkers support different specification languages, the specifications offered by VxBPM differ per enabled model checker. All specifications offered by VxBPM are described in two XML documents. The first document, *specification-languages.xml*, lists all specifications per specification language (e.g. CTL or LTL) and type (i.e. those directly applied to elements or those applied through connectors). The second document, *arrows.xml*, describes the visualization of each specification listed. For example, Listings 9.5 and 9.6 contain the XML definition and visualization of the CTL response specification (Specification 7.3.1.ii).

Listing 9.5: specification-languages.xml for the CTL Response Specification.

```
<specificationLanguages>
  <specificationLanguage>
    <id>CTL</id>
    <name>CTL</name>
    <constraints>
    ...
      <!--alwaysFinally-->
      <constraint id="CTL_alwaysFinally">
        <arrowId>alwaysFinally</arrowId>
        <formulas>
          <formula>AG($p -> AF $q)</formula>
        </formulas>
      </constraint>
    ...
    </constraints>
  </specificationLanguage>
</specificationLanguages>
```

Listing 9.6: arrows.xml Visualization for the Response Specification.

```
<!--AlwaysFinally-->
<arrow id="alwaysFinally" name="AlwaysFinally" dashed="true">
  <sourceShapes>
    <shape>eclipseFilled</shape>
  </sourceShapes>
  <centerShapes>
  </centerShapes>
  <targetShapes>
    <shape>arrowEastFilled</shape>
  </targetShapes>
</arrow>
```

9.2.4 Model Checkers

VxBPM supports both the NuSMV2 (Cimatti et al., 2002) and NuXMV (Cavada et al., 2014) model checkers , which support a wide range of verification techniques and specification languages. The model checkers, their locations, and supported specification languages are listed in the *model-checkers.xml* document (Listing 9.7). Other model checkers, which may support different techniques and specification languages can, however, easily be supported by adding an additional model checker to the *model-checkers.xml* document, implementing the *AbstractChecker*

class for that model checker, and implementing the *Formula* class for any unimplemented specification languages.

Listing 9.7: model-checkers.xml for the NuXMV Model Checker.

```
<modelCheckers>
  <ModelChecker>
    <id>NuXMV</id>
    <name>NuXMV</name>
    <location>c:/NuXMV.exe</location>
    <specificationLanguages>
      <specificationLanguage format="CTLSPEC {c};">CTL</
          specificationLanguage>
      <specificationLanguage format="LTLSPEC {c};">justice</
          specificationLanguage>
      <specificationLanguage format="JUSTICE {c};">LTL</
          specificationLanguage>
    </specificationLanguages>
  </ModelChecker>
  ...
<modelCheckers>
```

CHAPTER 10

Evaluation

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

– Isaac Asimov

The presented approach is not only able to return valuable compliance results, but it also allows greater insight into concurrently executing branches – while limiting model size and time to generate. To demonstrate this, we analyze its expressive power and performance.

In addition, we evaluate the approach against the requirements of Chapter 5 and consider the different case studies presented in Chapter 4.

10.1 Expressive Power

The expressive power of the transition graph while using CTL is evaluated through its application on four workflow patterns (van der Aalst et al., 2003b) and compared to relevant related work, i.e., formal cyclic preventative approaches. The approaches are categorized into three types of graphs: the *reachability graph* (Huber et al., 1986), the *transition occurrence graph* (i.e. a graph where transitions are included upon their occurrence) similar to the approaches of (Liu et al., 2007) and (Foster et al., 2003), and a version of the reachability graph where states are labeled with

enabled transitions (i.e., the *enabled graph*), similar in effect to (Esparza, 1993). The control-flow constructs used for comparison are deferred choice, exclusive choice, parallel split, and interleaved routing.

Figure 10.1 depicts the deferred choice and exclusive choice construct together with their representations using the four different graphs. Although the deferred and exclusive choice are different constructs, both fulfill a similar role; a choice of paths depending on some event or condition. After the occurrence of transition a, either transition b or c occurs. This functionality can indeed be seen in both the transition and occurrence graphs. The enabled graph, however, poses a problem when representing the deferred choice. After transition a is enabled, both b and c are

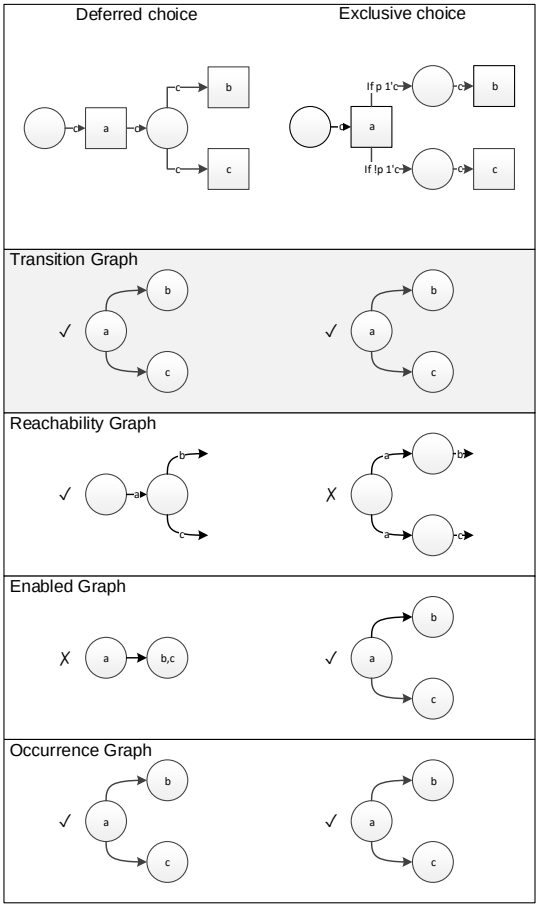


Figure 10.1: Graph comparison of the deferred and exclusive choice constructs.

enabled simultaneously. Without a way to foresee which of these two transitions actually occurs, verification of any logic formula on this graph can only guarantee the enabled state of transitions and never its occurrence. Although the reachability graph, in principle, produces correct graphs where after the occurrence of transition a either b or c occurs, its representation of the exclusive choice pattern does introduce issues with verification of branching time temporal logics. For example, the response formula $AG(a \Rightarrow EFb)$ (a is eventually followed by b in some path) would evaluate to false because the occurrence of a on the lower branch which is followed only by c. While the formula would evaluate true for the upper branch, it would evaluate false in its entirety due to the false result of the lower branch. This undesired behavior is introduced because the reachability graph is treating the oc-

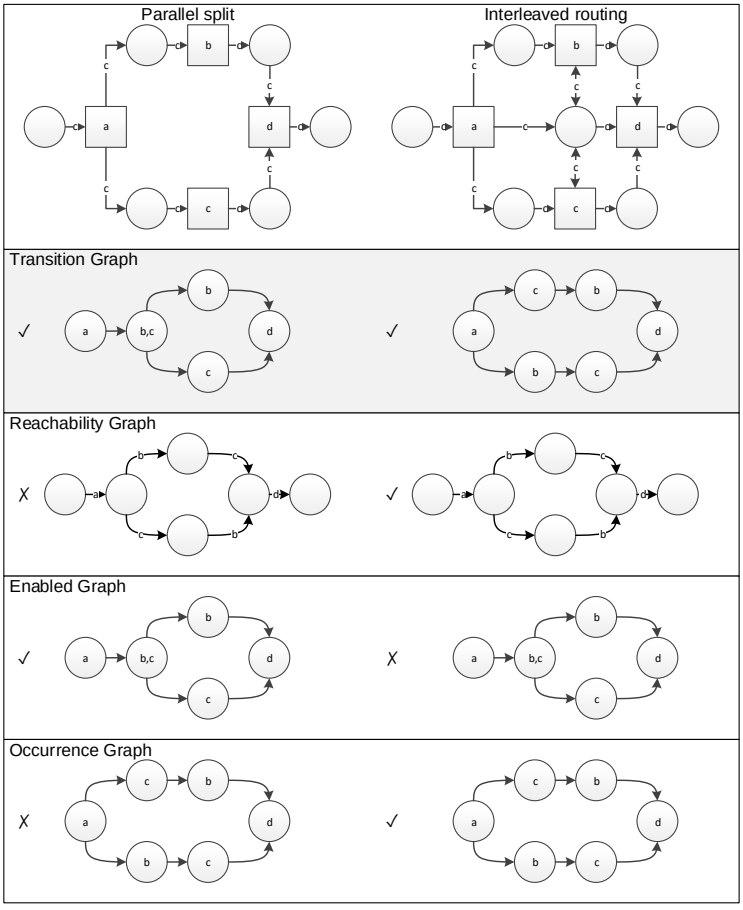


Figure 10.2: Graph comparison of parallel split and interleaved routing constructs.

currence of transition a with the boolean p evaluating true as a different occurrence of transition a with p evaluating false. In reality, when looking at the pattern, it is not the transition that differs, but the conditions on the outward arcs. The occurring transition executes the same with different values of p. Indeed, when mapping this pattern to a BPMN model, activity a would precede the exclusive choice gate which then splits into two paths leading to either activity b or c. A functionality which the reachability graph clearly does not represent correctly.

Figure 10.2 depicts the parallel split/join and interleaved routing patterns and their representations using the four different graphs. Although the two patterns look similar, their behavior is different. Where the parallel split/join pattern splits into two parallel executing branches which only synchronize at the join, the interleaved routing performs some form of synchronization during execution of each branch such that no two transitions can occur in parallel, but does so without inferring an explicit occurrence order between branches. In essence, the occurrence of the transitions on the different branches is interleaved. The functionality of these two patterns can indeed be seen in the transition graph. When representing the parallel split using the transition graph, transition b and c start occurring in parallel, after which either b or c has finished occurring (i.e. has occurred) and the other remains in an occurring state until it also has finished occurring and synchronizes into d. Furthermore, when representing the interleaved routing pattern, the transition graph indeed linearizes the occurring transitions on each branch as specified. For the remaining graphs one immediately notices an issue of expressive power. Indeed, the remaining graphs each represent both patterns with the exact same graph, even through the patterns behave in very different ways. For the enabled graph this means that the interleaved routing construct is not being linearized because, naturally, both transition b and c are enabled even though they may not occur simultaneously. In case of both the reachability graph and the occurrence graph, the parallel split/join construct is linearized as well. As the occurrence of transitions b and c is being linearized, any parallel occurrence information as well as local next occurrence information is lost. Where the transition graph can be used to verify that transitions b and c can occur in parallel by verifying the existence of both labels at one state (i.e. $EF(b \wedge c)$), the reachability and occurrence graphs can only be used to verify whether transitions b and c occur until d (i.e. $AG(a \Rightarrow A[(b \vee c)\, U\, d])$. Even when using LTL including always and exists path mechanisms, specifications can only verify that sometimes transition b occurs before transition c and vice versa (i.e. $\exists G(c \Rightarrow Fb)$ and $\exists G(b \Rightarrow Fc)$), a specification which could also be satisfied by a simple loop. The same is true for the local next occurrence, which can be ver-

ified using the transition graph by verifying whether a label holds until the next (i.e. $AG(b \Rightarrow E[b\ U\ d])$). A specification which correctly holds for the parallel split pattern and not for the interleaved routing pattern when considering the transition graph. Again, the other graphs fail to capture the difference.

As the transition graph can be used to verify parallel and local next occurrences, it bridges a clear gap in the expressive power required for verification of BP.

10.2 Performance Evaluation

The performance of the approach was first evaluated by executing an implementation of Definition 6.1.2 on the case study. Performance tests were attained using a system with an Intel Core I7-4771 CPU at 3.50 GHz, 32 GB of memory, running Windows 7 x64 and Java 7. The results are shown in Table 10.1.

Table 10.1: Conversion and reduction algorithm results of the telecom processes.

Kripke structure			**Red. Kripke structure**			**Performance**	
$\|S\|$	$\|R\|$	$\|AP\|$	$\|S\|$	$\|R\|$	$\|AP\|$	Conversion	Reduction
33	49	26	24	35	15	29 ms	1 ms

The case consists of 33 states and 48 transitions in its original form. After reduction with respect to the relevant AP used in the formulas, the amount of states is reduced by 27% and the transitions by 29%. However, it is clear from the table that the required execution time for the proposed conversion and reduction is negligible when applied to the real-life case. The customer support process, although consisting of a realistic amount of services, does not comprise excessively complex control-flow structures. In fact, it is (like many compositions), predominantly sequential. As such, the effect on performance remains rather limited.

To test the algorithm with respect to performance under increasing concurrency within parallel branches, the approach was subsequently evaluated by executing an implementation of Definition 6.1.2 on artificial service compositions (Groefsema and van Beest, 2015; Groefsema et al., 2016). These artificial compositions were specifically generated for performance evaluation purposes by specifying a gate type, number of branches, and branch length. The results can be found in Tables 10.2 and 10.3.

Table 10.2 displays information on the performance and results of the conversion process. Its columns describe the case number, the composition (containing se-

Table 10.2: Conversion algorithm results of compositions with n branches of m activities.

	Composition			Kripke structure			Performance
#	Type	n	m	$\|S\|$	$\|R\|$	$\|AP\|$	Conversion
1	SEQ	1	5	8	8	7	3 ms
2	XOR	2	5	13	14	12	3 ms
3	XOR	3	5	18	20	17	3 ms
4	XOR	4	5	23	26	22	3 ms
9	AND	2	5	38	63	12	4 ms
10	AND	3	5	218	543	17	27 ms
11	AND	4	5	1298	4323	22	96 ms
5	SEQ	1	50	53	53	52	5 ms
6	XOR	2	50	103	104	102	12 ms
7	XOR	3	50	153	155	152	16 ms
8	XOR	4	50	203	206	202	22 ms
13	AND	2	50	2603	5103	102	102 ms
14	AND	3	50	132653	390153	152	1.853ms
18	AND	4	50	6765203	26530203	202	184.758 ms
12	AND	4	8	6563	23331	34	214 ms
15	AND	3	80	531443	1574643	242	10.903 ms
17	AND	4	30	923523	3574923	122	20.345 ms
19	AND	5	15	1048578	4915203	77	22.444 ms
16	AND	3	100	1030303	3060303	302	25.023 ms
20	AND	6	10	1771563	9663063	62	44.364 ms
21	AND	7	8	4782971	29760699	58	158.008 ms

quence/exclusive/parallel branching, the number of branches n, and number of services per branch m), the resulting Kripke structure (number of states S, relations R, and atomic propositions AP), and the performance of the conversion algorithm.

Test cases 1-8 demonstrate that sequential compositions and compositions including exclusive paths are of no concern to the conversion performance, as they are converted within milliseconds. However, processes including parallel regions introduce an increased complexity. This increased complexity is introduced due to the interleaving of concurrent services on parallel branches. Although the interleaving is highly efficient due to the lack of complete linearization, it does produce $\prod_{i=1}^{n}(m_i + 1)$ states, where m_i is the length of branch number i out of n branches (for equal branch lengths the complexity is $(m+1)^n$). This is confirmed by test cases 9-11, which show that parallel interleavings of equal sizes to the exclusive compositions (cases 2-4) are converted within 96 milliseconds. When increasing the length of the branches in test cases 13-14, and 18 from 5 to 50 services, the time to convert is increased to 102 milliseconds for two branches, 1.8 seconds for three branches, and 185 seconds for four branches. Finally, test cases 15-17 and 19-21 demonstrate that compositions with extremely large parallel sections of 56 to 300 services are con-

Table 10.3: Reduction algorithm results of compositions with n branches of m activities after reduction by 50% of randomly chosen atomic propositions.

	Reduced Kripke structure			Performance
#	$\|S\|$	$\|R\|$	$\|AP\|$	Reduction
1	6 (25%)	6 (25%)	4	0 ms
2	10 (23%)	11 (21%)	6	0 ms
3	15 (17%)	17 (15%)	9	0 ms
4	19 (17%)	22 (15%)	11	0 ms
9	15 (61%)	22 (65%)	6	1 ms
10	67 (69%)	148 (73%)	9	10 ms
11	403 (69%)	1245 (71%)	11	78 ms
5	37 (30%)	37 (30%)	26	2 ms
6	76 (26%)	77 (26%)	51	2 ms
7	114 (25%)	116 (25%)	76	4 ms
8	151 (26%)	154 (25%)	101	5 ms
13	1409 (46%)	2741 (46%)	51	90 ms
14	50823 (62%)	148319 (62%)	76	704 ms
18	1915203 (72%)	7454605 (72%)	101	37.294 ms
12	2523 (62%)	8716 (63%)	17	100 ms
15	207363 (61%)	612122 (61%)	121	2.297 ms
17	241227 (74%)	924885 (74%)	61	4.481 ms
19	168003 (84%)	762205 (84%)	39	6.714 ms
16	453603 (66%)	1344692 (66%)	151	3.548 ms
20	396903 (88%)	2100427 (88%)	31	12.519 ms
21	774147 (84%)	4577290 (85%)	29	43.765 ms

verted in 10 to 158 seconds. However, as the number of parallel branches increases, there is a severe limit on the length of the branches supported. This is due to the fact that the number of branches n is the largest factor of the complexity formula. Although extremely large interleavings require several minutes to compute, this is negligible when verifying pre-runtime.

Next, we evaluate the model reduction algorithm. Model reduction is achieved through the removal of atomic propositions that are not relevant to specifications and equivalence with respect to stuttering. In order to evaluate the degree of reduction, we randomly remove 50% of atomic propositions of the cases listed in Table 10.2 before applying the reduction algorithm. Table 10.3 displays information on the resulting reduced Kripke structure (number of states S and percentage of original, relations R and percentage of original, and number of remaining atomic propositions AP) and the time it took to reduce the model. Note that reduced Kripke structure sizes may vary due to the random removal of atomic propositions.

Test cases 1-8, which contain sequential compositions and compositions including exclusive paths, are of no concern to the performance of the reduction algorithm either. These compositions are reduced within 0 to 5 milliseconds. Compositions

including parallel regions display the greatest reduction results. When increasing the number of parallel branches, reduction effects increase accordingly. Test cases 9-11 demonstrate that parallel interleavings of average sizes are reduced within 78 milliseconds. Increasing the length of the branches from 5 to 50 services increases the time to reduce to 90 milliseconds, 704 milliseconds, and 37 seconds for compositions with two, three, and four branches respectively.

The effect of model reduction displays varying results. Because sequential compositions generate relatively simple Kripke structures, model reduction shows limited effects with reductions of 15% to 30%. For sequential compositions, the worst case reduction with less than half of the AP removed is 0%. Processes with parallel interleaved paths, however, show a much larger effect with a 46% to 72% reduction. In this case, while the size of the Kripke structures increases with additional branches, model reduction naturally gains increased effect due to the particular applied interleaving. Because transition occurrence is interleaved with concurrent information, and not entirely sequentialized, reduction with regard to equivalence under stuttering gains increased effect. With each removed atomic proposition, a significant amount of interleaved states is reduced. Compositions including extremely large sections of parallel interleavings demonstrate reductions between 61% and 88%.

Although the resulting interleaving is responsible for a state explosion, this is of little concern for compositions with average and even large parallel areas. Normal sized compositions are generated and ready to be verified instantly. Extremely large parallel areas do introduce increased complexity. However, when a limited number of atomic propositions from these areas is used, these still can be reduced significantly and used for verification pre-runtime. Furthermore, when a model does turn out to be too large for model checking, our approach allows to split formulas into multiple sets, each resulting in a much smaller reduced Kripke structure. Each formula set can then be checked on its respective Kripke reduction, which results in a significant performance gain. In this respect, the size of the reduced model is directly related to the number of atomic propositions used within the set of formulas. As a result, the verification is possible even for business processes with extremely large amounts of concurrency.

10.3 Requirements Analysis

To evaluate the proposed approach, we survey the requirements detailed in Chapter 5 and analyze whether and how each requirement is satisfied. Table 10.4 lists the

Table 10.4: Requirements analysis of the Presented Approach.

Requirement	
Model	
5.1.1 Unstructured processes	✗
5.1.2 Parallel branching	✓
5.1.3 Arbitrary cycles	✓
5.1.4 Intermediary events	✓
Soundness	
5.2.1 Reachability	✓
5.2.2 Termination	✓
5.2.3 Proper Completion	✓
Compliance	
Occurrence	
5.2.4.i Absence	✓
5.2.4.ii Universality	✗
5.2.4.iii Existence	✓
5.2.4.iv Bounded existence	✗
Ordering	
5.2.5.i Precedence	✓
5.2.5.ii Response	✓
Resource specifications	
5.2.6.i Always performed by	✓
5.2.6.ii Performed by	✓
5.2.6.ii Never performed by	✓
Data conditions	
5.2.7.i Holds always	✓
5.2.7.ii Holds when	✓

Requirement	
Declarative Variability	
Inclusion	
5.2.11.i Include	✓
5.2.11.ii Prerequisite	✓
5.2.11.iii Substitution	✓
5.2.11.iv Corequisite	✓
5.2.11.v Causal selection	✓
Exclusion	
5.2.12.i Exclude	✓
5.2.12.ii Exclusion	✓
5.2.12.iii Admittance	✓
5.2.12.iv Exclusive choice	✓
Execution	
5.2.13.i Execute	✓
5.2.13.ii Requirement	✓
5.2.13.iii Replacement	✓
5.2.13.iv Backup	✓
5.2.13.v Causal execution	✓
Option	
5.2.14.i Option	✓
5.2.14.ii Avoidance	✓
Scheduling	
5.2.15.i Response	✓
5.2.15.ii Exists reponse	✓
5.2.15.iii Immediate response	✓
5.2.15.iv Exists immediate response	✓
5.2.15.v No response	✓
5.2.15.vi Exists no response	✓
5.2.15.vii No immediate response	✓
5.2.15.viii Exists no immediate response	✓
5.2.15.ix Coexecution	✓
5.2.15.x Cooccurrence	✓
5.2.15.xi Parallel execution	✓
5.2.15.xii Exclusive execution	✓

relevant requirements for both the model and specifications related to soundness, compliance, and variability. Requirements that are directly supported are marked with a ✓, while those that are not directly supported are marked with a ✗.

When evaluating the requirements related to the model, we notice that all requirements are satisfied except for the support of unstructured processes. Although most forms of unstructured processes do not present difficulties, there is one exception. In theory, the proposed model (i.e. the transition graph, see Definition 6.1.2) can become infinite. That is, in situations where a place of the CPN receives an infinite number of tokens over an infinite number of occurrences of the same set of transitions, the model will require an infinite number of states and relations as well. For example, a BP with a parallel branch leading into a loop that merges before the previously encountered parallel fork will produce such a result, and will not produce a model for verification. However, this can be easily overcome by collapsing the infinite amount of tokens at places into a single infinite value.

The requirements related to soundness specifications, on the other hand, have all been satisfied. However, the reachability specification presents some downsides. In order to evaluate the reachability of each structured activity, a reachability specification must be included for every structured activity. As a result, all atomic propositions related to all structured activities are being evaluated upon the model, denying the model the possibility of significant reduction. For the smaller models this is not an issue. Large models with large parallel regions, however, will produce a significant amount of states for verification.

The requirements related to the compliance specifications are all satisfied, except for those evaluating universality and bounded existence. Although both specifications can certainly be evaluated using the presented approach, their evaluation would not give the user valuable insights into the workings of the BP. This issue directly relates to the type of model that was proposed. That is, the transition graph (Definition 6.1.2) is an event-based model (i.e., it captures the event of structured activities occurring as transitions in the CPN) and not a state-based model (i.e, a model capturing the underlying data as structured activities occur). When evaluating universality, one seeks to verify whether the value of a piece of data remains universally true, and not whether a set of structured activities keep occuring universally. Bounded existence, on the other hand, seeks to evaluate whether one structured activity occurrence exists multiple times. Since the transition graph treats structured activity occurrences as single states, among which cycles are allowed, the bounded existence will not be able to evaluate its bound (i.e., the bound is always infinite).

The requirements related to variability specifications are all satisfied. Since the proposed approach is inherently declarative based, only those requirements related to declarative variability were evaluated.

The evolutionary requirements are all not directly satisfied. Most are, however, satisfied as soon as some form of versioning is introduced. In this case, once a new version of a template process is detected, variants must be reverified for compliance. The evolution of process instances (Requirement 5.3.3), however, is not satisfied since illegal behavior could already have been performed, as well as due to issues caused by the so called dynamic change bug (Ellis et al., 1995). In both cases rollbacks may be required to undo unwarranted behavior.

10.4 Case Study (continued)

In order to evaluate the proposed techniques, they are applied to the three case studies presented in Chapter 4. The first case study, the telecommunications customer support process, was used throughout Chapter 6 as a running example. The case is continued and finally verified for compliance with the specifications obtained from the TCP code of conduct. The second case, local Dutch eGovernment, is used to evaluate the automated generation of specifications, as well as the complexity of the resulting specification set. And finally, the third case, bouncer registration, is used to evaluate the application of the proposed techniques on collaborative BP.

10.4.1 Case 1: Telecommunications Customer Support

Listing 10.1 contains the code passed to the model checker when considering the customer support process as presented in Section 4.1. This code is obtained by taking the CPN depicted in Figure 4.2, generating the transition graph to form Figure 6.1, reducing the model to get Figure 6.6, and translating that to NuSMV2/ NuXMV input code. In addition, the specifications listed in Section 4.1 were translated to CTL in Table 7.2 and included for verification. Finally, a fairness condition is included to dismiss infinite loops from the set of valid verification paths.

Listing 10.1: Model Checker Code for the Customer Support Process.

```
MODULE main
  VAR
    state:{S0, S1, S2, S4, S5, S6, S7, S9, S10,S11,S16,S17,
           S18,S19,S20,S21,S22,S26,S28,S30,S31,S35,S37,S43};

  DEFINE
```

```
  start := ( state = S0  );
  t0    := ( state = S1  );
  t3    := ( state = S4  );
  t5    := ( state = S9  );
  t6    := ( state = S10 );
  t10   := ( state = S43 );
  t11   := ( state = S18 );
  t13   := ( state = S20 );
  t14   := ( state = S21 );
  t19   := ( state = S16 );
  t21   := ( state = S31 ) | ( state = S37 );
  t22   := ( state = S28 ) | ( state = S35 ) | ( state = S37 );
  t23   := ( state = S17 );
  t24   := ( state = S26 );
  end   := ( state = S5  );
  loop  := ( state = S18 ) | ( state = S19 );

ASSIGN
  init(state) := {S0};
  next(state) :=
    case
      state = S0  : {S1};
      state = S1  : {S2};
      state = S2  : {S4,S9};
      state = S4  : {S5,S7};
      state = S5  : {S6};
      state = S6  : {S6};
      state = S7  : {S5};
      state = S9  : {S10};
      state = S10 : {S11};
      state = S11 : {S16,S43};
      state = S16 : {S17,S28,S30,S35};
      state = S17 : {S18,S26};
      state = S18 : {S19,S21};
      state = S19 : {S18,S20};
      state = S20 : {S21};
      state = S21 : {S22};
      state = S22 : {S5};
      state = S26 : {S18};
      state = S28 : {S17};
      state = S30 : {S31};
      state = S31 : {S17};
      state = S35 : {S30,S37};
      state = S37 : {S28,S31};
      state = S43 : {S18};
    esac;

FAIRNESS !loop;
```

```
CTLSPEC !E[!(t11 | t3) U end] & AF(end);
CTLSPEC AG(t10 -> AF(t13 | t14));
CTLSPEC AG(t5 -> A[t5 U t6]);
CTLSPEC AG(t5 -> AF(t14));
CTLSPEC AG(t5 -> EF(t24));
CTLSPEC !EF(t21 & t23) & !EF(t22 & t23);
CTLSPEC AG(t13 -> A[t13 U t14]);
CTLSPEC EF(t21 & t22);
CTLSPEC AG(t19 -> EF(t23));
```

The resulting model is subsequently offered to NuSMV2/NuXMV in order to automatically verify its compliance with the rules specified in Section 4.1. Table 10.5 lists the results of model checking. The first two specifications returned false, signifying compliance issues. These two rules state that a complaint can result in multiple successive offers and that complaints should always result in an offer unless there is no contact with the complaining party. When inspecting the process, it is indeed clear that offers only take place in certain branches of the process where there has been contact with the customer. Additionally, it is true that the customer is informed of an offer only once, and that when this offer is rejected customer acceptance is sought multiple times for this offer without any possibility of revision of the offer.

Table 10.5: TCP Compliance Rules as CTL specifications.

#	CTL Formula	Result
1.	!E[!(t11 ∨ t3) U end] AF(end)	✗
2.	AG(t10 ⇒ AF(t13 ∨ t14))	✗
3.	AG(t5 ⇒ A[t5 U t6])	✓
4.	AG(t5 ⇒ AF(t14))	✓
5.	AG(t5 ⇒ EF(t24))	✓
6.	!EF(t21 ∧ t23) !EF(t22 ∧ t23)	✓
7.	AG(t13 ⇒ A[t13 U t14])	✓
8.	EF(t21 ∧ t22)	✓
9.	AG(t19 ⇒ EF(t23))	✓

10.4.2 Case 2: Local Dutch e-Government

The automated generation of specifications is realized using an implementation of the algorithms and definitions presented in Chapter 8 which take as input a set of

process models in PNML format.[1] Its output is a set of unique rules, as formally defined in Definition 8.5.1 and 8.5.2. Using this implementation, we conduct a quantitative evaluation of the proposed method, using three real-life process models from local e-Government in the Netherlands (Section 4.2). We assess the performance and number of produced variability rules for each category.

Rule Sets

To evaluate the variability rules generated by our method, we use the model of Municipality A as the general case, where Municipality B and Municipality C are variants. Table 10.6 provides an overview of the rules generated for the individual models. For each model, all existing response relations can be covered by V_{eiresp}, as there is only one model involved and, hence, no variability. Consequently, V^{red}_{eresp} equals 0 for each individual model. Furthermore, none of the models have concurrent activities, hence V_{par} equals 0 for each individual model. Municipality C has the most activities, of which the majority is involved in (multiple) loops. This is represented by the much higher V_{eiresp} and V_{conf} rules, particularly when compared to the involved events. Note that, although the rules and events of V_{eresp} are shown in the table, they are not included in the total as they comprise duplicate and overlapping rules and could, therefore, be omitted in favor of V^{red}_{eresp}.

Table 10.6: Output of variability specifications for each individual process.

$Model$	#	V_{iresp}	V_{iprec}	V_{eiresp}	(V_{eresp})	V^{red}_{eresp}	V_{conf}	V_{par}	$Total$
A	*Rules*	20	20	31	(195)	0	54	0	125
	Events	21	21	21	(21)	0	10	0	21
B	*Rules*	23	23	38	(219)	0	88	0	172
	Events	24	24	24	(24)	0	22	0	24
C	*Rules*	28	28	52	(346)	0	140	0	248
	Events	29	29	29	(29)	0	26	0	29

Subsequently, we have created three distinct variability specifications. The first specification combines Municipality A and Municipality B, which show low variability (i.e., the amount and complexity of the differences between the two models is relatively low). The second specification concerns Municipality A and Municipality C, which show high variability (i.e., the amount and complexity of the differences between the two models is relatively high). The third specification combines all three models.

Table 10.7 provides an overview of the sets of variability rules produced for each of the three variability specifications. Although V_{eresp} is shown in the table, it is not

[1] For a detailed description of the PNML standard, see http://www.pnml.org/

included in the total amount of rules, as the reduction (i.e. V^{red}_{eresp}) is used instead. It is easy to see that the amount of immediate response and precedence rules (i.e. V_{iresp} and V_{iprec}) is dependent on the amount of rules of the most complicated model in the specification. The amount of conflict rules (i.e. V_{conf}) are dependent on the model with the least amount of conflicts, as those are restrictive and should adhere to the least restrictive model (Municipality A). The reduction of V_{eresp} is significant, reducing the amount of required response rules by more than 99% for all specifications.

Table 10.7: Output of variability specifications composed of different processes.

$Spec$	$Models$	#	V_{iresp}	V_{iprec}	V_{eiresp}	(V_{eresp})	V^{red}_{eresp}	V_{conf}	V_{par}	$Total$
1	A, B	*Rules*	23	23	33	(223)	1	54	0	134
		Events	24	24	24	(24)	2	10	0	24
2	A, C	*Rules*	29	29	36	(276)	2	54	0	150
		Events	30	30	29	(29)	4	10	0	30
3	A, B, C	*Rules*	30	30	36	(297)	2	54	0	152
		Events	31	31	28	(30)	4	10	0	31

Clearly, an increasing variability and complexity results in a gradually increasing number of rules. Furthermore, Municipality C differs much more from Municipality A than Municipality B does, which is clearly shown by the number of rules resulting from the case featuring Municipality A and C. The amount of rules here are almost identical to the amount of rules required for the third case involving all models. Consequently, it can be observed that most of Municipality B fits within the behavior of both Municipality A and Municipality C, as the addition of Municipality B to the variability case hardly changes the required amount of rules.

Table 10.8 shows the percentage of overlapping rules between two specifications. Specifications 1 and 2 only show a total overlap of 75%, i.e., 75% of the rules are common between both specifications. In particular the immediate response and precedence show relatively low overlap (52% and 43%) respectively. This is because most of these rules overlap only in part as additional events are included as a preceding or responding event. However, the overlap of the exist immediate response and exist response (V_{eiresp}, V_{eresp} and V^{red}_{eresp}) are relatively high. This shows that, although both specifications have different exact paths, eventually similar activities are executed. This strongly supports the case of the necessity of such variability specifications, as *roughly* the processes are similar across different municipalities, but differ with respect to details and municipality specific policies. Consequently, a generic specification (like specification 3) allows for support of all variants, while maintaining the common required control-flow.

Table 10.8: Amount of overlapping rules between different variability specifications.

$Specifications$	V_{iresp}	V_{iprec}	V_{eiresp}	(V_{eresp})	V_{eresp}^{red}	V_{conf}	V_{par}	$Total$
1 (A, B) vs 2 (A, C)	52%	43%	73%	(85%)	100%	100%	N.A.	75%
1 (A, B) vs 3 (A, B, C)	61%	57%	82%	(95%)	100%	100%	N.A.	81%
2 (A, C) vs 3 (A, B, C)	69%	55%	92%	(99%)	100%	100%	N.A.	83%

Performance

For each variability case, the execution times were captured for each component of the procedure. The performance analysis was executed using a laptop with Intel i7 2.5GHz, running JVM 8 with 16GB of allocated memory. To eliminate load time from the measures, we executed each test five times and recorded average times of three executions, removing the fastest and the slowest executions. In all cases, the generation of the variability specification completed within 0.1 sec. Table 10.9 provides an overview of the execution times. Although the execution times of V_{eresp} are shown in the table, they are not included in the total execution time as they comprise duplicate and overlapping rules and could, therefore, be omitted in favor of V_{eresp}^{red}.

The construction of the compound labeled event structure (denoted by C-PES in the table) included the unfolding of each Petri net into a prime event structure as well as the subsequent building of the sets of compound behavior relations. From these compound behavior relations, the sets of variability rules could be easily derived. Consequently, the largest portion of the total time was dedicated to the construction of the compound labeled event structure, where the time required for the respective variability rules was comparatively negligible.

Furthermore, it can be observed that the cases with higher variability resulted in a slightly longer time for generation of the compound labeled event structure, but did not have a significant effect on the generation of the rules. This could be explained by the fact that the generation of the rules is linear in the amount of compound relations specific to the set of rules to be generated. The computational complexity of generating those compound relations, however, is part of the computational complexity of the construction of the compound labeled event structure.

However, even for the compound labeled event structure, performance seems to deteriorate only slightly despite a significant increase in the complexity of the models to be included (i.e. involving multiple nested loops).

Table 10.9: Performance of rules generation for each variability specification.

$Models$	C-PES	V_{iresp}	V_{iprec}	V_{eiresp}	(V_{eresp})	V^{red}_{eresp}	V_{conf}	V_{par}	$Total$
A, B	57ms	0.2ms	0.1ms	0.1ms	(0.4ms)	2.2ms	0.0ms	0.0ms	60ms
A, C	63ms	0.2ms	0.1ms	0.1ms	(0.6ms)	2.7ms	0.1ms	0.0ms	66ms
A, B, C	73ms	0.2ms	0.2ms	0.1ms	(0.7ms)	3.1ms	0.1ms	0.0ms	77ms

This can be explained by the theoretical complexity of the entire method. The complexity of the transformation step of each Petri net is defined by $\mathcal{O}((\frac{|B|}{\zeta})^{\zeta})$, where B is the set of places (conditions) of the unfolding and ζ is the maximal size of the presets of the transitions in the original net (Esparza et al., 1996). The subsequent transformation of the complete prefix unfolding into an event structure is linear time using the concurrency relation during the computation of the unfolding. Finally, the complexity of the construction of the compound behavior relation sets from Definition 8.4.1 is defined by $\mathcal{O}(|E_V|^2)$. As such, the total complexity is dominated by the transformation step from each Petri net to its respective unfolding and subsequent event structure, and, therefore, the amount of input processes involved in the specification.

10.4.3 Case 3: Bouncer Registration

The bouncer registration process is implemented in order to evaluate the effectiveness of the presented techniques when verifying properties over CBP. Since CPB are separate BP with messages as its only synchronization points, the CBP as a whole features a high amount of concurrency. The bouncer registration process, as described in Section 4.3, was implemented using the CPN depicted in Figure 4.10 and compared against the techniques examined by Corradini et al. (2015). Table 10.10 features the results.

Table 10.10: Comparison of CBP verification techniques (Corradini et al., 2015).

Approach	**States**	**Relations**
Unfolding (Corradini et al., 2015)	50	48
Stutter Optimized Transition Graph (Groefsema et al., 2016)	134	356
Transition Graph (Groefsema et al., 2016)	151	392
PIPE (Corradini et al., 2015)	638	1666

Of the three techniques, unfolding (Corradini et al., 2015) easily produces the least number of states and relations. Unfolding, however, fails to evaluate all concurrent

executions of the Petri net while evaluating the reachable markings (McMillan and Probst, 1995). Although this fact makes unfolding highly suitable for soundness verification, compliance and variability verification suffer from this lack of information. In addition, cycles are removed by decoupling any backward looping arcs, causing further loss of information. The proposed transition graph, however, does not suffer from such issues, while severely limiting the state space compared to the PIPE based approach (Corradini et al., 2015). Stutter optimization reduces the state space even further due to the two silent transitions used in the CPN of Figure 4.10. Note that no unused atomic propositions were removed before obtaining a transition graph that is equivalent with respect to stuttering. As a result, further optimization is available.

CHAPTER 11

Conclusion

The best thing about a boolean is even if you are wrong, you are only off by a bit.

– Anonymous

The main contribution of the presented work entails the evaluation of formal verification as an approach to business process variability. The central notion revolves around the fact that a BP variant must conform to a reference process. When comparing this to formal verification, we notice a clear similarity in that a model must conform to a formal specification. In other words, a reference process is the specification to which the model of a BP variant must comply. This, in turn, reflects approaches taken in the field of BP compliance verification. As a result, variability must be an extension of preventative compliance verification. One approach to formal verification is that of model checking. Supported by a large scientific community, model checking is a technique where a system model is automatically, systematically, and exhaustively explored while each exlored state is verified to be compliant with the specification. As a result, we must be able to verify whether a BP variant conforms to a reference process using model checking. To evaluate these statements, we defined the goals and developed the artifacts to support business process variability as an extension of formal preventative compliance verification through model checking.

11.1 Summary

In Chapter 2, we first discussed the background of BPM and formal verification. We introduced the informal models of BPM, the formal models for verification, and a branch of logics commonly used for specifications. In addition, we introduced a way of BP formalization using Petri nets to bridge the initial gap between the informal specification of BP models and the formal models used for verification.

In Chapter 3, we discussed the state of the art on the three goals of BP verification: BP soundness, compliance, and variability. Here, we noticed that BP soundness was solved through the same approaches used for BP formalization. In addition, we noticed that existing models used towards preventative BP compliance were either over-specifying the model, causing extreme amounts of overhead for the purpose of variability verification, or under-specifying the model in such a way that relevant information, like concurrent execution or the next execution in a branch, was lacking. At the same time, we noticed that the approaches towards compliance and variability either focused on runtime verification, consisted of design-time approaches with insufficient support of concurrent behavior, or defined new or newly extended logics which severely limits support by the field of model checking.

In Chapter 4, we step-wise introduced the scope of the problem through a series of real-life case studies. The first case study describes a compliance study at an Australian telecommunications provider which must comply to the Telecommunications Consumer Protections (TCP) code of conduct. The second case study extends the issue with that of variability, and details a number of Dutch municipalities which all are required by law to offer the same service to its residents, but tailored to local needs. Finally, the third case study increases the complexity of the problem by detailing a collaborative BP, where multiple BP occur concurrently.

In Chapter 5, we specified the requirements towards design-time BP verification. We specified both requirements for the model, as well as the specifications with respect to BP soundness, compliance, and variability. When taking the set of requirements to evaluate the level of requirement satisfaction throughout the state of the art, the lack of model support is evident. Where the specification requirements are often extensively compiled, requirements regarding the model – upon which the specifications are to be interpreted – are often neglected or even omitted, causing incorrect or incomplete support.

In Chapter 6, we presented a novel mapping of business process models to a system model. The resulting model allows the verification of preventative compliance and variability using well-known temporal logics and model checking techniques while providing full insight into parallel executing branches and the local next activity invocation. Furthermore, the mapping causes limited state explosion, and allows for significant further model reduction. In addition, we present an approach that allows verification using conditions, and define inheritance of specification sets.

In Chapter 7, we matched the set of requirements to specifications applicable to the presented model. To support the design of reference processes, visualization rules were specified that apply to each specification.

In Chapter 8, we presented an approach to apply the presented specifications and automatically obtain reference processes from sets of BP models. The approach not only captures the exact behavior of each presented model, but also that of models that exists in between those presented. Furthermore, the approach is extensible, allowing for more strict, or more loose, definitions. At the same time, the approach is also capable of incorporating ad-hoc runtime deviations of business processes by allowing the input of mined runtime traces in addition to the presented models.

In Chapter 9, we introduced a tool that supports full insight of the full verification process. The tool not only allows for the visual design of BPMN BPD (Appendix A) and the visual specifications presented in Chapter 7, but also supports configuration of the pattern-based formalization process as presented in Appendix B, the generation of the reduced (conditional) system model(s) as presented in Chapter 6, and the specification verification results. Furthermore, the tool is extensible and allows further specification, formalization, and modeling options to be defined.

Finally, in Chapter 10, we evaluated and demonstrated the presented artifacts. We established the expressive power of the system model (Chapter 6) by comparing the output of different types of system models when applied to a number of complex workflow patterns. Next, we determined that the performance of the system model is sufficient to provide instant results for small to large models. In addition, we established that the reduction techniques display considerable effects on the foremost source of state space inflation; parallel interleavings. The reduction technique even increases in effect as the number of parallel branches increases. Furthermore, the system models can be reduced to trivial sizes as the number of atomic propositions is kept low – by, for example, splitting the number of specifications using the same set of atomic propositions into separate manageable sets and verifying

those on a system model reduced with respect to that set of atomic propositions. We then evaluated the combination of the model and specifications against the set of requirements and found most of these satisfied. Finally, we demonstrated the applicability of the designed artifacts on the case studies presented in Chapter 4.

11.2 Contributions

Recalling the set of requirement of Balko et al. (2009), we consider the contribution of the presented work.

- **Reference process conformance**
  Reference process conformance is the ability to verify whether a process variant conforms to a reference process, and is directly supported. By defining a reference process as a BP model that includes a set of temporal logic specifications which formally define conformance of variants, and presenting variants as system models by a novel conversion process, variants are formally verified for conformance using well-supported model checking techniques.

- **Reference process patchability**
  Reference process patchability is the ability to patch, update, or change the reference process such that all changes are automatically propagated to every variant that is based on that reference process. Although not directly supported, reference processes do include a set of formal specifications that define conformance of variants. This set of specifications can simply be patched, updated, or changed without issues. Then, when versioning is introduced, variants only require a single re-verification after a patch is applied. In case the verification fails, the variant is illegal and should be re-evaluated.

- **Extension mining**
  Extension, or variant, mining is the ability to automatically detect manual ad-hoc deviations from a BP model and automatically derive extensions/variants. Although the detection of manual ad-hoc changes itself is out of the scope of this document, the automatic derivation of extensions/variants is related to the presented techniques of Chapter 8. This ability is supported by allowing mined runtime traces as input to the specification mining technique presented in Chapter 8. In case of manual ad-hoc deviations from a BP model, the original model and the mined trace of the deviation can be used as input. The output, then, is a set of reference process specifications which describe the combined behavior.

- **Stacked extensions**
  Stacked extensions describes the ability to define parent-child relations between both different reference processes and/or different extensions. This ability is supported by the definition of reference process conformance as sets of formal specifications. Reference processes can simply inherit the set of formal specifications of other reference processes. In this case, the model included in the reference process must not only adhere to its own specifications, but also to the inherited specifications. Furthermore, both variants and reference processes may choose to adhere to multiple other reference processes. Reference processes that contain contradictory specifications will, however, not produce any variants.
- **Design-time usability**
  Design-time usability refers to the support of toolsets to design reference processes and variants. We contributed with tooling that allows not only the visual design of specifications over new and existing BPMN BPD, but also offers full insight into the underlying verification process. That is, each step in the verification process from BPMN BPD with visual specifications, to CPN, to reduced Kripke structures, to model checker input, to model checker output, to logic specification, and back to visual element can be inspected by the user. Furthermore, the tool is fully extensible, allowing for more and alternative CPN pattern mappings, different specifications using different logics, and different model checkers.

11.3 Results

To address the open challenges, we specified a main research question and sub-questions in Chapter 1. We discuss our answers to each sub-question before discussing the outcome of the main research question.

1. *Which goals for design-time business process verification can be identified?*

   Three goals of design-time BP verification can be identified: soundness verification, compliance verification, and variability verification. Soundness verification aims to verify the correctness of BP by testing three basic properties: reachability, termination, and proper completion. Compliance verification, on the other hand, aims to verify whether a BP complies with a specification. The goal can be either to test for illegal states (using state-based verification),

to test for a series of illegal steps (using event-based verification), or to test whether an implementation matches the design. Finally, variability verification aims to verify whether a BP variant conforms with a reference process. The goal is to test for series of steps as specified by the reference process. Since they have similar goals, variability verification can be seen as an extension of event-based compliance verification.

2. *What system model adequately represents the business process for variability verification?*

   Since variability verification can be viewed as an extension of event-based compliance verification, an event-based system model is required. Most existing event-based system models, however, have issues with concurrency. First, the state space of the system model increases dramatically when encountering parallel executing branches. Second, any parallel execution is linearized completely, removing parallel execution information. And, finally, the system model behaves differently when describing parallel executing branches. That is, the next operator behaves as a global next (i.e., the next occurrence from all concurrently executing branches) when returning results from parallel interleaved branches, while returning seemingly local next results from non-parallel braches. Since BP define the order between structural activities in both parallel and non-parallel branches, this different behavior is undesirable and difficult to grasp due to its inconsistency.

   The transition graph (Definition 6.1.2) solves these issues through a novel conversion that introduces an event-based labeling to a state-based model. The resulting interleaving retains parallel execution information, while also offering consistent local next information through the until operator. At the same time, it also captures correct behavior of different, but similarly behaving, gateways. Furthermore, the required state space is smaller when encountering parallel executing branches. The result is a system model with increased expressive power that requires less state space.

3. *In which manner can the system model be reduced without relevant information loss?*

   Both existing system models and the transition graph (Definition 6.1.2) capture global next information, i.e., information on the next occurrence from all concurrently executing branches. However, where existing system models rely on this information to also offer local next information (when there are no concurrently executing branches), the transition graph does not. Instead, the transition graph features consistent local next information through the until

operator. In fact, this local next information is often more relevant than that of the global next. As a result, the global next information can often be seen as being superfluous, or causing overhead.

When removing the (global) next information, the resulting system model is equivalent with respect to stuttering. At the same time, any irrelevant labeling can be safely removed from the system model without the need for (global) next information. The resulting system model demonstrates increasing reductions as the number of concurrently executing branches increases. In fact, the reduced system model often becomes trivial when the number of used atomic propositions is kept low.

4. *What can be verified using well-supported specification languages?*

   Well-supported specification languages are specification languages that are supported by model checking tools, and include languages such as linear-time temporal logic (LTL) and the branching-time temporal logic computation tree logic (CTL). Since we aim to verify over structural activities in different branches of BP, a branching-time temporal logic, such as CTL, is required.

   When evaluating the specification requirements on expressivity against possible CTL specifications, we notice that all requirements are supported except those which are inherently state-based specifications. A natural consequence from the event-based nature of the system model. For example, the universality specification states that some condition must universally hold. A condition which is irregular for the changing nature of the event-based system model. On the other hand, specifications of a state-based nature can often be implied from a series of events leading up to such a state.

5. *In which way can specifications be obtained automatically?*

   Most specifications either define inclusion information of structural activities, or define ordering information of structural activities. To automatically obtain specifications from (sets of) source BP or runtime execution traces, a model is required that captures exactly this information. Labeled prime event structures (PES) are such a model.

   However, to obtain specifications that contain variability from sets of source BP or runtime execution traces, the inclusion and ordering information must first be consolidated into a compound model. From this compound PES (Definition 8.4.1), specifications can then be obtained directly using a simple translation (Definition 8.5.1). This direct translation, however, is only possible

when verified on a system model which maintains consistent local next information throughout the model, such as the transition graph (Definition 6.1.2). Without consistent local next information, next information is difficult to capture within parallel regions.

6. *For which business processes is the resulting system model verifiable?*

   Although the transition graph requires a smaller state space than other system models, there certainly is a limit to what BP can verified. When obtaining the system model, each and every possible state and relation is calculated before reductions are applied. When evaluating large, or erroneously designed, parallel regions, the number of states and relations can increase drastically. This is especially the case for increasing numbers of parallel branches. Although the length of each parallel branch also has an impact, it is far less severe. To counteract the effect, reductions gain superior results for increasing numbers of parallel branches. Furthermore, the reduction method allows formulas to be split into multiple sets, each resulting in a much smaller reduced system model. Each formula set can then be verified against its respective model reduction. In this respect, the size of the reduced model is directly related to the number of atomic propositions used within the set of formulas. As a result, even BP with extremely large parallel regions can be verified, as long as the set of atomic propositions can be kept low within sets of formulas and the number of parallel branches remains reasonable (e.g. see Table 10.2).

Now that each sub-question has been answered, only the main research question remains to be considered.

> *To which extent can formal verification through model checking be used to support verification of business processes variability as an extension of design-time, preventative, business process compliance?*

The main research question asks us to consider two elements. First, *to which extent can business process variability be seen as an extension of design-time, preventative, business process compliance?* And, secondly, *to which extent can formal verification through model checking be used to support the verification of such business process variability?*

BP variability can be seen as an extension of design-time, preventative, BP compliance, but only when the compliance verification is event-based. Since BP variability concerns itself with different versions of orderings of activities (i.e., events) within

BP, an event-based system model and specification is required. Most BP compliance verification methods, however, aim to verify state-based system models against state-based specifications. And most, do not do preventative verification, but verification during or after execution. Those that do preventative verification, often aim to verify the implementation of a BP against a BP model and generally incur in large amounts of overhead when considering variability verification purposes. As a result, a new optimized event-based system model, with corresponding event-based specifications, was proposed and evaluated for preventative compliance and variability verification. Results show that the proposed event-based system model is suitable for BP compliance verification and, as an extension, suitable for BP variability verification.

Formal verification through model checking can be used to support the verification of business process variability, but only for system models whose state space can be computed and reduced. BP models which contain extremely large parallel regions, or parallel branching with many branches, do remain difficult to compute. Reduction algorithms demonstrate promising results, exhibiting increased effect with each additional parallel branch, but can only perform when limited sets of atomic propositions can be used within sets of specifications.

11.4 Limitations

The presented artifacts and techniques solve several issues, but also feature several limitations. We discuss the most prominent limitations of data and time related specifications, specifications stretching over multiple models, and the application of logics with with past-time modalities.

Data and resources are important features of BP. However, values of data can scale up to infinite values and thus severely limit the application of verification through model checking. Although compliance verification may be interested in values of data throughout the control flow of BP, variability verification is only interested in the structure of BP found through the control flow. As a result, data values are only considered when multiple different markings of a CPN can be reached through different data values to reveal the full structure of the BP within the transition graph, and only in such a way that different states are revealed. The same considerations hold for specifications, which may hold for specific sets of states which are reachable under certain data conditions.

However, since the transition graph captures not only the different markings of a CPN but also the parallel enabled binding elements, and because CPN allow complex data as colors and functions over colors bound to transition occurrences, verification using data values can be achieved using the transition graph directly. In this case, one would simply require a more complex CPN as input. This may, however, increase the state space and performance of the transition graph due to the wide range of complex bindings introduced, as well as, severely limit the effect of the reduction technique. In addition, it would require additional tool support.

Most runtime compliance verification techniques include mechanisms to verify or track the progress of BP with respect to time. This is included because many specifications exist towards BP or activity completion within a given timeframe. Although timed automata and modal logics exist (Alur and Dill, 1994) and could provide valuable insights into the timed progress of BP, real compliance timed values can only ever be obtained at runtime and thus not supported through the presented design-time artifacts and techniques.

Although the presented artifacts and techniques allow specifications over multiple concurrent BP and, in some cases, specifications stretching sub-processes by means of multiple specifications, not all inter-process specifications are supported. For example, a compliance specification stating that a complaint must be escalated by immediately initiating a separate BP and closing the current complaint process can only be verified by runtime compliance techniques.

Although the transition graph as a system model solves many variability related issues such as the verification of local next and parallel occurrences, it does have its limitations with respect to Linear-time Temporal Logics with Past-time Modali-

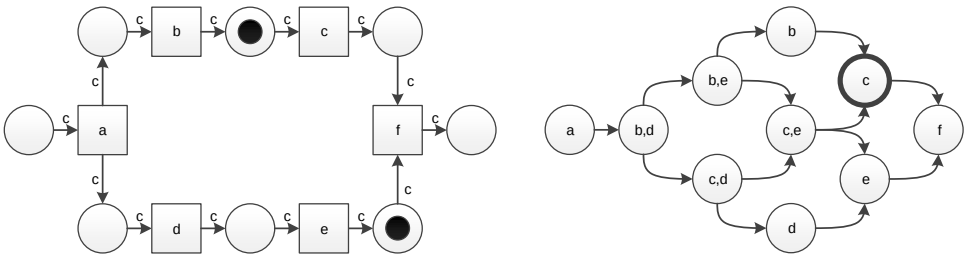


Figure 11.1: Parallel CPN pattern and related transition graph.

ties (PLTL). PLTL considers not only specifications over consecutive states, but also over previous states. Verification over previous states, however, behaves differently from that of subsequent states. In fact, it exhibits one of the same issues as verification over consequent states when considering most other system models.

Consider, for example, the parallel CPN pattern and resulting transition graph depicted in Figure 11.1. When considering previous information within the interleaving, all information seems to be present, including local previous occurrence information. This is, however, not the case for the state f. Although the transition f is always directly locally preceded by c (and e) in the CPN pattern, the transition graph does not reflect this. An argument could be made that this is correct behavior due to the synchronization occurring at f, but for structural verification of BP using past-time modalities, this is inconsistent behavior.

One possible solution is the labeling of tokens that do not enable a transition in the current marking with their previously enabled transition. Consider, for example, the marking highlighted in Figure 11.1 where one token exists on the place before transition c and one exists on the place after transition e. At this marking transition c is enabled, but not transition f. As a result, the state in the transition graph is normally only labeled with c, but now also with e, the transition which produced the token. The result is highlighted in Figure 11.2. As an additional benefit, the resulting graph can be reduced through model equivalence with respect to stuttering, removing the tail of the interleaving entirely. The effect on other patterns, however, should be explored further before applying such a solution.

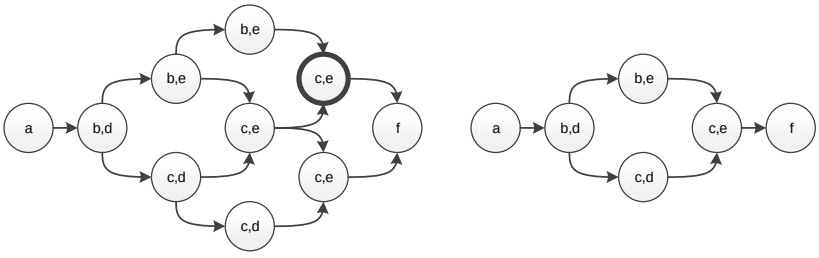


Figure 11.2: Transition graph supporting past-time modalities and related reduction.

11.5 Implications & Future Work

Although the artifacts presented throughout the previous chapters are mainly focused toward BP and BPM, we believe that the main impact of the presented research will be in the area of service oriented architectures, and specifically that of service compositions. The presented artifacts are particularly valuable in support of highly changeable environments featuring mass customization and personalization. The presented research forms the foundation of highly variable, customizable, and personalizable service-based software.

By applying specification sets, service compositions can be defined as incomplete, yet customizable templates. These templates can then be inherited and combined with other templates to form larger customizable templates. A personalized implementation can then be verified, in a formal way, to adhere to this structure of templates. An update, anywhere in the structure of templates, may trigger reverification and may invalidate any personalized implementation or allow more customization and personalization options. In this way, services can be composed with an incredible amount of customizabilty and personalization while at the same time remaining compliant with regulations.

Furthermore, the automated assembly of specification sets with high variability, from either input processes, or runtime traces, also has implications for automated composition of variable service compositions. In very much the same way as that these specification sets are assembled, rules for automated composition of service compositions with a high degree of variability, customizability, and personalization can be created. In addition, runtime adaptations of such a composition can be detected and automatically incorporated in the original rule set.

In addition to the results presented in the previous chapters, the transition graph, its specification interpretation, and related artifacts, offer the formal foundation required to accomplish these goals.

Appendices

A Business Process Model & Notation

The Business Process Model and Notation (BPMN) standard (OMG, 2011) describes a number of diagrams that are of use within the area of business process management. BPMN diagrams include Business Process Diagrams (BPD), Collaboration Diagrams, Choreography Diagrams, and Conversation Diagrams. The presented work, however, primarily focuses on the BPD and collaboration diagrams (i.e., collaborative business processes, or CBP) of the BPMN standard.

BPMN BPD can be either *private (internal)* or *public* processes. Private processes describe internal processes, whereas public processes describe the interactions between a private process and another participant. BPMN CBP, on the other hand, describe the interaction between two entities, each represented by its own BPD.

BPMN BPD and CBP are flow diagrams consisting of a large number of different elements. Tables A.1 through A.6 list each element visually. Table A.1 lists the BPMN events by type and trigger. The approaches presented within this thesis only describe events without specifying its trigger type and assume the general case. Note, however, that events with cancellation or termination triggers should be handled carefully. Table A.2 lists the possible activities as tasks and sub-processes, while Table A.3 lists the different gateways. Again, if no type is specified, one may assume the general case. Table A.4 lists the different flows as sequence flows, message flows and associations. Since associations do not affect the control flow of BP, the approaches presented within this document focuses on the other flows only. Finally, Tables A.5 and A.6 list additional BP annotations in the form of groups and data objects respectively.

Table A.1: BPMN events.

	None	Message	Timer	Error	Compensation	Link	Conditional	Signal	Multiple	Parallel multiple	Escalation	Cancel	Terminate
Start event (Interrupting)													
Start event (Non-interrupting)													
Intermediate catching event (Interrupting)													
Intermediate catching event (Non-interrupting)													
Intermediate throwing event (Interrupting)													
End event (Interrupting)													

Table A.2: BPMN activities.

	None	Call	Loop	Multi-instance (parallel)	Multi-instance (sequential)	Compensation	Ad-hoc	Transaction
Task								
Sub-process								

Table A.3: BPMN gateways.

	Exclusive	Exclusive	Parallel	Inclusive	Complex
Gateway (Data-based)					
Gateway (Event-based)					

Table A.4: BPMN flows.

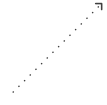
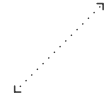
	Normal	Conditional	Default	Message	One-directional	Bi-directional
Sequence flow						
Message flow						
Association						

Table A.5: BPMN groups.

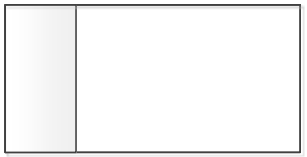
	Group	Pool/Lane
Group		

Table A.6: BPMN data objects.

	Object	Collection	Store
Data			

B BPMN BPD CPN Formalization

For the conversion of BPMN BPD to CPN, we provide a translation. The translation is based on the workflow patterns as defined in (van der Aalst et al., 2003b) and similar to the basic Petri net conversion of Dijkman et al. (2008). However, in some cases, additions are required in order to provide a generic translation of the respective BPMN element. Tables B.1 through B.6, provide an overview of the conversion of BPMN elements to CPN constructs. Patterns within service compositions can be easily identified by its description, motivation, context, and CPN overview. Patterns are then applied together using their CPN overviews to form the CPN of a BPMN BPD.

In general, BPMN sequence flows are represented by arcs in the CPN and activities are represented by a place connected to a transition. In the table, the elements, tokens, and weights that are part of the constructs are indicated with black lines and black text, whereas the surrounding elements (depicted where necessary for clarity) are represented with grey dotted lines. This is necessary, because in some cases the translation is not represented by a separate construct in CPN. Rather, it affects the preceding or succeeding elements.

To avoid any unnecessary complexity, the patterns have been adapted to use one color (e.g. complex merge variants). At the same time, intermediate catching events have been changed to occur once (or a set number of times) in order to avoid an infinite number of possible markings. Naturally, these changes do not affect labelings of states in the transition graph (Chapter 6).

In some cases it may be necessary to intoduce additional places and transitions (e.g. for consecutive forks and/or merges or the complex merge). In these cases, the introduced transitions are silent and do not perform actions. When used, silent transitions are depicted by solid black transitions and labeled s. Note that specifications may need to adjust for s (e.g. $AG(t_0 \Rightarrow [(t_0 \vee s)\ U\ t_1])$ in case of Specification 7.4.5.iii).

Table B.1 contains the conversion of the basic elements. Sequence flows are represented by arcs with a weight of 1‘c between a source transition and a target place. Activities, collapsed sub-processes, start-, end-, and intermediate throwing-events are all represented by place/transition pairs, with the exception that the start event features a token on its place, the end event features a sink place, and the intermediate throwing event may feature an outward message flow.

Table B.2 contains the conversion of several variants of intermediate catching events. The first row features the general inline intermediate catching event which may receive a message flow. When the intermediate catching event receives a message flow, it is a blocking event. When it does not feature a message flow, it is a non-blocking event instead. The second row features an intermediate catching event which receives an external message, and behaves similar to the start event. Finally, the third and fourth rows feature blocking and non-blocking versions of intermediate catching events attached to activities, respectively.

Tables B.3 through B.5 contain the conversions of the different fork and merge constructs. Table B.3 contains the conversion of the basic fork constructs; the exclusive fork, parallel fork, inclusive fork, and deferred choice. The conversions of the first three are very similar and all fork from the previous element's transition. They differ only in the conditions on the outward arcs. The conversion of the deferred choice, instead, collapses the places of the first elements on all branches into one place. When this place receives a token, the first element of an arbitrary branch will take the token and occur, blocking the other branches. Table B.4 contains the conversion of the basic merge constructs; the exclusive merge, parallel merge, and inclusive merge. The first row features the conversion of the exclusive merge, which simply merges on the place of the next element. The second row features the conversion of the parallel merge, which synchronizes all inward branches by providing as many places to the next element as there are inward branches. Only when all places contain tokens, the next element may occur. Finally, the third row contains the conversion of the inclusive merge. The inclusive merge behaves similar to the parallel merge, except for the fact that it allows each branch to be bypassed. Table B.5 contains two conversions for the complex merge. The complex merge features an n out of m merge on the next element which only occurs if n out of m branches have provided a token. The first row features the simple variant of this construct, while the second row features a variant save for use within cycles.

Finally, Table B.6 contains conversions for repeated executions of activities and message events. The first row features the conversion of a repeated execution of an activity in the form of a structured while loop, the second row features a structured repeat loop, and the third row features a multiple instance variant for repeated execution of an activity where the activities are executed in parallel. The fourth, and last, row features the conversion of message flows between throwing and catching intermediate events or activities.

Table B.1: Conversion of BPMN elements into CPN constructs based on the workflow patterns as defined in (van der Aalst et al., 2003b).

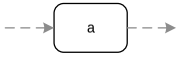
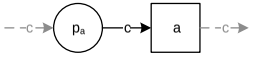
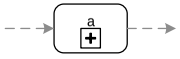
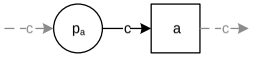
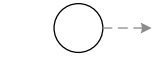
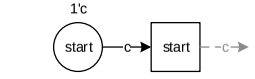
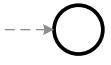
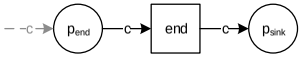
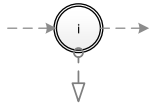
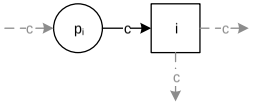
BPMN Element	BPMN Symbol	CPN Translation
Sequence Flow		
Task / Activity		
Sub-process		
Top-level Start Event		
Top-level End Event		
Intermediate Throwing Event		

Table B.2: Conversion of BPMN elements into CPN constructs (continued).

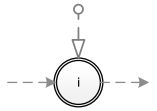
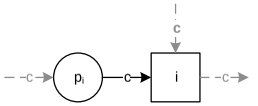
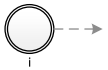
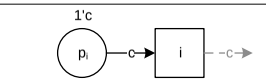
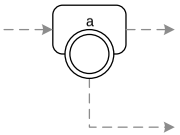
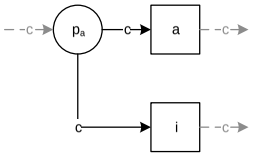
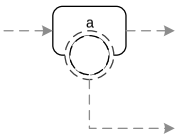
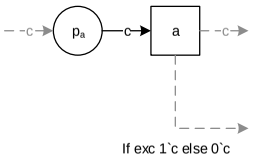
BPMN Element	BPMN Symbol	CPN Translation
Intermediate Catching Event		
		
		
		

Table B.3: Conversion of BPMN elements into CPN constructs (continued).

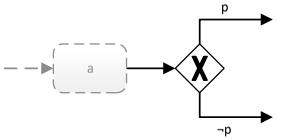
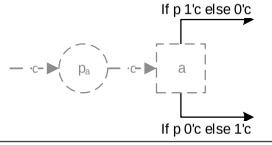
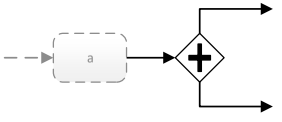
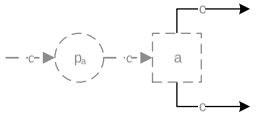
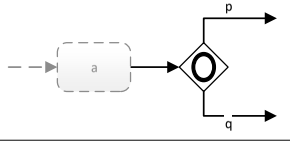
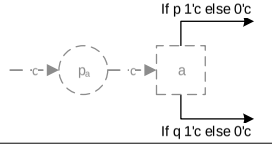
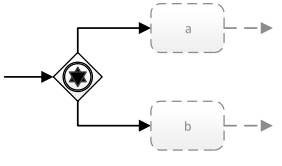
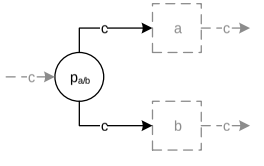
BPMN Element	BPMN Symbol	CPN Translation
Exclusive Fork		
Parallel Fork		
Inclusive Fork		
Deferred choice		

Table B.4: Conversion of BPMN elements into CPN constructs (continued).

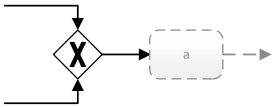
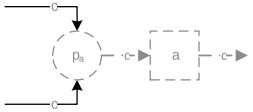
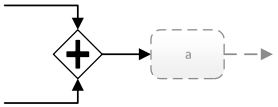
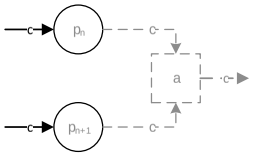
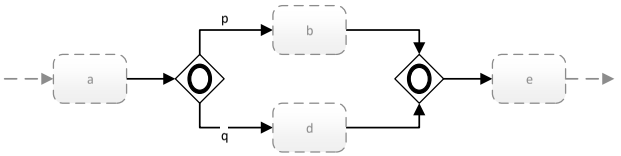
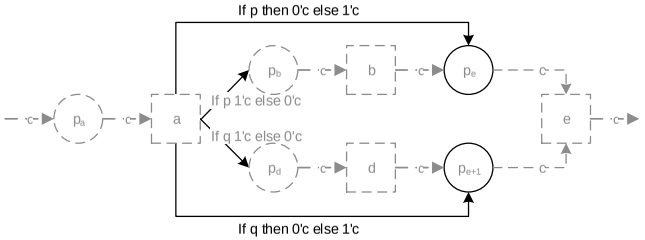
BPMN Element	BPMN Symbol	CPN Translation
Exclusive Merge		
Parallel Merge		
Structured Inclusive Fork and Merge		

Table B.5: Conversion of BPMN elements into CPN constructs (continued).

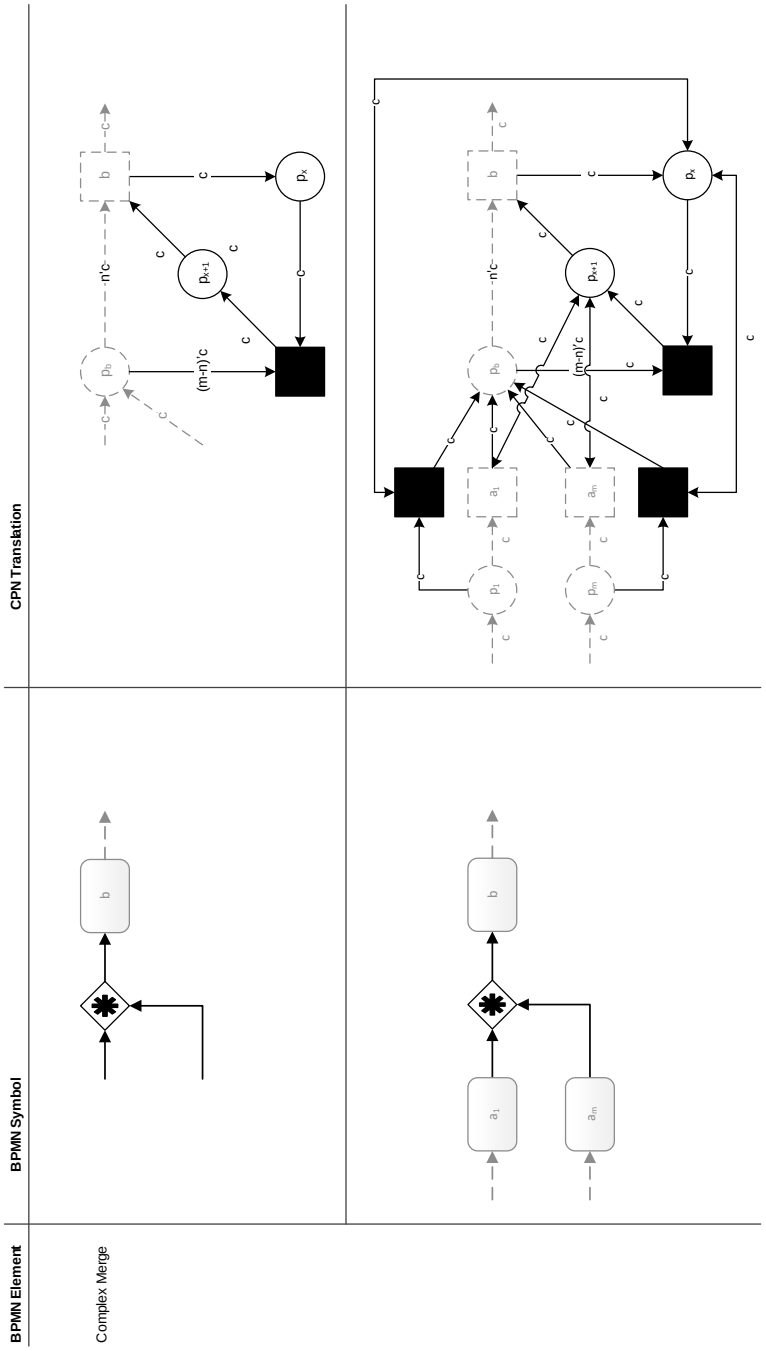
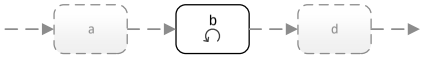
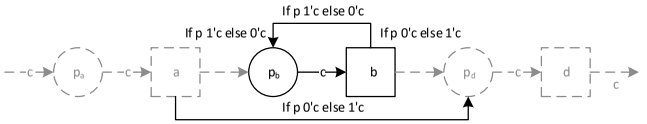
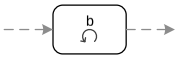
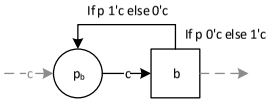
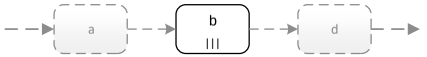
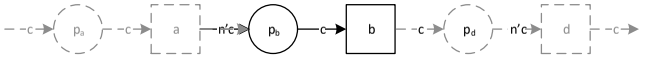
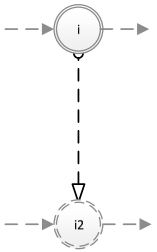
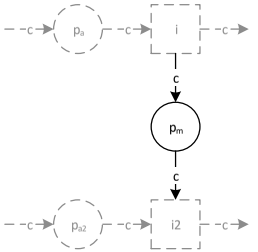


Table B.6: Conversion of BPMN elements into CPN constructs (continued).

BPMN Element	BPMN Symbol	CPN Translation
Structured Loop (While)		
Structured Loop (Repeat)		
MI Variant 2		
Message between activites or events		

Abbreviations

BP	Business Process
BPD	Business Process Diagram
BPEL	(Web Service) Business Process Execution Language
BPM	Business Process Management
BPMN	Business Process Model and Notation
BPMS	Business Process Management System
CBP	Collaborative Business Process
CPN	Colored PN
CTL	Computation Tree Logic
CTL*	Computation Tree Logic*
CTL-X	CTL minus the nexttime operator
ICT	Information and Communication Technology
IT	Information Technology
LTL	Linear Temporal Logic
LTS	Labeled Transition System
CPES	Compound PES
PES	(Labeled) Prime Event Structure
PLTL	LTL with Past-time Modalities/Past-time LTL
Pomset	Partially ordered multiset
PN	Petri Net
RG	Reachability Graph
SOA	Service Oriented Architecture
TG	Transition Graph
UML	Unified Modeling Language
WfM	Workflow Management
WfMS	Workflow Management System

WF-net	Workflow Net
WS-BPEL	Web Service Business Process Execution Language
WS-CDL	Web Service Choreography Description Language
XML	eXtensible Markup Language

English Summary

Business Processes (BP) are collaborations that aim to achieve certain value-added goals driven by external needs. Business Process Management (BPM) manages and optimizes BP with the intent to increase productivity and performance. BPM is a rapidly evolving field due to new requirements emerging at agile branches of business where BP are required to be less and less rigid. The advent of web services and service-orientation drives this evolution even further. Where BPM supported local user-specific rigid and repetitive units of work in the past, these days it is required to support loosely-coupled processes in cloud configurations among many users with each many different requirements.

As the field of BPM continues to manage an increasing number of rapidly evolving BP in agile environments, the evolution of each BP must continue to always behave in a correct manner and remain compliant with the laws, regulations, and internal business requirements imposed upon it. To manage the correct behavior of quickly evolving BP, or the definition of a wide variety of similar BP, we evaluate the application of formal verification techniques as a possible solution for the pre-runtime analysis of the correct behavior and compliant design of BP within possible process families. Formal verification entails proving or disproving the correctness of a system model with respect to a formal specification using formal methods of mathematics. When employing formal verification, a system model – often represented by a labeled transition system – is verified against a formal specification in the form of a set of logic formulas. One approach towards formal verification is model checking. When model checking a system model is automatically, systematically, and exhaustively explored while each exlored state is verified to be compliant with the formal specification.

A series of three case studies from business and government demonstrate the issues. First, a customer support process of an Australian telecommunications provider details the difficulties of compliant design while under enforcement of national regulations such as the Telecommunications Consumer Protections (TCP) code of conduct. Second, a study throughout a number of Dutch municipalities which all are required by law to offer the same service to its residents, but tailored to local needs, details how a large set of similar BP are defined locally and independently without regulation. Finally, a collaborative BP between multiple separate entities illustrates design with extreme concurrency.

A novel approach allowing pre-runtime verification that supports the different branching and merging constructs allowed by BP models and their service compositions is presented. A BP is first formulated as a colored Petri net (Jensen, 1981) (CPN) through the application of workflow patterns (van der Aalst et al., 2003b). The CPN is then translated into a Kripke structure (Emerson and Halpern, 1982) using a novel model conversion which maintains both parallelization information as well as information on the next local activity occurrence within individual parallel branches or processes (i.e. the local next). Since Kripke structures are transition systems used to interpret temporal logics, the model allows verification of control flow specifications over activity occurrences within BP. Furthermore, the resulting Kripke structure can be reduced before verification.

The state of the art is analyzed and a comprehensive set of specification requirements is compiled and extended from related work towards formal verification and BP compliance. The set of requirements is defined as temporal logic specifications and features visualization elements for visual design. In addition, an approach towards the automated generation of specifications from either a series of related BP or runtime traces of BP is presented.

The Verification extension for Business Process Modeling Tool (VxBPM) implements the main concepts. The tool is written using the Java programming language and features BP design using the Business Process Model & Notation standard, automated pattern-based model transformation to CPN, automated generation of the Kripke structures required for verification, automated verification using one of multiple model checkers, and transparent visual and textual feedback of the generated models and verification results.

Evaluations on expressive power demonstrate that, other than the generally employed transition systems, the proposed model correctly captures well-known BP

patterns. Furthermore, it maintains information on parallel occurrences of activities and the local next activity occurrence: an ability which is unique to the presented approach. Evaluations on performance confirm that the conversion algorithm performs well, even for BP with exceedingly large parallel branching. Moreover, very large BP can be significantly reduced before actual verification. Furthermore, the approach allows to split specifications in multiple sets, each resulting in a much smaller reduced Kripke structure. Each specification set can then be verified on its respective Kripke reduction, which results in a significant performance gain. As such, the size of the reduced model is directly related to the number of atomic propositions used within the set of formulas. Finally, the application of the presented approaches on the relevant case studies further demonstrate the applicability of formal verification as a solution of the analysis of correct and compliant behavior of BP within possible process families.

Nederlandse Samenvatting

Bedrijfsprocessen (BP) zijn collaboraties met de intentie om bepaalde, door de buitenwereld gedreven, waarde vermeerderende doelstellingen te behalen. Business process management (BPM) beheert en optimaliseert BP met het doel om productiviteit en bedrijfsprestaties te verhogen. BPM is een snel evoluerend veld door nieuw opkomende vereisten vanuit flexibele bedrijfstakken waar BP steeds minder star behoren te zijn. De opkomst van web services en service-orientatie drijft deze evolutie zelfs nog verder. Waar BPM in het verleden specifieke rigide en repetitieve werkeenheden ondersteunde voor de lokale gebruiker, wordt tegenwoordig vereist dat het losgekoppelde processen ondersteunt in cloud configuraties, te midden van vele gebruikers met elk vele verschillende eisen.

Zolang het BPM veld een stijgend aantal snel evoluerende BP in flexibele bedrijfstakken ondersteunt, moet de evolutie van elk BP aanhoudend correct gedrag vertonen en tevens voldoen aan de opgelegde wet- en regelgeving en interne bedrijfsregels. Om het aanhoudend correct gedrag te ondersteunen van snel evoluerende BP, of de definitie van een breed aantal soortgelijke BP, evalueren we de toepassing van formele verificatietechnieken als mogelijke oplossing voor analyse van het juiste gedrag en wettelijk conforme ontwerp van BP binnen mogelijke proces families, welke plaatsvindt voorafgaand aan de uitvoering van dat BP. Formele verificatie omvat het al dan niet bewijzen van de juistheid van een systeemmodel ten opzichte van een formele specificatie met behulp van formele wiskundige methoden. Bij de toepassing van formele verificatie wordt een systeem model, vaak weergegeven met een gelabeld transitie systeem, geverifieerd tegen een formele specificatie in de vorm van een reeks logische formules. Eén aanpak voor formele verificatie is model checking. Bij model checking wordt een systeem model op een automatische, systematische en uitputtende manier onderzocht, terwijl elke onderzochte staat wordt gecontroleerd op overeenstemming met de formele specificatie.

Een reeks van drie case studies uit het bedrijfsleven en de overheid demonstreren de problematiek. De eerste case study omvat een klantenservice proces van een Australische telecomprovider, en beschrijft de moeilijkheden van conform ontwerp terwijl het onder handhaving van nationale regelgeving staat (i.e. de Telecommunications Consumer Protections (TCP) gedragscode). De tweede case study omvat een onderzoek binnen een aantal Nederlandse gemeenten. Deze gemeenten zijn wettelijk verplicht dezelfde service te bieden aan haar inwoners, maar dan afgestemd op lokale behoeften. Hetgeen resulteert in een groot aantal soortgelijke BP welke lokaal en onafhankelijk van elkaar worden gedefinieerd zonder regulering. Tot slot illustreert de derde case study een coöperatief BP, tussen meerdere afzonderlijke entiteiten, een ontwerp met extreme parallel uitvoerende processen.

Een innovatieve benadering voor verificatie tijdens de ontwerpfase wordt gepresenteerd. De benadering ondersteunt de verschillende vertakkende en samenvoegende constructies zoals toegestaan in BP-modellen en hun service composities. Een BP wordt eerst geformuleerd als een colored Petrinet (Jensen, 1981) (CPN) door middel van het toepassen van workflow patronen (van der Aalst et al., 2003b). Vervolgens wordt de CPN vertaald naar een Kripke structuur (Emerson and Halpern, 1982) met behulp van een innovatieve vertaling, die zowel parallellisatie informatie handhaaft, alsmede informatie over de aanwezigheid van de volgende lokale activiteit in individuele parallelle takken of processen (d.w.z. de plaatselijk volgende aanwezigheid). Aangezien Kripke structuren transitie systemen zijn, die worden gebruikt om temporele logica te interpreteren, kan het model worden ingezet voor de verificatie van control-flow specificaties over activiteiten en hun uitvoer binnen het BP. Bovendien kan de resulterende Kripke structuur worden gereduceerd, voorafgaand aan de verificatie.

De state of the art wordt geanalyseerd en een uitgebreide verzameling van specificatie vereisten wordt verzameld en uitgebreid vanuit de literatuur op het gebied van formele verificatie en BP compliance. De vereisten worden gedefinieerd als temporele logica specificaties, welke visualisatie elementen bevatten voor een grafische weergave van het ontwerp. Bovendien wordt een aanpak voor het automatisch genereren van de specificaties vanuit zowel een reeks verwante BP als uitgevoerde BP gepresenteerd.

De Verificatie extensie voor Business Process Modeling Tool (VxBPM) implementeert de belangrijkste concepten. De tool is geschreven met behulp van de programmeertaal Java en voorziet in BP ontwerp met behulp van de Business Process Model & Notation standaard, automatische model transformatie naar CPN gebaseerd op

patronen, geautomatiseerde generatie van de Kripke structuren die nodig zijn voor verificatie, automatische verificatie via een model checker en transparante visuele en tekstuele feedback van de gegenereerde modellen en verificatieresultaten.

Evaluaties met betrekking tot expressiviteit bewijzen dat, anders dan doorgaans toegepaste transitiesystemen, het voorgestelde model bekende BP patronen juist vastlegt. Verder behoudt het model informatie over de aanwezigheid van parallelle activiteiten en de lokale volgende activiteit: een eigenschap uniek aan de voorgestelde aanpak. Evaluaties inzake de prestaties en snelheid bevestigen dat het conversie-algoritme goed presteert, zelfs voor BP met bijzonder grote parallelle vertakkingen. Bovendien kunnen zeer grote BP aanzienlijk worden gereduceerd voorafgaand aan de eigenlijke verificatie. Tevens laat de voorgestelde aanpak toe dat specificaties in meerdere sets worden opgedeeld, elk resulterend in een veel kleinere gereduceerde Kripke structuur. Elke specificatie set kan dan worden geverifieerd op zijn respectieve Kripke reductie, wat resulteert in een significante prestatiewinst. Als zodanig is de omvang van het gereduceerde model direct gerelateerd aan het aantal atomaire proposities gebruikt binnen de set van formules. Tenslotte toont de toepassing van de voorgestelde benaderingen op de desbetreffende case studies aan dat formele verificatie als een oplossing voor de analyse van het juist en conform gedrag van BP in mogelijke proces families toepasbaar is.

Bibliography

M. Aiello, P. Bulanov, and H. Groefsema. Requirements and tools for variability management. In *IEEE Workshop on Requirement Engineering for Services at IEEE COMPSAC*, 2010.

R. Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126 (2):183–235, 1994.

B.B. Anderson, J.V. Hansen, P.B. Lowry, and S.L. Summers. Model checking for E-business control and assurance. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(3):445–450, 2005.

A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. Liu, S. Thatte, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0, 2007. WS-BPEL TC OASIS, April 2007.

A. Armas-Cervantes, P. Baldan, M. Dumas, and L. García-Bañuelos. Diagnosing behavioral differences between business process models: An approach based on event structures. *Information Systems*, 56:304–325, 2016.

A. Awad, G. Decker, and M. Weske. Efficient compliance checking using BPMN-Q and temporal logic. In *Business Process Management*, BPM '08, pages 326–341. Springer, 2008.

S. Balko, A.H.M. ter Hofstede, A.P. Barros, M. La Rosa, and M.J. Adams. Controlled flexibility and lifecycle management of business processes through extensibility. Technical report, QUT, Australia, 2009.

H.H. Bi and J.L. Zhao. Applying propositional logic to workflow verification. *Information Technology and Management*, 5:293–318, 2004.

D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify BPEL4WS workflows. In *Int. Conf. on Service-Oriented Computing and Applications*, pages 13–20, 2007.

M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115–131, July 1988.

J.R. Büchi. On a Decision Method in Restricted Second-Order Arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. Mining configurable process models from collections of event logs. In *Business Process Management*, pages 33–48. Springer, 2013.

P. Bulanov, A. Lazovik, and M. Aiello. Business process customization using process merging techniques. In *Service-Oriented Computing (ICSOC)*, pages 1–4. IEEE, 2011.

R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M/ Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *Int. Conf. on Computer Aided Verification*, pages 334–342, 2014.

S.H. Chang and S.D. Kim. A variability modeling method for adaptable services in service-oriented computing. In *Int. Software Product Line Conference*, pages 261–268. IEEE, 2007.

F. Chesani, P. Mello, M. Montali, and P. Torroni. Web services and formal methods. In *Web Services and Formal Methods*, volume 5387 of *Lecture Notes in Computer Science*, chapter Verification of Choreographies During Execution Using the Reactive Event Calculus, pages 55–72. Springer, 2009.

Y. Choi and J.L. Zhao. Decomposition-based verification of cyclic workflows. In *Automated Technology for Verification and Analysis*, pages 84–98. Springer, 2005.

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Int. Conf. on Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

F. Corradini, A. Polini, and B. Re. Inter-organizational business process verification in public administration. *Business Process Management Journal*, 21(5):1040–1065, 2015.

P. Dadam and M. Reichert. The adept project: a decade of research and development for robust and flexible process support. *Computer Science - R&D*, 23(2): 81–97, 2009.

D. D'Aprile, L. Giordano, V. Gliozzi, A. Martelli, G.L. Pozzato, and D. Theseider Dupre. Verifying compliance of business processes with temporal answer sets, 2011.

G. De Giacomo, M. Dumas, F.M. Maggi, and M. Montali. Declarative process modeling in bpmn. In *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, pages 84–100. Springer, 2015.

R. Demeyer, M. van Assche, L. Langevine, and W. Vanhoof. Declarative workflows to efficiently manage flexible and advanced business processes. In *international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 209–218. ACM, 2010.

A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Int. Conf. on Database Theory*, ICDT, pages 252–267. ACM, 2009. ISBN 978-1-60558-423-2.

R.M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008.

M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Software Engineering*, pages 411–420. IEEE, 1999.

A. Elgammal, O. Türetken, W.J.A.M. van den Heuvel, and M. Papazoglou. On the formal specification of regulatory compliance: a comparative analysis. In *Int. Conf. Service-Oriented Computing (ICSOC)*, pages 27–38. Springer, 2010a.

A. Elgammal, O. Türetken, W.J.A.M. van den Heuvel, M. Papazoglou, and et al. Towards a comprehensive design-time compliance management: A roadmap. Technical report, Tilburg University, School of Economics and Management, 2010b.

A. Elgammal, S. Sebahi, O. Türetken, M.S. Hacid, M. Papazoglou, and W.J.A.M. van den Heuvel. Business process compliance management: an integrated proactive approach. In *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, 2012.

A. Elgammal, O. Türetken, W.J.A.M. van den Heuvel, and M. Papazoglou. Formalizing and appling compliance patterns for business process compliance. *Software & Systems Modeling*, pages 1–28, 2014.

C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of conference on Organizational computing systems*, pages 10–21. ACM, 1995.

E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proc. of the 14th annual ACM symposium on Theory of computing*, pages 169–180, 1982.

J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.

R. Eshuis and R. Wieringa. Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

J. Esparza. Model checking using net unfoldings. In *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 613–628. Springer Berlin Heidelberg, 1993.

J. Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3): 151–195, 1994.

J Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 87–106. Springer, 1996.

J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

C. Favre and V. Hagen. Symbolic execution of acyclic workflow graphs. In *Business Process Management*, volume 6336 of *Lecture Notes in Computer Science*, pages 260–275. Springer Berlin Heidelberg, 2010.

S. Feja, A. Speck, and E. Pulvermüller. Business process verification. In *GI Jahrestagung*, pages 4037–4051, 2009.

J.A. Fisteus, L.S. Fernández, and C.D. Kloos. Applying model checking to BPEL4WS business collaborations. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 826–830. ACM, 2005.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *IEEE Int. Conf. on Automated Software Engineering (ASE)*, pages 152–163, 2003.

X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Int. Conf. on World Wide Web*, pages 621–630. ACM, 2004.

L. García-Bañuelos, N.R.T.P. van Beest, M. Dumas, and M. La Rosa. Complete and interpretable conformance checking of business processes. Technical report, BPM Center, 2015.

C.E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *Int. Conf. Service-Oriented Computing (ICSOC)*, volume 4749 of *LNCS*, pages 181–192. Springer, 2007. ISBN 978-3-540-74973-8.

A. Ghose and G. Koliadis. Auditing business process compliance. In *Service-Oriented Computing (ICSOC)*, pages 169–180. Springer, 2007. ISBN 978-3-540-74973-8.

S. Goedertier and J. Vanthienen. Designing compliant business processes with obligations and permissions. In *Proc. Int. Conf. on Business Process Management Workshops*, BPM, pages 5–14. Springer, 2006. ISBN 3-540-38444-8, 978-3-540-38444-1.

F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa. Configurable workflow models. *International Journal on Cooperative Information Systems*, 17(2):177–221, 2008.

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 *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, pages 486–500. Springer, 2009.

G. Governatori, Z. Milosevic, and S.W. Sadiq. Compliance checking between business processes and business contracts. In *Int. Conf. on Enterprise Distributed Object Computing Conference*, pages 221–232. IEEE, 2006.

H. Groefsema and D. Bucur. A survey of formal business process verification: From soundness to variability. In *International Symposium on Business Modeling and Software Design*, pages 198–203, 2013.

H. Groefsema and N.R.T.P. van Beest. Design-time compliance of service compositions in dynamic service environments. In *Int. Conf. on Service Oriented Computing & Applications*, pages 108–115, 2015.

H. Groefsema, P. Bulanov, and M. Aiello. Declarative enhancement framework for business processes. In *Int. Conf. Service-Oriented Computing (ICSOC)*, pages 495–504, 2011.

H. Groefsema, P. Bulanov, and M. Aiello. Imperative versus declarative process variability: Why choose? Technical Report JBI 2011-12-6, University of Groningen, dec 2012.

H. Groefsema, N.R.T.P. van Beest, and M. Aiello. A formal model for compliance verification of service compositions. *IEEE Transactions on Services Computing*, 2016. To appear.

J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *ICALP '90*, pages 626–638, 1990.

H. Hadaytullah, K. Koskimies, and T. Systa. Using model customization for variability management in service compositions. In *IEEE Int. Conf. on Web Services (ICWS 2009)*, pages 687–694, 2009.

A. Hallerbach, T. Bauer, and M. Reichert. Managing process variants in the process lifecycle. In *Int. Conf. on Enterprise Information Systems (ICEIS'08)*, pages 154–161, June 2008.

T.T. Hildebrandt and R.R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *arXiv preprint arXiv:1110.4161*, 2011.

P. Huber, A.M. Jensen, L.O. Jepsen, and K. Jensen. Reachability trees for high-level petri nets. *Theoretical Computer Science*, 45(3):261–292, 1986.

W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying business processes using SPIN. In *Proc. of the 4th Int. SPIN Workshop*, pages 21–36, 1998.

K. Jensen. Coloured Petri Nets and the invariant-method. *Theoretical Computer Science*, 14(3):317 – 336, 1981.

C.T. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater. Model checking of workflow schemas. In *Int. Conf. on Enterprise Distributed Object Computing*, EDOC '00, pages 170–181. IEEE Computer Society, 2000.

A. Kheldoun, K. Barkaoui, and M. Ioualalen. Specification and verification of complex business processes - a high-level petri net-based approach. In *Business Process Management*, volume 9253 of *Lecture Notes in Computer Science*, pages 55–71. Springer International Publishing, 2015.

O.M. Kherbouche, A. Ahmad, and H. Basson. Using model checking to control the structural errors in bpmn models. In *IEEE Int. Conf. on Research Challenges in Information Science (RCIS 2013)*, pages 1–12, 2013.

R.K.L. Ko. A computer scientist's introductory guide to business process management (bpm). *Crossroads*, 15(4):4:11–4:18, June 2009.

J. Koehler, G. Tirenni, and S. Kumaran. From business process model to consistent implementation: a case for formal verification methods. In *Int. Conf. on Enterprise Distributed Object Computing Conference*, pages 96–106, 2002.

M. La Rosa. *Managing variability in process-aware information systems*. PhD thesis, Queensland University of Technology Brisbane, Australia 25, 2009.

M. La Rosa, M. Dumas, R. Uba, and R.M. Dijkman. Merging business process models. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 96–113. Springer, 2010.

M. La Rosa, M. Dumas, R. Uba, and R.M. Dijkman. Business process model merging: An approach to business process consolidation. *ACM Transactions on Software Engineering and Methodology*, 22(2):11, 2013.

L. Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

Y. Liu, S. Müller, and K. Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46:335–361, 2007.

R. Lu, S. Sadiq, and G. Governatori. On managing business processes variants. *Data and Knowledge Engineering*, 68(7):642–664, 2009. ISSN 0169-023X.

L.T. Ly, S. Rinderle-Ma, D. Knuplesch, and P. Dadam. Monitoring business process compliance using compliance rule graphs. In *Proc. Confederated Int. Conf. On the move to meaningful internet systems - Volume I*, OTM, pages 82–99. Springer-Verlag, 2011.

S. Ma, L. Zhang, and J. He. Towards formalization and verification of unified business process model based on pi calculus. In *Int. Conf. on Software Engineering Research, Management and Applications(SERA '08)*, pages 93–101, Aug 2008.

F.M. Maggi, M. Montali, M. Westergaard, and W.M.P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Business Process Management*, pages 132–147. Springer, 2011.

M. Marin, R. Hull, and R. Vaculín. Data centric bpm and the emerging case management standard: A short survey. In *Business Process Management Workshops*, pages 24–30. Springer, 2013.

N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, 2003.

C. Masalagiu, W.N. Chin, S. Andrei, and V. Alaiba. A rigorous methodology for specification and verification of business processes. *Formal Aspects of Computing*, 21(5):495–510, 2009.

K. L. McMillan and D. K. Probst. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

F. Milani, M. Dumas, N. Ahmed, and R. Matulevičius. Modelling families of business process variants: A decomposition driven method. *Information Systems*, 56: 55–72, 2016.

M. Montali, P. Torroni, F. Chesani, P. Mello, M. Alberti, and E. Lamma. Abductive logic programming as an effective technology for the static verification of declarative business processes. *Fundamenta Informaticae*, 102(3-4):325–361, 2010. ISSN 0169-2968.

S. Nakajima. Verification of Web service flows with model-checking techniques. In *Proc. 1st Int. Symp. on Cyber Worlds*, pages 378–385, 2002. doi: 10.1109/CW.2002.1180904.

S. Nakajima. Model-checking behavioral specification of bpel applications. *Electron. Notes Theoretical Computer Science*, 151(2):89–105, May 2006.

T. Nguyen, A. Colman, and J. Han. Modeling and managing variability in process-based service compositions. In *Int. Conf. Service-Oriented Computing (ICSOC)*, pages 404–420. Springer, 2011.

M. Nielsen, G.D. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science*, 13:85–108, 1981.

OMG. Business process model and notation (BPMN) version 2.0, 2011.

OMG. Omg unified modeling language (OMG UML) version 2.5, 2015.

M. Papazoglou and W.J.A.M. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

E. Pascalau, A. Awad, S. Sakr, and M. Weske. Partial process models to manage business process variants. *International Journal of Business Process Integration and Management*, 5(3):240–256, 2011.

K. Peffers, T. Tuunanen, M.A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

M. Pesic and W.M.P. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, pages 169–180. Springer, 2006.

C.A. Petri. *Communication with automata*. PhD thesis, Universität Hamburg, 1966.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

E. Pulvermueller, S. Feja, and A. Speck. Developer-friendly verification of process-based systems. *Knowledge Based Systems*, 23(7):667–676, 2010.

M. Razavian and R. Khosravi. Modeling variability in business process models using UML. In *Int. Conf. on Information Technology: New Generations*, pages 82–87. IEEE, 2008.

W. Reisig and G. Rozenberg. *Lectures on petri nets i: basic models: advances in petri nets*. Springer Science & Business Media, 1998.

M. Rosemann and W.M.P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.

N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view (2006). Technical Report BPM-06-22, BPM Center, 2007.

I. Rychkova, G. Regev, and A. Wegmann. Using declarative specifications in business process design. *International Journal on Computational Science and Applications*, 5(3b):45–68, 2008.

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.

A. Schnieders and F. Puhlmann. Variability mechanisms in e-business process families. In *Proc. Int. Conf. on Business Information Systems*, volume 85, pages 583–601. 2006.

H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W.M.P. van der Aalst. Process flexibility: A survey of contemporary approaches. In *Advances in Enterprise Engineering I*, pages 16–30. Springer Berlin Heidelberg, 2008.

D.M.M. Schunselaar, F.M. Maggi, and N. Sidorova. Patterns for a log-based strengthening of declarative compliance models. In *Int. Conf. on Integrated Formal Methods*, pages 327–342. Springer, 2012a.

D.M.M. Schunselaar, F.M. Maggi, N. Sidorova, and W.M.P. van der Aalst. Configurable declare: designing customisable flexible process models. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 20–37. Springer, 2012b.

D.M.M. Schunselaar, E. Verbeek, W.M.P. van der Aalst, and H.A. Raijers. Creating sound and reversible configurable process models using cosenets. In *International Conference on Business Information Systems*, pages 24–35. Springer, 2012c.

D.M.M. Schunselaar, H.M.W. Verbeek, W.M.P. van der Aalst, and H.A. Reijers. Petra: A tool for analysing a process family. In *PNSE@ Petri Nets*, pages 269–288, 2014.

M. Sinnema, S. Deelstra, and P. Hoekstra. The covamof derivation process. In *Int. Conf. on Reuse of Off-the-shelf Components*, LNCS, pages 101–114. Springer-Verlag, 2006.

C. Sun and M. Aiello. Towards variable service compositions using VxBPEL. In *High Confidence Software Reuse in Large Systems*, pages 257–261, 2008.

C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello. Modelling and managing the variability of web service-based systems. *Journal of Systems and Software*, 83:502–516, 2010.

J.L. Szwarcfiter and P.E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16(2):192–204, 1976.

J.C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Communications of the ACM*, 13(12):722–726, 1970.

N. Trčka, W.M.P. van Der Aalst, and N. Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, 2009.

N.R.T.P. van Beest, P. Bulanov, J.C. Wortmann, and A. Lazovik. Resolving business process interference via dynamic reconfiguration. In *8th International Conference on Service Oriented Computing (ICSOC-2010)*, volume 6470/2010, pages 47–60. Lecture Notes in Computer Science, 2010.

N.R.T.P. van Beest, P. Bulanov, J.C. Wortmann, and A. Lazovik. Automated runtime repair of business processes. Technical report, University of Groningen, 2012.

N.R.T.P. van Beest, M. Dumas, L. García-Bañuelos, and M. La Rosa. Log delta analysis: Interpretable differencing of business process event logs. In *Business Process Management*, pages 386–405. Springer International Publishing, 2015.

W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer Berlin Heidelberg, 1997.

W.M.P.. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8:21–66, 1998.

W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management*, pages 161–183. Springer, 2000.

W.M.P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1):125–203, 2002.

W.M.P. van der Aalst and S. Jablonski. Dealing with workflow change: identification of issues and solutions. *Computer systems science and engineering*, 15(5): 267–276, 2000.

W.M.P. van der Aalst and M. Pesic. *DecSerFlow: Towards a truly declarative service flow language*. Springer, 2006.

W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003a.

W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business process management: A survey. In *Business Process Management*, pages 1–12. Springer-Verlag, 2003b.

W.M.P. van der Aalst, A. Dreiling, F. Gottschalk, M. Rosemann, and M.H. Jansen-Vullers. Configurable process models as a basis for reference modeling. In *Business Process Management*, pages 512–518. Springer, 2005.

W.M.P. van der Aalst, M. Dumas, F. Gottschalk, A.H.M. Ter Hofstede, M. La Rosa, and Jan Mendling. Preserving correctness during business process model configuration. *Formal Aspects of Computing*, 22(3-4):459–482, 2010.

W.M.P. van der Aalst, Kees M. van Hee, A.H.M. ter Hofstede, Natalia Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.

W.M.P. van der Aalst, N. Lohmann, and M. La Rosa. Ensuring correctness during process configuration via partner synthesis. *Information Systems*, 37(6):574–592, 2012.

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. *Computer and Industry*, 58(6):578–601, August 2007.

T. van Eijndhoven, M.E. Iacob, and M.L. Ponisio. Achieving business process flexibility with business rules. In *Enterprise Distributed Object Computing Conference*, pages 95–104. IEEE, 2008.

B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66:438–466, 2008.

I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: on the verification of semantic business process models. *Distributed and Parallel Databases*, 27:271–343, 2010.

M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Business process verification – finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.