Automatic Synthesis and Verification
of Industrial Commissioning Processes

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Richard Mrasek

aus Oberkirch

Tag der mündlichen Prüfung:	27.06.2016
Erster Gutachter:	Prof. Dr.-Ing. Klemens Böhm
Zweiter Gutachter:	Prof. Dr. Andreas Oberweis

Acknowledgment

The completion of this thesis would not have been possible without the support of and being in exchange with so many people to whom I owe lots of gratitude.

First of all, my special thanks goes to my advisor Prof. Dr.-Ing Klemens Böhm. He guided me patience and encouragement. His comments on my publications drafts were plenty and always helpful. I deeply thanks my co-advisor Prof. Dr. Andreas Oberweis for his support.

My sincere thanks to the associates of Audi AG for the effective and enjoyable cooperation. Especially Dr. Christian Allmann, Michael Becker, and Sascha Nietert for their support with the project. I also deeply thank my colleagues at the Information Systems Group for fruitful discussions and a good working atmosphere, especially to Jutta Mülle for her expertise in the workflow research and my office mate Fabian Laforet for his support.

I am grateful to my girl friend Christina Vinke, who always had an open ear for me and helped me with her advice or just by listening.

Kurzfassung

Ein wichtiger Schritt in der Fahrzeugproduktion stellt die Inbetriebnahme des Fahrzeugs dar, die hauptsächlich aus den beiden nachfolgend beschriebenen Tätigkeiten besteht.

› Überprüfung der ordnungsgemäßen Funktion aller Komponenten des Fahrzeugs

› Installation (flashen) der Software auf den Steuergeräten des Fahrzeugs

Zur Durchführung dieser Tätigkeiten werden die Fahrzeuge zu verschiedenen Prüforten gefahren oder transportiert. An jedem Prüfort wird das Fahrzeug von einem Mitarbeiter an ein Diagnosesystem angeschlossen. Das Diagnosesystem ist ein spezielles *workflow system* für die Inbetriebnahme. Es aktiviert verschiedene Operationen im Fahrzeug und präsentiert dem Mitarbeiter die manuellen Aufgaben, die er auszuführen hat, über ein Handterminal. Die sequentielle Anordnung und Parallelisierung dieser Operationen wird durch den Inbetriebnahmeablauf beschrieben. Prüfprogrammierer entwickeln mit Hilfe von Entwicklungswerkzeugen diese Abläufe. In der Regel sind die Inbetriebnahmeabläufe komplex. Typischerweise enthält ein Ablauf mehrere hundert Operationen, angeordnet in bis zu 14 parallelen Kanälen. Jedem Prüfort, und damit jedem Prozessmodell, ist eine geplante Anzahl an Takten zugeordnet. Bevor das Zeitlimit der Takte erreicht ist, muss die Ausführung der Abläufe beendet sein. Anderenfalls kann dies zu großen Komplikationen und Störungen im Produktionsablauf führen. Aktuell werden diese Abläufe von einer Abteilung geplant und implementiert. Wie nachfolgend beschrieben, gibt es einige Trends und Entwicklungen, wodurch die Komplexität noch größer wird und die damit verbundenen Kosten für die Inbetriebnahme weiter ansteigen.

› Mit jeder neuen Fahrzeuggeneration wächst die Anzahl der Assistenz- und Medien-Systeme. Diese neuen Systeme benötigen oft neue Steuergeräte und Operationen, um in Betrieb genommen zu werden. Dies führt daher zu umfangreicheren und komplexeren Inbetriebnahmeabläufen.

› Zugleich steigt die Anzahl an unterschiedlichen Fahrzeugserien und Varianten. Jede Fahrzeugserie benötigt mehrere Inbetriebnahmeabläufe. Aufgrund der unterschiedlichen Komponenten in jeder Serie ist es kaum möglich, einen Ablauf wiederzuverwenden. Die wachsende Anzahl an Fahrzeugserien führt deshalb zu einem Wachstum an unterschiedlichen Inbetriebnahmenabläufen, die entwickelt und gewartet werden müssen.

› Der Lebenszyklus einer Fahrzeuggeneration verringert sich kontinuierlich. In immer kürzer werdenden Zeitabschnitten wird eine neue Generation entwickelt und auf den Markt gebracht. Dies führt zu neuen Inbetriebnahmeabläufen.

› Zur Erhöhung der Effizienz der Produktion werden die Taktzeiten in den Produktionsstätten immer weiter reduziert. Dies hat auch Auswirkungen auf die Inbetriebnahme. Die einzelnen Abläufe müssen daher immer zeiteffizienter geplant werden.

Diese Trends führen zu einem Anstieg der Anzahl und Komplexität der Inbetriebnahmeabläufe. Zugleich existieren für jedes Prozessmodell hunderte von Abhängigkeiten, die das Modell erfüllen muss. Zum Beispiel benötigen einige Operationen als Vor- bzw. Nachbedingung eine oder mehrere weitere Operationen. Zudem haben die verwendeten Ressourcen einiger Operationen nur eine beschränkte Kapazität. Die Qualität der Inbetriebnahme zu garantieren, wird daher immer aufwendiger und komplizierter. Folglich steigen somit auch die Entwicklungskosten. Aufgrund des hier beschriebenen Sachverhalts ergeben sich die folgenden Hauptbeiträge dieser Arbeit:

1 **Verifikation.** Zur Qualitätssicherung wollen wir verifizieren, ob ein Prozessmodell alle geforderten Abhängigkeiten erfüllt. Zu diesem Zweck führen wir eine Modellprüfung (model checking) durch. Die automatische Modellprüfung erfordert eine Interpretation des Prozessmodells als ein formales Modell. Eine solche Interpretation präsentieren wir in dieser Arbeit in Form einer von uns entwickelten Transformation.

2 **Synthese.** Zur Unterstützung der Entwicklung von Inbetriebnahmeabläufen entwickelten wir einen Algorithmus für die Synthese von Abläufen aus einer Spezifikation der Abhängigkeiten und zusätzlichen Optimierungskriterien. Unser Algorithmus exploriert den Zustandsraum der korrekten Prozessmodelle, um eine gute Lösung zu gegebenen Qualitätskriterien, wie z. B. der Durchlaufzeit, zu finden.

Sowohl die Verifikation als auch die Synthese benötigen eine Liste von Abhängigkeiten, die das Prozessmodell erfüllen muss. Die Spezifikation dieser Abhängigkeiten stellt eine komplizierte Problemstellung mit zahlreichen Herausforderungen dar. Einerseits müssen die Abhängigkeiten formal beschrieben werden, z. B. in einer temporalen Logik, um die automatische Verifikation oder

Synthese zu ermöglichen. Andererseits besitzen die Anwender häufig nur wenig Erfahrung bezüglich der formalen Notationen. Daraus erschließt sich der dritte Hauptbeitrag unserer Arbeit.

3 **Spezifikation.** Im Rahmen eines Vorstudiums konnte festgestellt werden, dass nur wenige Muster von Abhängigkeiten existieren, die ein Prozessmodell erfüllen muss. Diese Muster für abstrakte Abhängigkeiten wurden von uns gesammelt und formal beschrieben. Die konkrete Ausprägung der einzelnen Abhängigkeiten ist abhängig vom Kontext der Prozessmodelle und ihrer Ausführung, z. B. von den Komponenten eines Fahrzeugs oder vom Prozessort. Wir entwickelten ein Informationssystem, um diese Abhängigkeiten aus den Mustern zu instanziieren und anschließend zu verifizieren.

Wir evaluierten unsere Beiträge anhand realistischer Prozessmodelle unseres Industriepartners der Audi AG. Unsere Ergebnisse zeigen eine beträchtliche Verbesserung der Qualität in den Inbetriebnahmeabläufen. Die Synthese führt zu einer signifikanten Verbesserung gegenüber dem manuellen Entwurf. Durch Experteninterviews konnte festgestellt werden, dass unsere Spezifikation benutzerfreundlich und gut geeignet für ein echtes Produktionsumfeld ist.

Abstract

The topic of this doctoral dissertation is the verification and synthesis of processes, i. e., work-flows. Verification is the check if a given process model fulfills all necessary properties. Synthesis is the automatic generation of a process model from a set of properties. The running example of the thesis and the use case for the evaluation is the commissioning of vehicles in the automobile production.

Commissioning consists of two major exercises: First, testing if all parts of the vehicle function properly. Second, installment (flashen) of the software on the control units built in the vehicle. The sequential arrangement and parallelization of these operations are planned as a commissioning process model. Process developers define these process models with development tools. Commissioning processes are inherently complex. Typically, there are hundreds of tasks for each vehicle, arranged in up to 14 parallel lanes. The state of the art is that a dedicated department plans and implements the process models using authoring tools. There are several major factors that constantly increase the complexity and cost of the process model design over time. For instance the number of assisting/media components are increasing with each vehicle generation, leading to an increase in the size of each individual commissioning process. At the same time the number of different vehicle series and variants are increasing. Each vehicle series requires several commissioning processes. Due to the different components it is hardly possible to use the same process model for more than one vehicle series. This leads to an increase in the number of process models which must be designed and maintained. In combination, this leads to constant increase in the number of process models and their complexity. At the same time for each process model several hundred properties exist which the model needs to adhere to. For instance, some operations require other operations as pre- or postcondition, or some resources used by operations have limited capacity. We want to support the development of high-quality commissioning processes. In particular the major contributions of our work are:

1 **Verification.** To ensure a high quality of the commissioning processes we verify if a process model fulfills all required properties. To this end, we apply a model checking approach. For an automatic verification, e. g., model checking,

it is necessary to transform the process to a formal model, representing the necessary aspects.To this end, we develop a transformation of the process models to a Petri net representation preserving the necessary information for the verification. The high concurrent nature of commissioning processes leads to performance issues with model checking. To this end, we propose to tailor the model transformation to each respective property. For each property, our algorithm identifies the region of interest in the process model and only transforms these regions to a Petri net.

2 **Synthesis.** We want to support the development of the commissioning processes. To this end, we develop an algorithm to synthesize a process model from the properties the model needs to adhere to. In contrast to related work we do not limit ourselves to cases where the dependency only allows one possible process model. Additional constraints and optimizing criteria exist. Our algorithm explores the space of correct process models to find a good solution according to quality criteria, e. g., total processing time. We focus on the restricted case that there are no repetitions. There is a number of settings with this characteristic, for instance in manufacturing. In particular, loops are unnatural in commissioning processes, since a feature is tested only once. On the other hand, if a problem occurs and is fixed, a new commissioning process is started. Our algorithm is able to handle complex large specifications efficiently. In contrast to related work that are only able to process small scale specifications.

Both the verification and the synthesis require a list of properties a process model needs to adhere to. To specify these properties is itself a denoted problem with several challenges. On the one hand, the properties need to be in a formal language, e. g., temporal logic, to allow automatic verification and synthesis. On the other hand, domain experts hardly have the knowledge to specify those formal notations. This leads to the third major contribution, the user-friendly specification of properties:

3 **Specification.** We have observed that there are few pattern properties that testing processes adhere to, and we describe these patterns. They depend on the context of the processes, e. g., the components of the vehicle or the testing stations. We have developed a framework that instantiates the property patterns at verification time and then verifies the process against these instances.

We evaluate the approaches using realistic process models from our industrial partner, the Audi AG. The verification framework leads an improvement of the quality of commissioning processes. In our analysis we could detect issues in 90% of the commissioning processes. 23% contains major disturbances that may influence the production flow negatively. The synthesis features a significant improvement compared to the manual generation. Our simulation with vehicle-specific process models yields a runtime reduction of in average 44% compared to manual generation. From expert interviews we conclude that our framework for the specification is user-friendly and well suited to operate in a real production environment.

Contents

Chapter 1

Introduction

Putting a vehicle into commission presents an important step in the production process. The commissioning comprises of the two major exercises: the vehicle function test regarding to all parts of the vehicle and the installment (*flashen*) of the software on the control units build in the vehicle. To this end, each vehicle is driven to several testing stations. In each station a factory worker connects the vehicle to the diagnostic system, i. e., a workflow system for the commissioning, cf. [ZS11], using MPS[1]. MPS invokes several operations on the vehicle and displays the workers manual task over a hand-terminal.

The sequential arrangement and parallelization of these operations, both manual and automatic, are planned as commissioning process model. Process developers define these process models with development tools. Vehicle commissioning includes, say, checking for each vehicle produced, whether all electronic control units are integrated correctly and putting them into service. ECUs are components built in to the vehicle in order to control specific functionalities of the car, e. g., for controlling the engine electronics. Each ECU needs to be tested and put into operation, e. g., by installing certain software. To this end, the server of the diagnostic system sends the process model to the MPS, a portable computer system. The MPS is connected to the ECUs build in the vehicle in order to execute automatic tasks on the vehicle, and to delegate manual tasks to a factory worker over a hand terminal.

Figure 1.1 shows the general architecture of a diagnostic system. Commissioning processes displays characteristic to be complex. Typically there are hundreds of tasks for each vehicle, arranged in up to 14 parallel lanes. Each testing station, and thus process model, uses a preplanned number of production cycles. The execution of the models needs to be complete in this time limit or can either cause a major disturbance of the production.

The state of the art is that a dedicated department is in charge of the planning and implementation of those process models using authoring tools. There are

[1]Acronym of the german *Mobile Prüfstation* (engl. *mobile testing station*)

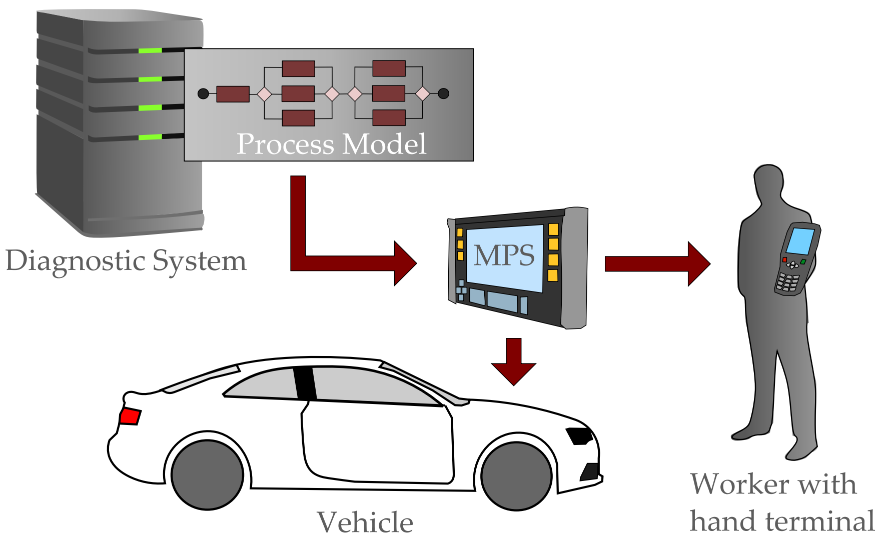


Figure 1.1: The Simplified Architecture of a Diagnostic System

several major factors for increasing the complexity and costs of the process model design.

› The number of assisting and media-components are increasing with each vehicle generation. Those new components require more control units and thus operations to put them into commissioning. This leads to an raise in the size of each individual commissioning process, cf. [Har09].

› At the same time the number of different vehicle series and variants are increasing. Each vehicle series requires several commissioning processes, cf. [Rei10, p.48]. Due to the different components it is hardly possible to use the same process model for more than one vehicle series. This leads to a raise in the number of process model to design and maintain.

› The lifespan of each generation is constantly decreasing, cf. [Rei10, p.50]. This results in shorter development times of a new vehicles and new commissioning process models.

› In order to increase the efficiency of the production the number and length of the production cycles are constantly decreasing. This means that the commissioning processes need to use the available time more efficiently.

In combination, this leads to a constant increase in the number of process models and their complexity. At the same time for each process model several hundreds of properties exist the model needs to adhere to. For instance some operations require other operations as pre- or postcondition, or some resources used by operations have limited capacities. In order to ensure the quality of the commissioning process models are getting more and more complex and their development costs are constantly increasing.

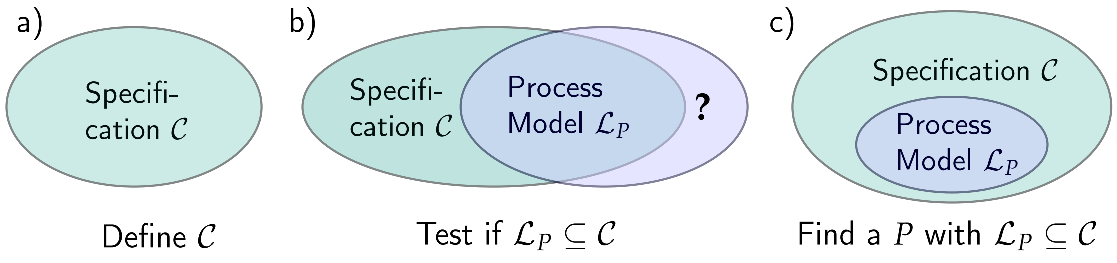

a)
Specification $\mathcal{C}$
Define $\mathcal{C}$
b)
Specification $\mathcal{C}$
Process Model $\mathcal{L}_P$
?
Test if $\mathcal{L}_P \subseteq \mathcal{C}$
c)
Specification $\mathcal{C}$
Process Model $\mathcal{L}_P$
Find a P with $\mathcal{L}_P \subseteq \mathcal{C}$

Figure 1.2: Our Problem Statements Specification (a), Verification (b), and Synthesis (c).

1.1 Contribution

In this thesis we are going to support process modeling, investigating testing schemes in responds to, whether a given process model fulfills all properties required (*process verification*), and approaches for semiautomatic generation of process models (*process synthesis*). For *process verification* and *process synthesis* one need the *specification* of the allowed behavior of the process models. Formally, let P be the process model of a commissioning process, and $\mathcal{L}_P$ denotes the complete log of the process, i. e., all possible traces of the process model. Let $\mathcal{C}$ denote the set of all traces allowed by the properties. We now can define *Specification*, *Verification*, and *Synthesis* as follows:

Specification	:	Define the set of allowed traces $\mathcal{C}$
Verification	:	For a given process model P check if $\mathcal{L}_P \subseteq \mathcal{C}$
Synthesis	:	Generate a process model P with $\mathcal{L}_P \subseteq \mathcal{C}$

Figure 1.2 illustrates our three major research fields: *specification*, *verification*, *synthesis*. In the following subsection we will describe our major contribution to each of those research fields.

1.1.1 Specification

Specification is one of the most frequent words in computer science. Therefore, the course of this thesis we use the definition of Axel van Lamsweerde:

> »Generally speaking, a formal specification is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy.« [Lamoo]

Regarding the process modeling field we can specify execution in two different ways. On the one hand, we can describe the actual execution using an imperative description or on the other hand describe the properties and constraints of all the allowed behavior in a declarative way. We call the imperative specification of a process its model, and the activity of imperative specification as the modeling of the process. If not stated otherwise, we employ the word specification only for the description of the declarative properties a process needs to adhere to. On the one hand, most of the workflow systems can only execute imperative process model such as **B**usiness **P**rocess **M**odel **N**otation (BPMN), **Y**et **A**nother **W**orkflow **L**anguage (YAWL), or **O**pen **T**est sequence e**X**change (OTX). On the other hand, it is only possible to describe the regulations and constraints of the processes in a declarative way in general. Therefore, *Verification* and *Synthesis* are two techniques employed in order to overcome this issue. Both techniques require the declarative specification of the properties beforehand, which itself is a complex issue.

The declarative specification gives way to several challenges. The knowledge about the properties processes need to adhere to is typically distributed between different employees in different departments. Documentation is often missing, and property issues known by individual employees. The properties itself are context-sensitive, i. e., holds only in specific context situation. For instance some properties only exist at some testing station for some vehicles. In general several hundreds of properties exist for an individual vehicle. In order to allow automatic verification and synthesis techniques the properties need to be available in a formal language, e. g., temporal logic. The specification in these languages is complicated and error-prone regarding to the domain experts. It is not possible for domain experts to give a succinct list of all properties by interview. The experts can decide, if a presented candidate is a property or not, but any given list is incomplete. Our major contributions of this thesis for specification are:

Property Instantiation. We systematically collect all properties and the relevant information for their specification by a series of interviews with domain experts.

We map most of the found properties onto a distinct set of classes, called property patterns. We collect and store the relevant information of the specification into a database of context knowledge. Before verification of a concrete process model, we extract the context, query the database with the context and use the results in order to instantiate the abstract property patterns to concrete property instances. Within this approach we automate most of the declarative specification, see Section 4.2 for further details.

Detection of Property Candidates. In order to support the initial detection of the properties we use an automatic detection of possible property candidates. To this end, we use a novel statistical analysis of a repository of existing process models. This analysis returns a set of frequent and strong property candidates. The manual classification reveals that the candidates contain actual properties and reveals additional useful information about the processes, see Section 4.3 for furthermore details.

1.1.2 Verification

According to the guidelines of the Institute of Electrical and Electronics Engineers (IEEE) verification is:

> »The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition.« [IEE11]

In our case the system is the imperative process model for commissioning and the specification are the declarative properties. The verification of the commissioning process models gives way to several challenges.

In order to allow an automatic verification of process models we need an interpretation of the execution semantic to generate the state space. The standard of our employed notation (OTX) does not provide a formal description of the semantic. The commissioning process models are in general highly concurrent and relative large in size. In combination this leads to an exploding state space and renders full-state verification impractical. Communicating the source and the reason of a property violation is not trivial. Verification techniques give at best a trace of transition that leads to a violating state. It is not efficient for the domain experts to detect the source of the property violation by this transitions trace. Therefore major contributions in the field of verification are:

Transformation to Petri Net. Automatic techniques in order to analyse OTX process models require an interpretation of the execution semantic. We provide this by the transformation of the OTX process to the formal nation of Petri Nets. Our transformation preserves the necessary information for the verification. Petri nets allow for an efficient generation of their state space, see Section 5.1 for furthermore details.

Relevance Optimization. In order to reduce the complexity of the verification to a comprehensive level we introduce a novel technique. Instead of generating one Petri net for all properties to verify, we generate Petri nets tailored to each property. To this end, we detect the relevant regions in the process model for the property. Next our algorithm prunes the transformation in such a manner that it only contain those regions. This allows us verifying several hundred properties on complex highly-concurrent process models in seconds, see Section 5.2 for further details.

Property Reporting. In the case of violation the verification algorithm returns a trace of transition leading to the violating state. For each property pattern we define a procedure in order to detect the relevant elements in the process model. We highlight those elements in a visualization for communicating the cause and reason of the violation to the domain experts, see Section 5.3 for furthermore details.

1.1.3 Synthesis

Synthesis means combining several objects to a new entity. In context of process modeling synthesis, this is the combination of properties in order to generate an imperative process model, see [Awa+11; Yu+08]. Several challenges exist for the synthesis.

The synthesized process models should be applicable to our use case, thus the process model has to be a block-structured ones. There often exists a great variety of models that fulfill the properties. For illustration, the sequential arrangement of n nodes, in the absence of any constraint, give way to $n!$ different process models. This is required in order to synthesize a good process model according to quality criteria out of the possible ones, e. g., processing time should be short and more important according to the lean production paradigm, cf.[Wom+07, p. 54] the waiting time of the worker should be reduced. It should be possible to apply these approaches to the large real-to life specifications of our use case. We target those challenges by the following major contributions.

Automatic Generation of Process Models. We developed a novel approach for the synthesis of block-structured process models from declarative properties. In contrast to related work we can synthesize from an incomplete specification, i. e., a solution space containing more than one valid process model. In those cases we apply a probability algorithm in order to find a good solution according to predefined quality criteria, e. g., throughput time, see Section 6.2 for furthermore details.

Use Case. We deploy our algorithm for synthesis to a large realistic application: The design of the new commissioning processes of a new compact executive car series. The use case leads to several new challenges for the generation and to a several additions regarding the general synthesis framework, see Section 6.3 for further details.

1.2 Overview

This thesis consists out of 7 chapters. Besides this section aiming at given short introduction into the topic the other chapters displays the following information.

Chapter 2: Commissioning of Vehicles. This chapter introduces to our scenario and our research application field: the commissioning of vehicles. In this section we focus on technical aspects of the commissioning relevant for the properties of the process model. In detail we introduce the bus systems and protocols of vehicle in Subsection 2.1, the commissioning operations in Subsection 2.2, and the architecture of a Diagnostic System in Subsection 2.3.

Chapter 3: State of Research. In this chapter we describe the state of the art in the relevant fields of business process research. We give an introduction to the fields that are close to our research agenda and describe the differences and similarities, i. e., imbedding our work into the recent field of research. Namely we name the differences between imperative and declarative modeling in Subsection 3.1, the specification of properties in Subsection 3.2, the verification in Subsection 3.3, the automatic generation in Subsection 3.4, and the discovery of process models from a log of previous executions in Subsection 3.5.

Chapter 4: Property Specification. This chapter focuses on the declarative specification of properties process models have to adhere to. Subsection 4.1 discusses the different temporal logics for formal specification and comprises our reasoning. Subsection 4.2 presents our approach for context-sensitive instantiation of property patterns. Subsection 4.3 displays our approach for the detection of property candidates from a repository of process models. Subsection 4.4

comprises the evaluation of the approaches. Subsection 4.5.1 concludes this chapter with a review of the related work.

Chapter 5: Process Verification. In this chapter we display the way to verify if a given process model fulfills all required properties. Subsection 5.1 presents our transformation of an OTX process tree into formal model of Petri nets. Subsection 5.2 presents our novel algorithm aiming to optimize the verification to the relevant aspects of each property. Subsection 5.3 describes the way to efficiently report violations of properties to the user. Subsection 5.4 evaluates the verification and we conclude with a review of the related work in process verification, see Subsection 5.5.

Chapter 6: Process Synthesis. This chapter focuses on the way to synthesize an imperative process model from declarative properties. Subsection 6.1 presents an approach based on the resource constraint scheduling and explains the major drawbacks of approaches based on scheduling. Subsection 6.2 shows our major synthesis algorithm. It is based on modular decomposition and a probabilistic search for the under-specified regions. Subsection 6.3 applies this algorithm to a real use-case: The commissioning of a new compact executive car series. Subsection 6.4 evaluates the synthesis approach and in Subsection 6.5 we discuss the related work for the synthesis.

Chapter 7: Conclusion. Within this conclusion chapter, Subsection 7.1 recaps the major contributions of our work and summarize our achievements. In addition to this, Subsection 7.2 gives a short outlook on the research being influenced by this work and sketches the leverages of our research in regard to different applications fields.

Chapter 2

Commissioning of Vehicles

In this chapter we introduce the scenario of this project, namely the commissioning of vehicles in the automobile industry. The chapter focuses on the relevant aspects for the project. For a more comprehensive overview see the literature, e. g., [ZS11], or the respective industry standards, e. g., [ISO12].

2.1 Bus Systems and Protocols

Figure 2.1 illustrate the simplified bus systems in a modern vehicle. The different bus systems are the result of different application fields for vehicle communication, described in [ZS11]. The application fields can be classified into on-board, i. e., communication between components inside a vehicle, and off-board ones, i. e., communication between components in the vehicle and systems outside the vehicle.

The on-board communication consists of three application fields with different communication protocols. The first field is the real-time controlling operations, e. g., the electronic controlling of the engine or the brakes. The necessary informations are often only a few bytes. Due to the security risk the bus systems must support a low latency and high availability. The **C**ontroller **A**rea **N**etwork (CAN) bus system, or the newer CAN FD or FlexRay are used for these applications. Second, for simple task, e. g. the controlling of the door or the light systems a simplified CAN systems in cooperation with **L**ocal **I**nterconnect **N**etwork (LIN) systems are used. LIN is a bus system for cost-efficient communication. The communications partner in a LIN bus are arrangement in a master/slave relationship. A master can use up to 16 slaves. Third, for the infotainment system, e. g., navigation and multimedia, a high amount of data are transported. The latency and availability is only secondary. The **M**edia **O**riented **S**ystems **T**ransport (MOST) protocol is often used for the infotainment system.

The off-board communication consist of two main applications fields. The first field is the diagnosis in a car workshop for repairing and exhaust emission test.

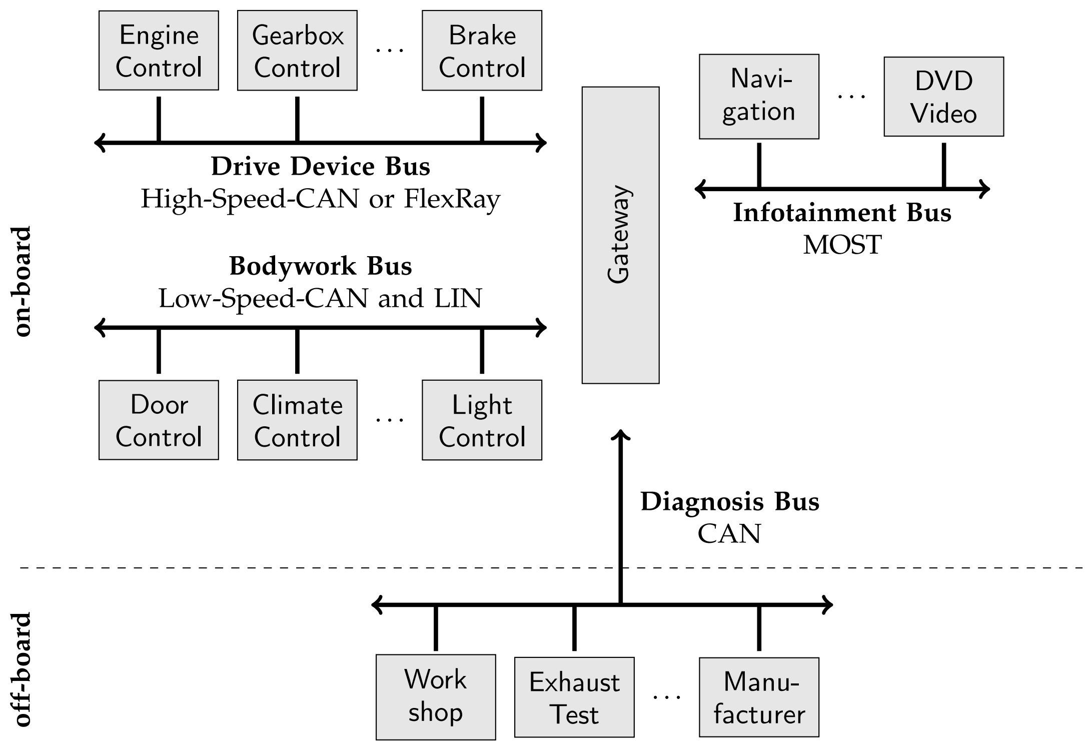


Figure 2.1: Bus Systems of an Automobile, simplified after [ZS11]

The communication require a high data rate and protection mechanisms, the latency and availability is only secondary. Federal standards exists for the communication (US OBD, European OBD) but manufacturer specific standards are common. The second field is the initial installing of software on the **E**lectronic **C**ontrol **U**nit (ECU) (flashen) and the testing of the components inside the manufacturing factory. For these application field the same communication protocols are used, often in a modus allowing for a higher data rate.

CAN, FlexRay, and LIN describes the first two Layers in the OSI-Model. The **K**ey**W**ord **P**rotocol 2000 (KWP2000) protocol and the **U**nified **D**iagnostic **S**ervices (UDS) protocol are communication protocols on the application layer of the OSI model for communication with control units.

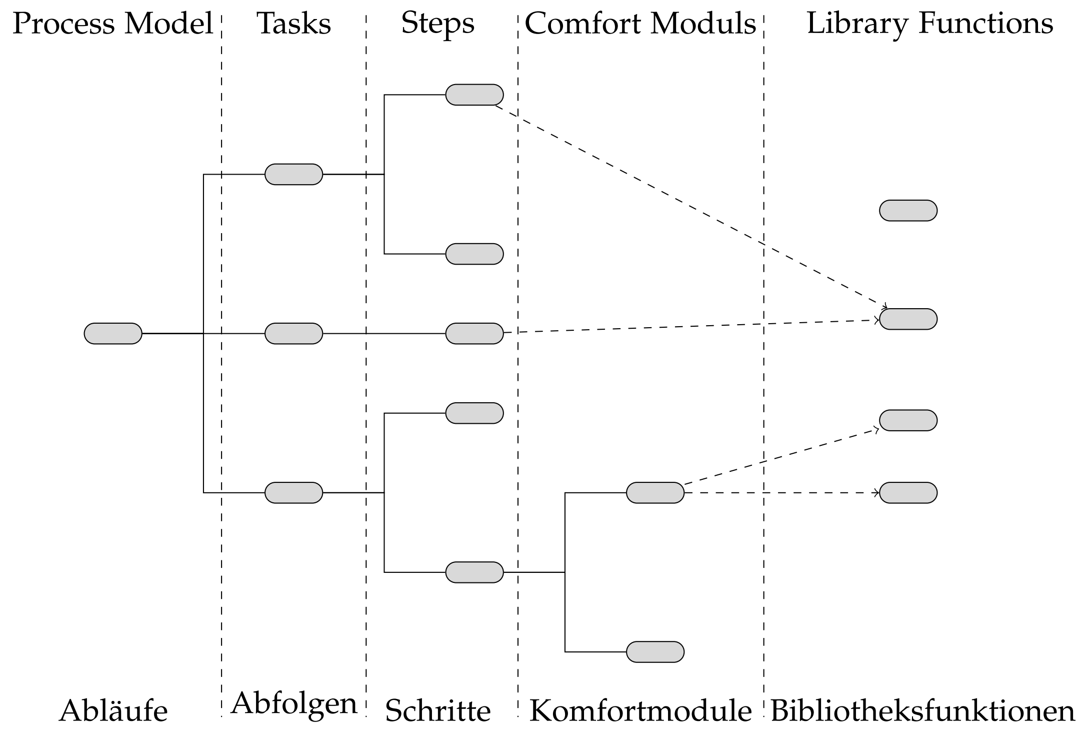


Figure 2.2: The Module Hierarchy of a Commissioning Process, with the Official German Names Below and the English Translation Above.

2.2 Commissioning Operations

The Figure 2.2 shows the module hierarchy of the commissioning operations. You find the nomenclature used in the later chapters. Below you find the nomenclature of the AUDI AG. These thesis mainly focuses on the first two layers the process model and the tasks.

› *Process Models (Abläufe)*: The commissioning process model is the highest entity in the hierarchy. Process models describe the way to put a given vehicle in commission. To this end, for each vehicle a series of operations are executed. The process model describes the arrangement and parallelization of those operations.

› *Tasks (Abfolgen)*: Tasks describe operations performed on electronic control units (ECU) in the vehicle. The operation can be automatic or requires a factory worker. Most of the tasks have preconditions that describe the execution conditions, e. g., specific vehicle settings. The operations consist of several steps. An example for a commissioning task would be `A_ENG__Check_EM`, i. e., the check of the error memory (EM) for the control unit ENG (Engine Control).

› *Steps*: Steps describe the concrete realization of a task. To this end, these steps use prebuild comfort modules or call directly library functions of the Diagnostic System.

› *Comfort Modules*: Comfort modules are prebuild modules to support development of tasks and steps.

› *Library functions*: Library functions are the lowest level in the module hierarchy. They provide the basic operations, e. g., data transfer. Steps or comfort modules call the library functions. The Diagnostic System, e. g., Sidis Pro of the Siemens AG, provides the library functions.

2.3 Diagnostic System

Diagnostic Systems in the automobile industry are workflow systems, see [AH04], for the commissioning of vehicles. The basic system architecture follows the workflow reference model of the *Workflow Management Coalition (WfMS)*, cf. [HSK04]. We will show examples using the diagnostic system *Open Test Framework (OTF)* of the emotive GmbH. Similar diagnostic systems exist from other vendors like PRODIS.Automation from DSA or SIDIS PRO from Siemens using different names for the components. The workflow reference model defines six major components in a workflow system:

› *Process Definition Tools:* Process definition tools are used to design the tasks and the process models. The final result output of the process definition tools are process models interpreted by the work flow engine at runtime. For the commissioning several different notations for process models exist, namely SIDIS PRO, PRODIS.Automation, and OTX. The process definition tools of the *Open Test Framework (OTF)* is the *OTX-Designer*.

› *Workflow Enactment Service:* The workflow enactment service consist of one or many workflow engines. A workflow engine provides the run time execution of an instance of a process model. In the *Open Test Framework (OTF)* the workflow engine is called *OTX runtime environment*. The engine interprets OTX process models and generates binary files for a high-performance execution.

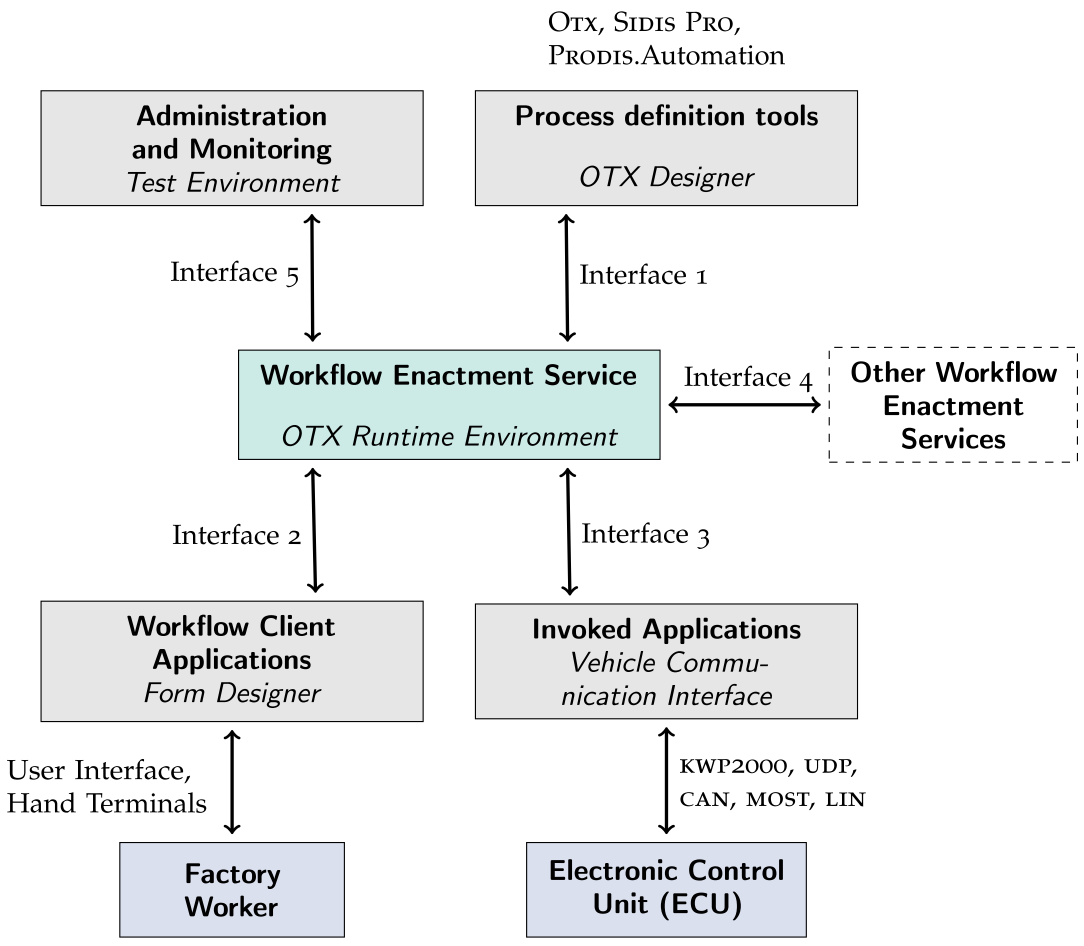

OTX, SIDIS PRO,
PRODIS.Automation
Administration and Monitoring
Test Environment
Process definition tools
OTX Designer
Interface 5
Interface 1
Workflow Enactment Service
OTX Runtime Environment
Interface 4
Other Workflow Enactment Services
Interface 2
Interface 3
Workflow Client Applications
Form Designer
Invoked Applications
Vehicle Communication Interface
User Interface, Hand Terminals
KWP2000, UDP, CAN, MOST, LIN
Factory Worker
Electronic Control Unit (ECU)

Figure 2.3: The Workflow Reference Model for Commissioning Processes

› *Workflow Client Applications:* The workflow client applications define how the workflow engine interacts with human workers. To this end, *Form Designer* of the *Open Test Framework OTF* allow to design custom user interfaces.

› *Invoked Applications:* The invoked Application components define how the workflow engine interacts with applications outside of the engine. In the context of commissioning these applications are mainly executed on ECUs build in the vehicles. To this end, the workflow engine communicate over the protocols in Section 2.1 with the the ECUs. **O**pen **D**iagnostic Data e**X**change (ODX) is an e**X**tensible **M**arkup **L**anguage (XML)-based data format defining the structure of the messages. In the *Open Test Framework OTF* the Vehicle Communication Interface (VCI) is used for invoked applications.

› *Other Workflow Engines:* The interface 4 defines the way the workflow engine interacts with other workflow engines. In general, diagnostic systems do not communicate with other diagnostic systems and thus no implementation is given in the *Open Test Framework OTF*.

› *Administration and Monitoring Tools:* The administration and monitoring tools define the way to administrate process models and to monitor the execution of process instances. The *Open Test Framework OTF* allows to debug process models in a test environment and allows for simulations.

Figure 2.3 shows the workflow reference model, with five interfaces and their counterpart in the *Open Test Framework OTF*. Other diagnostic system are Sidis Pro and Prodis.Automation. OTX is a new industry standard for commissioning process models.

Sidis Pro is a diagnostic system for the commissioning of vehicles from siemens. It contains an authoring tool for the graphical development of commissioning processes, simulation and analysis suites and as well as runtime environment for the execution of the process models. The block-based process models are serialized in a proprietary XML format.

Prodis.Automation is a diagnostic system for commissioning processes from DSA (Daten- und Systemtechnik GmbH). The main component is a graphical process definition tool for commissioning processes. Prodis.Automation provides a system to define Client Applications called *Controls*, the simulation of process models and their analysis. Prodis.Automation uses a proprietary XML format to serialize the commissioning process, but is able to support OTX process models. Prodis.RTS is the runtime environment compatible to Prodis.Automation.

Open Test Sequence eXchange

Open Test Sequence eXchange (otx) is an open ISO standard that »*[...] proposes an open and standardized format for the human- and machine-readable description of diagnostic test sequences.*«, cf [ISO12, p. 6]. OTX is an imperative block-structured language, i. e., the steps are given implicitly and no explicit jumps are allowed. OTX is serialized as XML document. The hierarchical format of an XML document enforces the block-structure and renders XML techniques, e. g., schema validation, possible. It is possible to directly write OTX in its XML source code. The standard recommends using visual specification tools. The standard does not describe explicitly a visualization of the OTX elements, however the examples are given as UML activity diagrams. It is possible to define several process models in one OTX document. Data objects can be global (for all process models in the file) or local (only visible in one process model). Next we will give a short overview over the most important node types in OTX.

`<otx>` is the root node of any OTX-file. It contains several attributes describing meta information like the time-stamp or the OTX version. The `<procedures>` node contains one or more `<procedure>` nodes. Each `<procedure>` is a process model. The `<realisation>` node contains the concrete realization of his parent node. In the case of a `<procedure>` node the steps of the process. `<action>` is used to represent an atomic operation. The `<realisation>` child node specifies the type of operation, e. g. data operators or the method call to an ECU in the vehicle. In our use case the `<action>` nodes model the tasks. Several node types exist to model control flow constructs. `<flow>` describe the sequential execution, `<parallel>` an AND-Gateway, `<branch>` an XOR-Gateway, and `<loop>` specifies iterations.

In Subsection 5.1 we will give a more detailed description of the OTX elements and provide a translation of the execution semantics to the formal model of Petri nets.

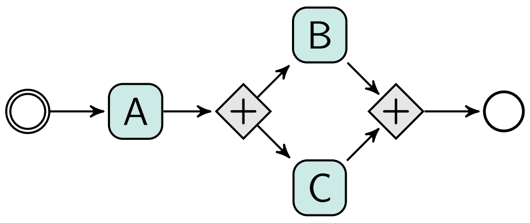


Figure 2.4: The Process Model Described in Example 2.3.1 in BPMN Notation

Example 2.3.1 . The following describes a simple OTX process file with three Tasks *A*, *B*, *C*. Figure 2.4 shows the same process model in BPMN notation.

```
<?xml version="1.0" encoding="utf-8"?>
<otx [...]>
  <procedures>
    <procedure name="p1" visibility="PUBLIC" id="1">
      <realisation>
        <flow>
          <action id="Action_A">
            <realisation xsi:type="ProcedureCall"
            procedure="Task_A"></realisation>
          </action>
          <parallel id="p1" name="par1">
            <realisation>
              <lane>
                <action id="Action_B">
                  <realisation xsi:type="ProcedureCall"
                  procedure="Task_B"></realisation>
                </action>
              </lane>
              <lane>
                <action id="Action_C">
                  <realisation xsi:type="ProcedureCall"
                  procedure="Task_C"></realisation>
                </action>
              </lane>
            </realisation>
          </parallel>
        </flow>
      </realisation>
    </procedure>
  </procedures>
</otx>
```

Chapter 3

State of Research

Process management is the art of monitoring the way activities are performed to reach business goals. Moreover, process management is not about an individual activity, rather it is about the interweaving of events, activities and decisions. Process models are the formal description of a process. The instance of a process model is one execution of the model. In general it is possible to execute a process model in more than one way, e. g. using different branches in the process model. For instance, we consider a process model describing that the task A is executed first followed by either B or C. The process model allows for two different executions either A and then B, or A and then C. During the execution different events are thrown for tasks or decisions. We call the set of possible sequences of events for a process model P its behavior $\mathcal{L}_P$.

3.1 Declarative/Imperative Process Models

There exist two major paradigms to specify a process model exists: the imperative and the declarative ones. Imperative Process Modeling means to explicitly define the possible executions, i. e., define the control flow of the process. Declarative process modeling means defining the forbidden behavior, i. e., constrains to the control flow. The possible executions are given implicitly, i. e., every execution that is not forbidden is allowed.

In general the declarative process models are more flexible than the imperative ones, i. e., allowing a larger set of possible executions. That can also cause unforeseen and unwanted process behavior in declarative models, rendering the model less controllable. [Fah+09b] argues that the understandability depends on the nature of the processes. Sequential processes are easier to understand in an imperative manner and processes with circumstantial information are easier to understand in a declarative manner. An empirical evaluation [Pic+12] resulted in a better understanding for imperative languages. [Fah+10] proposes propositions about the maintainability of imperative and declarative languages.

Table 3.1: The Differences Between the Declarative and Imperative Process Modeling

Imperative Process model	Declarative Process Model
› control flows given explicit	› control flow given implicit
› in general easier to control	› in general more flexible
› easier to understand	› difficult to understand
› major tool support exist	› prototype implementation
› suitable for structured processes, e. g., production	› suitable for unstructured processes, e. g., health-care

In summary sequential changes is easier in imperative language and change in the circumstantial information are easier in declarative languages.

The imperative paradigm is widespreadly used in practice and therefore major tool support exists. The declarative paradigm recently became a focus in research but until today major tool support is missing, especially for process models in the industrial settings. The imperative paradigm is best for structured processes following a strict plan with few alternations, e. g., in the industrial production domain. The declarative paradigm is best for unstructured process models where deviations and exceptions appear frequently, e. g., in the health care context [MGP15]. Table 3.1 summarizes the differences between the declarative and imperative paradigm.

Declare is a declarative process language and a associated framework. Declare tools have been referred to as DECLARE and the language as ConDec [PSA07] or in early stage as DecSerFlow for Web Services [AP06]. Dynamic Condition Response Graphs (DCR Graphs) [HMS12] allow a nested sub-graph for better understandability of declarative models. [Gia+15] extends the imperative process notation BPMN to support declarative modeling.

Example 3.1.1 . Figure 3.1 shows the same process in two different notations, the imperative BPMN notation and the declarative Declare notation. The model describes a process model that first execute once the task A followed by the execution of B and C exactly once in arbitrary order. The behavior of the Process P is $\mathcal{L}_P = \{ \langle A, B, C \rangle , \langle A, C, B \rangle \}$

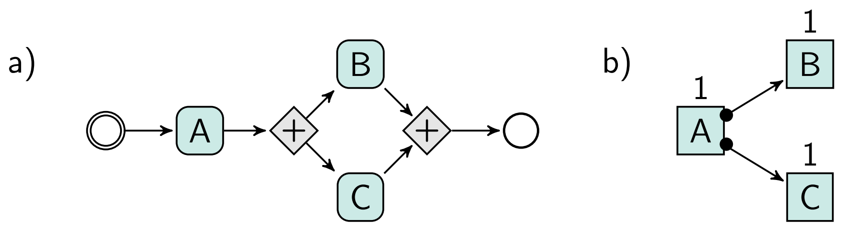


Figure 3.1: (a) A Process Model in Imperative BPMN Notation (b) The Same Process Model in the Declarative Declare Notation

It is possible to combine the two paradigms, e. g., using hybrid models with parts of the processes described imperatively and others declaratively [SSO01; Aal+09]. For our scenario regarding commissioning of vehicles predictability of the execution is more important than flexibility. To this end, we use imperative process models for commissioning process models. Additionally, no major framework exists to execute declarative processes for the commissioning. The declarative process models are related to our three major contributions and models are in fact properties that a process should fulfill. Our three major contributions of this thesis can then be formulated as the interaction of imperative and declarative process models.

› Specification: How to design a declarative process model D?

› Verification: Given an imperative process model P and a (incomplete) declarative process model D, is $\mathcal{L}_P$ a subset of $\mathcal{L}_D$?

› Process Generation: Given a declarative process model D, find a good imperative process model P, with $\mathcal{L}_P$ being a subset of $\mathcal{L}_D$.

3.1.1 Imperative Process Notations

In general, a graphical notation describes the process model. Over the last decades a plethora of different process model notations arise for different purposes and domains. For this thesis we want to categorize the process notations using two axes, Applied/Scientific Notations and Graph-based/Block-based Notations.

Applied notations are developed to support the domain experts like the business process designers to model their processes. The notations are characterized

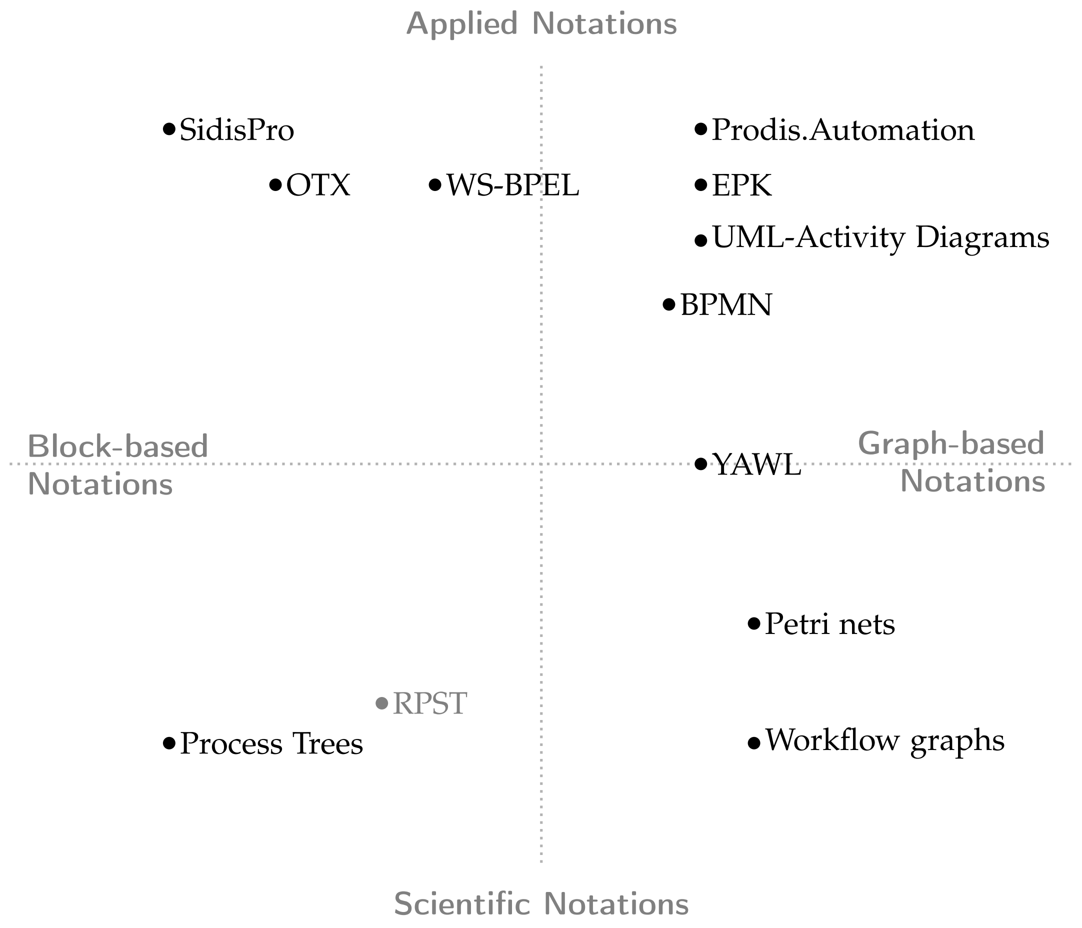


Figure 3.2: Classification of Process Notations to Applied/Scientific Notations and Graph-based/Block-based Notations

by several specific functions and a high usability. Applied notations often lack formal semantics, i. e., how to produce an instance $i \in \mathcal{L}_p$. Often only a vague textual description is given for contradicting interpretations. On the other side there exists academic notations [LVD09]. For the academic notations a formal semantic is given and, thus an unique interpretation. The academic notation originates from academic work about the expressive power or respectively verification. The formal semantic allows advanced verification and analysis techniques. But the academic notation often lacks domain specific constructs and tool support. A common approach to analyze applied notations is to transform them on an academic language language, and subsequent use the analysis technique of the academic language [LVD09]. The transformation itself is an interpretation of the applied notation and often ambiguous. YAWL [AH05] can not be classified definitely into one of two categories. YAWL has an unambiguously defined execution semantic and allows complex analysis techniques. At the same time it was developed to be used as a practical application.[Min11, S.39] shows that YAWL is despite of these properties *still* is not in widespread use. [FRM13] indicates that usability reasons regarding the graphical presentation may be the cause.

Graph-based notations describe the structure of the process model as directed graph. The different kinds of nodes model different behavior. Often graph-based notations distinguish between nodes that represents activities or events and nodes for the control structure, e. g., gateways. Block-based notations describe the structure of the process model as nested hierarchy of fragments, i. e., as a tree. The leaf nodes are often the activities or events and the inner nodes models the control structure. With repetition both block-based and graph-based notations are Turing complete, similar to Goto-Computability/While-Computability. Without the repetition of nodes it is possible to model a process in a graph-based notation that can not be represented as a block-based notation. This higher expressive power is not inherent an advancement. The modeling of a process that can not be represented in a block-based notation has shown to be hard to understand [MRC07] and often leads to errors in the process model [MNA07]. To this end, guidelines of modeling forbids the use of these constructs, cf. guideline G4 in [MRA10]. Similar guidelines hold for the modeling at the AUDI AG. Some graph-based notations like BPMN and YAWL allows a part hierarchical modeling by the use of sub graphs. The block-based notation OTX allows exceptions of the block-based structures in specific situations, e. g., guided jumps out of a loop body, but are forbidden by intern rules. **W**eb **S**ervices **B**usiness **P**rocess **E**xecution **L**anguage (WS-BPEL) allows a graph-based modeling beside the standard block-based notation. [Kop+09]. **R**efined **P**rocess **S**tructure **T**ree) (RPST) [VVK09] is not an actual process notation. RPST is a decomposition of a graph-based notation into a block-based hierarchy. The structure is similar to the block-based notations. We process the RPST similar to a block-based notation, thus are listed here.

3.1.2 State Space Explosion for Concurrency

Concurrent execution leads to an explosion of the behavior space. This means that the size of the behavior space $|\mathcal{L}_P|$ scales exponentially with the size of the process model $|P|$ in the case of concurrency.

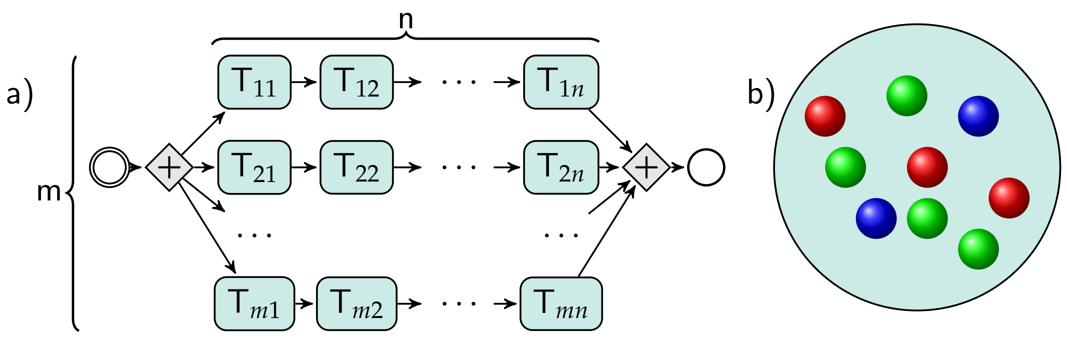


Figure 3.3: (a) Parallel Split with m lanes and n tasks each (b) Urn Model

Corollary 1. The size of a behavior space $|\mathcal{L}_P|$ for a Process P that consists of one parallel split with m-parallel lanes with $k_1, \dots, k_m$ tasks can be calculated with the multinomial coefficient, i. e:

$$|\mathcal{L}_P| = \binom{s}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!}$$

with $s = \sum_i k_i$.

Corollary 1 can be mapped on a problem of the combinatorics. Consider an urn with s different balls in it. Each ball has a color depending on the lane of the task. So for $k_1 = 3$, $k_2 = 2$, $k_3 = 4$ the urn would contain 3 red balls, 2 blue balls and 4 green balls, see Figure 3.3(b). Balls are drawn from the urn without placing a ball back to urn. If a color of a lane is drawn the next available task of the lane is executed. It can be easily seen that the number of possible draws correspond to the number of paths or $|\mathcal{L}_P|$.

Example 3.1.2. Consider a process with three lanes and $k_1 = 3$, $k_2 = 2$, $k_3 = 4$ each. This leads to $s = \sum_i k_i = 9$ and thus the following behavior space $|\mathcal{L}_P|$:

$$|\mathcal{L}_P| = \binom{9}{3,\ 2,\ 4} = \frac{9!}{3!\ 2!\ 4!} = \frac{362880}{288} = 1260$$

If $k_1 = k_2 = \dots = k_m = n$ the formula is simplified to $|\mathcal{L}_P| = \frac{(n\,m)!}{n!^m}$. Figure 3.4 shows $|\mathcal{L}_P|$ for 2 lanes, 5 lanes and 10 lanes (m) with 1 to 10 tasks ($n \in [1, 10]$).

Remark the logarithmic scale for $|\mathcal{L}_P|$. Commissioning processes contain up to $n = 14$ with over 20 tasks each. The number of atoms in the universe is around $10^{82} - 10^{89}$. The explicit construction of $|\mathcal{L}_P|$ thus is not possible for realistic size commissioning processes.

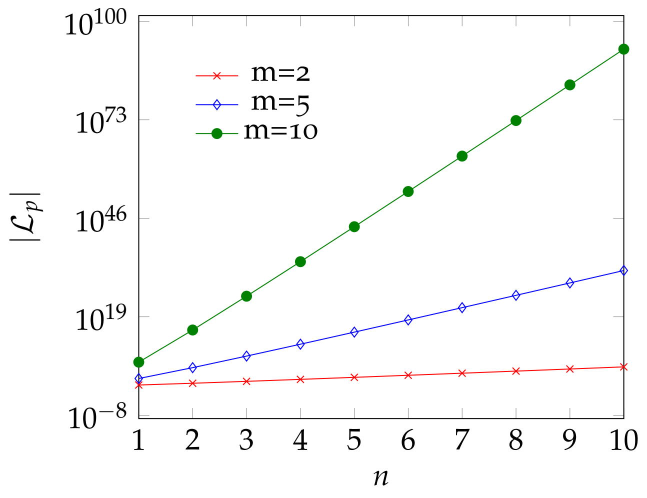


Figure 3.4: The Size of the Behavior Space $|\mathcal{L}_P|$ for $m = 2, 5, 10$ and $x \in [1, 10]$

3.2 Specification of Declarative Properties

In general specification is the act to formalize *what* a system should accomplish. In the course of this thesis we use the word specification for the formal description of the declarative properties a process model should fulfill. In contrast to the model of the process that specifies *how* to accomplish the goal. For instance, a property of a process is that each open connection should be closed at the end of each execution. The model of the process is the set of actual steps and their sequence. In contrast to a declarative model the specification is in general incomplete. In general, it is not possible to directly execute a property specification. First we will describe the formal notations to describe the allowed behavior. Formal notation can be automatically processed, e. g. verified, but their specification is error-prone and requires experience in mathematically modeling and abstract thinking. To this end, authors propose list of *property patterns* or property classes with semantic meaning, e. g., an event is the precondition of another event. The semantic information renders them easier to handle for domain experts. The patterns can be mapped on a formal notation to allow automatic processing. To further simplify the specification graphical notation are proposed. It is possible to map each graphical constructs in these notations maps to a property pattern, and thus to a construct in a formal notation.

3.2.1 Formal Notations

First order logic is an extension of the propositional logic with predicates and quantifier, e. g., $\forall, \exists$. First order logic is complete [Göd29] but in general not decidable. [LRD10] uses first order logic to express the compliance rule graphs. The logic formulas are evaluated over the set of all possible executions of a process model P, i. e., $\mathcal{L}_P$. For instance the $\mathsf{Response}(A, B)$-pattern are specified as:

$$\forall t(\mathit{ActivityType}\,(t, A) \rightarrow \exists d : (\mathit{ActivityType}\,(d, B) \wedge \mathit{Pred}\,(t, d))$$

Due to the fact that [LRD10] argues about the traces of a process model it is not possible to formulate constraints that require multiple futures, e. g., the weak anti-pattern of [TAS09].

Modal logics are another extension of the propositional logic with concept of necessarily $\Box$ and possibly $\Diamond$. Intuitively the $\Box$ operator denote that something is *true* in all possible worlds, the $\Diamond$ operator denotes that something is *true* in one possible world. The kripke-structure is a graph-based formalization of the world concepts. Node represents the possible worlds and edges the reachability of worlds. Temporal logics are an application of the modal logic. The worlds in the temporal logic refers to points in time, the $\Box$ operator is interpreted as *»something is true in all possible futures«* the $\Diamond$ operator as *»something is true in at one time in the future«*. It is possible to generate kripke structures for process models by generating their state space. The most famous temporal logics are Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and CTL*. CTL* is a superset of both LTL and CTL. LTL uses a linear concept of time, i. e., only one possible future exists while CTL and CTL* use a branching time concept. Past LTL (PLTL) is the extension of LTL with additional operators over the past. The expression power of PLTL and LTL is equal but some properties can be stated more succinct in PLTL. [DAC98] gives a mapping of their property pattern onto both LTL and CTL. [TAS09] formulates the anti-pattern as CTL*-formulas. [ADW08] maps BPMN-Q to LTL formulas for the purpose of verification. [Mon+10] maps Declare constructs on LTL formulas for the enactment. For instance the $\mathsf{Precedence}(A, B)$, $\mathsf{Response}(A, B)$ and $\mathsf{Existence}(A)$ pattern in LTL are respectively:

$\neg B \ \mathsf{U}\ (A \vee \neg\Diamond B)$ *(Precedence)* $\quad \Box(A \rightarrow \Diamond B)$ *(Response)* $\quad \Diamond A$ *(Existence)*

[Yu+06] specify their properties with finite state automatons (FSA). To this end, they give a mapping of the extended property pattern PROPOLS to finite state products. Figure 3.5 shows the FSA for the Precedence(A, B), Response(A, B) and Existence(A) pattern. Ω means any tasks not A or B. It is possible to express the process model M as finite state automatons. The verification can be solved by testing the intersection of the FSA of M with the complement of the property FSA [Yu+06]. This approach has scalability problems because the FSA of a process model scales exponential with the size of the model.

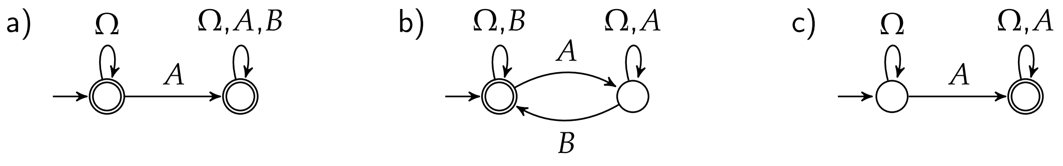


Figure 3.5: The FSA for the Precedence(A, B) (a), Response(A, B) (b) and Existence(A) (c) pattern

[RFA12] uses Petri nets to specify compliance properties. They verify if a given trace complies by testing if the Petri net can replay the path. Due to the trace based notation it is only possible to verify properties that can be expressed in LTL. The approach of [RFA12] is used to check the traces in a log in a retrospective matter. In general, it is not possible to generate all traces of a process model $\mathcal{L}_P$, i. e., the approach cannot guarantee the compliance of unobserved behavior.

3.2.2 Pattern Specification

[DAC98] describes a set of property patterns for the specification of systems. [DAC99] empirically evaluates if these patterns are sufficient to express most of the real world specification needs. The patterns are either of the type ordering or occurrence and are valid within a scope. Several mappings of the patterns to formal languages are given by the authors or others, e. g., CTL, LTL, or quantified regular expressions. The patterns are domain independent and on a rather high abstraction level. This high abstraction level makes them universally applicable but hinders their usefulness to more domain specific patterns. [CAC06] shows an approach to instantiate the property pattern with a question tree. [Yu+06] extend the work of [DAC98] with combined patterns to the specification language PROPOLS.

[TAS09] proposes a set of anti-patterns for data-flow errors. An anti-pattern A is a behavior that should not occur in a process model P. The problem is symmetric to the property verification and can be easily restated to verify whether $P \models \neg A$. [TAS09] proposes a mapping of the data-flow anti-pattern to CTL*-formulas.

3.2.3 Visual Specification

BPMN-Q [ADW08] as visual specification language is able to express two patterns: Response and Precedence. The graphical patterns are mapped to LTL formulas for the verification. The graphical notation resembles BPMN with additional edges. [För+07] proposes a visual language based on the UML activities supporting several variations of the response, precedence and successive patterns. [Bra+05] shows a visual representation of LTL-formulas in a language similar to BPMN. In contrast to the other work the visual representation directly maps to LTL and not to specification patterns. This results in high expression power for the cost of less abstraction and complicated specification. Compliance rule graphs (CRG), see [LRD10], are a visual specification language for compliance properties of process models. Each node in a CRG refers to either the occurrence or absence of an event, that either triggers the rule (*antecedent*) or is the respective *consequence*. Edges denote the necessary sequential ordering of the events. For instance the $\mathsf{Response}(A, B)$-pattern would consist of an *antecedent occurrence* node A connected to a *consequence occurrence* node B. For the verification the CRGs are mapped to first order logic formulas. Most of the declarative process modeling languages support a mapping to LTL [PSA07; AP06; HMS12; Gia+15]. Thus it is possible to use a graphical declarative process language for the specification of properties.

3.3 Process Verification

IEEE defines verification as: »*The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation.*« [IEE11, p.452]. In the case of BPM the system for verifying is the process model. We can differentiate between generic requirements that should be *true* for all process models independently of the domain, e. g., a process should always be able to reach a proper completion, and the domain-specific properties, e. g., in the commissioning each connection should be closed at the completion time. Section 3.3.1 discusses approaches for the verification of generic properties, and Section 3.3.2 approaches for the domain-specific verification. Another way to classify the verification approaches is the time in which the verification takes place. Most of the approaches verify a given process model at the design time, cf. Subsection 3.3.1 and 3.3.2. Monitoring means to verify if a process instance complies with the regulation during the execution, see Subsection 3.3.3. Conformance checking verifies if given previous executions were comply, see Subsection 3.3.4.

3.3.1 Verification of Generic Properties

Generic properties are properties that every process model should fulfill independently of the domain. The majority of research in generic property concentrates on the *soundness* of a process model.

Definition 1 (Soundness) .

A process model P is *sound* exactly if

› P does not contain a *local deadlock*, i. e., a state s is reachable with a token in the incoming edge e to an AND-Join, and each reachable state s' from s also contains a token in e.

› P does not contain a *lack of synchronization*, i. e., a state s is reachable with an edge containing more than one token.

Many different definitions of *soundness* exist. See [Fah+09a] for an overview. [Aal+10] analyzes the decidability of several soundness properties. A soundness property can be expressed as two temporal logic formulas, e. g., in CTL:

› Absence of local deadlock: $\mathsf{AG}\ \mathsf{EF}\,(\mathit{termination} > 0)$

› Absence of lack of synchronization: $\mathsf{AG}\,(\,\forall_{p \in P}\ m(p) \leq 1\,)$

The state *termination* is the last state of the process model. P refers to places in the Petri net representation.

Example 3.3.1 . The process model of Figure 3.6 contains both a *deadlock* and a *lack of synchronization*. If at the first XOR-Split Gateway B the above path is taken than a *deadlock* will occur at the AND-join *F*. If the below path of *B* and *E* is taken, the below path and in the next iteration at *B* the above path, than a *lack of synchronization* will occur in the edge of *D* to *F*.

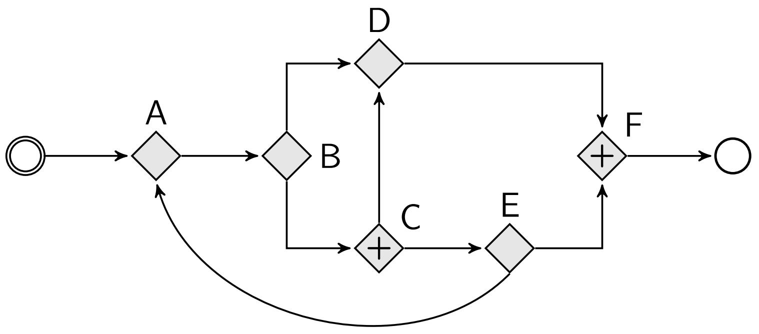


Figure 3.6: A Process Model in the BPMN Notation Containing a Deadlock and a Lack of Synchronization

Soundness was initially defined for workflow nets, i.e., a special Petri net, see [Aal98]. Woflan is a tool for soundness verification of a workflow net. It uses a complex algorithm that consists of several steps. First, [VBA01] use soundness preserving reduction rules on the net. Next, if the net is not trivial after the reduction, Woflan tests if the net is S-coverable [VBA01], utilizing the property that a not S-coverable net implies an unsound net. If the net is S-coverable Woflan constructs the state space of the net to analyze *soundness*. It is possible to restate the *soundness* as a model checking problem and use a model checking approach to analyze *soundness*, e. g., the LoLA tool kit [Scho0a]. SESE approaches try to solve the soundness by decomposition of the process model into smaller fragments. The complete process is sound if each of its fragments is sound. One of such decomposition is the refined process structure tree (RPST) [VVK09]. [Fah+09a] shows a study of three different types of soundness checkers (Woflan, LoLA, RPST) on a repository of industrial business processes.

In contrast to the verification of domain-specific properties the generic property is known beforehand. Therefore it is possible to optimize the approaches to verify the specific property. The optimization changes the result of the general domain-specific properties, e. g., reduction rules of Woflan [VBA01]. For a process model as a process tree, e. g., an OTX process model, it is not possible to model an unsound process.

3.3.2 Verification of Domain-specific Properties

In contrast to the generic properties like soundness, that every process model should fulfill, stand the domain-specific properties that hold for a specific domain. Due to the domain specific nature the concrete properties are unknown at the design time of the verification approach. Therefore, it is not possible to optimize the approach to a concrete property. The domain-specific approaches need to function for a variety of different properties with few if any assumption about the properties. The domain-specific properties given raise to related problems on how to specify the properties, see Section 3.2. Compliance checking is an application of domain-specific property verification. The properties in the compliance case are the regulations imposed by a third party. For instance the Sarbanes-Oxley Act (SOX) and Basel III for banking processes, or the guidelines of the *Verband der Automobilindustrie* (VDA, eng: *German Association of the Automotive Industry*) for the automobile industry.

[Yu+06] transforms the PROPOLS constraints and the WS-BPEL process model into a state machine. Next, their algorithm computes the intersection of the state machines and test if the resulting state machine is empty. [ADW08] verifies BPMN-Q graphs on process models. To this end, [ADW08] reduces the process model and subsequently verifies the model with an LTL model checker. [LMX07] verifies BPEL process models with properties, specified in BPSL, a XML-based specification language. The verification transforms the BPEL process models into the formal π-calculus and the BPSL into LTL formulas in order to allow a model checking verification. [Ly+11a] verifies the compliance of a process model to compliance rule graphs. For control-flow properties [Ly+11a] propose an abstraction with node relation to check the properties. For data-flow properties [Ly+11a] propose behavior verification.

3.3.3 Monitoring of Process Instances

Verification of a process model guarantees that all instances of the model comply with the properties. Another approach to guarantee the compliance is to monitor the instances either during the runtime or to test them after completion.

[Ly+11b] monitors process instances if they fulfill a set of compliance rule graphs (CRGS). [Mag+11] monitors Declare constraints by transforming their underlying LTL formulas into state machines. [RFA12] checks if a set of previous executions, i. e., a log, complies with the specification. For some contexts only an incomplete process model exists and not all steps are automated. In that case the process model verification is not possible and a monitoring approach is the only solution to check the compliance. In general, not all aspects that influence the execution are contained in the process model and often a further abstraction is done for the verification, e. g., ignoring the constraints on the branches. Those abstractions can lead to false positives, see Example 3.3.2. The monitoring approaches only focus on the actual executions and thus take all aspects into account that influence the execution.

Example 3.3.2 . Figure 3.7 shows a process model with the conditions of XOR-branches on the edges. The Property »The tasks A excludes the tasks D« is fulfilled for the process model. If we abstract from the condition the property is evaluated to be violated and thus a *false positive*.

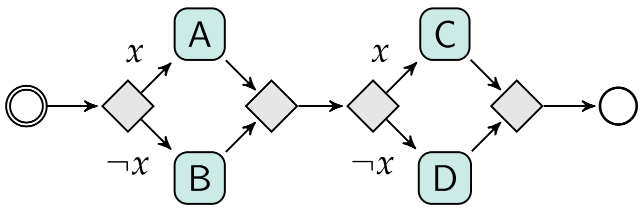


Figure 3.7: A Process Model with the Constraints of the Branches on the Edges

The use of monitoring is not possible in our use case. Commissioning processes only have a limited time frame for execution. If a property violation is detected as late as during execution time the countermeasures would cause a stop to the assembly lines. Some authors call the monitoring *after-the-fact approaches*, in contrast to the *compliance-by-design approaches*, i. e., the semantic verification [SGN07].

3.3.4 Conformance Checking

Conformance checking is the problem if a given process model P can replay the traces t in a log L_P [Aal11]. In general conformance checking evaluates if the model of a process discovery algorithm is correct, or tries to connect an event log and process model. If the process model is declarative conformance checking can be used to verify if the process that produces the log is correct. [ABD05] checks if a given log L_P complies with regulations given as LTL formulas. [AAD12] replays previous process executions, i. e., the log L_P, to detect performance issues.

3.4 Automatic Generation

In this section we will discuss approaches to generate process models from a declarative specification. First we will discuss approaches using scheduling algorithms and their application to BPM, next approaches for the synthesis of process models.

3.4.1 Scheduling

A schedule is the plan how to execute a set of Tasks $\mathcal{T}$. The scheduling problem is given by a set of Tasks $\mathcal{T}$, their processing time $p : \mathcal{T} \mapsto \mathbb{R}$ in order to find the optimal starting time of the tasks $s : \mathcal{T} \mapsto \mathbb{R}$ with respect to a precedence relation $\prec$ between the tasks. Extensions of the scheduling problem include additional constraint like resources, then called the Resource Constraint Project Scheduling Problem RCPSP, or assigning workers to the tasks. The scheduling problem is a generalization of the static job shop problem and thus NP-hard [BLK83]. In order to solve the scheduling problem optimal approaches, heuristics and genetic approaches [MMS02] have been proposed, see [Kol96] for an overview.

[Sen+15] detects bottlenecks in clinical processes by comparing a schedule to an actual process execution L_P. To this end, [Sen+15] generates from a schedule a process model P' and checks the conformance of P' to a log of executions $\mathcal{L}_P$.

The schedule describes one optimal execution of the tasks. A process model in turn describes several executions that are allowed. It is possible to generate a process model from a schedule that can replay the schedule. But this process model would be very constraint allowing only this one schedule. In general process model generated from a schedule are not block-structured.

3.4.2 Synthesis of Process Models

[Yu+08] synthesizes a process model directly from its specification. The specifications are in PROPOLS [Yu+08], a temporal constraint specification language. The specifications are transformed into finite state machines and then integrated into one machine. Next, each accepting path is generated from the state machine. An algorithm similar to the α-algorithm [AWM04] is applied to synthesize a process model from its set of paths. The approach of Yu et al [Yu+08] can only be applied if the specification, i. e., the number of state machines, is small ($\approx$

6). To this end, [Yu+08] divides the specification into small groups, synthesizes a process fragment for each group and manually combines the fragments. For our use case, this approach would require over a hundred state machines for each commissioning process model, and the manual combination would not be feasible. [Awa+11] has specifications with LTL as starting point. It generates a pseudo model from the specification. This model lists all paths that fulfill the LTL formula. [Awa+11] generates an ordering relation graph from the set of paths and uses it to synthesize a process tree. For our use case the generation of all paths would not be feasible. This is because the number of paths grows exponentially with the size of the specification. Even for the smallest process model we have evaluated calculating all paths has not been possible.

An approach different from generating the process model from scratch is to extract information from process models already specified and to create a similar process. [Chi+09] uses a CBR-based method to this end. The search is based on keywords that are annotations of the workflows. [Koo+08] guides the process designer with suggestions on how to complete data-oriented visualization models. The suggestions are generated from paths of existing visualization process models stored in a repository. [Koo+08] does not allow building a process model with an AND-Split and therefore is not sufficient in our case. [Lau+09] predicts which activity pattern, i. e., generic process fragment, will follow the partly modeled process. The paths of existing process models are extracted and analyzed with association rule mining. [Koo+08; Lau+09] extend an existing process model, while our approach generates one from a declarative specification. The approach of Chinthaka et al [Chi+09] requires annotations of the existing process models. None of the approaches mentioned optimize the runtime or consider constraints.

AI planning is the task of defining a set of actions that achieve a specified aim [HTD90]. In a nutshell, it is the retrieval of an applicable plan in the solution space. [VM91] uses a genetic algorithm in order to find a manufacturing plan. Some approaches that synthesize business processes are discussed next: [MR02] uses an AI planning approach to synthesize service compositions. Without calling it AI planning, [AHK05] uses a similar approach for configuration-based workflow compositions. [AK07] introduces a planning algorithm to compose data workflows. None of these studies focus on optimizing the runtime of the process or considers requirements similar to ours. These approaches are not applicable to our problem statement.

3.5 Process Discovery

Process Discovery is one of the major problem statements related to process mining. The problem statement is given to a set of traces L (called log) in order to find a process model P that is representative to L. Remark that representative does not mean that $L = \mathcal{L}_P$. In general the log L is *incomplete* and does not contain all possible behavior $\mathcal{L}_P \setminus L \neq \emptyset$ and L can contain infrequent and not representative behavior called *noise*, i. e., $L \setminus \mathcal{L}_P \neq \emptyset$. Four major competing quality criteria exist for the resulting process model: *fitness*, *simplicity*, *generalization* and *precision* [Aal11]. *Fitness* is the ability of the process model to replay the log L, *simplicity* is the size and complexity of the process model. Those two criteria on their own are not sufficient. Consider the BPMN model in Figure 3.8, called flower model [Aal11]. The process model is relative simple and can replay any log L with the tasks $\{A, B, \dots, Z\}$. This model is not helpful because it contains no information except the list of tasks. The *precision* quality criteria ensures that the model is not under-fitting the log thus preventing flower models. At the same time it is not helpful to overfit the log, i. e., the process model should reasonable abstract from the log, called *Generalization*. To illustrate consider a parallel split with 3 lanes, with 3 tasks each. 1680 possible ordering of tasks are possible for this split, see Lemma 1. It is not likely that the log L contains every possible arrangement.

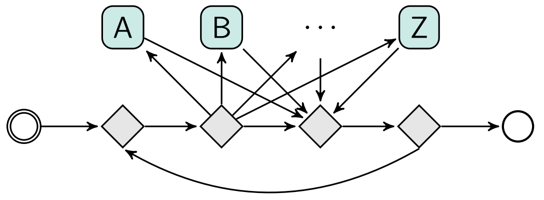


Figure 3.8: A Process Model in BPMN Notation that can Replay any Log with the Taks $\{A, B, \dots, Z\}$

3.5.1 Imperative Process Discovery

In this subsection we want to present two approaches for mining an imperative process model – the α-algorithm and the inductive miner. A plethora of approaches have been proposed to solve the process discovery of imperative process models since the mid-90s. [Aal+03a] gives an overview over the classical approaches up to 2003. Most of the earlier approaches cannot handle noise and incomplete logs. Recent approaches try to handle noise by heuristics [WA03], genetic algorithm [MWA07; AMW05; BDA12] or integer linear programming [Wer+08]. Decomposition based methods like [VA14; LFA13a] improve the scalability of process discovery. We focus on α-algorithm and the inductive miner because they are most relevant to our work and give an overview over the classical methods, i. e, α-algorithm, and the more recent approaches, i. e., inductive miner.

The α-algorithm is one of the *classical* process discovery algorithms [AWM04]. The algorithm is able to rediscover a sound Workflow net, i. e., a special Petri net, from a Log of task events. The algorithm consists of two phases. First we define several ordering relations over the log L. Second, we use those relations to define the places, transitions and edges in the Petri net.

Definition 2 (Log-based ordering relations, simplified after [AWM04]) .
Let L be a Log, and a, b tasks defined in the log L.

› $a > b$ iff there is a trace $\sigma = t_1 t_2 t_3 \dots t_n$ such that $\sigma \in W$ and $\exists i \in \{1, \dots, n-1\}$ with $t_i = a$ and $t_{i+1} = b$,

› $a \rightarrow b$ iff $a > b$ and $b \not> a$

› $a \# b$ iff $a \not> b$ and $b \not> a$

› $a \parallel b$ iff $a > b$ and $b > a$

Using those ordering relations the algorithm is straightforward by defining the elements of a Petri net according to the relations. See Algorithm 1, simplified after [AWM04].

Algorithm 1 AlphaAlgorithm (Log L) : Petri net PN

```
1: $T = \{t \in T \mid \exists_{\sigma \in L} t \in \sigma\}$
2: $T_I = \{t \in T \mid \exists_{\sigma \in L} t = \mathit{first}(\sigma)\}$
3: $T_O = \{t \in T \mid \exists_{\sigma \in L} t = \mathit{last}(\sigma)\}$
4: $X = \{(A, B) | A \subseteq T \wedge B \subseteq T \wedge \forall_{a \in A} \forall_{b \in B}\, a \rightarrow b$
       $\wedge \forall_{a_1, a_2 \in A}\, a_1 \# a_2 \wedge \forall_{b_1, b_2 \in B}\, b_1 \# b_2\}$,
5: $Y = \{(A, B) \in X \mid \forall_{(A', B') \in X}\, A \subseteq A' \wedge B \subseteq B' \Rightarrow (A, B) = (A', B')\}$,
6: $P = \{p(A, B) \mid (A, B) \in Y\} \cup \{i, o\}$,
7: $F = \{(a, p(A, B)) \mid (A, B) \in Y \wedge a \in A\} \cup \{(p(A, B), b) \mid (A, B) \in Y \wedge b \in B\}$
       $\cup \{(i, t) \mid t \in T_I\} \cup \{(t, o) \mid t \in T_O\}$,
8: **return** $PN(T, P, F)$
```

The α-algorithm can discovery a sound process model efficiently. The algorithm has several limitations. The algorithm requires the precondition that the log is complete, i. e., contains all traces. The resulting process models of the α-algorithm lack structure and are hard to understand, i. e., results in *spaghetti models*.

The inductive miner [LFA13a] is a process discovery algorithm to detect efficiently a fitting block-structured process model from a log L. The block-structured process model is inherent sound and death lock free and shown to be easier to understand than arbitrary structured process models [MRA10]. The algorithm first generates a graph called the direct-follows graph from the Log L. In the graph each node corresponds to a task. If task B directly follows A in at least one trace the graph contains an edge A to B. Next cuts are searched in the direct-follows graph. Each type of cut corresponds to a control element in a process tree, see Figure 3.9. The algorithm has been extended to deal with noise [LFA13b] and incomplete logs [LFA14].

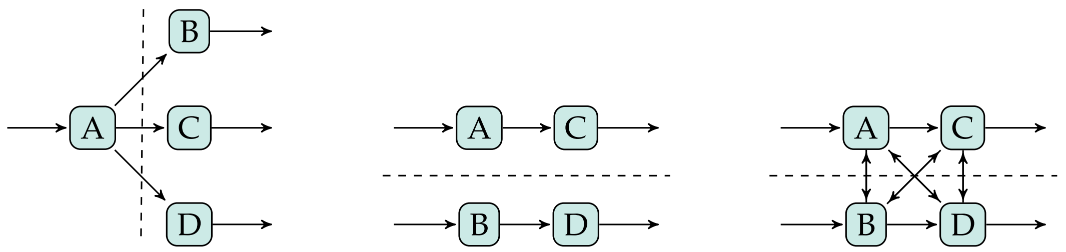


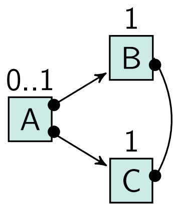
Figure 3.9: Cuts of the Directly-follows Graph for Operators $\rightarrow$, $\times$, and $\wedge$

3.5.2 Declarative Process Discovery

The declarative process discovery uses a log of previous executions of a model to (re)discover the declarative process model. In basic the approaches try to find patterns in the set of paths that relate to, e. g., Declare constructs.

Example 3.5.1. Consider the Log $L = \{ \langle a,b,c\rangle, \langle a,c,b\rangle, \langle c,b\rangle, \langle b,c\rangle \}$. Figure 3.10 shows a Declare model that can produce the log we want to rediscover.

Several approaches have been proposed for the declarative process discovery. [CM13; CM12; MBA12; MMA11; Mag+13; CMM14] mine process models in the Declare notation. They differ in the amount of declarative constructs they can



$\mathcal{L} = \{ \langle a,b,c \rangle, \langle a,c,b \rangle, \langle c,b \rangle, \langle b,c \rangle \}$

Figure 3.10: A Declarative Process Model with its Behavior Space $\mathcal{L}$

discover [Mag+13; CMM14] and in their scalability [MBA12]. [Lam+08] and [Che+09] use inductive logic programming for the declarative process discovery. [MSR14] describes an approach to rediscover hybrid models, i. e., process models partly defined imperative and partly declarative. Most of the approaches use a variation of the apriori algorithm for the detection of association rules. The declarative process discovery is related to our detection of behavior properties in a process repository. See Subsection 4.3. The difference is similar to the one of process synthesis and process discovery. For the detection of behavior pattern we want to discover frequent behavior in a collection, i. e., find a frequent subset of behavior. Declarative process discovery abstracts from the observed behavior.

Chapter 4

Property Specification

Specification is the formal description of the properties a process model should fulfill. Classical process modeling notations like BPMN, Petri nets, WS-BPEL, YAWL, or OTX specify imperatively the possible execution sequences of a process model. Imperative modeling is stating *how* to execute a process. The *goal* of the process is given implicitly. For instance, the imperative process model for a commissioning process is:

1. *Start the engine* (S).
2. *Test the hand-glove light* (H) *and the radio* (R) *in arbitrary order*.

The imperative specification in general omits some execution alternatives, i. e., the process model is over-specified and lacks flexibility.

In contrast to the imperative specification there exists declarative specification [PA06]. The declarative specification consists of the process model *goals* and the required properties for the execution. *How* to reach the goal is not specified and must be decided in runtime, i. e., the control flow is not specified. For instance, the declarative process model of a commissioning process is:

› *Goal*: Test the hand-glove light (H) and the radio (R).

› *Properties*: Testing of the hand-glove light requires a running engine ($\mathsf{S} \rightarrow \mathsf{H}$).

In this chapter, we focus on the declarative specification of process models, i. e., the properties of a process model. The declarative specification is required both for the verification of existing process models and for the automatic synthesis. A common definition of correctness of an imperative process model, e. g., a Petri net, is that it must fulfill properties, e. g., in **C**omputation **T**ree **L**ogic (CTL). Verification can be stated as testing if the imperative specification is correct, see the Venn diagram in Figure 4(a). The diagram shows the behavior $\mathcal{L}$, i. e., the possible execution traces of an imperatively specified process model (blue circle), and the allowed behavior $\mathcal{P}$ specified by a declarative specification of the properties (green circle). The verification tests if the process model

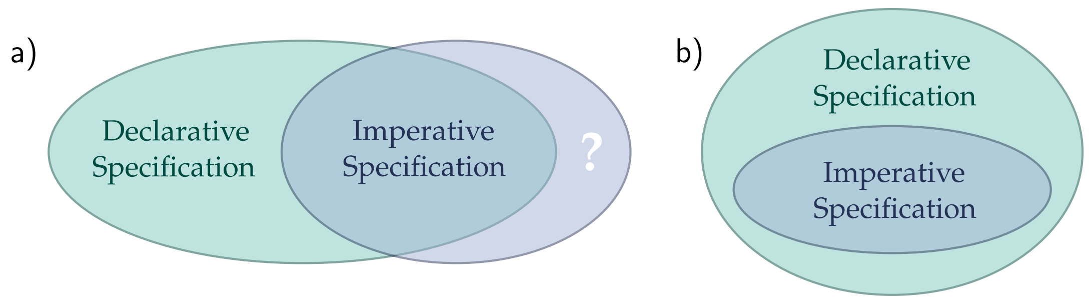


Figure 4.1: Use-Case Verification (a) and Use-Case Synthesis (b)

allows a behavior not in the declarative specification, i. e., if $\mathcal{P} \setminus \mathcal{L} = \emptyset$. The automatic synthesis generates an imperative specification from given declarative properties. The imperative specification has to comply with the properties, see the venn diagram in Figure 4(b), in other words finding a process model with a behavior $\mathcal{L}$ with $\mathcal{L} \subset \mathcal{P}$ for a property set $\mathcal{P}$. The declarative specification of the properties is thus a necessary step for the verification and the synthesis.

In this chapter we propose two approaches for the declarative specification of properties: Section 4.1 introduces the notation of temporal logics. Section 4.2 shows a novel approach to generate automatically declarative properties for a commissioning process from a domain data base. Section 4.3 presents a scalable discovery of behavior patterns in a process model collection. In detail, in Section 4.2 we analyze which properties a commissioning process should comply and furthermore the related context informations. We observed that the properties can be described with a few property patterns. We give a formal specification of these patterns. The individual properties are automatically generated by an instantiation of these property patterns. For the instantiation we developed a model of the context knowledge. The context consists of, but is not limited to, the electronic control units, their relationships and the dependencies of the vehicle projects. To populate the database we use several sources, e. g., information about the control units from the production planning, existing and specified dependencies, and the information of the process developers. In Section 4.3 we show an algorithm to discover behavior patterns in the process model repository. Behavior patterns are relationships between tasks that occur frequently in the behavior space of the process models. They are candidates for unknown declarative properties to discover. The behavior space grows exponentially with the size of the process models and is much larger than the set of executions that actually have occurred. This makes to discover the patterns difficult. Further, behavior patterns are more abstract than structural ones, making their detection even harder. The core of our solution is to propose a concise summary of the behavior space, to significantly reduce its size while preserving the characteristics needed for the discovery of those patterns. We use temporal logic as the formal foundation of our specification.

4.1 Temporal Logics

Linear **T**emporal **L**ogic (LTL) was designed by Amir Pnueli for the verification of software systems [Pnu77]. In 1983 Clarke et al [CES83] propose the language CTL for model checking. [EH86] introduces CTL* as a superset of both CTL and LTL.

Definition 3 (Computation Tree Logic*) .

The language of CTL* is defined by the following grammatic:

$$\Phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathsf{A}\phi \mid \mathsf{E}\phi$$
$$\phi := \Phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid [\,\phi\,\mathsf{U}\,\phi\,]$$

In our domain, p is a state equation holding true over a marking M of a Petri net. $\{\mathsf{A}, \mathsf{E}\}$ are path operators and $\{\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}\}$ are temporal operators.

Path operators:

$\mathsf{A}\ \phi$	:	The property ϕ must hold in all subsequent paths
$\mathsf{E}\ \phi$	:	The property ϕ must hold in one subsequent path

Temporal operators:

$\mathsf{X}\ \phi$	:	The property ϕ must hold at the next state
$\mathsf{G}\ \phi$	:	ϕ must always hold in the subsequent path
$\mathsf{F}\ \phi$	:	ϕ has to hold at least once in the subsequent path
$\phi\ \mathsf{U}\ \psi$	:	ϕ has to hold at least until ψ happens in the future

Definition 4 (Linear Temporal Logic) .
The language of can LTL is defined by the following grammatic:

$$\Phi := \mathsf{A}\phi$$

$$\phi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid [\phi \mathsf{U} \phi]$$

I. e., LTL is the subset of CTL*, that starts with the path operator A and contains no additional path operator. Often the preceding A operator is omitted from the formulas and the temporal operators X, G, F are written as $\bigcirc$, $\Box$, $\Diamond$ respectively. Due to the missing path operators the calculation model of LTL is over a set of traces. Each event has exactly one possible future hence the name linear time.

CTL is another subset of CTL*. Model checking algorithms exist to efficiently verify CTL properties, cf. [CES86]. The CTL syntax is as follows:

Definition 5 (Computation Tree Logic) .
The language of CTL is defined by the following grammatic:

$$\begin{aligned}\phi := {} & \mathit{true} \mid \mathit{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{AX}\phi \mid \mathsf{AF}\phi \mid \mathsf{AG}\phi \mid \mathsf{A}[\phi \mathsf{U} \phi] \\ & \mid \mathsf{EX}\phi \mid \mathsf{EF}\phi \mid \mathsf{EG}\phi \mid \mathsf{E}[\phi \mathsf{U} \phi]\end{aligned}$$

I. e., in CTL the operators always occurs in pairs: one path operator followed by a temporal operator. It is not possible to write two temporal operators in sequence, e. g. the LTL/CTL* formula $(\mathsf{A\,F\,G}\ \phi)$. In contrast to LTL (Linear Temporal Logic) where the formula describes the property of an individual execution, a CTL formula describes the property of a computation tree, i. e., a set of executions. This allows to express properties that cannot be expressed in LTL. An example is that after an event x there always exists a path executing y, i. e., $\mathsf{AG}(x \to \mathsf{EF}(y))$. The operators always occur in pairs: a path operator (A or E) and a temporal operator (X, G, F or U). The path operators allow to identify specific executions, i. e., the computation tree.

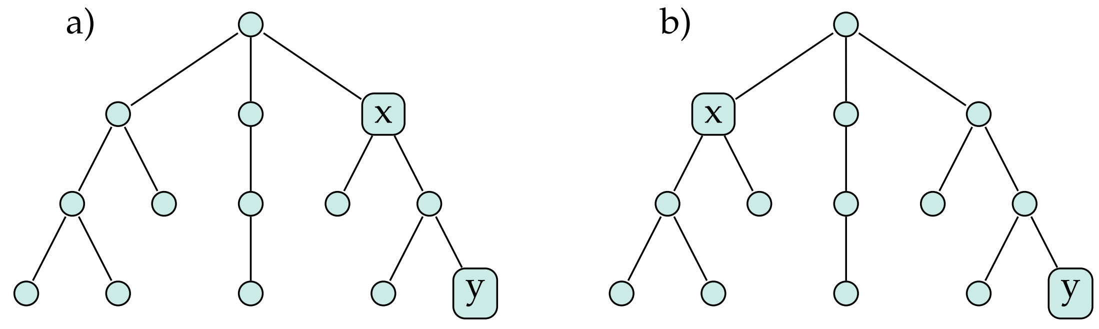


Figure 4.2: Two Computation Trees, (a) fulfills the CTL Formula $\mathsf{AG}(x \rightarrow \mathsf{EF}(y))$, (b) does not

Example 4.1.1 . The computation tree in Figure 4.2(a) fulfills the CTL formula $\mathsf{AG}(x \rightarrow \mathsf{EF}(y))$. For the whole tree at any point of time (AG), after the event x, a path exists (E) where the event y occurs at least once (F). The tree in Figure 4.2(b) does not fulfill the formula. This is because after the event x there is not any path that leads to the event y.

Past Linear Temporal Logic (PLTL) extends LTL with operators in order to argue about the past, see [LS95]. We denote them as $\mathsf{X}^{-1}, \mathsf{G}^{-1}, \mathsf{F}^{-1}, \mathsf{U}^{-1}$. For instance, $\mathsf{F}^{-1}\phi$ says that at some time in the previous trace, ϕ has been true. PLTL is of the same expressiveness as LTL, i. e., each PLTL formula can be rewritten to an equivalent LTL formula. We use PLTL because it allows to express some formulas more succinctly.

4.2 Instantiation of Property Pattern[1]

The overall goal of the promotion project is to verify if a given commissioning process is correct and besides that the synthesis of a process model from a specification. Verification means checking if a process model fulfills certain properties that are given. This is in contrast to validation, which is not at a formal level, but relies on the intuition of the users to ensure that a process meets their needs and is useful – For verification and synthesis it is necessary to specify properties. We have collected such properties in cooperation with

[1]Parts of this Subsection are published in [Mra+14] and in an extended version in [Mra+15]

domain experts from the AUDI AG by analyzing existing processes, and by closely observing experts of commissioning when designing these processes. To illustrate the notion of *property*, if a process uses more connections than available in the commissioning infrastructure, the process must halt. In this case process execution time is unnecessarily long. To this end, a commissioning process should never use more connections in parallel than the capacity for the protocol.

Properties typically are formulated as property rules, which are similar to compliance rules, see [Knu+10; LMX07]. For example, a property rule states that before executing Task X another Task Y has to be executed. This rule is called the *Precedence Pattern*. Verification itself is a process that consists of several phases, namely specifying the properties of the commissioning process, verifying them, and presenting the results to the users. Our concern is the design and realization of a framework supporting users throughout this entire process. In this chapter we focus on specification of properties. For the verification see Chapter 5 and Chapter 6 for the process synthesis.

Specification of properties gives way to several questions. First, in which form should one specify the properties to allow an automatic verification technique? Second, how one can utilize domain information for helping the users with the specification of their properties? Third, what is the usability of the solution?

Designing such a framework for the specification gives way to several challenges:

1. The knowledge on which characteristics an industrial process should fulfill is typically distributed among several employees in different departments. Often documentation is missing, and properties merely exist in the minds of the process modelers.
2. The properties frequently are context-sensitive, i. e., only hold in specific contexts of a commissioning process. For example, some tasks need different protocols to communicate with control units for testing at different factories. Due to this context-sensitiveness, the number of properties is very large, but with many variants with only small differences. This causes maintenance problems [KRL11]. For instance, an average process model from our use case has to comply with 39 properties. The properties and process models are constantly being revised.
3. To apply an automatic verification technique like model checking or for the synthesis of a process model, it is necessary to specify the properties in a formal language such as a temporal logic [SMS05]. With vehicle-commissioning

processes as well as in other domains, e.g., [DAC98; Ly+11a], specifying the properties in this way is error-prone and generally infeasible for domain experts who are not used to formal specification.

We have addressed these challenges based on the real-world use case of vehicle-commissioning processes. More specifically, we make the following contributions: We have analyzed which properties occur for vehicle commissioning processes and the respective context information. We have observed that there are few patterns these properties adhere to. We propose to explicitly represent these patterns, rather than each individual property. Next, we develop a model of the context knowledge regarding vehicle-commissioning processes. Here *context* consists of the components of a vehicle, their relationships and the constraints which the vehicle currently tested and configured must fulfill. We let a relational database manage the context information. To populate it, we use several sources, e. g., information on the vehicle components from production planning, constraints from existing commissioning processes, and information provided by the process designers themselves. Our framework uses this information to generate process-specific instances of the property patterns, transforms such instances to a Petri net, and verifies it against these properties.

Our approach is to use a system that uses the context information to automatically instantiate abstract classes of properties, called property patterns. Subsection 4.2.1 defines these property patterns. Subsection 4.2.2 presents the database to store the context information for the instantiation. Subsection 4.2.3 gives reasoning for the choice of the temporal logic to express the property instances. Subsection 4.2.4 describes the instantiation of the properties. Our evaluation in Subsection 5.4 will show that the framework as a whole does detect rule violations in actual real-world commissioning processes.

4.2.1 Property Pattern for Commissioning Processes

Together with domain experts of the department commissioning we collected typical properties for commissioning processes. From these collection we could conclude several property patterns covering nearly all found properties. The found patterns can be classified into five categories:

› *P1 Syntactical Correctness:* The process models should not contain any syntactical errors, e. g., a parallel-node in OTX has between 2 and *n* lane-nodes as child elements. Other kinds of nodes than lane are not allowed. Besides these syntactical constraints of the OTX notation syntactically constraints of the Audi AG exist. For Instance the naming conversation for tasks labels demands that a tasks starts with A_ followed by the short name of the control unit, and separated by two underlines the operation of the tasks, called Identifier. The short name of a control unit consists of three letters, followed by an optimal location separated by a underline.

$$\mathsf{Abfolge} = \mathsf{A}\ \_\ \mathsf{ECU}\ \_\ \_\ \mathsf{Bezeichner} \qquad \mathsf{ECU} = \mathsf{Bez}\ \_\ \mathsf{Ort}$$

› *P2 Resources of the control units:* Some control units need specific resources at a testing place to execution some testing operations. For instance some tasks require a bar code scanner. The scanner is only installed at some testing places. If not available the process will block when trying to execute the task.

› *P3 Connection to a communication protocol:* At the time two different communication protocols are in use to diagnose the electronic control units and to install the software. The protocols are KWP2000 (ISO 14230) and UDS (ISO 14229). Depended on the vehicle project and the version number a control unit uses one of these protocols. A maximum of 10 control units can communicate over a protocol at the same time. In total 14 connections can be used at one time without regard of the protocol. If 10 connections for one protocol are active a control unit needs to close the connection before another control unit can communicate over the protocol. This leads to a longer processing time and can cause a dead lock. Table 4.1 shows the classes of rules for the communication protocols.

› *P4 Dependencies between tasks:* The occurrence of some tasks is dependent on the occurrence of another task in the commissioning process. For Instance, some tasks cannot be executed in parallel due to technical reasons. Some tasks requires the previous execution of a second task as a precondition, e. g., before the error log of a control unit can be read another task has to check if the control unit is available. Table 4.1 shows the different property patterns for the dependencies between tasks. The property patterns are the result of an extension analysis of existing dependencies between tasks in real commissioning processes.

› *P5 Dependencies between control units:* Additionally to the dependencies between tasks, dependencies between control units exist. These dependencies hold true for all tasks that communicate with the specific control unit. For instance, each task using the control unit for the comfort electronic (KEL) cannot be used in parallel to a task using the control unit for the driving authorization (FBE). Table 4.1 gives the property patterns for the dependencies between control units.

Given this list, we conclude that for some properties a model-checking approach is feasible, while for others an algorithmic approach is more efficient. In general, model checking allows the efficient verification of properties that specify the temporal interaction of events in a process, e. g., properties regarding the control or data flow. However, properties that are static and refer to the process model at a whole, e. g., whether a certain resource is available for execution, can hardly be expressed in a temporal logic and verified by model checking.

Violations of those properties can result in undesirable characteristics of the process execution, subsequently referred to as *major disturbance*. An example is that it may block the execution of the process. This holds for properties R3, R4 and R5. Our approach is to define patterns for these properties. Table 4.1 shows the patterns. We use the term *minor disturbance* accordingly. This holds for properties R1 and R2. They are on a representational level, i. e., the syntax and the environment of the processes. Examples are violations of conventions or deviation from best practice or from guidelines. To ensure syntactical correctness (P1) we have implemented several checks, which take place before the model checking verification. First, our framework does an XML validation, in order to check, if the OTX document is valid against the XML schema. Additionally, we check whether the task labels comply with the company regulations. In order to check Property P2, our framework queries the database of context knowledge, see Subsection 4.2.2. It is done to check whether the process model given can use the respective resources. This resource check is static and is independent of the data flow. However, we do not exclude properties that specify the resource perspective in future iterations, cf. the data flow anti-patterns of [TAS09; Sta+14] or [Sch+11]. Such properties would require including the handling of resources at the Petri net level.

4.2.2 Database with Context Knowledge

Our goal is to generate instances of the property pattern of Table 4.1 for checking commissioning processes automatically, based on the information collected a priori. By definition, process context is any information that influences the process flow and not defined by the process model. [Ros+06] classifies context into *immediate context* (information that is related to the control flow), *internal context* (internal information of a company), *external context*, and *environmental context*. Our context information is mainly part of the first two categories (*immediate context* or *internal context*). As argued in [Ros+06], the *external context*, e. g., industry standards, influences the *internal context*. For instance, an industry standard can specify the use of a new communication protocol. The new protocol

Table 4.1: Property Patterns for Task and ECU Conditions

Property	Name	Description
P3.1	Maximal UDS Connections	The number of paralleled connections to communication protocol UDS should not exceed 10.
P3.2	Maximal KWP-2000 Connections	The number of paralleled connections to communication protocol KWP2000 should not exceed 10.
P3.3	Maximal Connections	The number of connections to the protocols UDS and KWP2000 should not exceed 14.
P4.1	Sequential before (Precedence)	If a task A occurs in the process, a task B has to occur before A.
P4.2	Optional Sequential before	If both A and B occur in the commissioning process, B has to occur before A. B can completely be missing.
P4.3	Sequential after (Response)	The occurrence of task A leads to the occurrence of task B.
P4.4	Non-Parallel	Tasks A and B are not allowed to occur in parallel.
P5.1	Restricted access	Only one task at the same time can access each ECU C.
P5.2	Non-Parallel	Some ECU C must never be tested in parallel with an ECU C_2.
P5.3	Close Connection	Task close-C must close the connection to an ECU C.

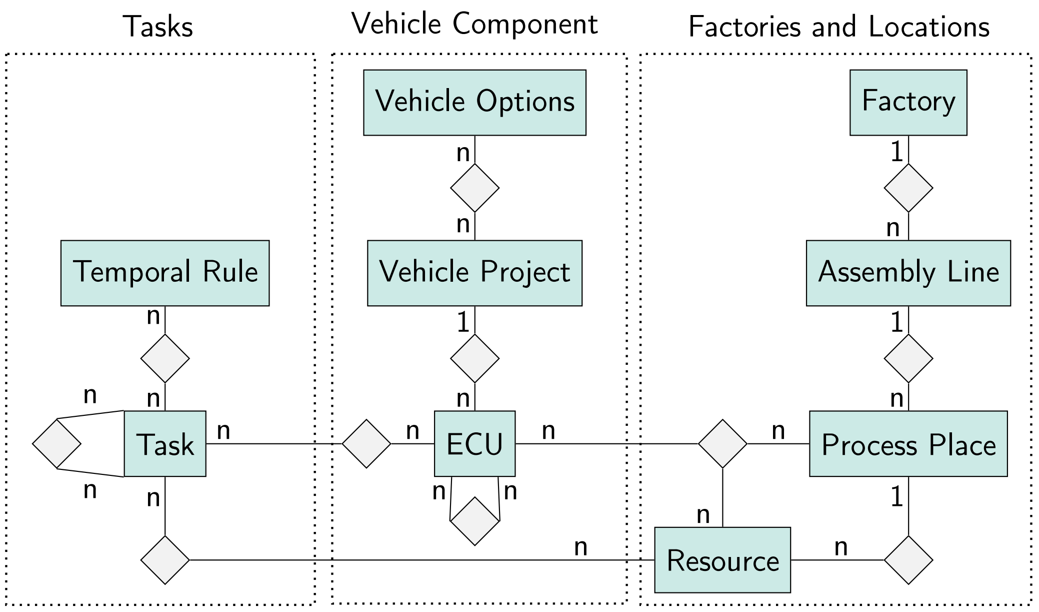


Figure 4.3: Excerpt of the Data Base Schema for the Context Knowledge

may have a different capacity and thus changes how often certain tasks can be in parallel. We have designed a relational database to manage this context information. The rationale is that the context information is represented in a user-friendly manner. The database needs to fulfill the following requirements:

› *DB-R1 Representing Contextual Information:* The database should contain the contextual information of the commissioning processes. First, the properties of the processes depend on the vehicle, i. e., components built into it to be tested,mostly electronic control units. The type of the vehicle and its concrete configuration determine the ECUs required. Second, the properties of the processes depend on the process places the component is tested at. The assembly lines for testing and configuring consist of these places. They vary in different factories. Third, there exist dependencies between the commissioning tasks, see Subsection 4.2.1.

› *DB-R2 User-Friendly Specification of the Properties:* Engineers should be able to specify the properties in a comfortable way. To this end, the structure of the database should support the perspective of these experts and not require extensive experience with formal modeling.

› *DB-R3 Use of Existing Documents and Information:* Defining the properties should use as much information from previous steps of the production life cycle as is available. Information on the vehicle and its components which have to be tested arises during the production design and production planning. The database should contain this information.

Figure 4.3 shows an excerpt of our database model illustrating the overall structure, see [Sch12] for more details. Our model consists of three parts, in line with *DB-R2*. One part comprises the vehicle components, e. g., the ECUs, including variants of the component configurations, so-called options of the vehicle. The product planning step delivers such information, which we use to populate the respective part of the database, cf. *DB-R3*. Another part contains the commissioning task objects, dependencies between tasks, and constraints on the tasks, specified as CTL formulas. A third part describes the assembly lines with process places and resources available there. Dependencies between the parts complete the model, e. g., the resources required to perform a testing task. The structure of the context knowledge given as database model allows to define and maintain the context in a form expert users are familiar with, cf. *DB-R1*, *DB-R2*.

4.2.3 Choice of Temporal Logic

The properties in Table 4.1 reason about events and their temporal relationships, e. g., that after an event a an event b occurs sometimes in the future. To this end, there is the need to specify the properties in a logic that allows for this temporal behavior, i. e., a temporal logic. The most common temporal logics are LTL, CTL, CTL*, and μ-calculus [Eme97]. CTL* is an extension of CTL without the limitation that the path operators and temporal operators have to always occur in pairs. For instance, it is possible to formulate $\mathsf{E}\,\mathsf{F}\,\mathsf{G}(\phi)$. LTL is another subset of CTL* where each formula always starts with the path operator A followed only by temporal operators. Often LTL formulas are written without the initial A, and the temporal operators (X, G, F, U) are replaced by $(\mathcal{O}, \square, \lozenge, \mathcal{U})$. Due to the absence of path operators, besides the initial A, LTL can only argue about linear sequences of events, i. e., there is only one possible future to be specified.

Formulas ϕ exist which can be expressed in CTL but not in LTL, while other formulas ψ can be expressed in LTL but not in CTL. The expressiveness of CTL* is a real superset of both CTL and LTL. The μ-calculus allows for an even larger expressive power than CTL*. Figure 4.4 shows the expressiveness of the four temporal logics. The property patterns of Table 4.1 lie in LTL $\cap$ CTL, i. e., can be expressed in all temporal logics. Related research has described patterns expressible in CTL but not in LTL, e. g., the weak data flow patterns of [TAS09].

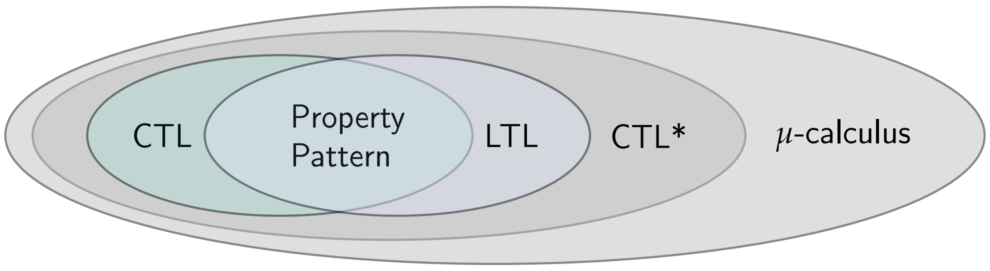


Figure 4.4: The Expressiveness of four Temporal Logics.

To allow future patterns to be expressed, at least the expressive power of CTL is necessary.

Another, but related issue is that we are dealing with large process models in real world settings, thus we need to consider the complexity of the verification. Model checking of a CTL formula is in complexity class $\mathcal{P}$, model checking of LTL is in complexity class PSPACE. The same holds for CTL* and the μ-calculus. On a closer look the differences in complexity classes can be to put into perspective. The complexity is determined by two parameters, the size of the model $|M|$, and the size of the property $|\phi|$. The runtime complexity for LTL scales linear with $|M|$ and exponentially with $|\phi|$. For a bounded $|\phi|$ the runtime of a model checking with LTL is comparable to CTL.

In conclusion, CTL is most appropriate in our framework. It allows expressing our commissioning properties, model checking is efficient, and mature tools exist. However, observe that our approach is not limited to CTL. Other temporal logics are possible if a respective model checker is integrated.

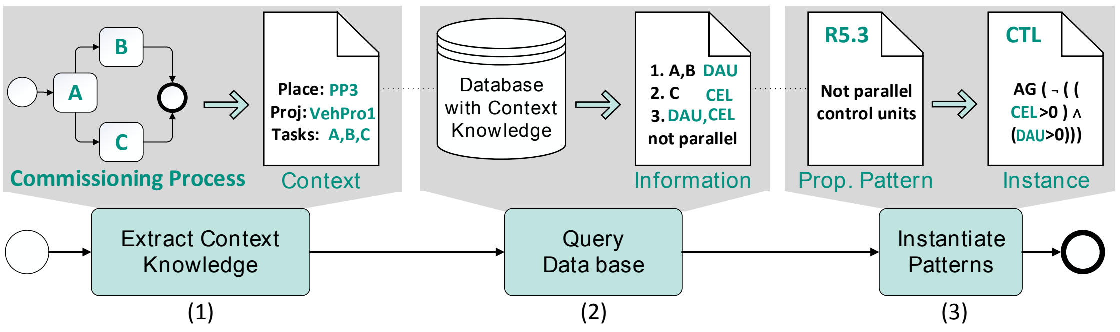


Figure 4.5: The Procedure for the Instantiation of Property Instances

4.2.4 Property Instances

Figure 4.5 shows the process for the instantiation of the property patterns. First, our approach determines the context of the commissioning process. The context consists of but not limited to the process place, the vehicle project, and the contained tasks, see Step 1 in Figure 4.5. The context is used to query the database with context knowledge to extract the required information for the instantiation, see Step 2 in Figure 4.5. For instance, the following queries to the database: Which control units use the tasks from the process model at the process place PP3 for the vehicle project VehPro1 and which dependencies exists between the control units? This context information is used to generate the instances of the property patterns from Table 4.1, see Step 3 in Figure 4.5. Table 4.2 shows the CTL-Formula for the property patterns from table 4.1. The atomic propositions are inequations referring to the distribution of tokens, i. e., the state of a Petri net. For instance, $(A_{\text{run}} > 0)$ refers to all states where the place A_{run} contains more than zero tokens, i. e., A is currently running.

Example 4.2.1 . The commissioning process contains the tasks $\{A, B, C\}$ and is executed at the process place PP2. The vehicle project is VehPro1. This context is used to query the database. The information contains that A and B use the control unit for comfort electronic (CEL) and the tasks C uses the control unit for driving authorization (DAU). A dependency exists for the vehicle project the tasks of CEL and DAU cannot be in parallel. The context information is used to instantiate the property patterns. For the property pattern R3.3 this results in the instance: $AG(\ \neg\ (\ (CEL > 0)\ \wedge\ (DAU > 0)\ )\ )$.

The dynamic generation of properties from the database has several benefits compared to their direct specification in, say, CTL. First, for a commissioning process given we only consider the properties relevant for it. An allocation of the correct rules to the correct processes, cf. [KRL11] is thus not needed. Second, the maintenance of the properties is simplified. For example, if a new ECU is available for a process place, one only needs to add the information into the database, i. e., to Relation *ECU*. With a direct specification in turn, one might have to specify several hundred properties. Third, the database stores the contextual knowledge in a centralized and non-redundant form, instead of managing all properties specified in CTL. For example, the Pattern »*A* leads to *B*« has a few hundred instances. If, for example, the need to change the pattern to »The first occurrence of *A* leads to *B*« arose, updating would be avoided.

Table 4.2: CTL-Formula for the Property Patterns

Property	Name	CTL
P3.1	Maximal UDS Connections	$\mathsf{AG}\ (\ \text{UDS} \leq 10\ )$
P3.2	Maximal KWP2000 Connections	$\mathsf{AG}\ (\ \text{KWP2000} \leq 10\ )$
P3.3	Maximal Connections	$\mathsf{AG}\ (\ (\text{UDS} + \text{KWP2000}) \leq 14\ )$
P4.1	Sequential before	$\mathsf{A}\ [\ (A_{run} = 0)\ \ \mathsf{W}\ \ (B_{run} > 0)\ ]$
P4.2	Optional Sequential before	$\mathsf{A}\ [\ (A_{run} = 0)\ \lor\ \mathsf{AG}(\ B_{run} = 0\ )\ )\ \mathsf{W}\ (\ B_{run} > 0\ )\ ]$
P4.3	Sequential after	$\mathsf{AG}\ (\ (A_{run} > 0)\ \rightarrow\ \mathsf{AF}\ (B_{run} > 0)\ )$
P4.4	Non-Parallel	$\mathsf{AG}\ (\neg(\ (A_{run} > 0)\ \land\ (B_{run} > 0)\ ))$
P5.1	Restricted access	$\mathsf{AG}\ (X_{\text{ECU}} \leq 1)$
P5.2	Non-Parallel	$\mathsf{AG}\ (\ \neg\ (\ (X_{\text{CON}} > 0)\ \land\ (Y_{\text{CON}} > 0)\ )\ )$
P5.3	Close Connection	$\mathsf{AG}\ (\ (\text{END} > 0)\ \rightarrow\ (X_{\text{CON}} = 0)\ )$

Fourth, domain experts only need to specify properties in CTL when there is a new property type, so the number of these error-prone and complicated tasks is reduced.

In Section 5.4.1 we evaluate our approach for the instantiation of property patterns. To this end, we show that the instantiation is able to generate sufficient properties for the verification of commissioning processes in the automobile industry.

4.3 Mining of Specifications

Behavior patterns are relationships between tasks or groups of tasks which occur frequently in the set of possible executions of a process model. A *property* in turn is a characteristic that all process models have to fulfill. Behavior patterns are good candidates for properties unknown so far.

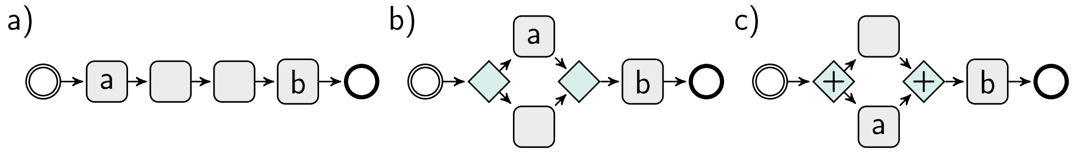


Figure 4.6: Three Structurally Different Process Models that Fulfill the Property: Response(a, b).

Example 4.3.1 . Think of the following behavior pattern: A task a occurs before a task b in nearly all cases. This is a hint that a might be a precondition for b.

In general, behavior patterns do not occur directly in the process model, i. e., no explicit structure in the graph representing the model exists corresponding to the pattern. Here, the differentiation between behavior and structure is the conventional one from process similarity; see [Dij+11].

Example 4.3.2 . Figure 4.6 shows three structurally different process models, in BPMN notation. One behavior pattern is Response, i. e., after the execution of a Task a, Task b occurs at some time in the future. Each model fulfills the pattern, but an analysis solely looking at the graph-structure level will not detect it.

Our concern is detecting many important and from an application perspective useful behavior patterns from real process models of true-to-life size efficiently. More specifically, patterns to be discovered have to be frequent in a collection of models. These patterns are useful for several very important use cases. We give three examples, namely *verification*, *synthesis*, and *improved flexibility*.

Verification : We can detect the rare cases where a property does not hold, which are likely modeling errors or at least a violation of best practices. New process models can be verified using these detected dependencies [LMX07; ADW08; Wei+11a].

Synthesis : A synthesis algorithm can use the dependencies found to generate more general process models [Awa+11; KRG07; Yu+08].

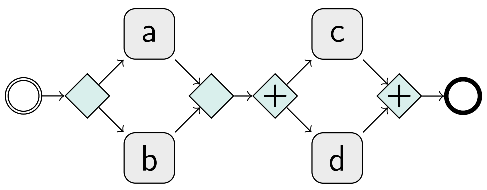
Improved Flexibility: There frequently are large number of process models. See [MBG14] for a study. Recent research has shown that declarative process models tend to be more concise and readable than imperative ones with the same flexibility [AP06; Mon+10]. However, the shift from imperative process models to a declarative one is not trivial. Nevertheless, a discovery approach such as ours can detect important dependencies, i. e., the core of the new flexible model.

Behavior patterns describe the control flow of business processes, in contrast to, e. g., data flow. In this study, we focus on block-based process models not containing loops. There is a number of settings with this characteristic, for instance in manufacturing. In commissioning processes, loops are unnatural as well, since a feature is tested once. If a problem occurs and is fixed, a new commissioning process is started. A further aspect is to consider properties not only for individual tasks but for groups of activities as well.

Example 4.3.3. With the commissioning of vehicles, each task performs an operation on an electronic control unit (ECU). There are dependencies of the form: Any task using ECU_1 has a certain dependency to any task using ECU_2. For instance, before any task on ECU_2 is executed, all tasks on ECU_1 have to be completed.

Process models often are represented as labeled graphs or trees. Behavior patterns cannot be discovered on the structural level of the process-model graph in a straightforward way, e. g., frequent subgraph mining [JCZ13], cf. Example 4.3.2. Each subgraph can be infrequent in the repository, but the pattern $\mathsf{Response}(a, b)$ can be frequent. A lot of related work on discovering patterns in process-model repositories is confined to the structure level [Gre+05; RDC14; Lau+09; LRW09]. Next, considering the behavior space of the process model explicitly, i. e., the possible execution paths, is not an option either. This is because this space grows exponentially with the size of the model. For real-word processes of normal size, it is not possible to generate that space explicitly. So our setting is also different from ones analyzing previous executions [Che+09; CMM14], e. g., logs whose sizes are limited.

As a first step, we collect descriptions of behavior patterns from the scientific literature. To avoid constructing the behavior space explicitly, we use a format for concise summaries of this space for a set of process models, originally proposed in Weidlich et al. [WMW11; Wei+11b], see Section 4.3.1. These summaries have two important characteristics. First, their generation can take place efficiently, as we will explain. Second, it is feasible to discover the model-based behavior



$$\mathcal{L}_{PM} = \{ \begin{array}{l} <a.s,\ a.e,\ c.s,\ c.e,\ d.s,\ d.e> \\ <a.s,\ a.e,\ c.s,\ d.s,\ c.e,\ d.e> \\ <a.s,\ a.e,\ c.s,\ d.s,\ d.e,\ c.e> \\ <b.s,\ b.e,\ c.s,\ c.e,\ d.s,\ d.e> \\ <b.s,\ b.e,\ c.s,\ d.s,\ c.e,\ d.e> \\ <b.s,\ b.e,\ c.s,\ d.s,\ d.e,\ c.e> \\ <a.s,\ a.e,\ d.s,\ c.e,\ c.s,\ c.e> \\ <a.s,\ a.e,\ d.s,\ c.s,\ d.e,\ c.e> \\ <a.s,\ a.e,\ d.s,\ c.s,\ c.e,\ d.e> \\ <b.s,\ b.e,\ d.s,\ d.e,\ c.s,\ c.e> \\ <b.s,\ b.e,\ d.s,\ c.s,\ d.e,\ c.e> \\ <b.s,\ b.e,\ d.s,\ c.s,\ c.e,\ d.e> \ \} \end{array}$$

Figure 4.7: A Simple Process Model P and its Behavior Space $\mathcal{L}_P$

patterns from the summary. While not trivial either, we present a respective efficient solution in Section 4.3.2. Our evaluation shows that our overall approach is applicable to real process models of realistic size. A case study with domain experts reveals that the results are indeed useful. The patterns discovered point to errors in process models already in use that had been undetected so far.

4.3.1 Behavior Patterns

In this dissertation, we focus on the control-flow aspect of processes. A process model P specifies which *events* are executed in *what order* [Aal11] (p. 31). An event can be the execution of a task or the sending of a message. A task a throws two events for its execution, $a.s$ for its start and $a.e$ for the end. The set of execution sequences a process model can generate is its behavior space $\mathcal{L}_P$. We refer to an individual sequence in $\mathcal{L}_P$ as σ. If an event is part of the sequence σ we refer to it as $a \in \sigma$.

Example 4.3.4 . Figure 4.7 shows a simple process model and its behavior space $\mathcal{L}_P$. Even for the tiny process depicted, $\mathcal{L}_P$ consists of 12 different executions.

In this dissertation we assume that there exists an unique mapping between task names and their meaning. A counterexample would be that an abstract notion, e. g., *payment*, can be represented by different tasks. In such settings, a preprocessing resulting in a mapping of meaning to tasks, e. g., [Leo+15; Kli+13], may be necessary.

Table 4.3: Control Flow Patterns

Pattern	Description
Existence	A given task a has to occur sometime during all executions
Bounded Existence	A given task a has to occur between *min* and *max* times in every execution
Parallel	Two tasks a and b must not occur in parallel
Precedence	A task a requires the previous execution of a task b
Response	A task a requires the subsequent execution of a task b
Succession	Task a is always followed by a task b and vice versa
Mutually Exclusive	If task a occurs, task b is absent and vice versa
Inclusive	Task a mandates that task b is present
Chain Ordering	Task a mandates that one of the tasks $b_1, b_2, \dots, b_n$ follows

Model-based behavior patterns are control-flow related occurrences of tasks frequently observed in the behavior space of process models. For instance, the occurrence of a task a is always followed by the occurrence of a task b. We have collected frequently used patterns from the literature. To this end, we have covered the specification language BPMN-Q [ADW08], the compliance rule graphs [Ly+11a], Declare [Mon+10], the Property Specification Patterns [DAC98], and Commissioning Property Patterns [MMB14a; Mra+14]. See Table 4.3 for a definition of the patterns. Table 4.4 shows which formalism supports which patterns. In the commissioning context, it is sufficient to let the properties refer to task labels, in contrast to, say, task parameter values. Observe the difference between Response(a,b) and Precedence(b,a). Figure 4.6(b) shows a process model that fulfills the pattern Response(a,b) but not Precedence(b,a). This difference is important in real contexts, see [ADW08; DAC99; Mon+10; Mra+14].

We do not consider *Chain Ordering* any further. An empirical study [DAC99] has shown that *Chain Ordering* occurs in only 9 out of 555 cases. While rare

Table 4.4: Five Declarative Specification Languages and the Patterns they can Represent Directly (Tick) or as a Combination of Patterns (Circle). The Omission of a Mark Means that the Language does not Support the Pattern.

	Existence	Bounded Existence	Parallel	Precedence	Response	Succession	Mutually Exclusive	Inclusive	Chain Ordering
Bpmn-q [ADW08]	✓			✓	✓	○			
Comp. Rule Graph [Ly+11a]	✓		○	✓	✓	✓	○	○	
Declare [AP06]	✓	○	✓	✓	✓	✓	✓	✓	✓
Prop. Spec. Pattern [DAC98]	✓	✓	○	✓	✓	✓	✓	○	✓
Comm. Prop. Pattern [Mra+14]	✓	✓	✓	✓	✓	✓	✓		

patterns might be interesting in general, they do not help in our specific context. [RFA12] has identified a set of frequently used specification elements similar to the one we consider. [DAC99] and [Mra+14] shows that the patterns from Table 4.4 occur in nearly any specification.

Table 4.5 shows the pltl formula for each pattern. Patterns 1 and 2 deal with the occurrence of a single event or task. Patterns 3–8 describe a relationship between two events or tasks. We call Patterns 1 and 2 *occurrence patterns* and 3 to 8 *relational patterns*. One can bring each pattern Pa from Table 4.5 into the form $Pa = \mathsf{G}(X \rightarrow Y)$. X is the trigger of the pattern and Y the cause. For instance, in the Response pattern the start event of a is the trigger, and the occurrence of $b.s$ is the cause. In the table, a and b are placeholders for tasks; replacing a and b with concrete tasks yields an *instance of the pattern*.[2] We want to detect the *frequent strong* instances of the model-based behavior patterns.

Definition 6 (Frequent and Strong Behavior Pattern) .

An instance of a model-based behavior pattern is *frequent* if the number of process models in a repository containing the instance is greater than a predefined threshold min sup. A model-based behavior pattern is *strong* if the share of process models in the repository containing the pattern among those meeting the precondition, i. e., containing the trigger, is greater than threshold min conf.

[2]Slightly abusing our own terminology, we may use the terms »pattern« instead of »instance of pattern« whenever clear from the context.

Table 4.5: Control Flow Patterns

No.	Pattern	PLTL
1.	Existence	$\mathsf{F}(a.s)$
2.	Bounded Existence	see [Mon+10]
3.	Parallel	$\mathsf{G}(a.s \vee b.s \rightarrow (a.s \wedge [\neg(b.s \vee b.e) \ \mathsf{U}\ a.e]$ $\vee (b.s \wedge [\neg(a.s \vee a.e) \ \mathsf{U}\ b.e]))$
4.	Precedence	$\mathsf{G}(b.s \rightarrow \mathsf{F}^{-1}\ a.s)$
5.	Response	$\mathsf{G}(a.s \rightarrow \mathsf{F}\ b.s)$
6.	Succession	$\mathsf{G}(a.s \ \vee \ b.s \rightarrow (a.s \ \wedge \ \mathsf{F}\ b.s) \vee (b.s \wedge \mathsf{F}^{-1}\ a.s))$
7.	Mutually Exclusive	$\mathsf{G}(a.s \vee b.s \rightarrow (a.s \wedge \neg\mathsf{F}\ b.s) \vee (b.s \wedge \neg\mathsf{F}\ a.s))$
8.	Inclusive	$\mathsf{G}(a.s \rightarrow \mathsf{F}\ b.s \vee \mathsf{F}^{-1}\ b.s)$

For the Bounded Existence pattern we want to discover the so-called *maximal bounds*, i. e., there do not exist any larger bounds that are frequent as well.

Example 4.3.5. Figure 4.6 shows three process models. The ones in (a) and (c) fulfill the Precedence pattern, i. e., Task a has occurred before b. The second model however does not fulfill the pattern. For MIN SUP $= 2$ and MIN CONF $= \frac{3}{4}$ the pattern is frequent but not strong. This is because in $\frac{1}{3}$ of the cases where b is in a process model, b can occur without a previous execution of a.

$\mathcal{L}_P$ grows exponentially with the degree of concurrency of P. The behavior space for one parallel split with n lanes and $k_1, k_2, \ldots k_n$ tasks in each lane can be calculated with the multinomial coefficient [MMB14a], cf. Subsection 3.1.2.

$$\binom{\sum_{i \in [1,n]} k_i}{k_1, k_2, \ldots, k_n} = \frac{(\sum_{i \in [1,n]} k_i)!}{k_1! \ k_2! \ \ldots \ k_n!}$$

Example 4.3.6. Figure 4.8(a) shows a process model that consists of a parallel split with m parallel lanes and n tasks per lane. Figure 4.8(b) shows the size of $\mathcal{L}_P$ with n on the x-axis, $m = \{2, 5, 10\}$. Observe that the y-axis is $\log_{10}$ scaled.

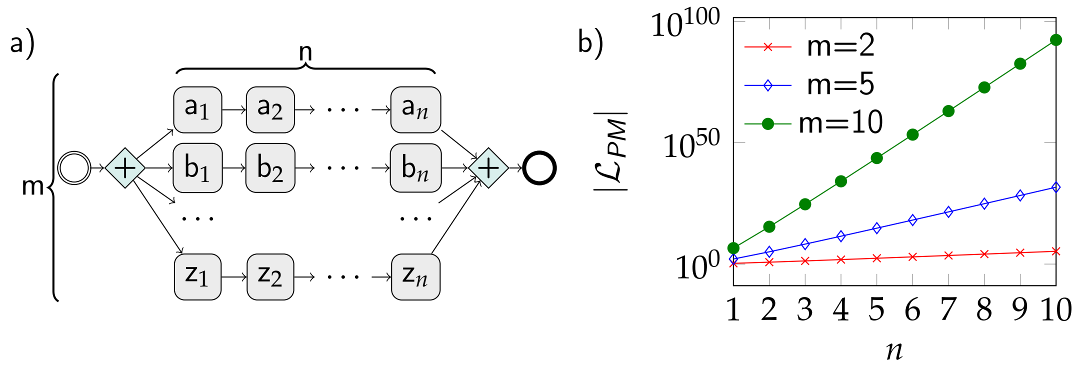


Figure 4.8: (a) Parallel Split with m Lanes and n Tasks Each (b) The Size of $\mathcal{L}_P$

Discovering model-based behavior patterns in $\mathcal{L}_P$ is not feasible for models of realistic size. Our approach is to use a summary of the process models. The summary should have two important characteristics:

1. One can compute the frequent model-based patterns from it.
2. It is possible to efficiently compute this summary.

Weidlich at al. [WMW11] introduces the notion of *behavioral profiles*. Behavioral profiles characterize the behavior space in terms of order constraints between the task of a process model. They are grounded on the weak order notation.

Definition 7 (Weak Order, after [WMW11]) .
Let P be a process model with the tasks T. A pair of tasks $(x, y) \in T \times T$ are in a *weak order relation* $\succ$ if an execution sequence $\sigma = t_1, \ldots, t_n$ exists with $1 \leq j < k \leq n$ with $t_j = x$ and $t_k = y$.

Using the weak order relation, it is possible to define three other relations whose combination forms the behavior profile.

Definition 8 (Behavior Profile, after [WMW11]).

Let P be a process model with the tasks T. A pair of tasks $(x, y) \in T \times T$ can be in the following relations:

› The strict order relation $\rightsquigarrow$, if $x \succ y$ and $y \not\succ x$

› The exclusiveness relation $+$, if $x \not\succ y$ and $y \not\succ x$

› The interleaving order relation $\|$, if $x \succ y$ and $y \succ x$

$B_P = \{\rightsquigarrow, +, \|\}$ is the behavior profile of P.

The interleaving order and the exclusiveness relation are symmetric. It is possible to define the reverse strict order relation $\rightsquigarrow^{-1}$ as $x \rightsquigarrow y \Leftrightarrow y \rightsquigarrow^{-1} x$. However, the behavior profile as defined so far does not allow expressing the causality of two tasks. Our context in turn requires this information in order to detect most of our behavior patterns. [Wei+11b] extends the basic behavior profile to the causal behavior profile C_P with an additional relation.

Definition 9 (Causal Behavior Profile, after [Wei+11b]).

Let P be a process model with the tasks T.

A pair of tasks $(x, y) \in T \times T$ is in the co-occurrence relation $\gg$ if for all execution paths σ in P it holds that $x \in \sigma \Rightarrow y \in \sigma$.

$C_P = \{\rightsquigarrow, +, \|, \gg\}$ is the causal behavior profile of P.

[Wei+11b] shows how to compute the causal behavior profile for a process model efficiently. [WMW11; Wei+11b] uses the behavior profile to check the consistency or similarity of two given process models. We in turn will use those relations as core of a summary in order to detect the frequent behavior patterns in a data base. As further constituent of our summary, we collect the minimum and maximum numbers of task a appearing in a process model. This is to cover the Existence and Bounded Existence patterns.

To detect the frequent patterns, we propose a mapping of the relations to PLTL formulas. Table 4.6 defines this mapping. Observe that the tasks are in exactly one of the classic behavior relations from Definition 2.3 ($\rightsquigarrow, \rightsquigarrow^{-1}, +, \|$).

Table 4.6: Appearance and Ordering Relationships

RELATIONSHIP	SYMBOL	IS TRUE IF
strict order relation	$a \rightsquigarrow b$	$\mathsf{G}(a.s \rightarrow \neg\mathsf{P}\ b.s)$
reversed strict order	$a \rightsquigarrow^{-1} b$	$\mathsf{G}(a.s \rightarrow \neg\mathsf{F}\ b.s)$
exclusiveness relation	$a + b$	$\mathsf{G}(a.s \lor b.s \rightarrow (a.s \land \neg\mathsf{F}\ b.s)$ $\lor\ (b.s \land \neg\mathsf{F}^{-1}\ a.s))$
interleaving order relation	$a \parallel b$	$\mathsf{F}(a.s \rightarrow [\neg a.e\ \mathsf{U}\ b.e])$
co-occurrence relation	$a \gg b$	$\mathsf{G}(a.s \rightarrow \mathsf{F}\ b.s \lor \mathsf{F}^{-1}\ b.s)$

Example 4.3.7. Figure 4.7 shows a process model and its complete log $\mathcal{L}_P$. The tasks of the process model are in the following relations.

BEHAVIOR PROFILE

$$\begin{array}{c|cccc} & \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{d} \\ \hline \mathsf{a} & + & + & \rightsquigarrow & \rightsquigarrow \\ \mathsf{b} & + & + & \rightsquigarrow & \rightsquigarrow \\ \mathsf{c} & \rightsquigarrow^{-1} & \rightsquigarrow^{-1} & + & \parallel \\ \mathsf{d} & \rightsquigarrow^{-1} & \rightsquigarrow^{-1} & \parallel & + \end{array}$$

CO-OCCURRENCE RELATION

$$\begin{array}{c|cccc} & \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{d} \\ \hline \mathsf{a} & \gg & & \gg & \gg \\ \mathsf{b} & & \gg & \gg & \gg \\ \mathsf{c} & \gg & \gg & \gg & \gg \\ \mathsf{d} & \gg & \gg & \gg & \gg \end{array}$$

OCCURRENCES

$$\begin{array}{c|cc} & \text{MIN} & \text{MAX} \\ \hline \mathsf{a} & 0 & 1 \\ \mathsf{b} & 0 & 1 \\ \mathsf{c} & 1 & 1 \\ \mathsf{d} & 1 & 1 \end{array}$$

4.3.2 Discovery of the Patterns in a Process Repository

This section describes the way our approach detects frequent patterns in a process-model repository, based on the summary consisting of relationships. First, we will describe the data structures behind our algorithms. Next, we will show how to discover the Existence and Bounded Existence patterns that are frequent and strong efficiently. Followed by a description of the discovery algorithm for the relational patterns. We conclude with the approaches of how to discover patterns for groups of tasks.

For each pair of tasks in the process models, we calculate the causal behavior profile and store it. Table 4.7 shows an illustration for two process models P_1 and P_2. For each pair of tasks, the relation Task contains both tasks (Task_1, Task_2), the process model (Process) and the behavior relations of the two tasks.

Additionally, Capacity contains for each task of each process model P how many times it occurs at least, and at most respectively, in any path in $\mathcal{L}_P$. The number of tuples in the database, with $\#Tasks_P$ being the number of tasks in P,

Table 4.7: The Task-Relation (Left) and the Capacity-Relation (Right)

Process	Task$_1$	Task$_2$	Occ.	Ord.
...	...	...	...	...
P_1	a	b	$\Rrightarrow$	$\twoheadrightarrow$
P_1	b	a		$\twoheadleftarrow$
...	...	...	...	...
P_2	a	b	$\Rrightarrow$	$\twoheadrightarrow$
P_2	b	a	$\Rrightarrow$	$\twoheadleftarrow$
...	...	...	...	...

Process	Task	min	max
...	...	...	...
P_1	a	1	1
P_1	b	0	1
...	...	...	...
P_2	a	1	1
P_2	b	1	1
...	...	...	...

is $\sum_{P \in \mathcal{P}} |\#Tasks_P|^2$. I.e., the number of tuples grows quadratically with the size of the process models. This can lead to several thousand entries. This number is easily processable; contrast this with the exponential growth of $\mathcal{L}_P$.

The discovery of the Existence and Bounded Existence Patterns is straightforward using the Capacity-Relation. Regarding Existence, recall that a pattern is frequent if the task occurs in more than MIN SUP process models.

Algorithm 2 calculates the maximal Bounded Existence pattern, somewhat reminiscent of the Apriori algorithm [AS94]. From the intervals of size k we generate candidate intervals of size $k+1$ (Line 5). If a candidate is infrequent, it is pruned (Line 6). The algorithm continues until the largest interval has been found.

Algorithm 2 generateBoundedExistence(Task a, $min\,sup$): Set$\langle\ \rangle$ Pattern Pa

```
1: $\mathcal{P} \leftarrow$ all process models that contain $a$
2: $Pa \leftarrow$ Find all frequent 0-intervals
3: for all $k = 0$; until $Pa$ is unchanged; $k{+}{+}$ do
4:     $\mathcal{C} \leftarrow$ combine the k-intervals to form candidates
5:     Remove the infrequent candidates in $\mathcal{C}$
6:     if $\mathcal{C} \neq \emptyset$ then $Pa = \mathcal{C}$
7: return $Pa$
```

Example 4.3.8 . Think of a Task a that is part of four process models. In one, a occurs 1 to 5 times, in the second one 3 to 7, in the third one 0 to 2, in the fourth one 1 to 6 times. For MIN SUP = 3 the maximal Bounded Existence of a has values 3 to 5. See Figure 4.9.

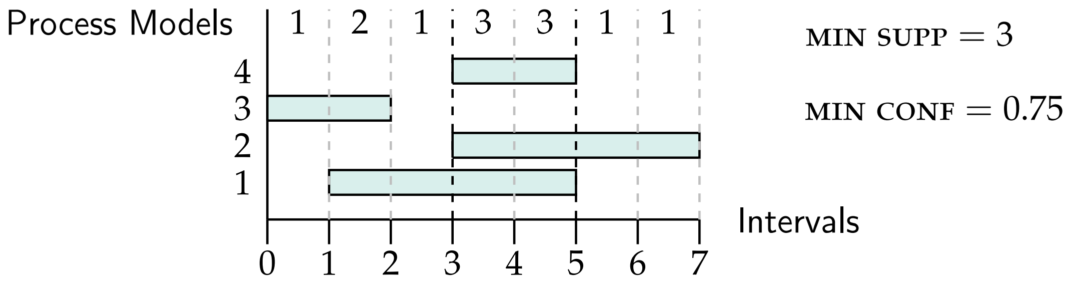


Figure 4.9: An Example for the Bounded Existence Pattern for four Process Models.

This subsection presents our approach for discovering relational patterns, i. e., Patterns 3–8. First, we map the relationships of Table 4.6 to patterns. Next, we define the conditions for *strong* and *frequent* patterns and the relationships between patterns. Lastly, we show our discovery algorithm in order to find the frequent strong patterns.

Each pattern consists of a *trigger* and a *cause*. The *trigger* always is the occurrence of a task or the co-occurrence of two tasks. A process model P for a pattern is activated if the Capacity-Relation contains the triggering tasks. For the *cause* condition, we check if the the tasks contained in it are in the Task-relation. Table 4.8 shows the trigger and the cause for each pattern.

Example 4.3.9. For the Response(a, b) pattern, any process model P that contains task a *triggers* it. We then check whether the *cause* is fulfilled, i. e., for each process model P containing a we check if the tuple $(P, a, b, \Rrightarrow, \twoheadrightarrow)$ is in the Task-Relation.

Lemma 1. If two tasks a, b are in the relations defined in Table 4.8, then a, b fulfill the respective pattern.

We can prove Lemma 1 by means of equivalence transformations. For instance, consider the Response pattern:

Table 4.8: The Trigger and Cause of the Patterns

Pattern	Trigger	Cause
Parallel	$a \vee b$	$(a,b) \notin \parallel$
Precedence	b	$(a,b) \in \twoheadrightarrow \ \wedge \ (b,a) \in \Rrightarrow$
Response	a	$(a,b) \in \twoheadrightarrow \ \wedge \ (a,b) \in \Rrightarrow$
Succession	$a \vee b$	$(a,b) \in \twoheadrightarrow \ \wedge \ (a,b) \in \Rrightarrow \ \wedge \ (b,a) \in \Rrightarrow$
Mutually Exclusive	$a \vee b$	$(a,b) \in \times$
Inclusive	a	$(a,b) \in \Rrightarrow$

$$(a,b) \in \twoheadrightarrow \ \wedge \ (b,a) \in \Rrightarrow \ \Leftrightarrow \mathsf{G}(a.s \rightarrow \mathsf{F}\ b.s \vee \mathsf{P}\ b.s) \wedge \mathsf{G}(a.s \rightarrow \neg \mathsf{P}\ b.s)$$
$$\Leftrightarrow \mathsf{G}((a.s \rightarrow \mathsf{F}\ b.s \vee \mathsf{P}\ b.s) \wedge (a.s \rightarrow \neg \mathsf{P}\ b.s)) \Leftrightarrow \mathsf{G}((\neg a.s \vee \mathsf{F}\ b.s \vee \mathsf{P}\ b.s)$$
$$\wedge(\neg a.s \vee \neg \mathsf{P}\ b.s)) \Leftrightarrow \mathsf{G}(\neg a.s \vee ((\mathsf{F}\ b.s \vee \mathsf{P}\ b.s) \wedge (\neg \mathsf{P}\ b.s))) \Leftrightarrow \mathsf{G}(\neg a.s \vee$$
$$(\mathsf{F}\ b.s \wedge \neg \mathsf{P}\ b.s)) \overset{(*)}{\Leftrightarrow} \mathsf{G}(\neg a.s \vee \mathsf{F}\ b.s) \Leftrightarrow \mathsf{G}(a.s \rightarrow \mathsf{F}\ b.s) \Leftrightarrow \mathsf{Response}(a,b))$$

(*) uses the property that each event occurs once in a path, i. e., $\mathsf{F}e \rightarrow \neg \mathsf{P}e$.

Reporting every co-occurrence of two tasks as a pattern would result in many patterns that are not representative. Thus we only report frequent patterns. The support of a relational pattern P is defined as follows:

$$supp\,(P) := \textit{number of tuples in the } \mathsf{Task}\textit{-Relation fulfilling the Cause condition}$$

We only want to detect strong relational patterns, i. e., causes that are likely if the trigger condition is fulfilled. The trigger depends on the pattern, see Table 4.8. The confidence of a pattern P is:

$$conf\,(P) := \frac{supp\,(P)}{\textit{no. of times the trigger condition holds}}$$

Definition 10. A pattern is *frequent* if its support exceeds the predefined threshold *min sup*. A pattern is *strong* if its confidence exceeds *min conf*.

Our algorithm described in the following will only discover frequent and strong relational patterns in a collection of process models.

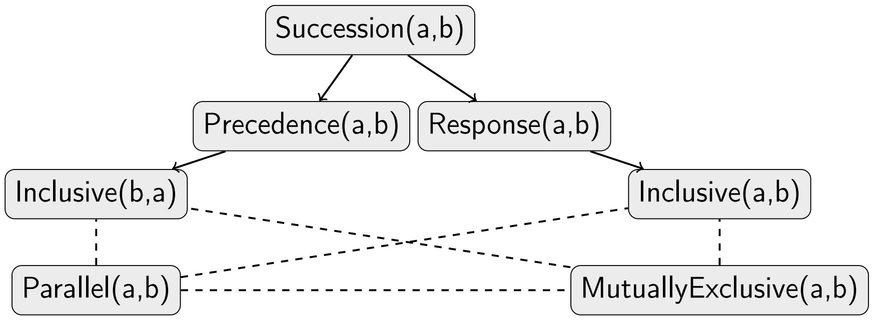


Figure 4.10: The Relationships of Relational Patterns. Directed edges represent implications and dashed edges exclusiveness relationships.

A pattern can have relationships to others, i. e., it can imply or exclude them. For instance, Response(a, b) implies Include(a, b), or Parallel(a, b) excludes MutuallyExclusive(a, b). Figure 4.10 shows some relationships between the relational patterns. These relationships allow improving the search for patterns. For instance, we first search for all Inclusive patterns. We only have to consider these pairs for the Response- and Precedence patterns. Succession$(a, b) \Leftrightarrow$ Precedence$(b, a) \wedge$ Response(a, b). I. e., our approach generates Succession patterns by a postprocessing of the Response and Precedence patterns.

The *GenerateRelationPatterns* Algorithm (Algorithm 3) incorporates the steps described so far: It first generates all pairs of frequent tasks, see Lines 1–2. For every such pair the algorithm checks if they form a MutuallyExclusive, Parallel, or Inclusive-Pattern, see Lines 3–11. For each Inclusive pattern the algorithm checks whether it forms a Response or Precedence-Pattern, see Lines 12–18. If two tasks t_1 and t_2 are in a Response- and Precedence-Pattern, the algorithm adds the Pattern Succession(t_1, t_2) to the set of patterns (Lines 19–23). The check if a candidate pair $c \in \mathcal{C}$ is a pattern is a lookup in the data structure. This takes place in logarithmic time if respective index structures are present. Thus, we observe the following:

Lemma 2 . The *Generate Relation Patterns*-Algorithm is in the complexity class $\mathcal{O}(\ |\mathcal{T}|^2 \cdot log(|D|)\ )$. $\mathcal{T}$ is the set of the tasks and D the data structure containing the relationships.

This complexity is extremely low. The lemma implies with process models of realistic size the focus of an evaluation does not have to be on performance; so that we rather can directly turn our attention to the usefulness of the patterns discovered from an application perspective. Next, the lemma features a very coarse upper bound: This is because we have the pruning condition that a behavior pattern can only be frequent if its tasks are frequent as well, see Line 1).

Relationships between individual tasks do not cover all information needs. To this end, we also search for group pattern GP, i. e., patterns between a set of tasks A and another set of tasks B. While straightforward at first sight, testing all possible group patterns is not possible due to the exponential growth of the number of possible combinations with the number of tasks. Instead, we combine the support and confidence of all patterns containing two tasks. Let GP be a group pattern and $\mathcal{GP}$ the set of 2-task patterns that relate to GP, i. e., that are covered by GP.

Definition 11 . The support and confidence of GP is:

$$supp(GP) = \sum_{\forall P \in \mathcal{GP}} supp(P), \quad conf(GP) = \frac{supp(GP)}{\sum_{\forall P \in \mathcal{GP}} \textit{number of times P triggers}}$$

The min sup of a group is defined as: $\textit{min sup}_{\mathcal{GP}} = |\mathcal{GP}| \cdot \textit{min sup}$

Example 4.3.10 . Let a, b be tasks. a performs the operation op_a ECU$_a$, and op_b on ECU$_b$ respectively. A Pattern Response(a, b) is redundant if the Pattern Response(A, B) is frequent and strong. A, B are the set of tasks corresponding to the same ECU as a and b respectively.

Even if a Pattern P_1 is frequent and strong, another frequent strong pattern P_2 could imply the pattern, rendering P_1 redundant. Thus, a refinement of our algorithm is that it does not return redundant patterns, defined as follows:

Definition 12 . A frequent strong pattern p_1 : Trigger$_1$ $\rightarrow$ Cause$_1$ is redundant if another frequent strong pattern p_2 : Trigger$_2$ $\rightarrow$ Cause$_2$ of the same type exists with Cause$_1$ $\subset$ Cause$_2$.

Example 4.3.11 . Suppose that the frequent strong pattern Response(A, B) exists. B is the set of tasks that operate on a specific ECU. Another frequent strong pattern Response(A, C) exists with C being the tasks that use a specific communication protocol. Every task that operates on the ECU uses the same protocol, i. e., $B \subset C$. Response(A, B) is redundant, and we do not report it.

Algorithm 3 generateRelationPattern(*min sup*, *min conf*): Set⟨ ⟩ Pattern

```
1:  Frequent tasks $\mathcal{T} = \{\ \textit{Task } t \mid \textit{capacity}(t) > \textit{min sup}\ \}$
2:  Candidates $\mathcal{C} = \mathcal{T} \times \mathcal{T}$
3:  for all $(t_1, t_2) \in \mathcal{C}$ do
4:      if Mutually Exclusive$(t_1, t_2)$ is a frequent strong pattern then
5:          Pattern.Add(Mutually Exclusive$(t_1, t_2)$)
6:          $\mathcal{C} \leftarrow \mathcal{C} \setminus (t_2, t_1)$
7:      else if Parallel$(t_1, t_2)$ is a frequent strong pattern then
8:          Pattern.Add(Parallel$(t_1, t_2)$)
9:          $\mathcal{C} \leftarrow \mathcal{C} \setminus (t_2, t_1)$
10:     else if Inclusive$(t_1, t_2)$ is a frequent strong pattern then
11:         Pattern.Add(Inclusive$(t_1, t_2)$)
12:     end if
13: end for
14: for all Inclusive$(t_1, t_2) \in$ Pattern do
15:     if Response$(t_1, t_2)$ is a frequent strong pattern then then
16:         Pattern.Add(Response$(t_1, t_2)$)
17:         Pattern.Remove(Inclusive$(t_1, t_2)$)
18:     else if Precedence$(t_2, t_1)$ is a frequent strong pattern then
19:         Pattern.Add(Precedence$(t_2, t_1)$)
20:         Pattern.Remove(Inclusive$(t_2, t_1)$)
21:     end if
22: end for
23: for all Response$(t_1, t_2) \in$ Pattern do
24:     if Precedence$(t_2, t_1) \in$ Pattern then
25:         Pattern.Add(Succession$(t_1, t_2)$)
26:         Pattern.Remove(Response$(t_1, t_2)$)
27:         Pattern.Remove(Precedence$(t_2, t_1)$)
28:     end if
29: end for
30: return Pattern
```

4.4 Evaluation

For the evaluation of the detection of property patterns we have evaluated our approach with commissioning processes of the AUDI AG. We want to find out whether our approach does help to detect properties in a collection of real process models. To this end, we let a domain expert validate the patterns found. For the evaluation of the property generation from patterns we refer to the evaluation of our verification approach, see Chapter 5.4, and for our automatic process generation, see Chapter 6.4.

Types of Patterns

Our industry partner currently has several hundred commissioning process models. A typical size is around 200 task nodes, with, say, 50 different tasks. While behavior patterns that are frequent exists in this entire set, and our algorithm has found them without any problem instantaneously, they are only moderately interesting. This is because the respective properties are generic in nature and are known to the industry partner. I.e., verification has not led to any improvement. However, models can be grouped by context, e. g., same vehicle project and locality of production. The number of models per group is small, typically 6 to 8. But discovering behavior patterns in such groups that not manifesting themselves as properties in all models of the group is important for verification purposes as well. Even more, if our results on such small sets already are useful from the perspective of domain experts, we can rightfully claim that this holds for larger sets with more complexity as well. So we now report on the results for one such group. We set MIN SUP to 5 and MIN CONF to 0.96 in this specific experiment.

Our approach discovers 79 non-redundant frequent strong patterns, with four different kinds of patterns, see Figure 4.11(a). An intermediate result is that more than 90,000 relationships have been extracted.

Validation of the Patterns Found

We have asked a domain expert to validate the quality of the patterns found. He is a professional process developer of our industrial partner with years of experience in commissioning. We have asked him to classify the patterns as Property, Covered, or Weak. Property is a characteristic that any process model has to fulfill. Covered means that there is a property which is similar. For

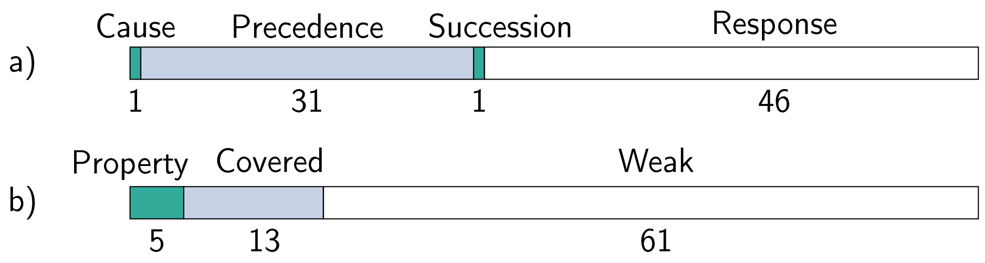


Figure 4.11: The Types of Patterns (a), and the Validation of the Patterns Found (b).

instance, Response(A, B) is Covered if Response(A, C) with $B \subset C$ exists as a property. Weak patterns are characteristics that a process model does not have to adhere to. Our domain expert has marked five patterns as Property, see Figure 4.11(b), 13 patterns have been of category Covered. This shows that our approach does detect real properties by discovering behavior patterns. 72.2% of the behavior patterns in Property and Covered have been of type Response(a, b), 27.8% of type Precedence(a, b).

Summing up, the evaluation shows that our approach does detect properties by discovering behavior patterns (Property and Covered). The Weak patterns are profitable from an application perspective as well. They provide useful information on the structure and the modeling of the processes. For instance, a Weak pattern can help with process synthesis. [MMB14b] shows that synthesis algorithms have to deal with underspecified cases, i. e., more than one process model is possible for a given specification. The Weak pattern can then support the synthesis by providing hints which model to select.

4.5 Related Work for Property Specification

In this Section we discuss the related work for the property specification. In Subsection 4.5.1 we discuss the work related to the instantiation of property patterns. Subsection 4.5.2 shows the related work for mining property candidates.

4.5.1 Related Work for the Instantiation of Properties

Related work includes the user-friendly specification of properties, their management and the property-specific verification of processes.

The direct specification of properties in formalisms like CTL is error-prone and not feasible for a user without experience in formal specification. To deal with this issue, different approaches have been developed. Most business processes are modeled in a graph-based modeling language like BPMN [OMG11], YAWL [AH05] or Petri nets [Aal98]. [Bra+05] extends the BPMN notation with new elements that directly represent LTL operators. BPMN-Q [ADW08] extends BPMN with new edge types that represent sequential ordering between tasks. Compliance Rule Graphs [Ly+11a] allow a specification of requirements in a graph-based formal language. Another approach is the specification of patterns. [DAC98] introduces the property patterns to specify concurrent systems. [Smi+02] extends the pattern system to cover variations of the property patterns (PROPEL, PROPerty ELucidation). [CAC06] uses a question tree to allow specifying PROPEL patterns. In our domain, only a few different property patterns exist. Dependent on the context, many instances are generated. Because of the small number of patterns but many similar instances, we have not found any of the approaches to be helpful in our specific case.

Only a few studies evaluates the usability of a verification system. [BGB14] analyzes the usability of the KeY Program Verification System [BHS07]. KeY is a verification system for a restricted class of Java programs. [BGB14] analyzes the cognitive dimension using a questionnaire. The insides of the questionnaire include the importance of the proof presentation. An issues we also encounter see Section 5.3. [BG12] uses a focus group method in order to evaluate the usability of an interactive theorem proofer.

[ASI12] builds an ontology for compliance management domain. However, it is not sufficient to capture the domain-specific information needed for the instantiation of our patterns. Managing compliance properties includes allocating properties to business processes. [KRL11] allocates the compliance properties to the processes using potentially relevant activities. We in turn dynamically generate only the relevant properties for the commissioning process using the context knowledge directly before verification.

Constraint programming is a programming paradigm that is an alternative to the usual imperative one. With the imperative paradigm, programming means writing a sequence of commands to solve a problem. In constraint programming, one must come up with a set of constraints $\mathcal{C}$ over a set of variables $\mathcal{X}$, in order to describe the problem. A solver then finds an allocation for all $x \in \mathcal{X}$ that comply with all $c \in \mathcal{C}$ [Apt03]. Constraint programming therefore consists of the following two steps: the programming step to write down the constraints $\mathcal{C}$ and the solution step, often automatically. Model checking has two inputs: the transition system M, in our case an imperative process model, and properties

Φ. The constraints $\mathcal{C}$ do not allow to expressing the temporal aspects of our properties. In principle, it would be possible to transform the process model into a set of constraints and to find a solution. This approach is often used for LTL model checking, called bounded model checking [Bie+03]. In our use case however, this is not practical. This is because of the fact that the set of equations grows exponentially, and finding a solution is in PSPACE, which has been proven formally [LP85].

The goal of our work is to check if an imperative process model complies with declarative properties. This stands in contrast to the declarative workflow paradigm, at which the process model solely consists of a set of declarative properties. Declarative workflows allow for any behavior as long as it fulfills the specification [Mon+10]. A declarative specification is similar to the properties in our context. However, properties in general are not sufficient to describe an executable process model. The enactment of declarative workflows is not trivial [PBA10], and tool support by major vendors is missing. To our knowledge, there does not exist any tool that executes declarative process models comparable to the commissioning of vehicles. Another aspect is that, in contrast to an imperative process model, all possible executions in a declarative process model are not easily comprehensible at specification time, leading to an unexpected and possibly false execution of the process model. In consequence, our approach has been to increase the quality of an imperative process model by verifying it against relevant properties.

4.5.2 Related Work for Property Candidate Detection

In what follows, we first review approaches that find patterns on the structural level, then work on declarative process mining and, lastly, on process similarity.

Smirnov et al. [Smi+12] detect action patterns in a process repository. They use the behavior profile of a process schema in order to mine the patterns. In contrast to our relationships, this profile does not contain the appearance information. This renders the detection of the response and precedence patterns impossible. Leopold et al. [LPM15] extend the work of Smirnov et al. [Smi+12] toward handling the semantic heterogeneity of labels.

[Gre+05] discovers frequent patterns in the execution log of a process model. [Gre+05] proposes two algorithms: w-find for an iterative level-wise exploration and c-find for an exploration by composing connected components. The patterns found are connected graphs. It is not possible to detect our patterns in this way. [Lau+09] analyses co-occurrences of activity patterns in process models. Activity patterns are generic process fragments [TRI09] common in business processes, e. g., notification or approval. [Lau+09] uses a frequent subgraph mining approach. Frequent subgraph mining as is cannot identify our control-flow patterns. [Lau+09] only identified information on the structural level. [RDC14] used a crowd-based approach to mine patterns in a repository of data mash-up process models. [RDC14] has detected interesting process fragments. For our case, a crowd-based approach is not applicable. Many patterns are not local, i. e., relate to different tasks at different positions in the process, hard to detect by an expert or even crowd worker. A manual inspection also is more time- and cost-consuming than our automatic discovery.

The so-called problem of process discovery is an instance of process mining, see [Aal11]. The objective is to rediscover a process model P from a Log $L_P \subseteq \mathcal{L}_P$. Recent approaches use heuristics, [WA03], decomposition [LFA14], or genetic approaches [BDA12]. We in turn look for the properties in a repository of process models, the models themselves are given. More related to our problem is the discovery of declarative process models [Che+09; CMM14], i. e., the model is defined by constraints [PA06]. Our approach would have to meet the challenge of summarizing $\mathcal{L}_P$. In declarative process discovery in turn, L_P is of reasonable size and can be analyzed.

Process similarity means defining a metric between process models [DDM08]. Approaches using label, structural, and behavior similarity have been proposed. [KWW11] approximates $\mathcal{L}_P$ by a set of relationships and defines two process models as similar if their set of relationships is similar. In contrast to our work, [KWW11] does not consider the appearance of relationships, required for most patterns. The 4C Spectrum [Pol+14] describes and classifies the relations, i. e., different variants of co-occurrence, conflict, causality, concurrency, that may summarize the behavior space. They cover a total of over 100 relations existing between two transitions. However, our smaller set of relations given in Section 4.3.1 has been sufficient to detect all of our patterns. Thus we have not needed the computation effort necessary to calculate all relations of Polyvyanyy et al. [Pol+14].

Chapter 5

Process Verification[1]

Verification derives from the latin word *veritas* »truthfulness« and *facere* »create«, and means in computer science testing if a system conforms to the specification. In our case the system is the commissioning process and the specification are the context-sensitives properties, see Chapter 4. Recent research has shown that real-life process models often do not comply with all properties [Men+08; Fah+09a; Men09]. Many techniques for process verification are limited to specific properties, e. g., *soundness* [AAB97; Bar+07]. To support the properties of our scenario a general verification technique is required. Model checking verification is such a general technique [Kar+00]. Model checking answers the question, if the model of a system M satisfies a certain property ϕ, formally If $M \models \phi$. In our case, the system is a process model in form of a process tree and the properties are dependencies regarding to a commissioning process we want to check.

Example 5.0.1 . Due to technical limitations it is only possible to open 10 connections to the communication protocol KWP2000 at the same time. We want to check if a given process model never is able to open more connections than 10.

To unambiguously define when a system violates a property ϕ, it has to be formally specified. Chapter 4 shows the specification of the properties. In this Chapter, we focus on the problem of how one can verify if a given process model as a process tree satisfies a given property Φ. To enable the verification of process model by model-checking techniques we need a way to generate the state space of the process model. The commissioning process notation do not define a direct way to generate the state space, see Section 5.1. To allow the generation we provide an interpretation by a transformation of the commissioning process model into a formal language. We choose Petri nets (*PN*) as the formal language. See Subsection 5.1.1 for the reasoning. We refer to

[1]Parts of this Chapter have been Published in [MMB14a] and in [Mra+15].

the transformation of a Commissioning Process Model M to a Petri net PN as co2pn : $M \rightarrow PN$, see Step 2 in Figure 5.1. The next step is to verify a temporal formula ϕ on the Petri net. I. e., we want to know if PN is a model of ϕ. The notation is $PN \models \phi$. Let $\text{verify}(PN, \phi)$ be an operator that checks if $PN \models \phi$, then the verification is defined as:

$$\text{verify}(\text{co2pn}(M), \phi)$$

The verification itself consists of two steps:

1. Construct the state space of the Petri net

2. Search in the state space for a state that is incorrect

One challenge is that the size of the state space grows exponentially with the size of the Petri net, particular in the case of highly parallel processes common for commissioning processes. This problem is called state space explosion, see [Pel09]. A lot of work has been done to optimize the creation of a state space, see [Wol07; Cla+01] for an overview. Usually, information on the structure of process model languages gets lost when transforming the process to a formal representation and cannot be used to optimize the creation of the state space. An example is the information on which tasks of the process relate to which organization unit. The transformation loses such information, and a requirement that only holds for a specific organizational unit cannot be verified any more. In contrast to other work, like [DAV05], [ADW08], and [TAS09], we want to use such information to optimize the transformation otx2pn, in order to reduce the size of the formal representation of the process. The subsequent reduction of the state space with existing techniques for Petri nets is orthogonal to our approach. In Section 5.4.2 we show that, in practice, it is advantageous to use such techniques in addition.

Another issue we target is how to report the verification result to the user. A simple message is not sufficient for the user to understand the cause of the violation. To this end, we analyzed the counter example (a sequence of transitions fired leading to a violating state) and used a pattern based reporting approach to highlight the relevant aspects in the process model.

Subsection 5.1 shows our transformation of an OTX Process model to a Petri net. Subsection 5.2 gives our reduction of the state space to allow an efficient model checking for highly parallel process models. Subsection 5.3 shows how we report the found violations to the user.

5.1 Process Transformation[2]

To allow an automatic verification, e. g., model checking, the process representation has to allow searching its state space. Unfortunately, there is no implementation that analyzes the state space for the proprietary notation for commissioning processes that we want to verify. At the moment, Audi AG uses two concurrent proprietary Diagnostic Systems for commissioning vehicles: Sidis Pro by Siemens AG and Prodis.Automation by DSA, see Chapter 2 for details. Each of these systems uses its own WfMS, its own process notation, and terminals to communicate with the workers. To simplify the maintenance of the process model repositories, the new ISO-standard OTX should standardize the process models. Therefore, we want to support three notations Sidis Pro, Prodis.Automation and OTX. To this end, we have specified and implemented a transformation of Sidis Pro and Prodis.Automation to OTX containing the structural properties to verify, see Step 1 in Figure 5.1. It is not possible to generate the state space for analyzing efficiently directly from an OTX process. A common approach is to transform the process model in a formal language, cf. [RWM10; LVD09], allowing the state space generation. To this end, we transform the OTX process to a Petri net beforehand, see Step 2 in Figure 5.1. [Sch99; Schoob] show efficient ways to generate the state space of a Petri net for different applications, e. g., model checking. We use the LoLA-Framework for producing the state space in the form of a graph, see Step 3 in Figure 5.1.

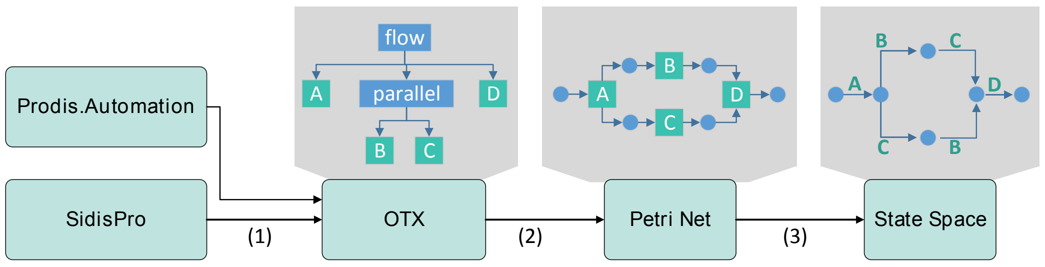


Figure 5.1: The Transformation Steps for the Verification of a Process Model

5.1.1 Formal Language

Formal verification techniques require an explicit representation of the execution semantics. Unfortunately, the ISO standard of OTX does not include a formal

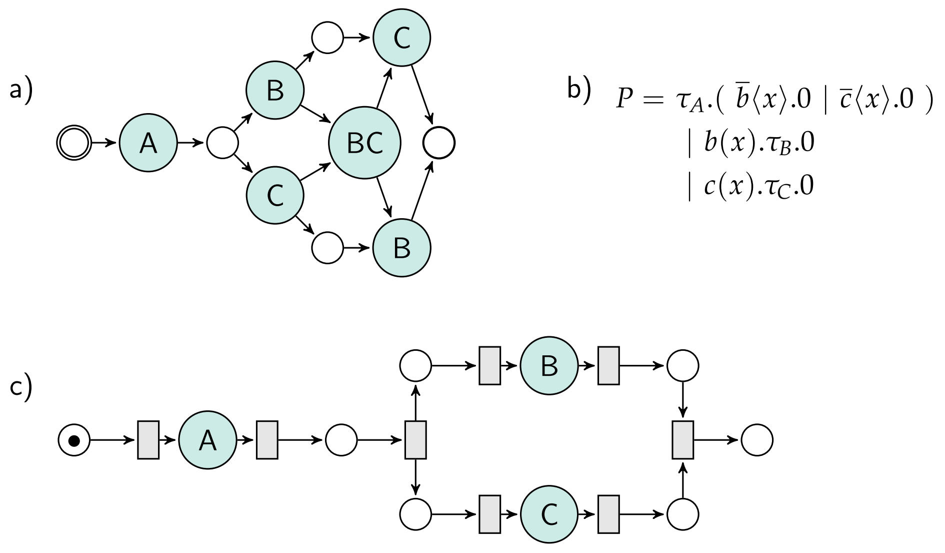
[2]An early version of this chapter was published in [Mra+15]

description of this. To this end, we define an interpretation of the core elements of the OTX notation ourselves. We do so by specifying a transformation of an OTX process model to a formal language that allows analyzing the state space directly. This formal language has to fulfill several core requirements. First, analysis tools should be available for the verification. Second, all the core elements of OTX should have a representation in that language. Third, the formal language should be as close as possible to the OTX processes, to allow a mapping of the violations found to OTX constructs. Three classes of formal languages seem possible, Kripke structures, π-calculus, and Petri nets.

Kripke structures are an extension of a transition system. Kripke structures are connected, directed graphs (S, S_0, R, L). Each node $s \in S$ represents a state of the system, with S_0 being the set of start states. Each edge $r \in R$ represents a transition from a state to another one. A labeling function $L : S \mapsto 2^{AP}$ maps each state s to a set of atomic propositions *true* in s [CGP99]. A plethora of tools exist for model checking Kripke structures, e. g., the SPIN framework, cf. [Hol97]. However, commissioning processes contain a lot of parallel structures. It is not possible to represent these concurrent structures directly as Kripke structures. It would be possible to list all each combinations of executions as a state. But this would lead to a Kripke structure too complex to handle, i. e., for the average commissioning process approximately 10^{60} states. In contrast to a Kripke structure, the π-calculus is able to directly express concurrent executions. Model checking tools exist for the π-calculus, e. g., the PRISM model checker, cf. [KNP02]. The π-calculus is based on a textual, i. e., rather a linear, description, cf. [Aal03]. Most process notations either are graph-based or block-structured. The mapping of a graph-based model to a textual one is not trivial. There does not exist a clear mapping of the original process to elements in the π-calculus. Thus, the reporting of violations found will be hard to understand. Finally, Petri nets are based on bipartite graphs, see Section 5.1.2. They allow a direct representation of concurrent systems. Tool support exists for model checking, e. g., the LoLA-Framework cf. [Sch00a]. Petri nets are structurally similar to most process notations and allow a simple mapping of subnets to constructs of other notations.

Example 5.1.1 . Figure 5.2 shows a process in the three notations. The process consists of three tasks (A, B, C). First, Task A is executed, followed by a parallel execution of B and C. Figure 5.2(a) shows the process as a Kripke structure. The labels in the states describe the execution of tasks. Figure 5.2(b) shows the process in π-calculus. The process model consists of three sub models in parallel. Only the first one can execute initially, because

$b(x)$ and $c(x)$ require a signal on the channel b or c respectively. The first sub model executes A by τ_A and then writes a signal on the channels b and c. This allows the other sub models to be active, either B or C. Figure 5.2(c) shows the process as a Petri net. For each tasks a place exist representing the execution of that tasks. Transitions before the task states model the start and ending of the tasks. Many authors, e. g., [Aal98] model the tasks as a single transition. This modeling does not allow to argue about the parallel execution of tasks.



$$\begin{aligned} P = {} & \tau_A.(\ \bar{b}\langle x\rangle.0 \mid \bar{c}\langle x\rangle.0\ ) \\ & \mid b(x).\tau_B.0 \\ & \mid c(x).\tau_C.0 \end{aligned}$$

Figure 5.2: A Process as a Kripke-Structure, in the π-Calculus, and as a Petri Net

We decided to use Petri nets as our formal representation. Tool support exists. It allows representing concurrency directly and is graph-based. So a direct mapping of constructs is possible. Table 5.1 shows a summary of the core requirements for Kripke-structures, π-calculus, and Petri nets. We do not see any problems when using our approach with other formal languages if a respective transformation of OTX to the formal language is given.

5.1.2 Petri Nets

A Petri net is a directed bipartite graph with two types of nodes called places and transitions. It is not allowed to connect two nodes of the same type.

Table 5.1: Core Requirements for the three Formal Languages

	Kripke-structure	π-Calculus	Petri net
Tool Support	$\times$	$\times$	$\times$
Direct Concurrency		$\times$	$\times$
Graph-based	$\times$		$\times$

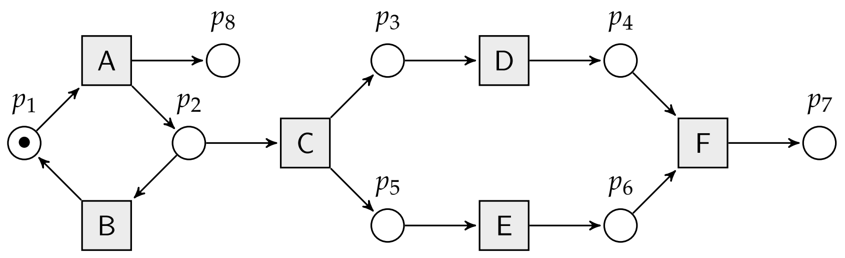


Figure 5.3: The Graphical Representation of a Petri Net (P, T, F, m_0)

Definition 13 (Petri net) .
A Petri net is a tuple (P, T, F, m_0)

› P is a set of places

› T is a set of transitions ($P \cap T = \emptyset$)

› $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs

› A initial distribution of tokens over places $m_0 : P \to \mathbb{N}_0$

$p \in P$ is an input place of $t \in T$ if $(p, t) \in F$ and an output place if $(t, p) \in F$. $\bullet t$ denotes the set of input places of t and $t\bullet$ the set of output places. A mapping $m : P \to \mathbb{N}_0$ maps each $p \in P$ to a positive number of tokens. The distribution of tokens over places represents a state m of the Petri net with m_0 being the initial state. A Petri net is visualized as a graph. Circles represents places $p \in P$, boxes or bars represents transitions $t \in T$, directed edges represents arcs, and markers inside places represent tokens. The state m of the net is the distribution of tokens over places.

Example 5.1.2. Figure 5.3 shows the graphical representation of a Petri net $PN = (P, T, F, m_0)$ with:

› $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$

› $T = \{A, B, C, D, E, F\}$

› $F = \{(p_1, A), (A, p_2), (A, p_8), (p_2, B), (B, p_1), (p_2, C), (C, p_3), (p_3, D), (D, p_4), (p_4, F), (C, p_5), (p_5, E), (E, p_6), (p_6, F), (F, p_7)\}$

› $m_0(p) = \begin{cases} 1 & \text{if } p = p_1 \\ 0 & \text{else} \end{cases}$

A transition $t \in T$ is activated in a state m if $\forall p \in \bullet t : m(p) \geq 1$. A transition $t \in T$ in m can fire, leading to a new state m' with:

$$m'(p) = \begin{cases} m(p) - 1 & \text{if } p \in \bullet t \\ m(p) + 1 & \text{if } p \in t\bullet \\ m(p) & \text{else} \end{cases}$$

For instance, the Petri net in Figure 5.3 in state m_0, have only one active transition, the transition A. The firing of transition A leads to the new state m' by removing the token in p_1 and adding one token both p_8 and p_2.

We denote $m \xrightarrow{t} m'$ that m leads to m' by firing the transition t. m^* denotes the set of states reachable from the state m, i. e., $m' \in m^*$ iff a sequence of transitions $\sigma = t_1, t_2, \ldots, t_n$ exists with

$$m \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} m'$$

Definition 14 (Reachability Graph).
A Petri net (P, T, F, m_0) induce a transition system (S, S_0, A, T') with:

› $S = m^*$ as the set of states

› $s_0 = m_0$ as the initial state

› $A = T$ as the alphabet

› $T' = \{(m, t, m') \in S \times A \times S \mid \exists_{t \in T} m \xrightarrow{t} m'\}$

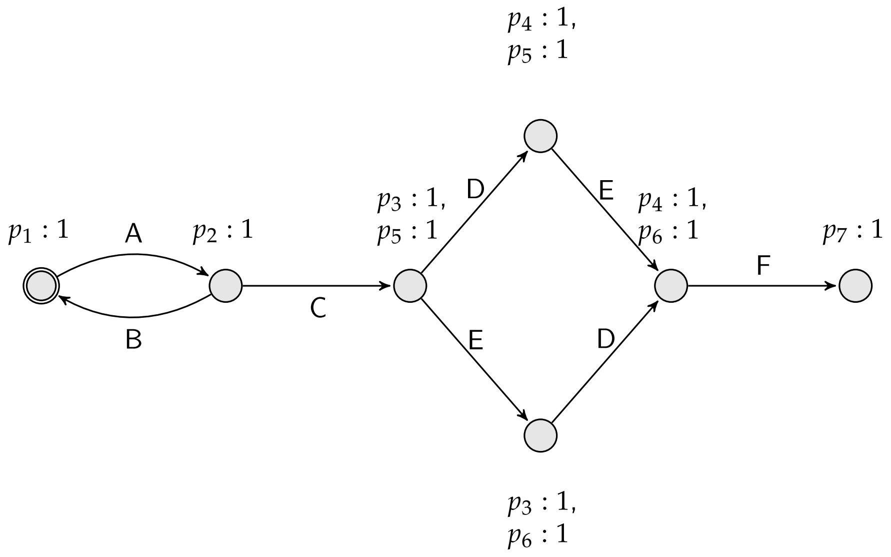


Figure 5.4: The Reachability Graph for the Petri net of Figure 5.3 (without Place p_8).

In general the reachability graph of a finite Petri net is not limited. Consider the net of Figure 5.3, due to the loop between A and B the place p_8 can contain any number of tokens $n \in \mathbb{N}_0$, i. e., m_0^* is not limited. Figure 5.4 shows the reachability graph for the Petri net of Figure 5.3, without Place p_8 and its adjacent edge. The labels over the states show the distribution of tokens, the labels on the edges shows the sequence of transitions fired.

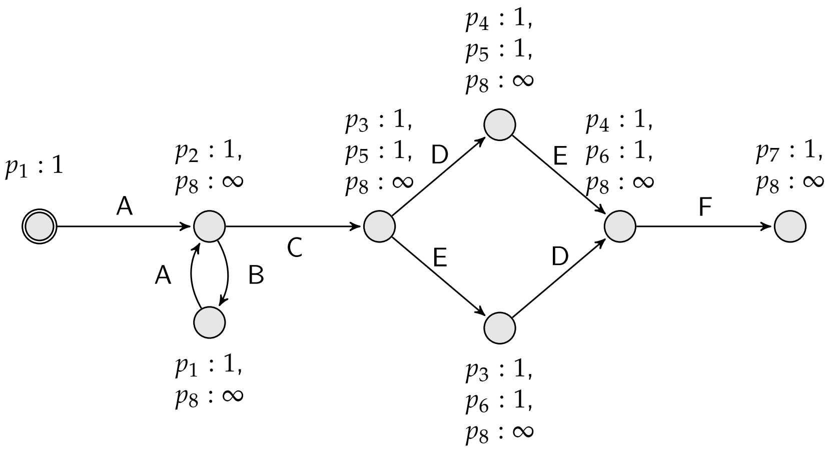
The possible unlimited nature of the reachability graph hinders analysis techniques. The coverability graph of a Petri net does not have these limitations. A coverability set C is the a set of mapping of the a place to a natural number or ∞, i. e., $P \mapsto \mathbb{N}_0 \cup \{\infty\}$ with the following properties:

1. For each $m \in m_0^*$ there is a $m' \in C$ with $m' \in m^*$
2. For each $m \in C \setminus m_0^*$ there is an infinite strictly increasing sequence of markings converging to m'.

A coverability set C is minimal iff no proper subset is a coverability set.

Definition 15 (Coverability Graph).
A Petri net (P, T, F, m_0) induce a transition system (S, S_0, A, T') with:

- $S = C$ be the minimal coverability set
- $S_0 = m_0$ be the initial state
- $A = T$ be the alphabet
- $T' = \{(m, t, m') \in S \times A \times S \mid \exists_{t \in T} m \xrightarrow{t} m'$ or an infinite strictly increasing sequence of markings converging to m'. $\}$



$p_1 : 1$
A
$p_2 : 1,$ $p_8 : \infty$
A
B
$p_1 : 1,$ $p_8 : \infty$
C
$p_3 : 1,$ $p_5 : 1,$ $p_8 : \infty$
D
E
$p_4 : 1,$ $p_5 : 1,$ $p_8 : \infty$
$p_3 : 1,$ $p_6 : 1,$ $p_8 : \infty$
E
D
$p_4 : 1,$ $p_6 : 1,$ $p_8 : \infty$
F
$p_7 : 1,$ $p_8 : \infty$

Figure 5.5: The Coverability Graph for the Petri net of Figure 5.3.

Figure 5.5 shows the Coverability Graph for the Petri net of Figure 5.3. The Coverability Graph for a limited Petri net is also limited.

5.1.3 OTX2PetriNet

OTX is an XML-based process notation for commissioning processes. It allows the definition of structured process models, i. e., an OTX process model can be represented in a notation similar to process trees [LFA13b]. OTX defines a commissioning process as nodes arranged in a tree. The nodes are of two categories: *atomic nodes* (leaf nodes in the tree), and *compound nodes* (inner nodes). *Compound nodes* describe the structural behavior of the process, and *atomic nodes* describe the commissioning tasks. In its core, OTX allows five different types of *compound nodes*: *flow*, *loop*, *branch*, *parallel*, and *handler*.

For each node type of OTX we define at least one Petri net template. A Petri net template is a Petri net subnet with an input place *In* and a certain Output Place *Out*. If a node type allows child nodes, the template contains specific regions for the insertion of the templates of the child nodes. Figure 5.6-5.9 shows the insertion regions as dotted boxes.

Action Node

Action nodes are used to represent atomic operations, i. e., commissioning tasks, in the commissioning process. In the Petri net representation we represent the fact that a task *n* is running as a state $label_n$. As stated earlier, each task communicates with an electronic control unit ECU in the vehicle. We present the ECU with three transitions. *ECU* represents the use of the control unit. *ECUcon* and $\overline{ECUcon}$ represents whether the connection to a certain ECU is open (*ECUcon*) or closed ($\overline{ECUcon}$). To communicate with the ECU the WfMS uses one of two protocols. The place *prot* represents the connection to a protocol (either UDS or KWP2000). The connection to an ECU implicitly opens with the first task that uses the ECU. Specific tasks close the connection. To this end, we define two templates: one for the tasks that close the connection, see Figure 5.6(b), and one for the tasks that can implicitly open the connection, see Figure 5.6(a). At the beginning of the process execution, each place $\overline{ECUcon}$ contains a token, meaning that all connections are closed. Figure 5.6(a) shows the template for the action node that opens a connection. After the *In* place two transitions are possible: *t*_1 and *t*_2. If the connection to the ECU is open, i. e., a token is in *ECUcon*, the *t*_2 can fire and generate a token in the places label$_n$ and ECU. If the connection to the ECU is closed, i. e., a token is in $\overline{ECUcon}$, the *t*_1 is active. This means that it removes the token in $\overline{ECUcon}$ and generates one in the places *ECUcon*, *prot*, $label_n$, and *ECU*. The transition *t*_3 ends the execution of the task, removing the token in $label_n$ and *ECU*. The places *prot*, *ECU*, *ECUcon* and $\overline{ECUcon}$ are shared between tasks. Each task *n* has its own $label_n$, *In*, and *Out* Place.[3]

[3]Other authors, e. g., [Aal98], use a simple model for tasks consisting of a single transition. This model does not allow verifying properties about the parallelization of tasks and the usage of communication protocols.

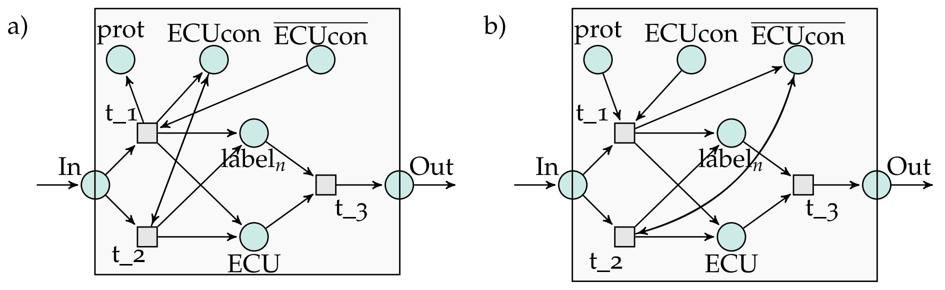


Figure 5.6: Template of the *Action Node* (a) and the *Close Connection Action Node* (b).

Loop Node

The *loop node* is used for a structured repetition of a part of the process until a condition is met. ISO defines the *loop node* as:

> » For repetitive execution of flows, the Loop node shall be utilized. [...] As long as the condition holds, the loop flow is repeated. In OTX loops, the condition can be checked before or after the flow. « [ISO12]

This is equivalent to the structured iteration control flow pattern of [Aal+03b]. We use two templates for the *loop node* depending on whether the condition is checked before or after the execution. Figure 5.7 shows the two templates for the pre-test and the post-test *loop node*. The dotted region is the position where the algorithm inserts the sub net for the unique child node.

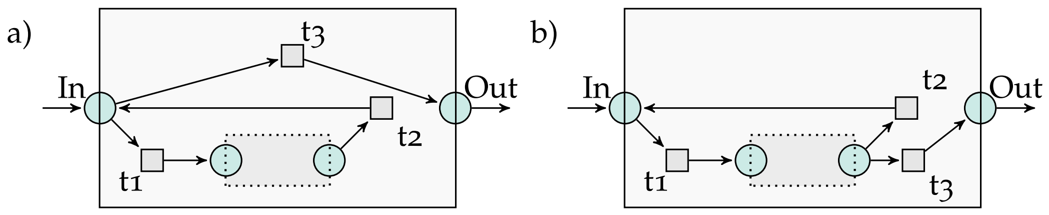


Figure 5.7: Template of the *Loop Node* with Checks Before the Execution (a) and after the Execution (b).

Branch Node

Process designers use the *branch node* to model alternatives in the process, e. g., two or more cases that exclude each other. ISO defines the *branch node* as :

> » A Branch node contains one or more `<case>` elements. Every case includes a boolean `<condition>` together with a `<flow>`. The first case with a true condition is executed. If no condition is true, the `<default>` flow will be executed. « [ISO12]

The *branch node* is equivalent to the exclusive choice control flow pattern of [Aal+03b]. Figure 5.8(a) shows the templates for the branch node. The dotted box is the region for the sub nets corresponding to the child nodes.

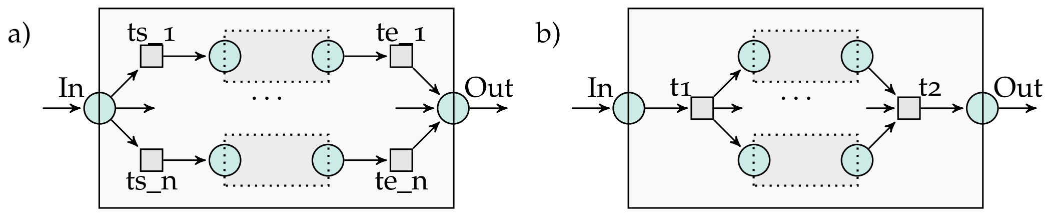


Figure 5.8: Template of the *Branch Node* (a) and the *Parallel Node* (b).

Parallel Node

The *parallel node* is used to model flows that are executed in parallel. ISO defines the *parallel node* as :

> » A parallel node consists of one or more flows that shall be executed at the same time. « [ISO12]

This is equivalent to a combination of the *Parallel Split* and *Join* control flow pattern of [Aal+03b]. Figure 5.8(b) shows the template for the *parallel node*. The dotted region identifies where the sub nets for the child nodes are inserted.

Flow Node

The *flow node* is used for two cases: first, to define the sequential execution of tasks and, second, to define sub processes that can be collapsed:

> » In general, `<flow>` is used for grouping a sequence of nodes together. When a `<flow>` is used stand-alone (not nested in a control structure like loop, branch, etc.), it supports authors to partition the procedure flow into logical blocks for providing clarity through modular sequence design. « [ISO12]

The *flow node* can be used as the *sequence* control flow pattern of [Aal+03b]. Figure 5.9(a) shows the templates for the *flow node*. The dotted region is the place where to insert the sub nets for the child nodes.

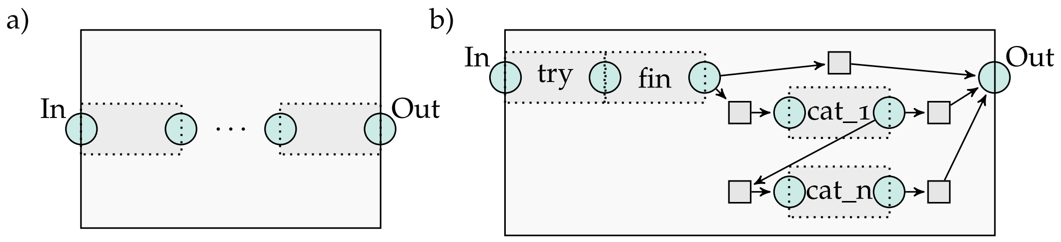


Figure 5.9: Template of the *Flow Node* (a) and the *Handler Node* (b).

Handler Node

The *handler node* contains a *try* subprocess. If an error in the *try* subprocess occurs, one or more *catch* subprocesses handle the error subsequently. An optional *final* subprocess executes after the *try* and before the *catch* subprocess:

> » A handler node contains a so called `<try>` flow followed by an additional `<finally>` flow as well as `<catch>` flows for exception treatment. The `<handler>` node monitors the `<try>` flow for exceptions, [. . .] « [ISO12]

Figure 5.9(b) shows the templates for the *handler node*. The dotted region specifies the sub nets for the *try*, *final* and the *catch* subprocesses.

5.1.4 Algorithm transformNode()

Algorithm 4 transform (OTX node n)

```
1: $t \leftarrow type\ of\ n$
2: $template_t(n)$
3: **if** $n$ is a compound node **then**
4:     **for all** child nodes $c$ of $n$ **do**
5:         $transform(c)$
6:     **end for**
7: **end if**
```

Algorithm 4 transforms an OTX process model into a Petri net representation. To this end, we call the method transformNode() of the root node. First, the algorithm determines the type of the node n. The possible types are *flow*, *branch*, *parallel*, *loop*, *handler*, and *action*, see Line 1. Next, the algorithm adds the places and transitions that have been defined by means of the template of the type, see Line 2. If the node is a compound node, the algorithm calls the function recursively.

5.1.5 Property Instantiation

As explained in Section 4.2, the database of context knowledge stores under-specified CTL-formulas. Our framework uses the context information to generate the instances for the verification. Each under-specified CTL-formula contains place holders that our framework replace during the instantiation with concrete places in the Petri net of the commissioning process. Their exist five types of place holders, see Figure 5.6:

1. A_{run} : Replaced by the place *run* of an individual task.
2. X_{ECU} : Replaced by the place *ecu* of the ECU.
3. KWP2000 or UDS : replaced by the place *prot* for the communication protocol.
4. X_{CON} : The connection place *ECUcon* for the ECU X.
5. END : The *end* place of the process, i. e., the place *Out* of the root element in OTX.

Table 5.2 shows the under-specified CTL-formula for each pattern of Subsection 4.2.

Table 5.2: CTL-Formula for the Property Patterns, Repetition of Table 4.2

Property	Name	CTL
P3.1	Maximal UDS Connections	$\mathsf{AG}\ (\ \mathsf{UDS} \leq 10\ )$
P3.2	Maximal KWP-2000 Connections	$\mathsf{AG}\ (\ \mathsf{KWP2000} \leq 10\ )$
P3.3	Maximal Connections	$\mathsf{AG}\ (\ (\mathsf{UDS} + \mathsf{KWP2000}) \leq 14\ )$
P4.1	Sequential before	$\mathsf{A}\ [\ (A_{run} = 0)\ \ \mathsf{W}\ \ (B_{run} > 0)\ ]$
P4.2	Optional Sequential before	$\mathsf{A}\ [\ (A_{run} = 0)\ \lor\ \mathsf{AG}(\ B_{run} = 0\ )\ )$ $\mathsf{W}\ (\ B_{run} > 0\ )\ ]$
P4.3	Sequential after	$\mathsf{AG}\ (\ (A_{run} > 0)\ \rightarrow\ \mathsf{AF}\ (B_{run} > 0)\ )$
P4.4	Non-Parallel	$\mathsf{AG}\ (\neg(\ (A_{run} > 0)\ \land\ (B_{run} > 0)\ ))$
P5.1	Restricted access	$\mathsf{AG}\ (X_{\text{ECU}} \leq 1)$
P5.2	Non-Parallel	$\mathsf{AG}\ (\ \neg\ (\ (X_{\text{CON}} > 0)\ \land\ (Y_{\text{CON}} > 0)\ )\ )$
P5.3	Close Connection	$\mathsf{AG}\ (\ (\text{END} > 0)\ \rightarrow\ (X_{\text{CON}} = 0)\ )$

Example 5.1.3. The process to be verified contains the *ECUs* = [AWG, CEL, DAU]. For the process place PP2 and the vehicle series M3, an ECU dependency exists that CEL and DAU are not allowed to be used in parallel. For Property Pattern P5.2 one of the properties our framework generates is following:

$$\mathsf{AG}(\ \neg\ (\ (\mathsf{CEL}_{\text{CON}} > 0)\ \land\ (\mathsf{DAU}_{\text{CON}} > 0)\ )\ )$$

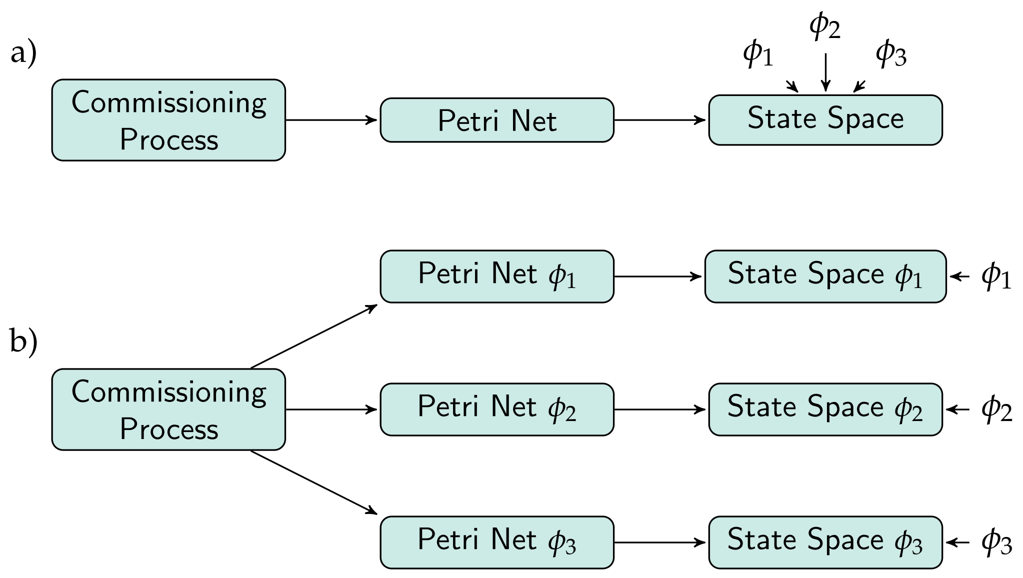


Figure 5.10: (a) The Classic Approach for Process Verification (b) our Approach

5.2 Relevance Optimization[4]

As suggested by theoretically observation, see Subsection 3.1.2, and proven by our own experiments is the state space for realistic commissioning processes too large to generate and thus to verify. Our basic idea is, instead of generating one state space for the complete process and subsequent analyze this one state space for each property $\phi \in \Phi$, we want to generate a smaller state space for each property ϕ, see Figure 5.10. To this end, we have to define which region of the process model is relevant for a property ϕ, see Subsection 5.2.1. Subsection 5.2.2 explains the general algorithm. Subsection 5.2.3 shows how we use the relevance function to find the set of relevant tasks. Subsection 5.2.4 explains how we prune the irrelevant parts of the process for a property. Subsection 5.2.5 shows the extension of the transformation algorithm and Subsection 5.2.7 discusses the limitations.

5.2.1 Relevance Function

To improve the performance of the verification, we seek a transformation that creates a smaller Petri net PN_c that is a model to ϕ iff PN is a model to ϕ, i. e.,

[4]This chapter is published in [MMB14a]

$PN_c \models \phi \Leftrightarrow PN \models \phi$. To this end, we want to exploit the hierarchical structure of the process tree. We only want to explore the regions of the process that are relevant for the verification.

Definition 16 (Abstract Relevance Function).

Let *relevant* be a function for a temporal formula $\phi \in \Phi$ and $n \in N$

$$relevant : \Phi \times N \rightarrow \{\text{true}, \text{false}\}$$

$$relevant\ (\phi, n) = \begin{cases} \text{true} & \text{if } n \text{ is relevant for the verification} \\ & \text{of property } \phi \\ \text{false} & \text{otherwise} \end{cases}$$

An inner node n of the process tree is relevant if at least one of its child nodes is relevant. We define T_ϕ as the set of tasks relevant for a property. Formally describe:

$$T_\phi := \{\ t \mid t \text{ is a task node} \land relevant(\phi, t) = true\ \}$$

The concrete definition of *relevant* depends on the application domain and on the execution model of the process. Our algorithm works with any definition of relevance that features this abstract structure. For our use case, the commissioning of vehicles, we provide two definitions of the relevance function, one for task-related properties, see P3.1–P4.4 in Section 4.2, and one for ECU-related properties, see P5.1–5.3 in Section 4.2. At first we explain the execution and resource model for a task n in our main use case.

Remark that Figure 5.6 shows the Petri net template for a task node. With these definitions, we are now able to define the relevance functions for our domain of commissioning processes. Our approach considers properties that focus on sequential and parallel ordering and on objects a certain task requires, c. f. Section 4.2. As consequence, we define two relevance functions, one for each of the categories of properties identified in our domain.

Definition 17 (Task-relevance Function).

Let n be a task n and ϕ a temporal formula that argues about the occurrence or ordering of tasks. L_ϕ is the set of labels referred to in ϕ.

$$relevant\ (\phi, n) = \begin{cases} \text{true} & \text{if } \exists\, l \in L_\phi\ :\ label_n = l \\ \text{false} & \text{otherwise} \end{cases}$$

The templates P3.1–P4.4 use this relevance function. We want to include n if the label of n is identical to one of the labels in L_ϕ. Further, it is possible to consider labels of n when they are similar enough to a label in L_ϕ by replacing $label_n = l$ with $\mathrm{sim}(label_n, l) \geq \gamma$, $\gamma \in (0,1]$. γ is a predefined threshold for the similarity and *sim* a similarity function. A simple similarity function is the Levenshtein distance, see [Lev66]. In the field of label matching, better metrics have been proposed recently with higher recall, cf. [Kli+13] and using semantic information, cf. [EKO07]. The rationale is that the labels of the tasks in real processes often differ slightly. Therefore, the tasks mentioned in the properties often do not perfectly match the labels in the processes. For instance, a property could state that each task *payment* leads to a task *confirm the payment*. In the processes the task label may be *confirmation of the payment*. Thus, we want a task to be relevant if the similarity exceeds a domain-specific threshold γ. In our concrete use case, company restrictions limit the labels to a predefined set.

Definition 18 (ECU-relevance Function) .

Let n be a task and ϕ a temporal formula that argues about ECUs. R_ϕ is the set of ECUs referred to in ϕ.

$$relevant\ (\phi, n) = \begin{cases} \text{true} & \text{if } n \text{ communicates to one of the ECUs in } R_\phi \\ \text{false} & \text{otherwise} \end{cases}$$

The templates P5.1–P5.3 use this relevance function. We have to consider a task n if it has access to one of the ECUs in R. An example for applying the ECU-relevance function is the following.

Example 5.2.1 . An ECU can only be accessed by one task at a time. In CTL this is $\mathsf{AG}(r \leq 1)$. Each task n with $r \in require(n)$ is considered relevant as well as all inner nodes containing such a task as one of its descendant nodes, see Definition 16.

Other important classes of properties covered by our approach are the response and cardinality constraints, cf. [Mon+10]. Also see the Example 5.2.2.

Example 5.2.2. Think of the following properties: Each task A leads to a task B, cf. Property R3 in Section 4.2. A occurs never or once, while B needs to occur once or more. The first part of this property is a response constraint, the second one a cardinality constraint. In CTL this can be formulated as follows:

$$\mathsf{AG}((A) \rightarrow \mathsf{AF}(B)) \wedge \neg\mathsf{AG}((A) \wedge \mathsf{AF}(A)) \wedge \mathsf{AG}(B)$$

with A representing all states of the Petri net with at least one token in the *run*-place of the task nodes for A, cf. Figure 5.6.

If a property ϕ argues about both ECUs as well as task ordering, it is possible to combine these relevance functions. A node n is relevant when it is relevant for either the task-relevance function or the ECU-relevance function. In Subsection 5.2.4 Lemma 3 states that our relevance functions preserve the properties for all CTL formulas except for those with a Next-Operator. For some properties, deciding which tasks are important is not trivial. To illustrate, dependencies that cannot be determined a priori may lead to a large set of tasks, because many tasks cannot be excluded and must be seen as relevant. The algorithm still works, but the reduction is less efficient.

Example 5.2.3. For instance the two properties »b leads to a« as CTL: $AG(b \rightarrow AF(a))$ and »b occurs 0 or 1 time« $AG(a \rightarrow AX(AG(\neg a)))$ in combination lead to a hidden dependency: The occurrence of a leads to the absence of b in the subsequent paths.

5.2.2 Algorithm

We want to use the information on the relevant tasks from ϕ to arrive at a smaller Petri net. We call the optimized transformation $\text{OTX2PN}_c(OTX, \phi)$, and the verification changes to:

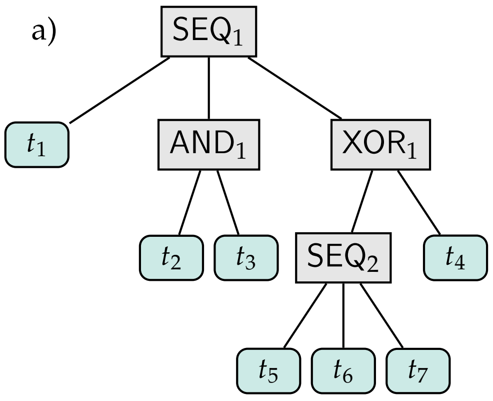
$$\text{verify}(\text{OTX2PN}_c(OTX, \phi), \phi)$$

To complete this overview, we present our approach in pseudo-code. Our algorithm receives as input a process model P in the form of a process tree and a set Φ of temporal formulas as input. $\text{prune}(P, T_\phi)$ is the actual reduction algorithm. T_ϕ is the set of tasks that are relevant to property ϕ.

Algorithm 5 constrain(P, Φ)

```
1: Constructing for each node $n$ its set of tasks $lt_n$
2: for all $\phi \in \Phi$ do
3:     $P_\phi \leftarrow$ prune($P$, $T_\phi$)
4:     $PN \leftarrow$ OTX2PN($P_\phi$)
5:     verify($PN$, $\phi$)
6: end for
```

a)



b)

Node		Set of Tasks
SEQ$_1$	:	t_1, t_2, t_3, t_4, t_5, t_6, t_7
AND$_1$	:	t_2, t_3
XOR$_1$	:	t_4, t_5, t_6, t_7
SEQ$_2$	:	t_5, t_6, t_7

Figure 5.11: The Process Tree of a Simple Process (a) and the Set of the Tasks of its Inner Nodes (b)

Section 5.2.3 describes the construction of the set of tasks for each node in the tree. Section 5.2.4 explains prune(P, T_ϕ) in detail. Section 5.2.5 gives the changes to the transformation of Section 5.1, and the verification itself is subject of Section 5.2.6.

5.2.3 Constructing the Set of Tasks

At first we parse recursively the tree, and for each subtree s in the process tree we build the set of tasks belonging to s. Figure 5.11(b) shows the sets for each inner node of the process tree from Figure 5.11(a). Let n be a node. Then lt_n is the set of tasks in the subtree with the root node n. In the pruning step we need to check relevant(ϕ, n). To do so, we need the set lt_n. To speed up subsequent steps, we build a HashMap that maps each node n to lt_n.

5.2.4 Prune(P, ϕ)

In this step we reduce the process tree in order to speed up the verification task later on. Starting point are the property ϕ and the root node $n_0 \in P$, see Line 2 of Algorithm 6. We check if the node is relevant for ϕ, see Line 2 of Algorithm 7. If relevant(ϕ, n) = true, we parse each child n_{c1}, n_{c2}, ..., n_{cn} of n_0 and check recursively if n_{ci} is relevant, see Line 4 of Algorithm 7. If relevant(ϕ, n) = false we prune this subtree. The XOR-node needs a different treatment, as we explain later in this subsection. If the type of the parent is not XOR, we delete the node, see Line 10 of Algorithm 7. Otherwise, we replace it with an empty node λ, see Line 8 of Algorithm 7. Next, we check if an inner node n is left with only one child node, see Line 2 of Algorithm 8. If so, we delete n and connect the parent node of n with its child node, see Lines 3 and 4 of Algorithm 8.

Algorithm 6 prune(P, ϕ)

```
1: procedure PRUNE($P, \phi$)
2:     $n_0 \leftarrow$ get root node of $P$
3:     pruneNode($n_0, \phi$)
4:     trimTree($n_0, \phi$)
5: end procedure
```

Algorithm 7 pruneNode(n, ϕ)

```
1: procedure PRUNENODE($n, \phi$)
2:     if relevant($\phi, n$) then
3:         for all children $n'$ of $n$ do
4:             pruneNode($n', \phi$)
5:         end for
6:     else
7:         if type of parent of $n$ = XOR then
8:             replace $n$ with $\lambda$
9:         else
10:            delete $n$ from the process tree
11:        end if
12:    end if
13: end procedure
```

Algorithm 8 trimTree(n, ϕ)

```
1: procedure TRIMTREE($n, \phi$)
2:     if $|n.childs| = 1$ then
3:         connect the children of $n$ to the parent of $n$
4:         delete $n$
5:         trimTree(parent of $n, \phi$)
6:     else
7:         for all children $n'$ of $n$ do
8:             trimTree($n', \phi$)
9:         end for
10:    end if
11: end procedure
```

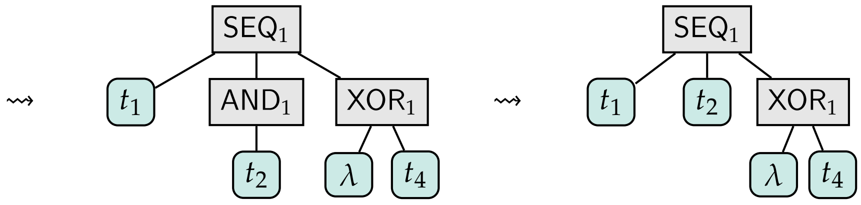


Figure 5.12: First and Second Pruning Step

Example 5.2.4. Let the process tree of Figure 5.11 be given, and let the set of tasks that influence ϕ be $T_\phi = \{t_1, t_2, t_4\}$. First, we delete the nodes with $l_n \cap T_\phi = \emptyset$. This results in the left tree of Figure 5.12. Then, we delete the nodes with only one child node. See the right-hand side of Figure 5.12.

In order to preserve the result of the verification, the empty node for the XOR-node must remain. Take as example the temporal logic formula »After an occurrence of t_1 there always is an occurrence of t_4 at some time in the future«. In CTL this is $\mathsf{AG}(t_1 \rightarrow \mathsf{AF}(t_4))$. This evaluates *false* for the process in Figure 5.11. But it would be *true* if we had removed the other child of the XOR-node.

Lemma 3. Our approach is correct for CTL without the Next-Operator X (see Section 5.2.7).

Proof 1 (of Lemma 3). Let S be the set of states and $\mathsf{EG}(\phi)$ the CTL formula, i. e., there exists a path $p = (s_1, s_2, \dots, s_n)$ where ϕ is always true. As a result of our reduction, the paths are limited to the states that are able to influence ϕ, called S_ϕ, $p' = (s'_1, s'_2, \dots, s'_n)$, $s'_i \in S_\phi \subseteq S$. If a path p exists in the original Petri net, a path p' exists in the reduced Petri net, too. If no path p exists in the original Petri net, no path p' can exist in the reduced Petri net. The same reasoning applies for the formula $\mathsf{E}[\phi \ \mathsf{U} \ \psi]$ and for $(\mathsf{EF}\phi, \mathsf{AF}\phi, \mathsf{AG}\phi, \mathsf{A}[\phi \ \mathsf{U} \ \psi])$, by use of equivalences.

Often the Next-Operator is used to argue that the formula ϕ is *true* in the following states, however *false* in the current, i. e., $\mathsf{AX}\,\mathsf{AG}\,\phi$, for instance in [TAS09]. It is possible to use our algorithm in those cases. Only if the actual distance of states is relevant our algorithm cannot be applied, e. g., three states in the future ϕ is *true* $\mathsf{AX}\,\mathsf{AX}\,\mathsf{AX}\,\phi$.

5.2.5 hl2pn(P_ϕ)

After the pruning step we use the process tree to build a Petri net. We start with the root node and transform it to a Petri net. We recursively parse the tree and transform each node. For each control structure we define a pattern that describes its execution semantics as a Petri net, see Section 5.1. Figure 5.13 shows the pattern for an xor-node. The locations where the child nodes should be inserted into a pattern are labeled with »Child 1« and »Child 2«. If one of the children of an xor-node is λ, the respective branch is reduced to a simple transition. Let n be an xor-node with two child nodes n_1 and n_2. If $n_1 \neq \lambda$ and $n_2 \neq \lambda$, then the pattern of Figure 5.13 is included into the Petri net. If $n_1 \neq \lambda$ and $n_2 = \lambda$, the second branch can be reduced to a simple transition, see Figure 5.14. If $n_1 = n_2 = \lambda$, then the complete xor-node is deleted.

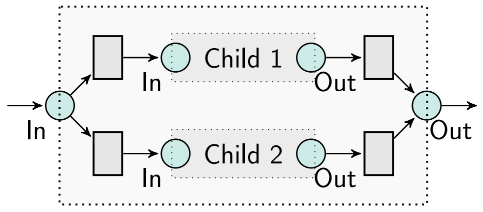


Figure 5.13: The Pattern for an xor-Node

5.2.6 Verify(PN, ϕ)

After the transformation, we generate the state space of the Petri net. We, for our part, use the LoLA-Toolkit, cf. [Schooa] for this task. Then a model-

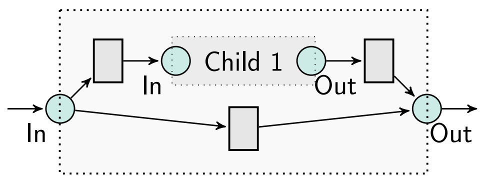


Figure 5.14: The Pattern for an XOR-Node with Empty Second Child Element

checking algorithm verifies if the Petri net satisfies the temporal logic formula, i.e., $PN_c \models \phi$. Again, we use the CTL model checker of the LoLA-Framework, which is an implementation of the ALMC-algorithm, see [VL93].

5.2.7 Limitations

The reduction just proposed does not change the sequence of the tasks, but it may change the distance of tasks in the traces of the executions of a process schema. For example, trace $a \rightarrow b \rightarrow c$ in the original process P can be reduced to trace $a \rightarrow c$ by pruning b. The distance between a and c changes from two to one. The CTL formula $\mathsf{AG}(a \rightarrow \mathsf{EX}\ c)$ evaluates to *false* for the original trace, but not for the reduced trace. Thus, our specification language is limited to argue only about the ordering of tasks and not about their exact distance. However, this is not as severe as it may sound at first sight: Namely, the state distance may be non-intuitive. This is because how many states are between the tasks depends on the Petri net model transformation. A solution would be extending the notion of explicit time to tasks, see [Mon+08; GL06]. But this would require a different formal model, e. g., timed Petri nets, and a different specification language, e. g., RTCTL. In the domains studied here, such explicit time constraints do not play a role, and our approach does not support them. Therefore, we do not allow the Next Operator (X) of CTL to be used to argue about the distance of states in the property. It is possible to use the Next Operator (X) to argue that the property should hold in the following states but not in the current. The other operators of CTL can be used without limitation.

Our approach is efficient for requirements in which the set of relevant tasks can be determined a priori, see Section 5.4.2. As stated earlier in Subsection 5.2.1 and in Example 5.2.3, this could be an issue with hidden dependencies. In particular our approach is efficient for ordering constraints on tasks, cardinality constraints, negation constraints, i. e., a task excludes another task. All requirements found in Subsection 4.2 fall into these categories. Moreover, the majority of specification languages for business processes are limited to these types of requirements, see

[ADW08; Ly+11a; AP06]. As we showed in the evaluation, cf. Subsection 5.4.2, the runtime of our optimization algorithm is negligible compared to the runtime of the verification itself. Thus, even in the few cases when our optimization does not bring a significant gain it causes only little effort.

5.2.8 Using the Algorithm for Graph-based Processes

Until now we described how our relevance optimization algorithm can improve the verification for a process model as a process tree. In the case that the process model is in a graph-based notation like BPMN we want to give an outlook about possible application. [Pol12] shows an approach in order to find a structured process model that is the bi-simulation of an unstructured one. This approach consists of calculating the causal net of the unstructured process model, generating an ordering relationship graph (ORG) from the causal net, as well as performing a modular decomposition. If the decomposition contains no prime element than the modular decomposition can be transformed into a bi-simulation process tree. If the decomposition contains a prime element it is not possible to find a structured process model. Chapter 6 discusses the details about the ordering relationship graph and modular decomposition.

We can leverage the work of [Pol12] to apply our relevance optimization algorithm to unstructured process model. See Algorithm 9. Our pruning algorithm handles prime elements like an XOR-Split.

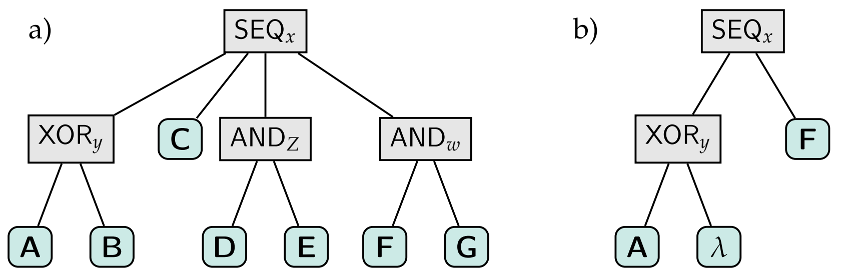
Algorithm 9 constrainUnstructured(P, Φ)

```
1: $causal \leftarrow$ Calculate Causal net of $P$
2: $ORG \leftarrow$ Calculate Ordering Relation Graph using $causal$
3: $MDT \leftarrow$ Perform Modular Decomposition of the $ORG$
4: constrain($MDT$, $\Phi$)
```

5.3 Violation Reporting[5]

The LoLA-Framework returns a sequence of transitions fired leading to a violating state in the case of a property violation. Our goal is to highlight the important aspects in a graphical interface, supporting the user to understand

[5]This chapter was published in [Mra+15]



<**y_s2**, b_s, b_e, **y_e2**, c_s, c_e, z_s, d_s, e_s, e_e, d_e, z_e, w_s, g_s, **f_s** >

<y_s2, y_e2, f_s >

Figure 5.15: The Reduction of the Counter Example

the property violation. Reporting all sequences of transitions fired that lead to a state violating a property is not practical. In general, only some of the transitions are important for the property violation and should be visualized. To detect which elements of the counter examples our framework should mark bears two major challenges: First, the complete counter examples are not a very efficient means to report the errors to the end user. The sequence often contains a large number of transitions, and most of them are unimportant for the specific property. Second, the important information depends on the property pattern we are analyzing.

To address these issues, we propose a two-step approach. First, we detect the relevant entries in the counter example, see Subsection 5.3.1. Second, we collect the important information in the remaining path for each pattern and report it to the user, see Subsection 5.3.2. Subsection 5.3.4 gives an example for a commissioning process and the task highlighted.

5.3.1 Reduction of the Counter Example

In Section 5.2 we show a reduction of the process tree, which is good from a performance perspective. The approach reduces the OTX process tree to the regions of relevance for each property. This reduction not only allows us to efficiently verify the properties, it also significantly reduces the size of the counter example.

Example 5.3.1. Consider the OTX process tree in Figure 5.15(a) and the property »A precedence F«. Under the process tree there is one counter example. The events refer to the transitions of the branch y, the parallel node z, w, and the tasks b–f. Observe that most of the events reported in the example are not important for the property, e. g., the fact that the execution of C, D, E or G has started. The cause for the property violation, i. e., the execution of the second branch of y, is hard to perceive, due to these unimportant events listed. Figure 5.15(b) shows the process tree reduced with the algorithm described in Section 5.2. This reduction leads to a much smaller counter example, which contains only events directly related to the property. Only three events are sufficient to show the cause of the violation.

5.3.2 Property Pattern Reports

Our goal is to report the property violations to the end user in a user-friendly manner. We want to report by means of a visualization of the process where the modeling error has occurred. This information depends on the property. For instance, for the *Response*-Pattern, see Section 4.2, the information of the sequential arrangement is important, in contrast to the maximal connection pattern that requires the parallel arrangement. To this end, we propose a specific reporting for each pattern. More specifically, we will describe how we extract the information from the counter example and detect which elements our visualization should highlighted. Elements are nodes in the graph, i. e., control nodes or tasks. We will propose algorithms for each pattern. Their input is the counter example, i. e., a sequence of transitions fired that leads to a violating state. We call this sequence the log L. The algorithm returns a set of elements to be highlighted in the visualization. The transitions in the counter example can belong to tasks or to control structures, see the templates in Subsection 5.1.3.

Maximal Connection P3.1-P3.3 Algorithm 10 for the maximal connection patterns P3.1-P3.3 returns a set of tasks which we highlight in the visualization to report the error to the user. For the pattern Maximal Connection P3.1-P3.3 we are only interested in transitions belonging to tasks, see Line 1 in Algorithm 10. Line 2 reduces the event log to the tasks which open or close a connection to the respective protocol, i. e., UDS for P3.1, KWP2000 for P3.2, both for P3.3. Figure 5.6 shows that for each task only transition t_1 opens or closes a connection. Therefore the result consists of tasks that execute transition t_1. Next, we check which connections are open for the respective counter state. In other words, which transition »Open a Connection to the ECU X« is not followed by a transition »Close a Connection to the ECU X«, see Line 3 in Algorithm 10. Last, we return all tasks that have a starting event in L, see Line 4 in Algorithm 10. These tasks are then highlighted in the visualization for the user.

Algorithm 10 report-P3 (event log L)

```
1: $L \leftarrow \{e \mid e \in L \wedge e$ is a start transition for a task$\}$
2: $L \leftarrow \{e \mid e \in L \wedge e$ opens or closes a connection to the respective protocol$\}$
3: $L \leftarrow \{e \mid e \in L \wedge e$ opens and is not followed by closing event e'$\}$
4: $T \leftarrow \{t \mid \exists e \in L :$ e is the starting event of t$\}$
5: return $T$
```

Sequential Before P4.1 The Sequential before, also called precedence pattern, refers to a Task A that requires the previous execution of another Task B. If this property is violated A is executed without the execution of B. So we have to report something that does not happen. This is more difficult than reporting something does that happen. For this pattern we highlight A and a non-existing instance of B.

Optional Sequential Before P4.2 and Not-Parallel Tasks P4.4 These two patterns are relatively easy to handle. If they are violated, then both A and B have to exist in the log. We can simply highlight the two tasks, i. e., the set of highlighted elements is $T = \{A, B\}$.

Sequential After P4.3 This pattern, also known as *Response* or *Leads-to*, describes a Task A that requires the subsequent execution of another Task B. The pattern is symmetric to the Sequential-Before Pattern, and its processing is analogous.

Restricted Access P5.1 The template for the actions has a place for the ECU used, see Subsection 5.1.3. This place models the access to the component. The pattern restricts the access to one action at the same time. In other words, the place ECU is restricted to 1. For the pattern, we identify two tasks that access the component ECU at the same time. Each task has an assignment to the ECU it uses. *First*, we reduce the event log to tasks for the component we are looking for. Next, we are only interested in tasks that are active, i. e.,

$\mathcal{A}_L = \{A \mid A\_s \in L \Rightarrow A\_e \notin L\}$. Exactly two tasks A and B are active for a counter example, see Lemma 4. We highlight the tasks A and B.

Lemma 4 . For P5.1 exactly two tasks are active, i. e., $|\mathcal{A}| = 2$.

Proof 2 (of Lemma 4) . Because of the relevant region reduction described in Subsection 5.3.1, only transitions belonging to tasks with the same ECU are in the log. Now suppose that $|\mathcal{A}| < 2$. Then the two tasks would never access the same component at the same time. If it held that $|\mathcal{A}| > 2$, three tasks $A, B, C \in \mathcal{A}_L$ would exist. Without loss of generality let $A_s < B_s < C_s$ be within the sequence L. The shorter sequence until B_s would already cause the property violation. The algorithm would abort after B_s and report the shorter counter example.

Algorithm 11 report-P5.1 (event log L)

```
1: $L \leftarrow \{e \mid e \in L \wedge e$ is a start or end transition of a task$\}$
2: $L \leftarrow \{e \mid e \in L \wedge e$ uses the respective control unit ECU$\}$
3: $L \leftarrow \{e \mid e \in L \wedge e$ is of type A_s and not followed by an event A_e$\}$
4: $T \leftarrow \{t \mid \exists e \in L :$ e is the starting event of t$\}$
5: return $T$
```

Non-Parallel P5.2 We handle this pattern in a way similar to P5.1., except that we are not only looking for the tasks for one Control Unit X but for two, X and Y. Out of the active task (thus t_3 has not fired) in the log, only two tasks exist, one Task A with access to Control Unit X and one Task B with access to Y. We highlight the two Tasks A and B.

Close Connection P5.3 The counter example aborts at the first task X that does not have its connection closed by at least one subsequent execution path. We highlight this task X. Additionally, we look for each task in the process that could close the connection for these respective tasks and highlight them as well. Recall that the situation is not *symmetric*, i. e., while a connection may be opened implicitly within a task, there must always exist designated activities whose only responsibility is to close the connection.

5.3.3 Observation

The construction described in Subsection 5.3.2 has the following characteristic:

Observation 1. For each property pattern a reporting has been defined. This reporting delivers a succinct set of elements for each pattern.

We will show that this reporting presents the violations to the user in a way that is understandable.

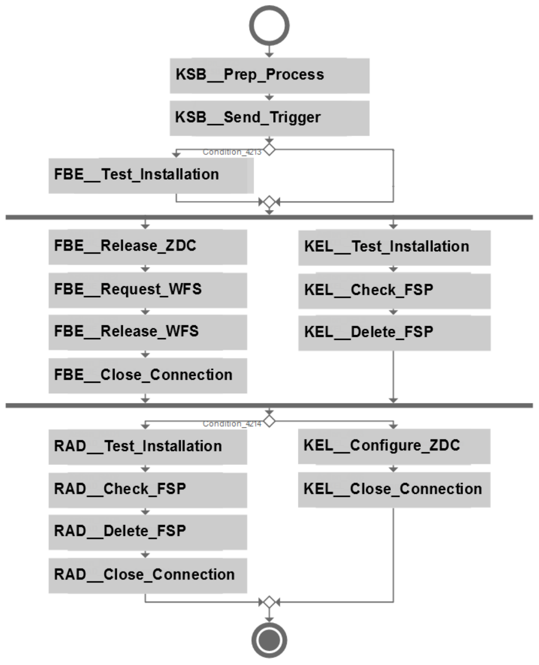


Figure 5.16: A Commissioning Process

5.3.4 Example for Commissioning Process with Violations

Consider the commissioning process in Figure 5.16. The notation is similar to the one of a UML-Activity diagram. The process contains three property violations. First, for every control unit the task *Test_Installation* precedes task *Request_WFS* (P4.1). Second, the control units FBE and KEL must not be used in parallel (P5.2). Third, the connection to the control unit is not closed in all cases (P5.3). The only task that can close the connection is *Close_Connection*. Our framework has verified the commissioning process and has been able to

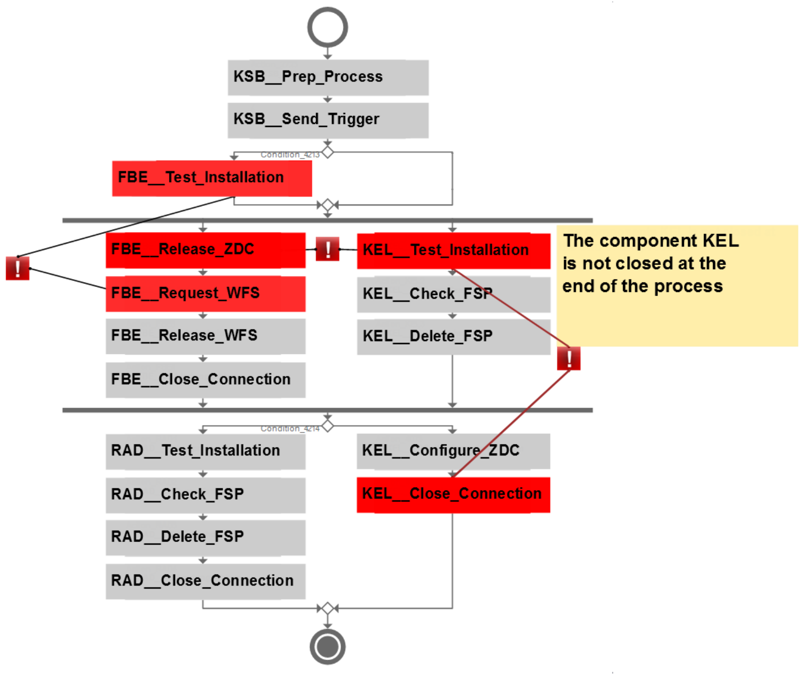


Figure 5.17: The Highlighted Violations in the Commissioning Process

detect all three errors. Using the reporting just described the tool finds the tasks relevant for the errors and highlights them, see Figure 5.17. The large red boxes with exclamation points contain a description of the violations, when the mouse hovers over them as well.

5.4 Evaluation[6]

First we will present an evaluation of the verification framework AAAFT and its visualization frontend CoVA, see Subsection 5.4.1. In Subsection 5.4.2 follows an evaluation of the optimizing technique for the verification.

5.4.1 Verification Framework (AAAFT/CoVA)

According to ISO 9241-11, cf. [ISO98], usability has three different aspects to be evaluated separately:

[6]The results of the evaluation have been published in [Mra+15] and [MMB14a]

Effectiveness: Whether the user can complete his tasks and achieve the goals.

Efficiency: The amount of the resource usage to achieve the goals.

Satisfaction: The level of comfort the users experience achieving the goals.

Subsection 5.4.1 evaluates if the way the information on property violation is presented to the user is effective. Subsection 5.4.1 evaluates the efficiency of our framework to identify property violations in real processes as a whole. In order to test the effectiveness, we count the number of violations found in the process models, see Figure 5.18, and interview an expert to detect false positives and negatives, see Subsection 5.4.1. To measure the satisfaction with our tool, we use the standardized System Usability Score (SUS), see [Bro13]. The SUS is a questionnaire that has proven to be applicable to a variety of scenarios. According to [Sau11], the SUS is reliable and valid, see Subsection 5.4.1 as well.

Efficiency of the Reporting

In order to evaluate our reporting scheme we compare it to two baselines. First, given the counter example L, we highlight each element in L corresponding to a transition (BASELINE 1). Second, we use the counter example reduction of Subsection 5.3.1, without the reporting from Subsection 7.2, and highlight every element surviving the counter example reduction and corresponding to L (BASELINE 2). The third line (REPORTING) shows our pattern-specific reporting, see Subsection 5.3.2. Our evaluation criterion is the size of the event log of the counter example. We have evaluated nine property violations in commissioning processes (A – I). The properties belong to 6 different patterns: Precedence (P4.1), Response (P4.3), Not-Parallel ECU (P5.2), Closed Connection (P5.3), and Restricted Access (P5.1). We have selected these property violations because their complexity is representative for the ones of violations found.

Table 5.3: The Number of Highlighted Elements for each Approach

VIOLATION	A	B	C	D	E	F	G	H	I
PATTERN	4.1	4.1	4.3	4.3	4.1	4.1	5.2	5.3	5.1
BASELINE 1	5	29	11	70	27	31	35	98	19
BASELINE 2	2	3	2	4	4	4	15	10	6
REPORTING	1	1	1	1	2	2	2	1	2

Table 5.3 lists the number of elements of our reporting scheme and of the two baselines. The Baseline 1 counter examples often contain dozens of elements. The Baseline 2 counter examples perform well for the majority of the properties. For some violations however, the number of elements still is rather large. The ecu conditions in particular lead to many elements with the Baseline 2 approach. This is because many tasks with the respective ecus often are executed before the violation occurs. Reporting finally performs much better than the alternatives for all properties and yields concise output.

Efficiency of the Framework as a Whole

We have used our prototype in order to verify 60 commissioning processes, newly generated or modified ones, before their execution. These processes refer to four vehicle series: the middle class car M1, the upper-middle class car M2, the executive car M3, and the sports car M4. They are executed at 34 stations. We have discussed the verification results and have categorized the processes into three categories: *correct*, with *minor* process disturbance and with *major* process disturbance. Figure 5.18 shows the number of processes in the three categories for each vehicle series. Most of the *minor* disturbances result from incorrect labels of tasks and missing values in the database. For few processes, the verification framework has reported false positives, due to the fact that we do not consider guard conditions. These false positives have also been categorized as *minor*. In a significant share of the processes ($\approx$ 23%), we could detect a *major* disturbance. All this shows that our framework can detect control flow disturbances in real commissioning processes. *Major* disturbances are property violations that hinder the execution of the process particularly, e. g., cause a deadlock.

Next, we study the key characteristics of the verification. In particular, depending on the size of the commissioning process, i. e., how many commissioning tasks *#Task* and electronic control units *#ECU* such processes contain, we are interested in the number of property instances our tool generates for each process *#PropIns*. We also are interested in the *runtime* of the specification and verification, i. e., the combined time our framework needs to specify the properties, as well as run the verification and report the violations. The framework runs on a working notebook with 2.6 GHz (dual core) and 8 GB main memory. We use a local MySQL database running on the same machine.

Table 5.4 shows the five-number summary of these characteristics and the respective boxplots. The five-number summary consists of the five most important

	M1	M2	M3	M4
No. of Processes	13	17	25	5
Correct	3	3	0	0
Minor Disturbance	9	8	18	5
Major Disturbance	1	6	7	0

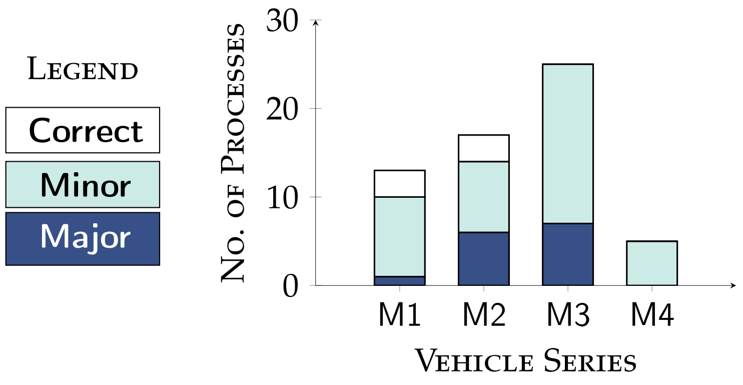


Figure 5.18: Process Disturbances Found in the Evaluated Processes

percentile: the minimum value found Minimum, the first Quartile Q_1, the median, i. e., second Quartile Median, the third Quartile Q_3 and the maximum value found (Maximum). Even for the most complex commissioning process with 870 tasks, 103 ecu, and 280 different property instances our framework is able to do the verification in 10.5 s. In 75% of the cases the verification needs less than 3 seconds in total. The runtime is strongly correlated with the size of the process, i. e., 0.926 for #tasks and 0.904 for #ECU, and even stronger with the number of property instances, i. e., 0.973 for #PropIns. We have generated and verified 2801 property instance and verified them in 104.1 seconds, i. e., an individual property instance takes only 37.2 ms to be verified on average. The boxplots in Figure 5.18 show that most of the commissioning processes contain between 50 and 200 tasks and have between 15 and 100 properties and take between 0.5 and 3 seconds to verify.

Expert Interviews

To evaluate our approach we have held semi-structured expert interviews. We aim to test the three characteristics process quality, generality and usability. The

Table 5.4: The Five-Number Summary for *#Task*, *#ECU*, *#PropIns*, and the *runtime*

Characteristic	*#Task*	*#ECU*	*#PropIns*	*runtime*
Q_0 Minimum	5	2	2	108 ms
Q_1	39	10	15	567 ms
Q_2 Median	125	29	39	1 458 ms
Q_3	211	54	95	2 973 ms
Q_4 Maximum	870	103	280	10 544 ms

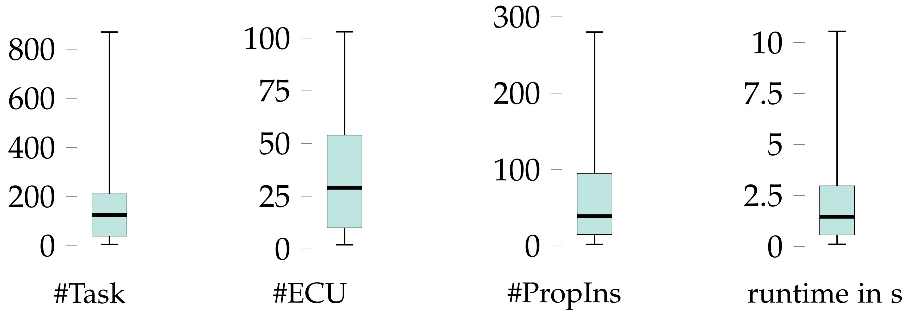


Figure 5.19: The Boxplots for the #Task, #ECU, #PropIns, and *runtime*

interview guide is available on our website:
`http://dbis.ipd.kit.edu/2027.php`.

Process Quality: *Has the framework increased the quality of the commissioning processes?* This criterion includes the change in the development time of processes, the number of *false positives* and the number of *false negatives*.

Generality: *Can the framework be used in a different context within the company?* For instance, is the framework general enough to be used in another factory? How can the framework can be integrated into the tool chain?

Usability: *Can the framework used in an intuitive way? Is the help of a technical person needed in order to use the framework?* Regarding usability we have used the Standard System Usability Test (SUS), see [BKM08]. SUS is a 10 item test that is scored on a 5-point scale of strength of agreement or disagreement. The SUS has the advantage that it is technology-agnostic, i. e., it can be used in different application domains. Due to its wide usage, a meta-test and guidelines exist to interpret the results, see [BKM08].

Participants Participants in our study are domain experts, i. e., employees who have developed commissioning processes. We have limited our interviews to experts who had used our framework intensively and had enough expertise to give feedback. We have been able to gain four experts who met these requirements for a qualitative interview. Their experience in developing commissioning processes ranges between 1 and 14 years, with an average of 7 years. The experts are waiting at different factories and departments at the AUDI AG.

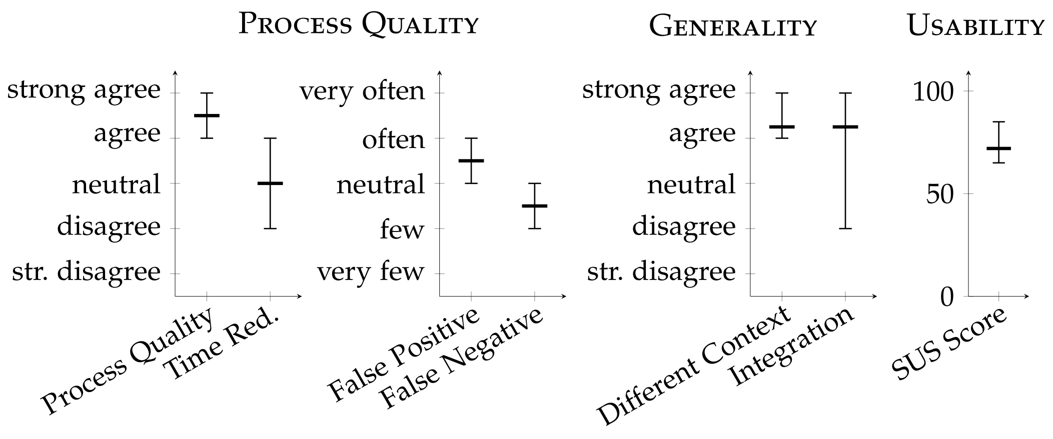


Figure 5.20: Results of the Empirical Evaluation

Results and Discussion Figure 5.20 shows the results of our qualified interviews. The experts do not think that our framework will influence the development time negatively. The number of false positives and of false negatives are acceptable but should be improved. Our framework detected slightly more false positives than false negatives. The experts saw a great potential of our framework to be used in other testing environments as well. The rating of how well the framework can be integrated into the tool chains varies between the experts. The SUS score, i. e., a measure for the usability, ranges between 65 and 85 with an average of 71.67. This is slightly above the average (69.69) and median (70.91) of reported studies using the SUS score, cf. [BKM08]. The fact that this value is above average is a positive result given the complex nature of a formal specification and verification tool. All experts see great potential in improving the quality of the commissioning processes by means of our framework.

Discussion The evaluation has shown that the experts deem our framework very useful to enhance process quality. One issue has been the number of false positives that our tool generates. We have found out later that outdated specification documents have been the reason for these false positives. We have now updated the database and have added additional checks before adding information to the database. False negatives result from properties that are unknown at verification time and are not yet specified. The costs of a false positive are rather small. This is because, it is only a small manual effort of an engineer to check the error and mark it as a false positive. The costs of a false negative are much higher in general. They are however difficult to quantify in a general fashion. For instance, one error might only lead to a slightly larger processing time, while another error could cause a stop of the production line. A minor issue is that the experts have criticized the amount of information presented by our framework. To address this point, we plan to have two modes. A debug mode that presents detailed information on the model checking process, and a normal mode that only shows the information required by the domain experts. In order to improve usability further, the experts had suggested presenting the results in more than one language. Some experts doubt that our framework can be easily integrated into the tool chain. After having received this suggestion, we have reimplement the tool in C#, at the time of the survey the tool has run in Java. At the moment we are integrating existing databases of the AUDI AG into our tool.

5.4.2 Relevance Optimization for Process Verification

To evaluate our relevance optimization with some generality, we study two use cases, namely »testing workflows at a car manufacturer«, see Subsection 5.4.2 and »data-flow anti-patterns« see Subsection 5.4.2. In the first use case, we verify complex industrial processes. We hypothesize that our algorithm leads to a significant performance within with complex processes. The stubborn set reduction and the invariant based compression are activated. In the second use case we evaluate our approach on another domain, »detection of data-flow errors«, to study its generality.

Verification of Industrial Processes

Our industrial partner has provided us with 40 processes to carry out this study. We have selected six representative processes, i. e., A, B, C, D, E, and F, whose characteristics we will present in detail later. These processes are indeed representative due to various criteria: number of requirements, duration of the verification, number of tasks in the process, number of ECUs used, number of

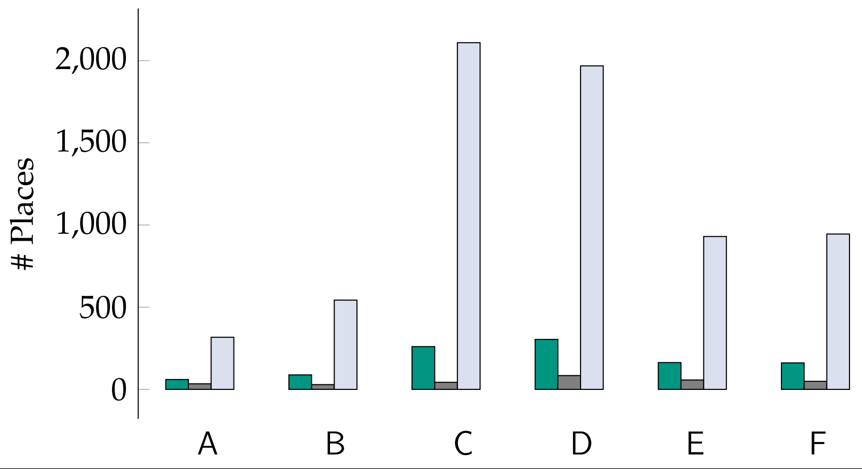
Table 5.5: Criteria of the Selected Processes

	Nr.Req.	Duration	Tasks	ECUs	parallel tasks	sequential relation
Arithmetic Mean ∅	65	3 225 ms	174	38	7 188	24 527
Standard Deviation	56	4 223 ms	165	25	12 709	49 592
A	33	956 ms	72	19	393	2 739
B	100	2 803 ms	162	59	6 275	7 931
C	275	9 498 ms	632	103	43 152	187 878
D	117	4 765 ms	482	68	25 090	121 292
E	104	3 775 ms	264	54	19 001	21 231
F	99	3 574 ms	267	64	10 855	32 348

parallel tasks, and number of sequential relationships. See Table 5.5. Each of the requirements in Table 5.5 relates to exactly one requirement of Section 4.2. These processes cover all requirements of Section 4.2 except for R2. R5 and R8 are most frequent.

Evaluation Procedure For each process and each requirement a reduced Petri net is generated at the runtime of verification. Figure 5.21 shows the number of places in the original net (org) and, in comparison, the number of places of the reduced Petri net for the average case (avg) and the maximal case (max). As shown in Figure 5.21, the Petri net is reduced significantly to a size between 2.0% and 10.1% of the original size. The biggest process yields the largest percentage reduction. The percentage reduction of the state spaces is even larger than the percentage reduction of the Petri nets. It has not been possible to compute the state spaces of the unreduced nets for an exact comparison because of their sheer size, see Subsection 3.1.2.

Result Our reduction algorithm gives way to a significant reduction of the runtime of the verification algorithm. Let R_p be the set of requirements for a process p. Table 5.6 lists the number of requirements $|R_p|$ generated from our knowledge database. Let c_i be the costs, i. e., the runtime to evaluate requirement ϕ_i for the original Petri net, i. e., to verify$(\text{hl2pn}(p), \phi_i)$. Then, the total runtime



		A	B	C	D	E	F
🟩	MAX	60	88	260	304	163	161
■	AVG	34	29	43	84	57	49
□	ORG	317	543	2109	1968	930	945

Figure 5.21: Size of the Petri Nets from the Evaluation

of the transformation and evaluation for all requirements for the original Petri net is:

$$t = \sum_{i=1}^{|R_p|} c_i$$

Let cc_i be the costs, i.e., the runtime to evaluate requirement ϕ_i for the reduced Petri net, i.e., to verify$(\mathrm{hl2pn}_c(p, \phi_i), \phi_i)$. Then the total runtime of the transformation and evaluation for all requirements by our algorithm calculates to:

$$t_c = \sum_{i=1}^{|R_p|} cc_i$$

We aborted the verification when the state space exceeds 1 mio states and mark the requirement as incomplete. In most cases of our evaluation, the verification has not been computable for the original Petri net representation without reduction. Even on a server with 128GB RAM it has not been possible to build the state space even for the SH-process, which contains only eight parallel

Table 5.6: Number of Requirements $|R_p|$ and Runtime t_c

Process p	$\|R_p\|$	t_c in s	completed	t in s	completed
A	33	0.956	100.00%	658.445	54.50 %
B	100	2.803	100.00%	1577.277	0.00 %
C	275	9.498	100.00%	4410.480	1.45 %
D	117	4.765	100.00%	1995.171	0.00 %
E	104	3.775	100.00%	1722.020	4.81 %
F	99	3.574	100.00%	1813.100	4.04 %

Table 5.7: Maximal Degree of Parallelization of the Processes

process	A	B	C	D	E	F
max. degree	8	11	10	11	12	10

branches. The processes have a maximal degree of parallelization of up to 12, see Table 5.7.

Our reduction algorithm prunes efficiently these parallel nodes with a complex structure. The size of the state space decreases by a much higher factor than the number of places and transitions in the Petri net. For example, the biggest one of our reduced Petri nets, the one for the D-process, is only 4.3% smaller than the smallest unreduced process (A). But on the other hand, while the reduced Petri net could be verified in seconds, it has not been possible to generate the state space for the unreduced process.

Use Case 2: Data-Flow Errors

In order to prove that our approach is general, i. e., can be used in other domains as well, we use our algorithm to detect data-flow errors. Besides modeling the control flow, another important aspect of business-process-management systems is to model the data-flow of the process. A common way to model the data in a workflow is to include data objects and associate them to tasks with CRUD-Operations (create, read, update, delete) [TAS09]. E. g., the authors of [TAS09] annotate the transitions with data object access with the operations performed

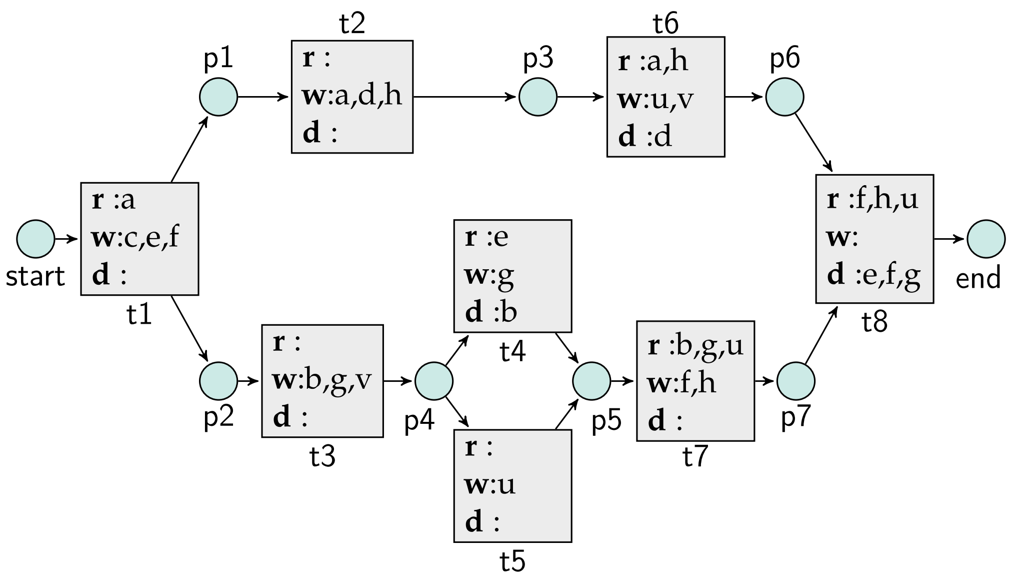


Figure 5.22: Workflow Net with Data-Flow Errors from [TAS09]

on the data object. The operations are: **r**ead, **w**rite and **d**elete. For instance, Transition t1 in Figure 5.22 reads data object a and writes the data objects c, e, and f. Errors in the process model can cause anomalies like lost updates or data objects never deleted. [TAS09] describes a set of data-flow anti-patterns. If a process model contains such an anti-pattern it contains a respective data anomaly. We want to verify that a given process does not contain any of the data-flow anti-patterns described in [TAS09]. To keep the presentation simple, we only consider the anti-patterns 1 to 8 and leave out the guard condition. The requirements in our case are the absence of these data-flow anti-patterns from the process, see Table 5.8.

For the use case *Data-Flow Errors* each anti-pattern is associated with a data object o. A task n is relevant for this anti-pattern if it accesses the data object, i.e., if n has either read, write or delete access to o.

Table 5.8: Data-Flow Anti-Patterns According to [TAS09]

ANTI-PATTERN	FORMALIZATION IN CTL
Missing Data	$\mathsf{E}[\neg\text{write}(d)\ \mathsf{U}\ ((\text{read}(d) \vee \text{delete}(d))]$
Redundant Data (Strong)	$\mathsf{EF}(\text{write}(d) \wedge \mathsf{AF}(\text{read}(d)))$
Redundant Data (Weak)	$\mathsf{EF}(\text{write}(d) \wedge \mathsf{EF}(\text{read}(d)))$
Lost Data (Strong)	$\mathsf{EF}(\text{write}(d) \wedge \mathsf{AX}(\mathsf{A}[\neg\text{read}(d)\ \mathsf{U}\ \text{write}(d)])$
Lost Data (Weak)	$\mathsf{EF}(\text{write}(d) \wedge \mathsf{EX}(\mathsf{E}[\neg\text{read}(d)\ \mathsf{U}\ \text{write}(d)])$
Inconsistent Data	$\mathsf{AG}(\neg(\ (\text{write}(d) \wedge \text{read}(d)) \ \vee\ (\text{write}(d) \wedge \text{delete}(d)) \ \vee\ (\text{read}(d) \wedge \text{delete}(d))\ )$
Never Destroyed	$\mathsf{EF}(\text{write}(d) \wedge \mathsf{EX}(\mathsf{EG}(\neg\text{delete}(d)))$
Twice Destroyed	$\mathsf{EF}(\text{delete}(d) \wedge \mathsf{EX}(\mathsf{E}[\neg\text{write}(d)\ \mathsf{U}\ \text{delete}(d)])$

Definition 19 (Data-Flow Relevance Function).
Let n be a task and ϕ a data-flow anti-pattern for a data object o

$$\text{relevant}(\phi, n) = \begin{cases} \text{true} & \text{if } n \text{ may performs a read, write or delete operation on the data object } \phi \text{ is a data-flow anti-pattern for} \\ \text{false} & \text{otherwise} \end{cases}$$

Example 5.4.1. For the Petri net in Figure 5.22 and the data-flow anti-pattern $\phi' =$ »Missing data with respect to data object o« the data-flow relevance function is:

$$\text{relevant}(\phi', n) = \begin{cases} \text{true} & \text{if } n \in \{t_2, t_6\} \\ \text{false} & \text{otherwise} \end{cases}$$

We have evaluated our approach using the process of [TAS09], see Figure 5.22. Our algorithm has detected all occurrences of any data-flow anti-pattern in this process model. It could reduce the state space significantly, see Figure 5.23. The x-axis of Figure 5.23 shows the requirements analyzed, the y-axis shows the

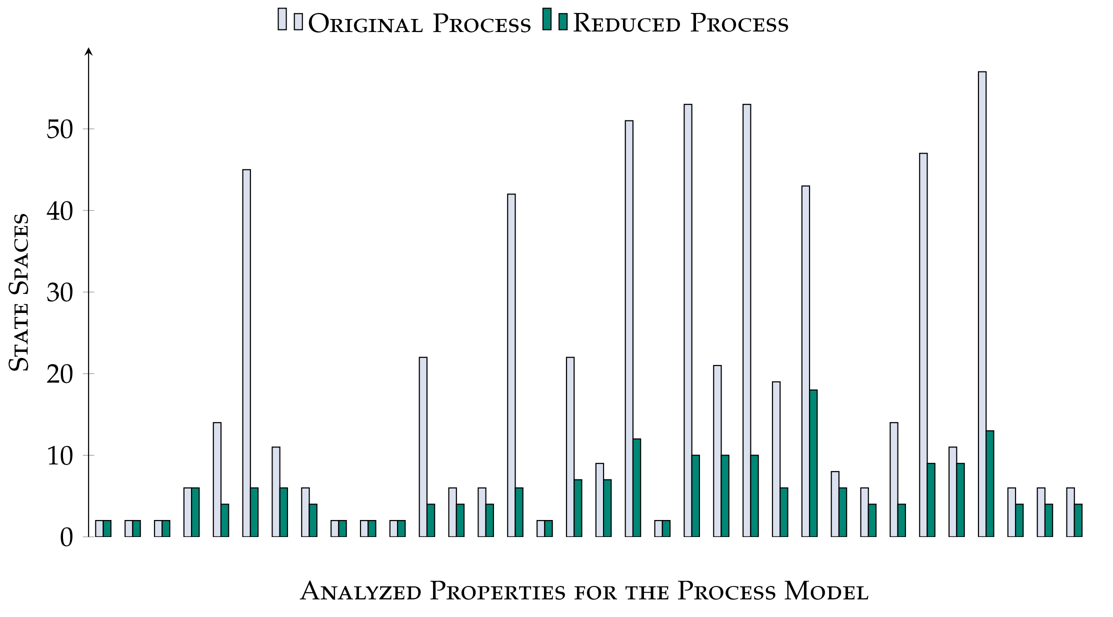


Figure 5.23: The State Space before and after the Reduction of the Process in Figure 5.22

number of the states in the state space for the original process and after our optimization step.

5.5 Related Work for Process Verification

First, we discuss the *verification of soundness*. Next, we give an overview on research on *reduction on the level of the state space*, then on work on *reduction of high-level process models* and reduction in the case of processes as result of *process mining*. We close the section with related work on the *transformation from high-level process languages to Petri nets, the verification of declarative processes* and *frameworks for compliance checking*.

Verification

We aim to check if a business process complies with the properties given. [Tag+13] uses an approach that checks if the event log L, i. e., a set of execution traces, complies with properties. In our case, there exist violations of properties that are not related to an event during process execution. For example, we do not see how to recognize a violation of a non-parallel property from the log of a process. Further, we use model checking in order to verify the processes. Most high-level process languages lack the direct construction of the state space required for model checking. To this end, a transformation to a formal language like Petri nets is required. [LVD09] gives an overview of transformations from BPMN, YAWL and WS-BPEL to Petri nets. Regarding the transformation aspect, our approach is similar to [HSS05]. [Men09] empirically evaluates different approaches for soundness verification. The criteria include error rate, process size and verification time.

Business Process Compliance

A related field of research is the business process compliance, see [Ly13]. Compliance is ensuring those process model are in accordance with prescribed norms, cf. [SGN07], e. g., Sarbanes-Oxley, Basel II, HIP AA. In general two approaches toward process compliance exists: the manual and expensive after-the-fact checking of the process model and the automated detection. For the automated detection the norms have to be specified formally. As well as in our use case, the norm specification leads to maintenance problem, cf. [Ly+08]. Approaches exist to ensure the compliance of an existing process model by, e. g., model checking [ADW08; LMX07; För+07] or synthesize a new process model that complies with the norm [Awa+11; GV06]. Another approach is to use process fragments that are compliance fragments to model the process, see [Sch+10a; Sch+10b].

Reporting

[LF14] reduces the counter examples to give the end user a compact report of the property violation found. To this end, [LF14] reduces the path using conflict clusters, spurious conflicts, and distributed runs. The approach is able to reduce the counter examples significantly, i. e., omit information that is unimportant for the shareholder. For our processes, the approach of Subsection 5.3.1, which is easier to compute, performs well. In consequence, we do not have to apply the approach of [LF14] as a preprocessing step for the reporting patterns. [LF14] could be used for the reporting of general properties not covered by the patterns.

Verification of Soundness

The soundness of a process is an important property. It means to guarantee that there is not any deadlock or any lack of synchronization, cf. [Aal+10]. Soundness verification differs from the verification of activity-based CTL formula. Soundness verification allows for reducing the state space with methods that are not applicable to our problem. This is because these methods, e. g., stubborn sets, would change the result of our verification. An example is given at the end of this subsection, after we have addressed the stubborn-set reduction. A vast amount of techniques exist to verify the soundness property for a given process. The approach of [Fah+09a] evaluates three of these techniques on a collection of 1368 industrial processes. In comparison to our production processes, see Section 4.2.1, most of these processes are small. Only 8.59% of them yield a state space with over 1 million states. In contrast, 78% of the processes from our use case yield a state space of at least this size. The first technique in [Fah+09a] uses the LoLA-Framework with a stubborn set reduction. The second technique is the Woflan tool, see [VBA01]. Woflan uses a combination of techniques from the Petri net structure theory and state space exploration. The third technique decomposes the workflow graph and generates a process tree, cf. [VVK09]. Soundness is compositional with respect to Single-Entry-Single-Exit fragments (SESE), and therefore each fragment can be verified in isolation. Our verification technique is more general than just verifying soundness as in [Fah+09a]. It is however interesting to see that some insights at an abstract level are similar to ours. In particular, the size of the processes correlates with the number of rule violations, and a significant share of processes in industrial settings contains rule violations. Over the past two decades, many authors have proposed alternative notions of soundness, e. g., weak soundness or relaxed soundness, and more expressive languages, e. g., cancellation regions or OR-joins, see [Aal+10]. One fundamental approach is to reduce the complexity of the verification task by reduction rules while preserving the property in question. [Wyn+09b] speeds up the soundness verification of a reset workflow net by deploying reduction rules on the net. [Wyn+09a] proposes a set of reduction rules to verify the soundness of an YAWL workflow with cancellation regions and OR-Joins. In general, the reduction rules depend on the property analyzed. For each reduction rule, it must be guaranteed that applying the rule does not change the verification result. A reduction rule that preserves the soundness property may not preserve other properties, see Example 5.5.1.

Example 5.5.1 . The »Fusion of series places rule« of [Wyn+09b] preserves the soundness property, but it is not possible to evaluate the sequential arrangement of places any more after such a reduction. To this end, reduction rules for soundness verification cannot be used for model-checking verification without further examination.

Reduction on the Level of the State Space

The stubborn set reduction, cf. [Sch99], is a partial-order reduction technique. It selects a subset of the operations enabled in a state s, i. e., stubborn set for s, and only explores these subsets. The generation of the stubborn sets for standard properties of a Petri net, e. g., boundedness, reachability, liveness, is described in [Sch99]. [Ger+95] presents an approach for CTL formulas without the Next operator. The verification can find smaller stubborn sets if the CTL formula is limited to the EF and AG-Operators, see [Schoob]. With these operators, it is not possible to formalize all of the requirements of Subsection 4.2. For instance, it is not possible to formalize R3 that A responds to B, $\mathsf{AG}(A \rightarrow \mathsf{AF}(B))$. We have evaluated the 40 processes from Section 4.2.1 without the stubborn set reduction, to compare it to the results with the stubborn set reduction turned on. Without the reduction, the average runtime increases by 64.2%. For the majority of processes, the runtime only slightly increases ($< 3\%$). The identification of the stubborn sets is not effective for branching time model checking, e. g., CTL or CTL* on complex processes, like the ones we want to verify. One reason is that a dependent operation can activate another operation that then affects the property analyzed. These results in big stubborn sets which tend to render the verification infeasible for our application area. Having said this, the stubborn set reduction and other approaches on this level are orthogonal to our approach and can be used in addition.

Reduction on the Level of the High-Level Process

There already exist approaches which reduce the process schema focusing on relevant aspects of the process, see [SO99] and [SO00]. Their goal is to find structural errors, e. g., deadlocks or missing synchronization. This is done by applying iteratively a chain of reduction rules to the process. These rules remove elements that do not contain a structural error. If the process contains a structural error, only the split and join gateways causing the error, remain in the process. Otherwise, the process is reduced to a start- and an end-event. [AHV02] and [Lin+02] extend the original incomplete algorithm and feature correctness proofs. This approach finds only structural properties and does not verify more complex requirements expressed in temporal logic.

[DAV05] verifies a process in **E**vent-driven **P**rocess **C**hains (EPC) notation. The authors propose a two-step approach: First, they reduce the EPC by means of reduction rules, and then they run the verification algorithm. [DAV05] verifies if an EPC model is sound; it is not possible to check more complex requirements with their approach. [ADW08] extends the set of reduction rules of [DAV05] in order to verify a BPMN process using requirements defined as queries in BPMN-Q. BPMN-Q is a visual language to query business process models. [ADW08] is restricted to some temporal operators, like leads-to ($\mathsf{AG}(\phi \rightarrow \mathsf{AF}(\psi))$ in CTL) or precedence ($\neg\mathsf{E}[\ \neg\psi\ \mathsf{U}\ (\phi \wedge \neg\psi)\ ]$ in CTL). BPMN-Q does not allow specifying that nodes must not occur in parallel. To be precise, the property pattern P3.1–P3.3 and P5.1–P5.3 of Section 4.2 cannot be expressed in BPMN-Q. With BPMN-Q, it is possible to specify only 242 out of the 728 requirements from our application scenario. Further, the approach of [DAV05] and [ADW08] cannot remove entire regions of the process in one step. After each iteration of the reduction algorithm of [ADW08] it may be necessary to check the process as a whole if another reduction rule has become applicable. Our algorithm covers each of their reduction rules. By using more information than [ADW08] on the relevance of regions we can reduce more regions than that approach. For example, [ADW08] regards the resource condition. By including this additional information into the *relevance* function, our approach can reduce the Petri net further.

Data-aware requirements not only concern the execution of tasks but also variables that influence the control flow, e. g., »If variable x is greater than 800, task *A* needs to be executed«. The requirements in our setting do not depend on data, thus we have not taken data-aware requirements into account. A difficulty with data-aware requirements in general is that modeling every possible state of a variable can lead to a state space explosion. [Knu+10] improves the runtime of the verification of data-aware requirements, in comparison to the time needed to evaluate all states of a variable. It does so by abstracting from the individual possible states of a variable in a preprocessing step. They face the same problem as we do, i. e., confine the growth of the state space, but the context and therefore the approaches differ. The work of [Knu+10] is orthogonal to our approach, and a combination to analyze data-aware requirements seems feasible.

Process Mining and Conformance Checking

[MCA13b] and [MCA13a] use a related approach to check the conformance of process schemata discovered by process mining techniques. Process mining techniques aim to discover a process schema from a set of execution traces,

i. e., from a log cf. [Aal11]. A recently proposed algorithm discovers processes as block-based structures, cf. [LFA13b]. As in our approach, they explore the hierarchical structure of a process tree to accelerate the computation.

Conformance checking validates if the discovered process is consistent with the log. [MCA13b] and [MCA13a] decompose a discovered Petri net process into SESE fragments in order to build a process tree for conformance checking. The algorithm of [MCA13b] checks the conformance for each inner node in the process tree. The hierarchical structure of the process tree lets them identify conformance problems and limit the complexity of the conformance analysis by only exploring certain nodes. This algorithm explores only leaf nodes and inner nodes containing a number of tasks below a threshold value. [MCA13b] is based on the assumption that the conformance can be calculated meaningfully limited to a SESE region. In our verification case we cannot partition the problem in this way. For example, a task A in one SESE region can have a response requirement for another task B in another SESE region.

Transformation from Process Languages to Petri Nets

Many high-level process modeling languages (e. g., BPMN, WS-BPEL, EPC) lack execution semantics adequate to create the state space that gives way to verification. To this end, the high-level processes are first transformed to a formal representation. [Rae+07] has developed a transformation from BPMN models to Petri nets. For a transformation of WS-BPEL to Petri nets see [HSS05; Sta05]. [LVD09] provides a survey of transformations from various process languages to Petri nets. We follow a similar transformation approach, providing for each control-structure element of the source language a Petri net pattern that reflects its execution semantics. For OTX, the process language we are supporting, we are not aware of any transformation described in literature so far. But because of the similar structure of OTX and WS-BPEL, we have used the approach of [HSS05] as the basis of our transformation.

Verification of Declarative Process Models

Classic procedural process models only explore a subset of the possible behavior allowed, due to their implicit assumption that »all that is not explicitly modeled is forbidden« [Mon+10]. This is problematic in domains that require high flexibility like service choreographies, cf. [Mon+10]. To overcome this limitation, declarative process languages do not define the imperative execution of the flow, but the set of rules that the process should fulfill. The flow of the process includes all instantiations that do not violate the rules. The rules often are defined in a graphical notation like ConDec [PA06] or [AP06]. For verification or enactment, these models are mapped to a formal logic-based notation, e. g., LTL [Mag+12] or SCIFF [Mon+08]. Verification in declarative process models means to test either if the rules are in conflict, thus no instantiation is possible as in [Mag+12], or if the behavior matches predicted behavior at runtime or a-posteriori, see [Mon+08]. This kind of verification is different from our approach. We test at design time if the process model overlaps with forbidden behavior.

Compliance Checking Frameworks

The SeaFlows toolset is a framework for the specification and verification of compliance rules on business processes, see [Ly13; Ly+11a]. For the specification a new visual graph-based language, Compliance Rule Graphs (CRG), has been developed. The language tends to be more usable for domain experts than the formalization in, say, LTL. CRGs can be mapped unambiguously to rules specified in first-order predicate logic. It would be possible to use SeaFlows graphs as a specification language within our approach, but with two difficulties. First, the LoLA-Framework can only simulate Petri nets with CTL formulas. We would need to use or develop a different tool for the verification and state-space generation of predicate logic. Second, because of the large number of requirements of a certain requirement type, see Section 4.2, we have used an automatic transformation to generate the requirements directly from domain information. In such a setup, a graphical language would not bring any benefit. [KRL11] uses an activity-oriented clustering to determine which requirement C needs to be verified for which process model P. In more detail, the approach of [KRL11] uses an *IsTriggered* function comparable to our relevance function. In their approach, an antecedent occurrence node represents the occurrence of a task triggering the requirement it belongs to, cf. [Ly13]. A requirement is considered *IsTriggered* in a process model P if it either does not have any antecedent occurrence node, or if the set of antecedent occurrence tasks is a subset of the nodes in P. For example, in Pattern »*B responds to A*«, A is the antecedent occurrence node, in pattern »*A precedes B*« B is the antecedent occurrence node. [KRL11] focuses on entire process models. Our approach in turn addresses regions within a process. The *IsTriggered* function cannot be used as a relevance function because it would not preserve the requirements for verification if applied to regions of the process model, see Example 5.5.2.

Example 5.5.2. Consider the Process Tree of Figure 5.11(a) and the requirement ϕ: »t_2 responds to t_1«, i. e., $\phi = AG(t_1 \rightarrow AF(t_2))$ in CTL. ϕ evaluates *true* on the original process model. Only t_1 is an antecedent occurrence node. If we use our pruning algorithm with the *IsTriggered* function the resulting process model only contains SEQ_1 and t_1. On this process model, ϕ would be evaluated to *false*. Thus the *IsTriggered* function does not preserve the requirements in the general case and is not a valid relevance function for our optimization.

Chapter 6
Process Synthesis

Process Synthesis means automatically generating an imperative process model from specifications. The specification can be in several forms, e. g., bill-of-material (BOM) [Aal99], obligations [GV06], artifacts [Loh13], or in form of a temporal property [Awa+11; Yu+08]. Due to the increase in components the complexity of the commissioning process models is steadily raising, e. g., over 1000 tasks per vehicle. In case of a manual generation, a large amount of design iterations of a process model are needed until the model fulfills all properties required, i. e, is correct. Chapter 5 describes the way to verify a given process model if it fulfills the properties. In this chapter we want to synthesize an imperative commissioning process model automatically.

However, the process synthesis gives way to several major challenges. First, the resulting process model should be applicable for our use case, thus the process model has to be block-structured. Second, often the specification allows more than one possible process model. Our approach should be able to generate a good process model according to quality criteria out of the possible ones. Third, the approach should be scalable, i. e., it should be possible to apply the approach to the large real-to life specification of our use case. To target those challenges we want to propose a novel synthesis algorithm for the automatic generation of process models.

We develop two algorithms for the synthesis. Section 6.1.1 presents the first algorithm to synthesize a process model from a schedule. We will show that such an approach is possible but fails to deliver block-structured process models and thus is not suitable for our use case. To this end, we develop a second algorithm in Section 6.2. The approach synthesizes a process model by a modular decomposition of the specification graph. Section 6.3 applies the algorithm to the case study of a new vehicle model for a compact executive car and its commissioning process, and gives some heuristics developed during the case study. The case study reveals some new and novel requirements and shows the applicability of our approach to a real scenario. Section 6.5 discusses the related work for the process synthesis.

6.1 Resource Constraint Scheduling Synthesis

A schedule is the planned execution of a set of activities. The **R**essource **C**onstraint **S**cheduling **P**roblem (RCSP) is an optimizing problem in order to find an optimal execution for a given set of activities. The algorithm only considers predecessor relations between the activities, as well as capacity dependencies of resources. The resource constraint scheduling problem is related to our problem statement the synthesis of imperative process models from a declarative specification. The dependency graph is similar to our sequential properties, and with resources non-parallel dependencies can be expressed.

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of activities, and a dependency graph $G(A, E)$ with $(a_1, a_2) \in E$ iff the activity a_2 requires the previous execution of a_1. The function $run : A \mapsto \mathbb{R}^+$ gives the expected processing time of an activity. Given a set of resources R, a function that for every activity and every resources gives its usage $rv : A \times R \mapsto \mathbb{N}_0$, and a function that gives the capacity of each resource $cap : R \mapsto \mathbb{N}_0$. We are searching for a Schedule s, i. e., a function $s : A \mapsto \mathbb{R}^+$ that states for each activity its starting time. The schedule should be minimal according to total processing time of the process, i. e., we are looking for a schedule s that is minimal to the following function:

$$\min_s(\quad \max(\, s(b) + run(b) \mid a \in A \,) \quad - \quad \min(\, s(a) \mid a \in A \,) \quad)$$

Under the constraint that the schedule s fulfills the predecessor relations, i. e.:

$$(a_1, a_2) \in E \Rightarrow s(a_1) + run(a_1) < s(a_2)$$

and at each time during execution the resource consumption should not exceed the capacity for any resource, i. e.:

$$\forall_{r \in R} : \sum_{a \in A, t \in [s(a), s(a) + run(a)]} rv(a) \leq cap(r)$$

The RCSP problem is NP complete. Existing approaches using heuristics or genetic algorithms can solve the RCSP in acceptable runtime.

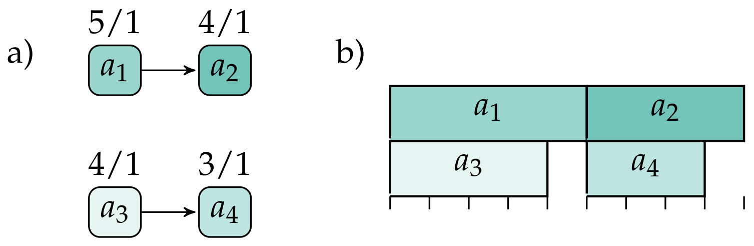


Figure 6.1: Dependency Graph with Processing Time and Resource Consumption (a) a Possible Optimal Schedule (b)

Example 6.1.1. Let $A = \{a, b, c, d\}$ bet set of activities. The processing times are $run\,(a_1) = 5$, $run\,(a_2) = 4$, $run\,(a_3) = 4$ and $run\,(a_4) = 3$. Only one resource r exists with $cap\,(r) = 2$. For each activity $a \in A$, $rv(a, r) = 1$. One predecessor relation between a_1–a_2 and between a_3–a_4 exist, i. e., $(a_1, a_2), (a_3, a_4) \in E$. Figure 6.1(a) shows the dependency graph for the problem statement. The processing time and resource consumption are given above the nodes ($run\,(a) / rv\,(a, r)$). A possible optimal schedule s is given as $s(a_1) = 0$, $s(a_2) = 5$, $s(a_3) = 0$, $s(a_4) = 5$. Figure 6.1(b) shows the schedule.

The generation of a process model from a schedule is possible, but the process model will lack properties required for our use case, the commissioning of vehicles. Subsection 6.1.1 will show a simple algorithm to construct a process model from a schedule s. Subsection 6.1.2 discusses the limitation of the algorithm, i. e., that the algorithm and all approaches from a schedule are not able to generate a flexible process model.

6.1.1 PNSyn Algorithm

The algorithm is able to generate Petri nets from a schedule s. The algorithm is effective and we will show that the Petri net is able to execute the schedule. However, the algorithm has two major drawbacks. First, the algorithm generates an unstructured process model, in general. This holds even in cases the specification allows a structured equally effective process model. For some unstructured process models no trace-equivalent structured process model can exist. Second, the generated models lack flexibility. The models are more constrained than required and restrict the execution unnecessary. These inflexibility leads to longer process processing time when activities deviate from the estimation.

We define a relation $\prec$: $A \times A$. The relation is true if two activities are in a predecessor relation in the schedule s:

$$\forall a, b \in A \ : \ (a,b) \in \ \prec \quad \Leftrightarrow \ s(a) + run\,(a) \ \leq \ s(b)$$

The relation $\prec$ is transitive, irreflexive and asymmetric. The relation $\prec$ induce a graph $G_2(A, \prec)$. The dependency graph G is a sub graph of G_2. For the Petri net $PN(P, T, F, m_0)$ we define the set of transition as:

$$T \ = \ t_s \ \cup \ t_e \ \cup \ \{t_a \mid a \in A\}$$

t_s is a silent *start* transition, required if more than one activities are started first. t_e is a silent *end* transition, required if more than one activity end as last. The set of places P is defined as:

$$\begin{aligned} P \ &= i \ \cup \ o \ \cup \ \{p_{ab} \mid (a,b) \in \ \prec\} \cup \{ps_a \mid \nexists b \in A : (b,a) \in \ \prec\} \\ &\cup \{pe_a \mid \nexists b \in A : (a,b) \in \ \prec\} \end{aligned}$$

i is the start place, o the end place, p_{ab} connects the transitions of two activities, ps_a are the places for the starting activities, i. e., the activities executed at first, and pe_a the place for the end activities, i. e., the activities executed at last.

$$\begin{aligned} F \ &= (i, t_s) \ \cup \ \{(t_s, ps_a), (ps_a, t_a) \mid \nexists b \in A : (b,a) \in \ \prec\} \\ &\cup \ \{(t_a, p_{ab}), (p_{ab}, t_b) \mid (a,b) \in \prec\} \\ &\{(t_a, pe_a), (pe_a, t_e) \mid \nexists b \in A : (a,b) \in \ \prec\} \ \cup \ (t_e, o) \end{aligned}$$

The starting state m_0 is:

$$m_0(p) = \begin{cases} 1 & \text{Falls } p = i \\ 0 & \text{else} \end{cases}$$

The relation $\prec$ can be computed in linear time, using the transitive of the relation. The generation of the Petri net can be calculated in constant time. Therefore the algorithm functions in linear time to the size of the schedule.

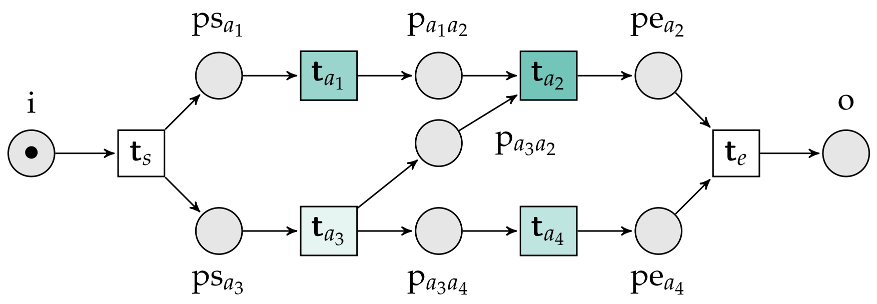


Figure 6.2: Generated Petri net for the Schedule in Figure 6.1(b)

Example 6.1.2. For the schedule in Figure 6.1(b) the following relations holds: (a_1, a_2), (a_3, a_2), $(a_3, a_4) \in \prec$. This gives way to the following transitions T, places S and edges F:

$$T = \{t_s, t_e, t_{a_1}, t_{a_2}, t_{a_3}, t_{a_4}\}$$

$$S = \{i, o, p_{a_1a_2}, p_{a_3a_2}, p_{a_3a_4}, ps_{a_1}, ps_{a_3}, pe_{a_2}, pe_{a_4}\}$$

$$\begin{aligned} F = &\{(i, t_s), (t_s, ps_{a_1}), (ps_{a_1}, t_{a_1}), (t_s, ps_{a_3}), (ps_{a_3}, t_{a_3}),\\ &(t_{a_1}, p_{a_1a_2}), (p_{a_1a_2}, t_{a_2}), (t_{a_3}, p_{a_3a_2}), (p_{a_3a_2}, t_{a_2}), (t_{a_3}, p_{a_3a_4}),\\ &(p_{a_3a_4}, t_{a_4}), (t_{a_2}, pe_{a_2}), (pe_{a_2}, t_e), (t_{a_4}, pe_{a_4}), (pe_{a_4}, t_e), (t_e, o)\} \end{aligned}$$

Figure 6.2 shows the Petri net for the schedule of Figure 6.1.

6.1.2 Limitation of the *PNsyn*-algorithm

The *PNsyn*-algorithm often generate an unstructured process model. In general no structured process model exists that is trace-equivalent to the unstructured process model. This fact holds true even if the specification allows a structured optimal process model.

Proof 3. Consider the Petri net of Figure 6.2. The net is obvious unstructured. The set of execution paths is $\mathcal{L}_P = \{ \langle a_1, a_3, a_2, a_4\rangle, \langle a_1, a_3, a_2, a_4\rangle, \langle a_3, a_1, a_2, a_4\rangle, \langle a_3, a_1, a_4, a_2\rangle, \langle a_3, a_4, a_1, a_2\rangle \}$. This gives way to following behavior profile:

$$\begin{array}{c|cccc} & a_1 & a_2 & a_3 & a_4 \\ a_1 & + & \rightarrow & \| & \| \\ a_2 & \leftarrow & + & \leftarrow & \| \\ a_3 & \| & \rightarrow & + & \rightarrow \\ a_4 & \| & \| & \leftarrow & + \end{array}$$

Figure 6.3(a) shows the **O**rdering **R**elation **G**raph (ORG) for the behavior profile. The graph contains a prime component and thus no structured process model exists that is a trace-equivalent, cf. [Pol12]. The Petri net in Figure 6.2 without the place $p_{a_3 a_2}$ and the adjacent edges is able to reproduce the schedule s. The behavior profile is:

$$\begin{array}{c|cccc} & a_1 & a_2 & a_3 & a_4 \\ a_1 & + & \rightarrow & \| & \| \\ a_2 & \leftarrow & + & \| & \| \\ a_3 & \| & \| & + & \rightarrow \\ a_4 & \| & \| & \leftarrow & + \end{array}$$

Figure 6.3(b) shows the Ordering Relation Graph for the new behavior profile. The ORG does not contain a prime component, i. e., the model is structured, see Figure 6.3(c).

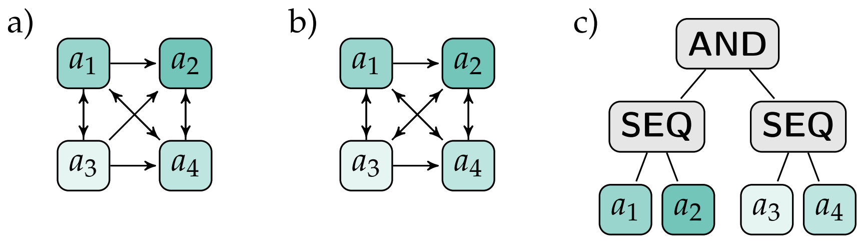


Figure 6.3: Ordering Relation Graph for the Petri net of Figure 6.2

The counter example in proof 3 shows that in general the *PNSyn*-Algorithms generates unstructured Petri nets. It is not possible to transform an unstructured Petri net into a process tree. Therefore the generation of an OTX or SIDIS.Pro process model is not possible. This aspect is not limited to the *PNSyn*-Algorithm. It is true for all approaches solely using a schedule s. The predecessor relation $\prec$ contains all information of a schedule. It is not possible to distinguish the reason why two activities are in a $\prec$-relation. It could be to comply with a given constraint, because of optimizing aspects, or just random. Due this aspect the process models are inherent inflexible and over-specified. This leads to a unstructured process models and a lack of flexibility. This lack of flexibility could lead to a sub optimal execution if the processing time of activities deviates from the estimation. For instance, in one case the execution of a_3 takes 6 time units instead 4. For the Petri net in Figure 6.2 the processing time of the complete process model would be larger, due to the additional dependency between a_3 and a_2. In a flexible process model, without the additional dependency, the unexpected processing time could be compensated and a better runtime achieved.

[Sen+15] generates a Petri net from a schedule to test the validity of a schedule s. To this end, the resulting process model is tested to conform to a log L of actual executions, see the section conformance checking in 3.3.4. In the use case of [Sen+15] the goal is to detect discrepancies between the schedule and the actual execution, i. e., the lack of flexibility is not an issue.

6.2 Automatic Generation of Optimized Process Models[1]

In Section 6.1 we have shown an algorithm to generate a process model from a schedule. The generation from a schedule has some unwanted characteristics, e. g., overly constraint and inflexible. In this section we will introduce a new algorithm to synthesize a process model that does not constrain the execution any more than necessary. First we want to motivate the algorithm and its requirements using our use case, the commissioning of vehicles.

Example 6.2.1 . Our application scenario is commissioning. Commissioning means configuring and testing the electronic components of a vehicle during its production. Process models describe the arrangement of the configuration and testing tasks. For instance, a factory worker has to configure the transmission and to activate the anti-theft system. The transmission can either be manual, i. e., Task M does the configuration, or automatic (Task A). Task T activates the anti-theft system. Before the activation, a central computer needs to generate a master key (Task G), and it opens the connection to the specific control unit (Task O). The connection has to be closed before the process finishes (Task C). The configuration of the transmission and the

[1]Parts of this Subsection are published in [MMB15] and the extended version in [MMB14b]

activation of the anti-theft system require a running engine. Task E turns it on. Figure 6.2(a) shows the tasks that may be part of the commissioning process. The second column is the expected processing time. Commissioning always has a context, i. e., the variation of the vehicle, its components, their relationships and the constraints the vehicle currently tested must fulfill. The variation determines which tasks have to be executed, e. g., a car with a manual transmission requires different tasks than a car with an automatic one.

a)

		Task	time
E	:	Start **E**ngine	1 s
M	:	Conf. **M**anual transmission	5 s
A	:	Conf. **A**utomatic transmission	2 s
T	:	Activate anti-**T**heft system	1 s
C	:	**C**lose Connection	1 s
O	:	**O**pen Connection	1 s
G	:	**G**enerate Master Key	5 s

b)

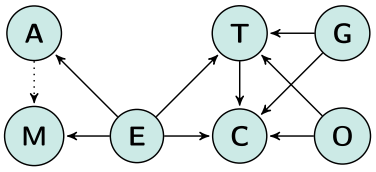


Figure 6.4: The Tasks for Commissioning (a) and the Ordering Relations Graph (b).

A context c determines the tasks $\mathcal{T}_c$ required for a process. It is not feasible to model all processes for each possible set of required tasks by hand. This calls for generic process models covering several contexts. With such generic models, however, an optimal arrangement of tasks for any context does not exist.

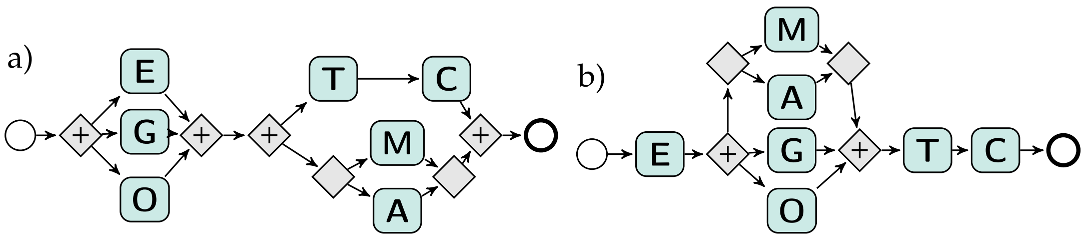


Figure 6.5: Two Distinct Process Models for the Graph in Figure 6.2(b)

Example 6.2.2 . The context characteristic *transmission* determines the required tasks as follows: If the vehicle has an automatic transmission, the commissioning requires execution of the tasks $\mathcal{T}_c$ =*{E, A, T, C, O, G}*. For a manual transmission in turn the tasks are $\mathcal{T}_c$ =*{E, M, T, C, O, G}*. Figure 6.2(b) shows the dependencies between the tasks as a graph. Directed edges represent ordering dependencies, while dashed lines represent exclusive dependencies. The graph is the declarative specification we generate the process model from. Section 6.2.2 shows how one can generate such a specification from input in other languages.

Figure 6.5 shows two generic process models that comply with the dependency graph of Figure 6.2(b). Figure 6.5(a) has a shorter processing time if the transmission is automatic (7 to 10 seconds). For a manual transmission in turn, the process model in Figure 6.5(b) has a shorter processing time (8 to 10 seconds).

With at least one process model for each possible context, the number of such models increases exponentially with the number of context characteristics.

Example 6.2.3 . Say a vehicle has 10 different context characteristics, e. g., the kind of transmission, the navigation system, the safety system, etc. If each option can occur or not $2^{10} = 1024$ different contexts are possible.

Next, several models typically are possible for a given context. The problem studied here is how to generate a good process model for a given context from a declarative specification. The model should be good according to predefined quality criteria, e. g., throughput time. Process models that comply with the dependencies can be very different with respect to quality and performance criteria. Section 6.4 shows that models generated with our approach are about 50% faster than the ones designed by professionals with several years of experience. We focus on the restricted case that there are no repetitions, i. e., we focus on process trees with inner nodes SEQ, AND, XOR, but not LOOP. There is a number of settings with this characteristic, for instance in manufacturing. In particular, loops are unnatural in commissioning processes, since a feature is tested only once. On the other hand, if a problem occurs and is fixed, a new commissioning process is started. Another assumption, which also holds for commissioning and elsewhere, is that context information together with experience from the past allow to reliably estimate the processing time of individual tasks.

The generation of a process model from a declarative specification bears several challenges. There often exists a great variety of models that fulfill the specification, as mentioned before. To illustrate, the sequential arrangement of n nodes, in the absence of any constraint, give way to $n!$ different process models. For the largest process in our evaluation, 4.12×10^{340} models are possible. Generating all possible models is not possible. It is challenging to detect a good process model that does not violate any constraint.

Example 6.2.4. There are four tasks A, B, C, and D. Suppose that the following constraints exist: B must always occur before A and C ($B \rightarrow A,\ B \rightarrow C$), and D always occurs before C ($D \rightarrow C$). It seems reasonable to put A and C in parallel, because this might reduce the throughput time, but putting A and C in parallel rules out having A and D in parallel.

Related work in process synthesis is fully automatic only for processes that are fully specified by their dependencies, see [Awa+11; Yu+08]. In case of an under specification, [Awa+11] requires a process modeler to manually make decisions, and [Yu+08] requires a manual clustering of the constraints. This is not practical, because of the daunting number of possible models. To this end, we propose a novel process synthesis algorithm whose output on the one hand complies with the dependencies and on the other hand is good according to predefined criteria. Our approach is as follows: First, it uses a modular decomposition of the dependencies to detect the fully specified regions of the process as well as the under-specified ones, so-called prime components. For each prime component, our approach partitions the corresponding ordering graph systematically, as follows. It selects a pivot element and generates several smaller ordering graphs from the pivot partition. We reduce the problem in a divide and conquer fashion until it is small enough to explicitly generate all possible models. We repeat this for different pivot elements to have a better coverage according to our quality criterion the throughput time of the process. Other criteria such as overall energy consumption are possible as well. As we show in the evaluation with thousands of non-trivial process models, our approach is efficient, i. e., is able to test thousands of models in under a second, checking for complex constraints. On average, our approach nearly halves the processing time compared to the reference processes, which already are the output of a careful intellectual design. Our approach can handle complex real-world specifications containing several hundred dependencies as well as more than one hundred tasks. In our evaluation, the process models generated contain between 98 and 185 tasks, and their arrangement typically is nontrivial.

6.2.1 Preliminaries

A meaningful input for process synthesis is the declarative specification in the form of an ordering relation graph (ORG), see [Pol12]. The modular decomposition of a graph yields its components and implies a hierarchical structure of components called the Modular Decomposition Tree (MDT) [MM05], see Subsection 6.2.3. The MDT separates the under-specified regions from the fully specified ones.

Ordering Relation Graph

In an **O**rdering **R**elation **G**raph, each node represents a task. Each edge represents a dependency between tasks. The dependencies consist of ordering dependencies, i. e., in which order do the tasks occur, and exclusive dependencies, i. e., two tasks exclude each other.

Definition 20 (Ordering Relation Graph) .

The ordering relation graph is a directed attributed graph $G = (V, E)$, with V being nodes and $E \subseteq V \times V$ the edges. Each node corresponds to a task. E consists of two subsets $E_\rightarrow$ and $E_\#$ such that $E = E_\rightarrow \cup E_\#$ and $E_\rightarrow \cap E_\# = \emptyset$. $E_\rightarrow$ defines the ordering relation, i. e., two tasks that should be in a specific order have an edge in $E_\rightarrow$. $E_\rightarrow$ is transitive and anti-symmetric:

$$\begin{aligned} (\textit{transitive}) \quad & \forall (x,y),(y,z) \in E_\rightarrow : (x,z) \in E_\rightarrow \\ (\textit{antisymmetric}) \quad & \forall (x,y) \in E_\rightarrow : (y,x) \notin E_\rightarrow \end{aligned}$$

$E_\#$ defines the exclusiveness relation, i. e., if two tasks exclude each other they share an edge in $E_\#$.

$E_\#$ is symmetric, i. e., $\forall (x,y) \in E_\# : (y,x) \in E_\#$. We do not allow self-edges, i. e., $\forall v \in V : (v,v) \notin E$.

Note that $E_\rightarrow$ does not contain any cycle. For each task we determine the processing time. The average error of the estimated execution times of our tasks from our application scenario is less than 17%. We had calculated these times by analyzing the logs of existing traces.

Definition 21 (Neighborhood).
The neighborhoods $N^{\text{out}}(v)$, $N^{\text{in}}(v)$ of a node v are defined as:

$$N^{\text{out}}(v) := \{w \mid w \in V \wedge (v,w) \in E_{\rightarrow}\}$$
$$N^{\text{in}}(v) := \{w \mid w \in V \wedge (w,v) \in E_{\rightarrow}\}$$

$N^{\text{out}}(v)$ is the set of nodes with an incoming ordering edge from v. $N^{\text{in}}(v)$ is the set of nodes that have an outgoing ordering edge to v. For a set of nodes V the incoming and outgoing set are defined as $N^{\text{out}}(V) := \bigcup_{v \in V} N^{\text{out}}(v)$ and $N^{\text{in}}(V) := \bigcup_{v \in V} N^{\text{in}}(v)$ respectively.

In contrast to an imperative process language like BPMN, ORG is a declarative description and not necessarily fully specified.

Process Tree

We want to generate the process model in the form of a process tree (PT). In contrast to a graph-based process models, the process tree has two important characteristics. First, it can be easily transformed into an executable process language, see [MMB14b]. Second, a process tree is sound by default [Kop+09]. This means the following: First, the process will terminate properly. Second, for each task at least one process instance contains it. Each *process tree* $PT = (\mathcal{V}, \mathcal{E})$ is an ordered tree, thus a rooted tree for which an ordering is specified for the children of each vertex. $\mathcal{V}$ consists of leaf nodes $\mathcal{V}_t$ and inner nodes $\mathcal{V}_c$, $\mathcal{V}_t \cup \mathcal{V}_c = \mathcal{V}$, $\mathcal{V}_t \cap \mathcal{V}_c = \emptyset$. Each leaf node corresponds to a task, and each inner node corresponds to a control structure. In this paper we consider three control structures, namely sequence SEQ, parallel AND and exclusive XOR. These control structures correspond to the basic control workflow patterns, see [Aal+03b]. This study focuses on the synthesis of process models without cycles. Hence, we do not define a loop operator. It is possible to model the commissioning processes using those control structures. Each control structure can be translated to another block-oriented language, e. g., WS-BPEL, OTX, or to a graph-oriented process language, e. g., Petri nets, BPMN.

Table 6.1: Four Different Declarative Specification Language and the Patterns they can Represent, Directly (Black Mark) or by Combination (Gray Mark).

	Existence	**Resp. Existence**	**Not co-existence**	Bounded Existence	**Response**	**Precedence**	**Succession**	Chain Ordering	Data Constraints
BPMN-Q [ADW08]	✓				✓	✓	✓		
Comp. Rule Graph [Ly+11a]	✓	✓	✓		✓	✓	✓		✓
Declare [AP06]	✓	✓	✓	✓	✓	✓	✓	✓	
Prop. Spec. Pattern [DAC98]	✓	✓	✓	✓	✓	✓	✓	✓	

6.2.2 Generating an Ordering Relation Graph

In this subsection, we explain how to generate an ORG from a declarative specification, e. g., Declare, Compliance Rule Graphs or BPMN-Q. In our application domain, each task is executed exactly once and for simplicity we consider here the core set of specification elements that are supported by the majority of graphical specification languages, see Table 6.2.2.

Figure 6.6 shows the core set of these elements and the representation for Declare. The study of [RFA12] indicates a similar set of mostly used specification elements. Empirical studies, e. g., [DAC99] as well as our experience in [MMB14a] show that this core set is sufficient for the majority of the specifications. Bounded existence and chain ordering only occur rarely. A study in [DAC99] reveal that only 10 out of 555 models contains Bounded existence and chain ordering patterns.

We describe our algorithm for Declare, for Compliance Rule Graphs or BPMN-Q the algorithm would function equivalently. We start with a set of tasks $\mathcal{T}_c$ the commissioning process has to comprise for the context c. Next, we check if a task $t \in \mathcal{T}_c$ has an outgoing response, succession, responded existence or co-existence edge to a task t_2 not in $\mathcal{T}_c$. If so, we add t_2 to $\mathcal{T}_c$. Our algorithm repeats these steps until no further change happens, see Step 1–3. For each task we generate a node in the ORG, see Step 7–9. If an ordering edge, an edge with an arrow, exists

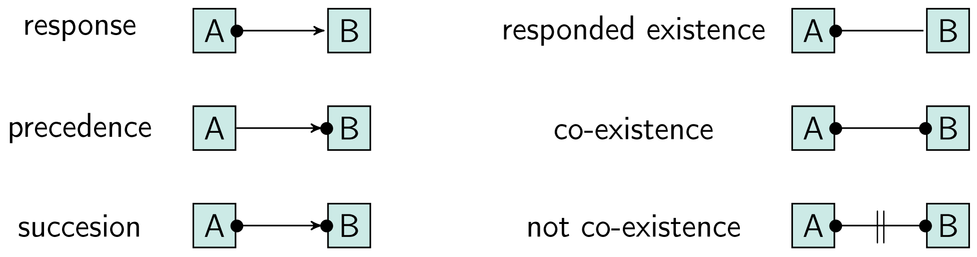


Figure 6.6: Core Set of the Declare Elements

in the Declare graph between nodes $v, w \in \mathcal{T}_c$ we add an edge $(v, w) \in E_{\rightarrow}$ to the ORG, see Step 10–12. At last, the algorithm delivers the ordering relation graph ORG, see Step 13.

Algorithm 12 generateORG (Declare graph DG, task set $\mathcal{T}_c$) : ORG

```
1: while $t \in \mathcal{T}_c$ has an activation edge to a task $t_2 \notin \mathcal{T}_c$ do
2:     $\mathcal{T}_c \leftarrow \mathcal{T}_c \cup \{t_2\}$
3: end while
4: for all $t \in \mathcal{T}_c$ do
5:     Add a state $t$ to ORG
6: end for
7: for all response, precedence or succession edges between tasks $t_1, t_2 \in \mathcal{T}_c$ do
8:     Add an edge $(t_1, t_2) \in E_{\rightarrow}$ to ORG
9: end for
10: return The ordering relation graph ORG
```

Example 6.2.5 . We want to generate the ordering relation graph for the Declare graph in Figure 6.8(a). The commissioning requires executing the tasks B and E, highlighted in dark red in Figure 6.8(a). E needs the occurrence of C, and C the precedence of task D. The algorithm extends the set of tasks for the response and precedence pattern, see Figure 6.8(b)–(c). Finally we transform the Declare graph to an ordering relation graph, see Figure 6.8(d).

6.2.3 Modular Decomposition

We want to generate a process tree from the declarative specification, i. e., from the ORG. Let $G = (V, E)$ be such a graph. For any $W \subseteq V$ we say that $G_W(V^W, E^W)$ is the sub-graph induced by W, i. e., $V^W = W$ and $E^W = E \cap$

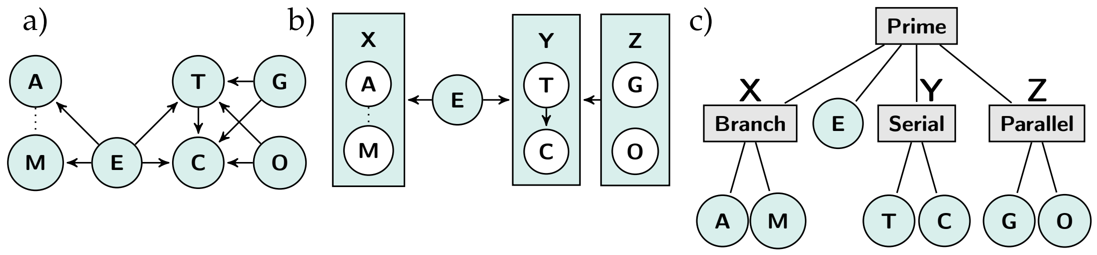


Figure 6.7: An Ordering Relation Graph (a), its Modular Decomposition (b) and the Corresponding Modular Decomposition Tree (c)

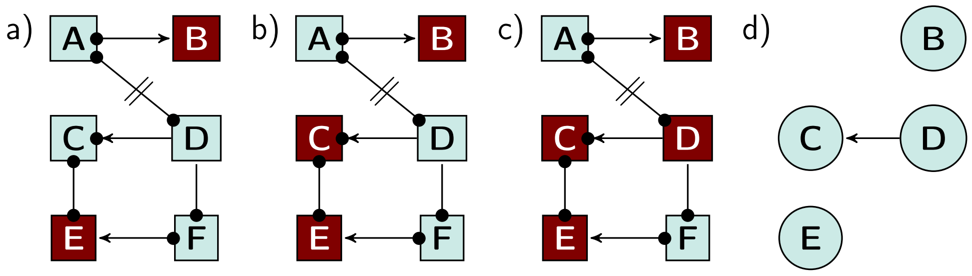


Figure 6.8: Generation of an ORG From a Declare Graph and a List of Tasks $\mathcal{T}_c$

$(W \times W)$. We call W a module iff $\forall v, v' \in W$, $N^{out}(v) \backslash W = N^{out}(v') \backslash W$ and $N^{in}(v) \backslash W = N^{in}(v') \backslash W$. Thus v and v' have identical neighborhoods outside of W. In other words, a module consists of tasks with the same dependencies regarding tasks outside of the module.

Example 6.2.6 . Figure 6.9 shows an ORG with three nodes. The complete graph $\{A, B, C\}$ and all that sets with one element $\{A\}$, $\{B\}$, $\{C\}$ form a (trivial) module.
$\{A, B\}$ is a module because of $N^{out}(A) \backslash \{A, B\} = N^{out}(B) \backslash \{A, B\} = \emptyset$ and $N^{in}(A) \backslash \{A, B\} = N^{in}(B) \backslash \{A, B\} = \{C\}$. $\{A, C\}$ is a module because of $N^{out}(A) \backslash \{A, C\} = N^{out}(C) \backslash \{A, C\} = \{B\}$ and $N^{in}(A) \backslash \{A, C\} = N^{in}(C) \backslash \{A, C\} = \emptyset$. No other module exists in the graph. $\{B, C\}$ is not a module because of $N^{out}(B) \backslash \{B, C\} = \emptyset$ and $N^{out}(C) \backslash \{B, C\} = \{A\}$.

In our use case, a module often consists of tasks operating on the same electronic control unit of the vehicle. W is a strong module if, for each module $W' \subseteq V$, one of the following holds: $W \cap W' = \emptyset$, $W \subseteq W'$, or $W' \subseteq W$.

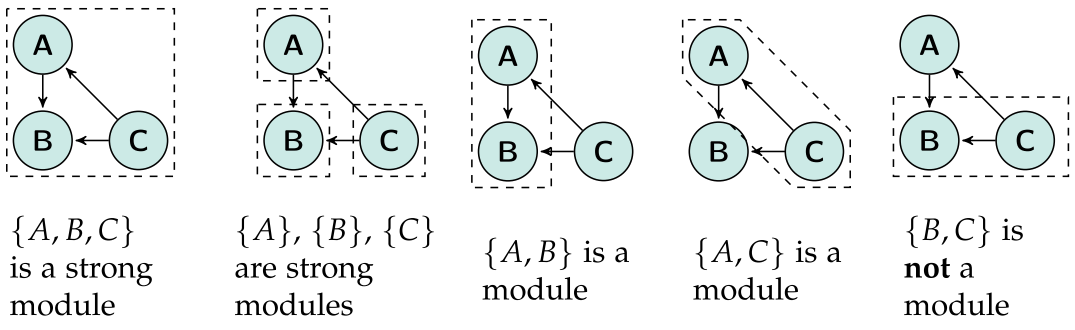

A
B
C
$\{A,B,C\}$ is a strong module
$\{A\}$, $\{B\}$, $\{C\}$ are strong modules
$\{A,B\}$ is a module
$\{A,C\}$ is a module
$\{B,C\}$ is **not** a module

Figure 6.9: All Modules for an ORG with Three Nodes A, B, C.

Example 6.2.7. For the graph in Figure 6.9 the set $W = \{A,B,C\}$ is a strong module because other module W' is a subset of W. The module $W = \{A,B\}$ is not a strong module because for the module $W' = \{A,C\}$, $W \cap W' = \{A\} \neq \emptyset$.

The decomposition of a graph into strong modules is called Modular Decomposition, and the resulting hierarchical structure is called Modular Decomposition Tree (MDT). Figure 6.7(a) shows the simple ordering relation graph of Figure 6.2, its decomposition in four modules, see Figure 6.7(b), and the corresponding modular decomposition tree, see Figure 6.7(c). [MM05] shows that a node W in a MDT with children $S_1, S_2, \ldots, S_k$ is of one of the following:

$$\begin{aligned} \text{Complete :} \quad & \forall I \subset \{1,\ldots,k\},\ \text{with } 1 < |I| < k : \bigcup_{i \in I} S_i \text{ is a module} \\ \text{Prime :} \quad & \forall I \subset \{1,\ldots,k\},\ \text{with } 1 < |I| < k : \bigcup_{i \in I} S_i \text{ is not a module} \end{aligned}$$

Example 6.2.8. The graph in Figure 6.10(a) is a prime module, i. e., none of the subsets W of $\{A,B,C,D\}$ with $1 < |W| < 4$ is a module. The graphs in Figure 6.10(b)–(e) are complete, i. e., all subsets are modules.

A complete module W with the induced graph $G_W(V^W, E^W)$ either does not contain any edges or is a clique in $E^W_{\#}$ or $E^W_{\rightarrow}$, see the proof of Lemma 5. A complete module can easily be transformed to a process tree deterministically, see [Pol12]. For a prime module our approach will use a heuristic optimization.

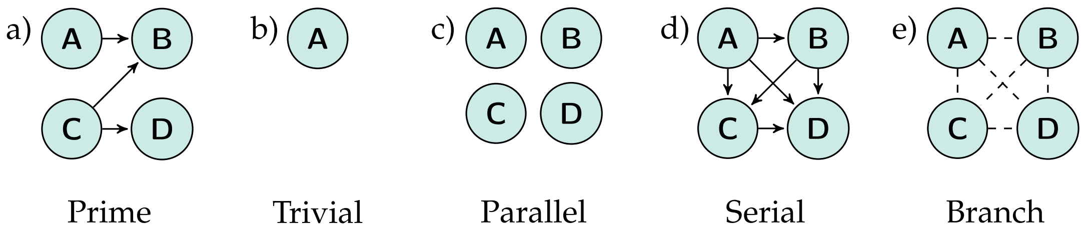


Figure 6.10: An Example for a Prime, Trivial, Parallel, Serial, and Branch Module.

Lemma 5. A strong complete module W is of exactly one of four types:

trivial	:	$\lvert V^W\rvert = 1$
serial	:	For every $v, v' \in V^W : (v, v') \in E^W_{\rightarrow} \vee (v', v) \in E^W_{\rightarrow}$. Recall that the edges in $E^W_{\rightarrow}$ are cycle-free.
branch	:	For every $v, v' \in V^W : (v, v') \in E^W_{\#}$
parallel	:	For every $v, v' \in V^W : (v, v') \notin E^W$

Proof 4 (of Lemma 5). For $|V^W| = 1$ the module is trivial. For $|V^W| = 2$ the pair (v, v') is either $(v, v') \in E^W_{\rightarrow}$ (serial), $(v, v') \in E^W_{\#}$ (branch), or $(v, v') \notin E^W$ (parallel). For $|V^W| \geq 3$ there exist two pairs of nodes (v_1, v_2), (v_3, v_4) with $(v_1, v_2) \neq (v_3, v_4)$. At least one element in the pair differs. Without loss of generality let $v_1 \neq v_3$. Each set of two elements is a module, and as shown earlier it is either a serial, branch, or parallel. We call this the *type* of the pair, namely $type_{v_1,v_2}$. Assume that the lemma is false. Then at least two pairs of nodes exist with $type_{v_1,v_2} \neq type_{v_3,v_4}$. W is a strong module and thus all combinations of child elements are strong modules. The set of nodes $\{v_1, v_3\}$ is a strong module $\Rightarrow N(v_1) = N(v_3) \Rightarrow type_{v_2,v_3} = type_{v_1,v_2}$. If $v_2 = v_4$ this is a contradiction to the assumption, so let $v_2 \neq v_4$. The set of nodes $\{v_2, v_4\}$ is a strong module $\Rightarrow N(v_2) = N(v_4) \Rightarrow type_{v_3,v_4} = type_{v_2,v_3} \Rightarrow type_{v_3,v_4} = type_{v_1,v_2}$. This is a contradiction to the assumption.

Example 6.2.9. Figure 6.10(b)–(e) shows an example for a trivial, parallel, serial, and branch module.

[MM05; CH94] prove that the decomposition of a directed graph (V, E) can be done in $O(|V| + |E|)$, thus in time linear with the size of the graph. We use the MDT to transform the ORG into a process tree.

Algorithm 13 synthesize(ORG G, context c): Process Tree PT

```
1: Determine $\mathcal{T}_c$ from $c$
2: $G \leftarrow$ subgraph $G_W$ of $G$ with the nodes $W = \mathcal{T}_c$
3: $PT \leftarrow$ Modular Decomposition of $G$
4: for all prime nodes $\mathcal{P} \in PT$ do
5:     Process tree PT$_{\mathcal{P}} \leftarrow$ $synPrime(\mathcal{P})$
6:     Replace $\mathcal{P}$ with $PT_{\mathcal{P}}$
7: end for
8: for all leaf nodes $l \in PT$ do
9:     if $l$ is a partition leaf node then
10:        $G_l \leftarrow$ ORG of $l$
11:        Process tree PT$_l \leftarrow$ $synthesize(G_l, c)$
12:        Replace $l$ with $PT_l$
13:    end if
14: end for
15: return $PT$
```

6.2.4 Overview of the Automatic Generation

Our goal is to automatically generate a process model from a declarative description. Algorithm 13 synthesizes a process tree from an ORG and a context c. The context c determines the required tasks $\mathcal{T}_c$, see Line 1. We then reduce the ORG G to the subgraph G_W with the nodes $W = \mathcal{T}_c$, see Line 2. The algorithm then computes a modular decomposition of the ORG, see Line 3 in Algorithm 13.

The resulting modular decomposition tree (MDT) may contain both complete and prime components. For complete components, a transformation to process fragments exists, cf. [Pol12]. For a prime component in turn, several fragments are possible, see Figure 6.11. In other words, each prime component stands for an under-specified region. For each prime component $\mathcal{P}$, we use a probabilistic optimization to find a solution , see Line 5. We replace $\mathcal{P}$ with the solution found, see Line 6.

synPrime() splits the ORG of the prime components into partitions. It generates a graph with one node for each of these partitions. The algorithm recursively calls itself, in order to replace each node with a subtree. Finally, our approach transforms the process tree into a process language, e. g., BPMN, WS-BPEL.

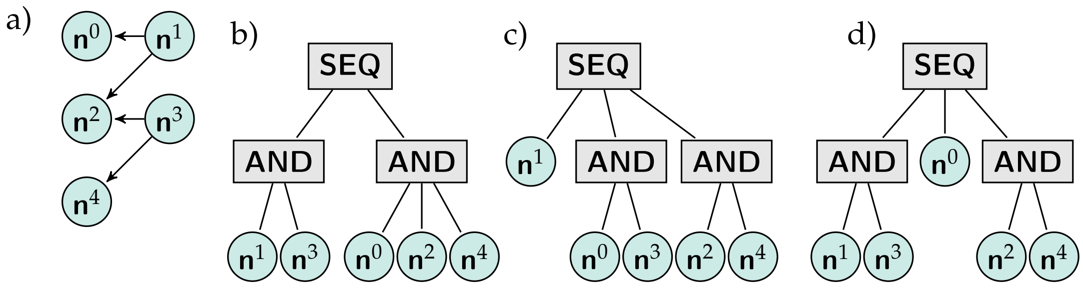


Figure 6.11: The Neighborhood Graph to Directly Generate a Process Tree (a), three Possible Process Trees (b), (c), (d) for the Graph (a).

6.2.5 Under-Specified Regions

Each prime component $\mathcal{P}$ induces a graph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$. $V_{\mathcal{P}}$ denotes the set of strong components that belong to $\mathcal{P}$. Figure 6.7 shows that the graph $G_{\mathcal{P}}$ for the prime component $\mathcal{P}$ consists of $V_{\mathcal{P}} = \{\mathsf{X}, \mathsf{R}, \mathsf{Y}, \mathsf{Z}\}$ with $E_{\mathcal{P}} = \{(\mathsf{R} \to \mathsf{X}), (\mathsf{R} \to \mathsf{Y}), (\mathsf{Z} \to \mathsf{Y})\}$. $\mathcal{P}$ is not fully specified and thus no unique corresponding process tree exists. Due to the large number of possible process models for a prime graph $G_{\mathcal{P}}$ it is not feasible to construct every possible one.

The modular decomposition detects the fully specified and the under-specified regions of the process. Even for small prime components it is not possible to generate and test all possible process models. Our overall idea is to reduce the size of the graph induced by a prime component iteratively until the number of remaining solutions is low (< 100). So that we can solve the problem, see Figure 6.11. Our intuition for the reduction is to select a pivot node v and detect which nodes (V_1) must occur before v, and which nodes (V_2) can be scheduled in parallel to v. V_1 as well as V_2 imply two smaller ordering graphs. We repeat this steps with several different pivot nodes. Our approach randomly selects a node $v \in V_{\mathcal{P}}$ with $N^{out}(v) = \emptyset$ as pivot node. Lemma 8 will show why we need this characteristic. The ORG $G_{\mathcal{P}}$ is cycle-free, and thus a node v with $N^{out}(v) = \emptyset$ always exists.

Definition 22 (Zero Neighborhood).

The zero neighborhood of a pivot node v is $N^{(0)}(v) := \{v\}$, $N^{(1)}(v) := N^{in}(v)$. For $i \in \mathbb{N}$, $i > 1$ we define the i-neighborhood as:

$$N^{(i)}(v) := \begin{cases} \left(\bigcup_{v' \in N^{(i-1)}(v)} N^{out}(v')\right) \setminus N^{(i-2)}(v) & \text{if } i \in \{2,4,6,\ldots\} \\ \left(\bigcup_{v' \in N^{(i-1)}(v)} N^{in}\ (v')\right) \setminus N^{(i-2)}(v) & \text{if } i \in \{3,5,7,\ldots\} \end{cases}$$

Lemma 6. The neighborhoods for a pivot node v of a prime component $G_{\mathcal{P}}(V,E)$ contain each node exactly once. This means that:

1. $\bigcup_{i \in \mathbb{N}} N^{(i)}(v) = V$
2. $\forall i, j \in \mathbb{N}, i \neq j \ :\ N^{(i)}(v) \cap N^{(j)}(v) = \emptyset$

Proof 5 (of Lemma 6).

1. Assume that $G_{\mathcal{P}}$ is not connected $\Rightarrow$ two sub-graphs exists G' with the nodes in the connected graph of v and $G'' := G_{\mathcal{P}} \setminus G'$. This is a contradiction to the assumption that $G_{\mathcal{P}}$ is a prime, because G' and G'' would form a component. Let $w \in V_{\mathcal{P}}$ be a node, with $w \neq v$. $G_{\mathcal{P}}$ is connected thus a shortest undirected path $p = (a_0, a_1, a_2, \ldots, a_{n-1}, a_n)$ exists between $v = a_0$ and $w = a_n$. p is alternating thus $\forall i \in [0,n]$: $(a_i, a_{i+1}) \Leftrightarrow (a_{i+2}, a_{i+1})$, or a shorter path would exist. $a_i \in N^{(i)}(v)$ and thus $w \in N^{(n)}(v)$.

2. Assume $N^{(i)}(v) \cap N^{(j)}(v) \neq \emptyset$ then a node w exists with $w \in N^{(i)}(v) \wedge w \in N^{(j)}(v)$. This means that two shortest undirected paths would exist between w and v. This is a contraction because only one path can be shortest.

We use the neighborhood information for the partition of the graph. Each partition $n^{(i)}$ is a subgraph of the ORG $G_{\mathcal{P}}$ with the nodes $N^{(i)}(v)$. In other words, the partitioning implies a graph G_v where each $n^{(i)}$ is a node. We refer to this graph as the neighborhood graph. Formally, given a pivot node v, the neighborhood graph $G_v = (V_v, E_v)$ is as follows

$$V_v \ = \ \{n^i \mid N^{(i)}(v) \neq \emptyset\}$$

$$\begin{aligned} E_v \;=\; & \{(n^i, n^{i+1}) \mid i \in \{1,3,\ldots\} \;\wedge\; n^i, n^{i+1} \in V_v\} \;\cup \\ & \{(n^{i+1}, n^i) \mid i \in \{0,2,\ldots\} \;\wedge\; n^i, n^{i+1} \in V_v\} \end{aligned}$$

The graph contains each non-empty neighborhood as a node.

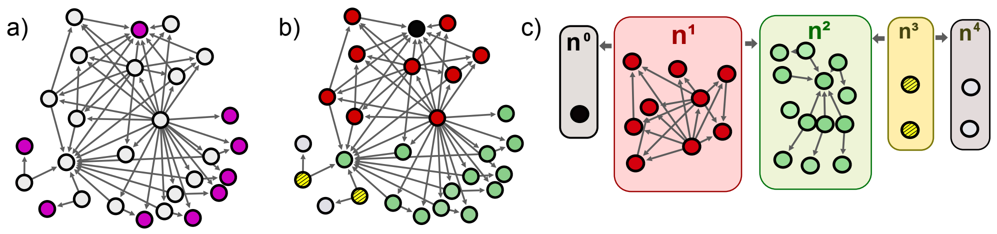


Figure 6.12: A Prime Component (a), its Partitioning (b), and the Neighborhood Graph (c)

Example 6.2.10. For the graph in Figure 6.7(b) and the pivot Y the neighborhoods are: $N^{(0)}(\mathsf{Y}) = \{\mathsf{Y}\}$, $N^{(1)}(\mathsf{Y}) = \{\mathsf{R},\mathsf{Z}\}$, $N^{(2)}(\mathsf{Y}) = \{\mathsf{X}\}$, and for $i > 2$ $N^{(i)}(\mathsf{Y}) = \emptyset$. The neighborhood graph $G_\mathsf{Y}(V_\mathsf{Y}, E_\mathsf{Y})$ for the pivot Y is:

$$G_\mathsf{Y} = (\ \{n^0, n^1, n^2\}\ ,\ \{\ (n^1, n^0)\ ,\ (n^1, n^2)\ \}\ )$$

Example 6.2.11. Figure 6.12(a) shows a more complex graph that is a prime component, i. e., there is no unique corresponding tree. The possible pivot nodes are in violet. The pivot node at the top of Figure 6.12(a) leads to the partitioning in Figure 6.12(b). Figure 6.12(c) shows the respective neighborhood graph.

Lemma 7. The partitioning into the neighborhood graph for a pivot v preserves all order dependencies. In other words, for each edge $(v_1, v_2) \in E_\mathcal{P}$, one of the following holds:

(a) $\exists i \in \mathbb{N}_0 : v_1, v_2 \in N^{(i)}(v)$

(b) $v_1 \in N^{(i)}(v), v_2 \in N^{(j)}(v), i \neq j \Rightarrow (n^i, n^j) \in E_v$

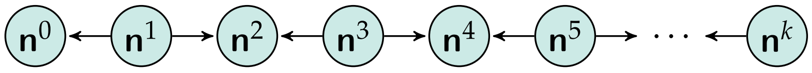


Figure 6.13: Structure of the Neighborhood Graph

Proof 6 (of Lemma 7). If $v_1, v_2 \in N^{(i)}(v)$ then the edge is in the subgraph for $N^{(i)}(v)$. We now focus on (b).
CASE 1 – $i \in \{1,3,5,\ldots\}$: If $j = i-1$ then $(n^i, n^j) \in E_v$. Otherwise, if $j \neq i-1$ then $v_2 \in N^{out}(v_1) \wedge v_2 \notin N^{(i-1)}(v) \Rightarrow v_2 \in N^{(i+1)}(v) \Rightarrow j = i+1$, and $(n^i, n^j) \in E_v$.
CASE 2 – $i \in \{0,2,4,\ldots\}$: In this case, a node v_3 exists with $(v_3, v_1) \in E_{\mathcal{P}}$. The transitivity of $E_{\mathcal{P}}$ leads to $(v_3, v_2) \in E_{\mathcal{P}}$. $v_2, v_1 \in N^{out}(v_3) \Rightarrow v_1, v_2 \in N^{(i)}(v) \Rightarrow i = j$, i. e., there is a contradiction to Lemma 7.

Lemma 7 states that our approach does not lose any dependencies. A symmetric solution would be to select pivots with $N^{in}(v) = \emptyset$ and change the definition of the neighborhood accordingly. However, a pivot v with $N^{out}(v) \neq \emptyset \wedge N^{in}(v) \neq \emptyset$ would lose a dependency, see Lemma 8.

Lemma 8. The neighborhood graph G_v for a pivot node v with $N^{out}(v) \neq \emptyset \wedge N^{in}(v) \neq \emptyset$ does not preserve the order dependencies.

Proof 7 (of Lemma 8). Let v with $N^{out}(v) \neq \emptyset \wedge N^{in}(v) \neq \emptyset$ be the pivot node. Then a node $v^- \in N^{in}(v)$ and a node $v^+ \in N^{out}(v)$ exists. The ORG is cycle free, thus $v^+ \neq v^- \neq v$. The following dependencies, i. e., elements of $E_{\mathcal{P}}$, exist: (v^-, v), (v, v^+), and, because of the transitivity of the ORG, $(v^-, v^+) \in E_{\mathcal{P}}$. $v \in N^{(0)}(v)$, $v^+ \in N^{(1)}(v)$, $v^- \in N^{(2)}(v)$ altogether imply that $(v^-, v) \in E_{\mathcal{P}}$ and $(n^2, n^0) \notin E_v$. This means that the dependencies between v and v^- have been lost.

Algorithm 14 synPrime (Neighborhood Graph $G(V, E)$) : ProcessTree PT

```
1: Pivot v ← randomly select a node $v \in G$ with $N^{out}(v) = \emptyset$
2: $G_v(V_v, E_v)$ ← calculate neighborhood of $v$
3: if $N^{(\lambda)} = \emptyset$ then
4:     return (select tree pattern randomly)
5: else
6:     return $synPrime(G_v)$
7: end if
```

Algorithm 14 generates a process tree for an under-specified region, i. e., a prime component. First, the algorithm randomly selects a pivot node v, see Line 1 and calculates its neighborhood graph G_v, see Line 2. The parameter $\lambda \in \mathbb{N}^+$ defines when the neighborhood graph is small enough to generate a process tree. If the neighborhood graph is too large, the algorithm calls *synPrime()* again, and everything is repeated until the graph is processable. Figure 6.14 shows the reduction of a neighborhood graph. If our approach selects n^2 as the pivot element then it builds the smaller graph on the right hand side.

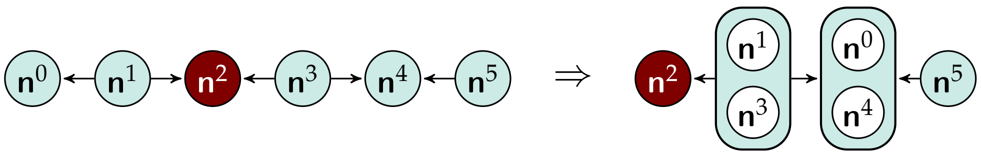


Figure 6.14: Reduction of a Neighborhood Graph with the Pivot n^2.

If the neighborhood graph is small enough, i. e., $N^{(\lambda)} = \emptyset$), Algorithm 14 randomly selects a tree pattern for it, see Line 4. A tree pattern is a process tree for the neighborhood graph. The neighborhood graph in Figure 6.11(a) contains 5 nodes and 4 edges. For a graph with five nodes thousands of process trees are possible. For the graph in Figure 6.11(a) 53 trees are possible, given the constraints. For most of these 53 process trees, there is another tree with a lower overall processing time, for any processing times of the tasks. If we exclude these dominated trees, three trees remain. Figures 6.11(b) and (c) show two of them, randomly selected. The tree patterns define which additional dependencies have to be added to generate a block based process model for the specification. Section 6.2.6 shows and explains all tree patterns for $\lambda \in [1, 5]$. Figure 6.11(d) shows a process tree fulfilling the constraints in Figure 6.11(a), but the processing time of the tree in Figure 6.11(b) is always shorter.

Example 6.2.12 . The Figure 6.15(a) shows the modular decomposition of the introduction of the example from Figure 6.2. The prime node is small enough to directly select a tree pattern. Our algorithm selects the tree pattern shown in Figure 6.15(b) leading to the process tree in Figure 6.15(c). This process tree can be easily transformed into the BPMN process in Figure 6.15(d).

For each ORG we have started out with, we calculate κ different process trees. The resulting trees differ depending on the probabilistic choices in Algorithm 14, see Line 1 and Line 4. We select the best process tree found according to quality criteria, e. g., the processing time. We calculate a quality value of each

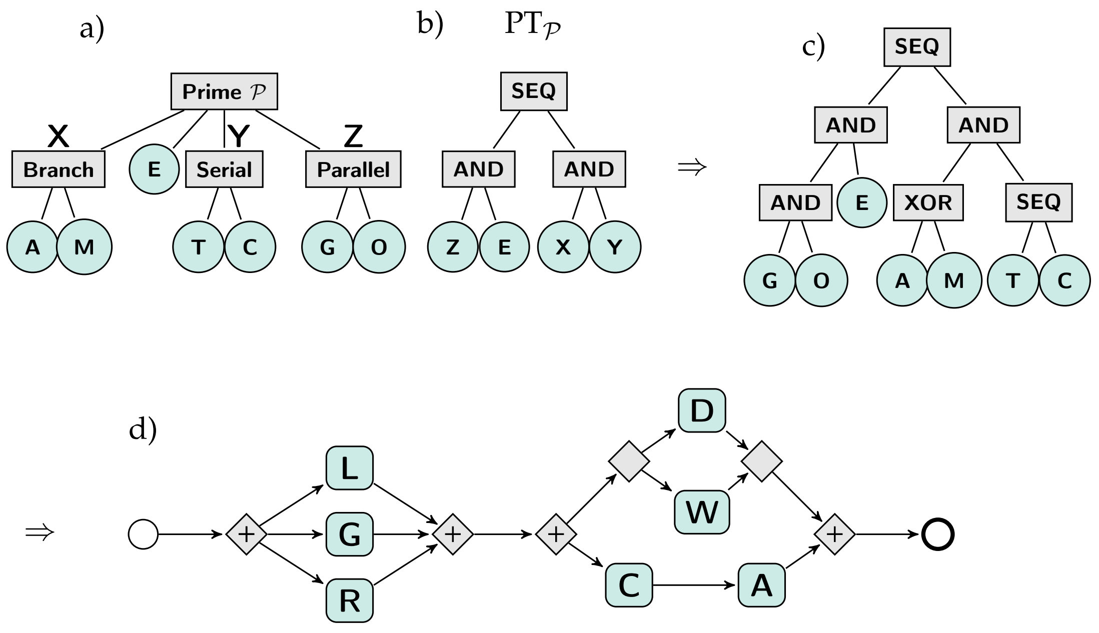


Figure 6.15: The MDT for the ORG of Figure 6.7 (a), the Process Tree for the Prime Node $PT_{\mathcal{P}}$ (b), the Resulting Process Tree (c) and the Process Model in Bpmn Notation (d).

tree as follows. The average processing time for each node in a process tree $PT(\mathcal{V}, \mathcal{E})$ is calculated recursively with function $fit \colon \mathcal{V} \to \mathbb{R}$.

$$fit(n) := \begin{cases} runtime(n) & \text{if } type(n) = task \\ \max_{c \in child_n} fit(c) & \text{if } type(n) = \textsc{and} \\ \sum_{c \in child_n} fit(c) & \text{if } type(n) = \textsc{seq} \\ \max_{c \in child_n} fit(c) & \text{if } type(n) = \textsc{xor} \end{cases}$$

$type \colon \mathcal{V} \to \{task, \textsc{and}, \textsc{seq}, \textsc{xor}\}$ is a function to determine the type of the tree node. $child_n := \{c \mid (n, c) \in \mathcal{E}\}$ is the set of nodes in the process tree with parent node n. The estimation for the xor-Split is a worst case analysis, i.e., the processing time is smaller than the estimated. If the probabilities of the splits are known a priori a more precise average case assumption is possible, see [Yan+12]. The fitness of a process tree $fit(PT)$ is the fitness of its root node. The algorithm returns the process tree with the highest fitness value. The resulting process tree can easily be transformed to the notation required.

6.2.6 Tree Patterns

In this section we explain the tree patterns for a neighborhood graph G_v. For all possible runtimes of nodes in the neighborhood graph at least one of the tree patterns is optimal according to the fitness function *fit*. To minimize the fitness value we are trying to set tasks in parallel.

$\lambda = 2$ and $\lambda = 3$

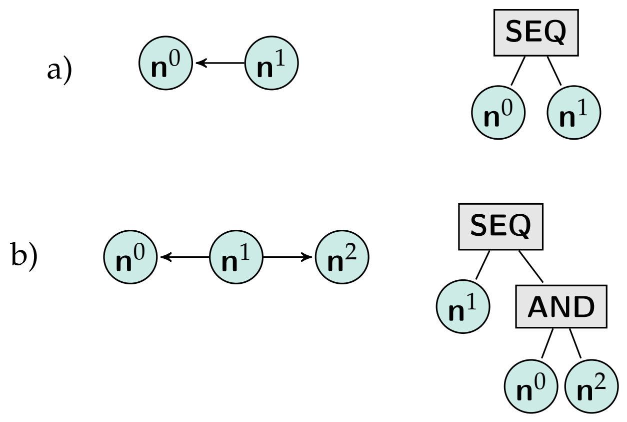


Figure 6.16: Tree Patterns for $\lambda = 2$ (a) and $\lambda = 3$ (b).

For $\lambda = 2$ only one process tree is possible that fulfills the requirements. Figure 6.16(a) shows the ORG for $\lambda = 2$ and the only possible process tree. For $\lambda = 3$ it is only possible to execute n^0 and n^2 in parallel. This parallel arrangement leads to one possible process tree, see Figure 6.16(b).

$\lambda = 4$

For $\lambda = 4$ we can set in parallel three pairs of nodes (n^0, n^2), (n^0, n^3), (n^1, n^3). If we set (n^0, n^2) in parallel, it allows setting (n^1, n^3) in parallel and leads to the pattern in Figure 6.17(a). The dotted line shows the dependency added according to the tree pattern. If we set (n^0, n^3) in parallel, we have to add the two dependencies in Figure 6.17(b). This leads to the second tree pattern.

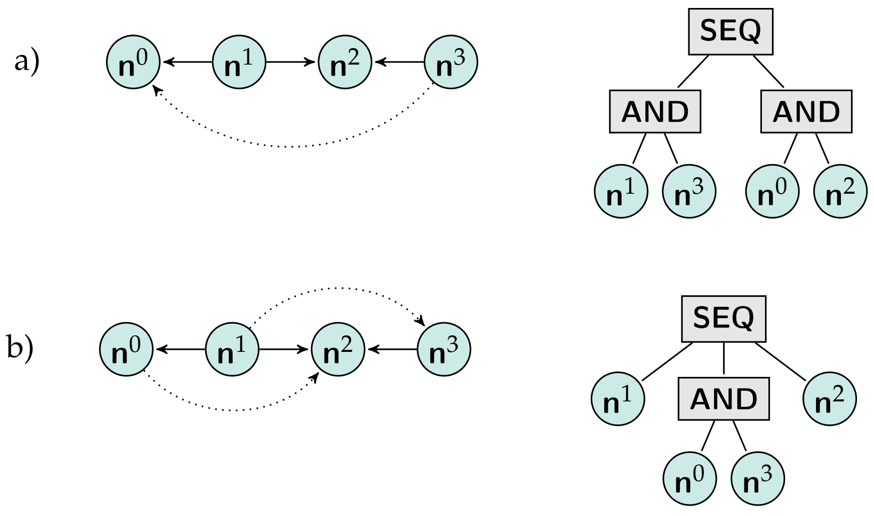


Figure 6.17: The Two Tree Patterns for $\lambda = 4$

$\lambda = 5$

For $\lambda = 5$ we can set five pairs of nodes in parallel (n^0, n^2), (n^0, n^3), (n^0, n^4), (n^1, n^3), (n^1, n^4), and (n^2, n^4). If we set (n^0, n^2) in parallel, then we additionally can set in parallel either (n^1, n^3) or (n^1, n^4). If we choose (n^1, n^3) we can add n^4 to the parallel (n^0, n^2) resulting in the process tree of Figure 6.18(a). If we choose (n^1, n^4) no other parallelism is possible resulting in the process tree of Figure 6.18(b).

We have implemented the algorithms in C#. The program receives the ORG as input, see [MMB14b] on how to generate an ORG from a declarative specification. The output of the program is a process tree that is then transformed to the commissioning process notation OTX by a proprietary XSLT script written by us. The implementation can handle specifications with several hundreds of tasks and thousands of dependencies in a few minutes, see Section 6.4.

6.3 Case Study – Compact Executive Car

We applied our synthesis framework to support the development of the new commissioning process for a novel compact executive car series.

During the use case some aspects of the modeling arose that cannot directly be handled by our approach. To this end, we designed several heuristics that

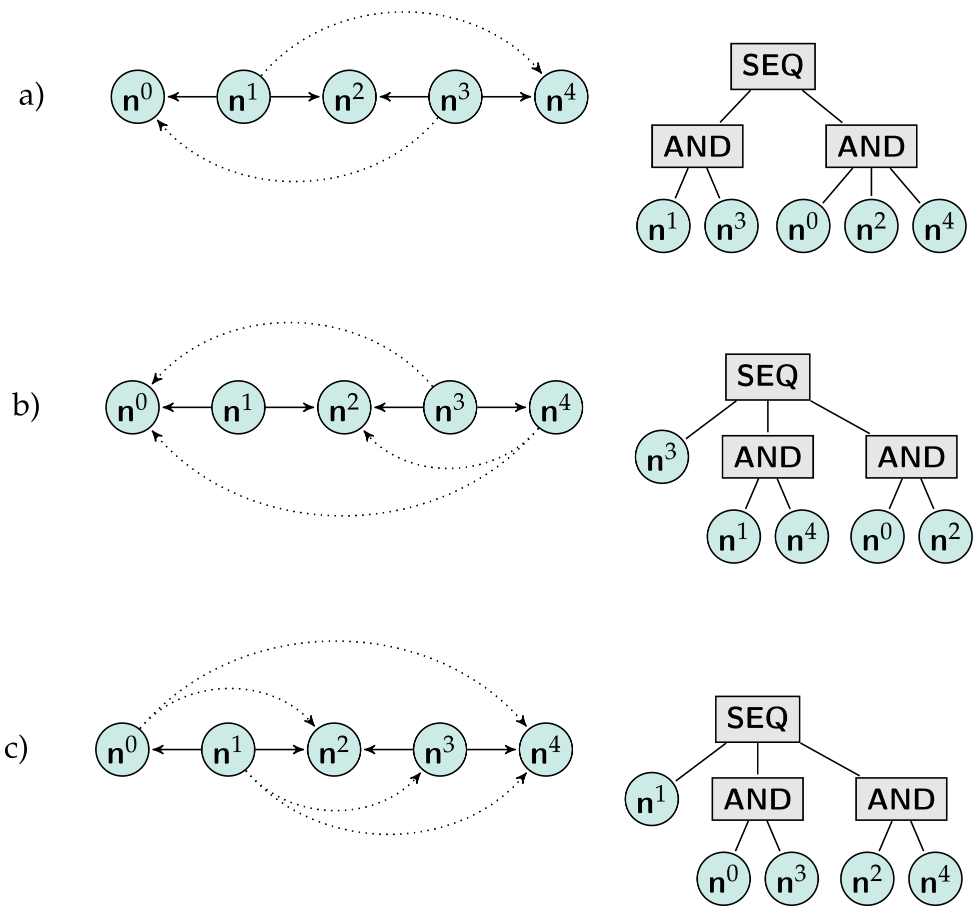


Figure 6.18: The Three Tree Patterns for $\lambda = 5$

either add dependencies before the execution, see Subsection 6.3.1 and 6.3.2, or are applied as a post-processing after the synthesis, see Subsection 6.3.3 and 6.3.4.

6.3.1 Arranging the Data Writing Tasks

To configure the electronic control units the software has to be installed on the electronic control units. This requires a vast amount of data communicated over a bus protocol with a low data rate. These configuration tasks, i. e., ZDC schreiben, are the most time consuming for the new commissioning processes. To use the communication protocol to its full potential only four parallel connections should be used for the data transfer.

To this end, we want to apply a post-processing of the commissioning process that schedules the tasks more sequentially. The idea of our approach is to combine two parallel lanes in each iteration until the constraint is fulfilled.

Algorithm 15 describes this heuristic. For each iteration of the algorithm we collect the possible candidates for parallel lanes, see Line 4. We call the set of all possible candidates $\mathcal{C}$. A candidate $c \in \mathcal{C}$ is a pair of two parallel lanes on any hierarchy level of the process model. For each candidate $c \in \mathcal{C}$ we calculate the reduction of the resources if we combine the two lanes $\Delta_{\text{res}}(c)$. Additionally we calculate how the candidate would extend the processing time $\Delta_{\text{run}}(c)$ and the human worker time Δ_{hum}. If the combination of the two lanes reduces the process model under the limit for the resource we set Δ_{res} to the limit of the resource. For instance if we assume that the process model before the reduction uses 5 resources and the limit of the resources is 4, then Δ_{res} is at most 1. Next, we calculate the gain of the candidate with $g(c) = \frac{\Delta_{\text{res}}(c)+\lambda}{\sigma\Delta_{\text{hum}}(c)+\Delta_{\text{run}}(c)+\lambda}$ (Line 5). σ is a constant with $\sigma > 1$ that ensures that the human processing time is considered more important than the total processing time. λ is a constant with $\lambda > 0$ to adjust the quality and performance, i. e., a larger λ leads to a broader search with a possible better quality result but if a longer runtime. Our goal is to select candidates with a high gain to efficiently reduce the resource consumption. Unfortunately, the selection of a candidate influences the values of the other candidates subsequently. The selection of the best candidate at one step could lead to a semi-optimal end result. To this end, we randomly select the candidates and iterate the algorithm several times to find the best arrangement. A completely random selection could lead to a long runtime of our algorithm. We resolve this by weighting the probability of the candidates by their gain with the probability mass function $p(c)$, see Line 10-12.

Lemma 9 . $p(c) = \frac{g(c)-\min+\epsilon}{\sum_{c'\in\mathcal{C}} g(c')-|\mathcal{C}|(\min-\epsilon)}$ is a probability mass function.

Proof 8 (of Lemma 9) . For this we need to proof $\sum_{c\in\mathcal{C}} p(c) = 1$.

$$\sum_{c\in\mathcal{C}} p(c) = \sum_{c\in\mathcal{C}} \frac{g(c)-\min+\epsilon}{\sum_{c'\in\mathcal{C}} g(c')-|\mathcal{C}|(\min-\epsilon)} = \frac{\sum_{c\in\mathcal{C}} g(c)-\min+\epsilon}{\sum_{c'\in\mathcal{C}} g(c')-|\mathcal{C}|(\min-\epsilon)}$$

$$= \frac{\sum_{c\in\mathcal{C}} g(c)+|\mathcal{C}|(-\min+\epsilon)}{\sum_{c'\in\mathcal{C}} g(c')-|\mathcal{C}|(\min-\epsilon)} = \frac{\sum_{c\in\mathcal{C}} g(c)-|\mathcal{C}|(\min-\epsilon)}{\sum_{c'\in\mathcal{C}} g(c')-|\mathcal{C}|(\min-\epsilon)} = 1$$

The factor ϵ in the probability function p allows us to scale how much the gain

Algorithm 15 HeuristicZDCschreiben (ProcessTree *original*): ProcessTree *best*

```
1: while number of iterations is not fulfilled do
2:     ProcessTree $tmp \leftarrow original$.copy
3:     while resource constraint not fulfilled do
4:         Calculate all candidates $\mathcal{C}$ of parallel lanes
5:         for all $c \in \mathcal{C}$ do
6:             Calculate $\Delta_{\text{res}}(c)$, $\Delta_{\text{run}}(c)$, and $\Delta_{\text{hum}}(c)$
7:             $$g(c) = \frac{\Delta_{\text{res}}(c) + \lambda}{\sigma\Delta_{\text{hum}}(c) + \Delta_{\text{run}}(c) + \lambda}$$
8:         end for
9:         $\textsc{min} \leftarrow \min\left(g(c) \mid c \in \mathcal{C}\right)$
10:        for all $c \in \mathcal{C}$ do
11:            $$p(c) = \frac{g(c) - \textsc{min} + \epsilon}{\sum_{c' \in \mathcal{C}} g(c') - |\mathcal{C}|(\textsc{min} - \epsilon)}$$
12:        end for
13:        $rand \leftarrow$ Random Number between 0 and 1
14:        $last \leftarrow 0$
15:        for all $c \in \mathcal{C}$ do
16:            if $rand > last \vee rand \leq last + p(c)$ then
17:                $choosen \leftarrow c$
18:                break
19:            else
20:                $last \leftarrow last + p(c)$
21:            end if
22:        end for
23:        ProcessTree $tmp \leftarrow$ combine the two lanes of the candidate $choosen$
24:    end while
25:    if $tmp$ has a larger processing time than $limit$ then
26:        continue
27:    end if
28:    if $tmp$ has shorter processing time than $best$ then
29:        $best \leftarrow tmp$
30:    end if
31: end while
32: return $best$
```

influence the probability. Next in Line 13 we select a candidate $c \in \mathcal{C}$ according to the probability function p and combine the lanes of candidate c (Line 14). We iterate (Line 3–15) until the resource constraint is fulfilled. In Line 16 we check if the found solution *tmp* has a shorter overall processing time than the best found solution so far. We repeat the algorithm until the maximal number of iterations is reached, see Line 1–19. Other abortion criteria, e. g., maximum computation time would be possible. Last, we return the best found solution for the resource constraint, see Line 20. The problem is similar to the multiprocessor scheduling problem, cf. Coffman:

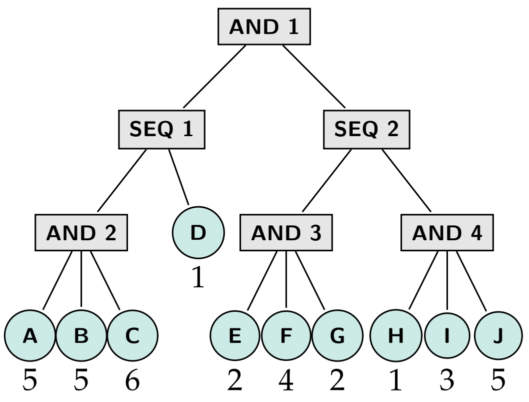
> » Formally, we are given a set $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$ of *tasks*, each task T_i having *length* $l(T_i)$, and a set of $m \geq 2$ identical *processors*. A *schedule* in this case can be thought of as a partition $\mathcal{P} = \langle P_1, P_2, \dots, P_m \rangle$ of $\mathcal{T}$ into m disjoint sets, one for each processor. The i-th processor, $1 \leq i \leq m$, executes the tasks in P_i. [. . .] The *finishing time* for the schedule $\mathcal{P}$ is then given by
>
> $$f(\mathcal{P}) = \max_{1 \leq i \leq m} l(P_i)$$
>
> where for any $X \subseteq \mathcal{T}$, $l(X)$ is defined to be $\sum_{T \in X} l(T)$.
>
> An *optimum* m-processor schedule $\mathcal{P}^*$ is one that satisfies $f(\mathcal{P}^*) \leq f(\mathcal{P})$ all partitions $\mathcal{P}$ of $\mathcal{T}$ into m-subsets. « [CGJ78]

The multiprocessor scheduling is known to be NP-hard, cf. [CGJ78]. In contrast to the multiprocessor scheduling our tasks or lanes cannot be arranged freely. Complex sequential constraints exist limiting the possible scheduling. To this end we could not leverage the work in the multiprocessor scheduling.

Example 6.3.1 . Figure 6.19 shows a process tree. Below the tasks their respective runtime is written. The table on the right hand side shows the gain $g(c)$ for each candidate with $\lambda = 0.1$. The sum of all gains is $\sum_{c' \in \mathcal{C}} g(c') = 3.33$, the minimum is 0.03 and $|\mathcal{C}| = 10$. We choose $\epsilon = 0.01$ to get the probabilities $p(c)$ in the table on the right hand side of Figure 6.19.

6.3.2 Same Electronic Components

With the next heuristic we want to address an ordering of tasks communicating with the same control unit. It is not possible to access the same control unit



Candidate	g(c)	p(c)
(seq 1,seq 3)	0.18	0.051
(A , B)	0.27	0.080
(A , C)	0.22	0.064
(B , C)	0.22	0.064
(E , F)	0.05	0.010
(E , G)	1.00	0.313
(F , G)	0.05	0.010
(H , I)	1.00	0.313
(H , J)	0.09	0.022
(I , J)	0.03	0.003

Figure 6.19: A Process Tree with its Candidates for the Reduction and their Respective Gain $g(c)$ and Probability $p(c)$

more than ones. Tasks that communicate with the same unit should be in serial. A common task FSP loeschen (**F**ehler**SP**eicher) deletes and initializes the error log of a control unit. After the deletion of the error log other tasks FSP schreiben and FSP pruefen access the error log. The deletion of the error log takes a long time. While other tasks can operate on the control unit the access of the error log by FSP schreiben and FSP pruefen should be postponed as long as possible. To this end, tasks should be scheduled in between the deletion of the error log and the new access. The heuristic Algorithm 16 describes how we handle the two challenges.

Algorithm 16 iterates over all pairs of tasks $\langle t_1, t_2 \rangle$ with the same electronic control unit (ecu), see Line 1. If a relationship between t_1 and t_2 exists we continue with the next pair. If one of the tasks deletes the error log (FSP loeschen) we schedule the tasks first, and if one tasks access the error log (FSP pruefen oder schreiben) we schedule the task second, see Line 4 – 7. Else we randomly either schedule t_1 or t_2 first, see Line 9 – 13. If a new succession rule is added we have to recalculate the transitive hull to avoid adding a cycle, see Line 15, see Example 6.3.2. If we iterate over all pairs we return the new succession relationships, see Line 17.

Algorithm 16 HeuristicSameComponent () : SuccessorRelationships

```
1: for all pair of tasks $\langle t_1, t_2 \rangle$ with the same ECU do
2:     if a relationship $t_1 \to t_2$ or $t_2 \to t_1$ exists then
3:         Continue
4:     else if $t_1 = \mathsf{FSP\ loeschen} \vee t_2 = \mathsf{FSP\ schreiben} \vee t_2 = \mathsf{FSP\ pruefen}$ then
5:         succesRel.add( $t_1 \to t_2$ )
6:     else if $t_2 = \mathsf{FSP\ loeschen} \vee t_1 = \mathsf{FSP\ schreiben} \vee t_1 = \mathsf{FSP\ pruefen}$ then
7:         succesRel.add( $t_2 \to t_1$ )
8:     else
9:         if random number between 0 and 1 is smaller than 0.5 then
10:            succesRel.add( $t_1 \to t_2$ )
11:        else
12:            succesRel.add( $t_2 \to t_1$ )
13:        end if
14:    end if
15:    Recalculate transitive hull
16: end for
17: return succesRel
```

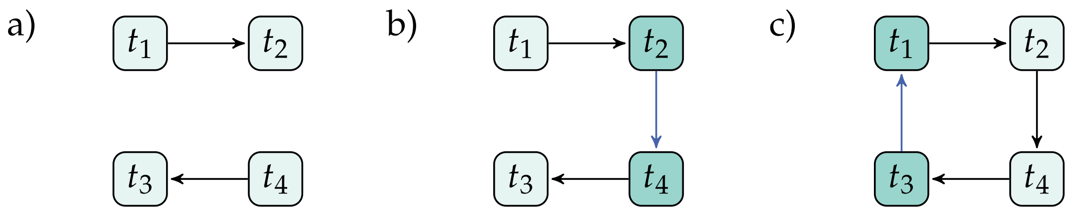


Figure 6.20: Without Recalculating the Transitive Hull a Cycle Can Occur.

Example 6.3.2. To consider why we recalculate the transitive hull, see Figure 6.20(a). In the first iteration the pair $\langle t_2, t_4 \rangle$ is selected and the edge (t_2, t_4) is added, see Figure 6.20(b). Without recalculating the transitive hull the pair $\langle t_1, t_3 \rangle$ is possible and the additional edge (t_3, t_1) would lead to a cycle.

6.3.3 Reducing Depth of the Process Model

To minimize the amount of connection a commissioning process should use only few parallelisms. To this end, we use a post-processing and combine short running parallel lanes. The combination does not influence the processing time of the process if the combined runtime is shorter than the longest running lane.

Algorithm 17 HeuristicFewerParallelisation ()

```
1:  for all parallel nodes $P$ in the process tree do
2:      while true do
3:          $l_1$ ←Lane with the shortest processing time in $P$.
4:          Remove $l_1$ from $P$
5:          $l_2$ ←Lane with the shortest processing time in $P$.
6:          Remove $l_2$ from $P$
7:          $l_{max}$ ←Lane with the longest processing time in $P$.
8:          if $runtime\,(l_1) + runtime\,(l_2) \leq \lambda\ runtime\,(l_{max})$ then
9:              $l_3$ ← combine $l_1$ and $l_2$
10:             Add $l_3$ to $P$
11:         else
12:             Add $l_1$ and $l_2$ to $P$
13:             break
14:         end if
15:     end while
16: end for
```

Algorithm 17 iterates over all *parallel nodes* in the process tree, Line 1. For each *parallel node P* the algorithm selects the two nodes with the shortest processing time t_1 and t_2 and the one with the longest t_{max}, Line 3 – 7. Next the algorithm checks if the combined processing time of t_1 and t_2 is shorter than the runtime of t_{max} times a factor λ, Line 8. The processing times are only guesses and vary. To this end, the algorithm uses the factor $\lambda \geq 1$ and only combines if the processing time of t_{max} is for instance 20% higher ($\lambda = 1.2$). In this case the algorithm combines l_1 and l_2 to l_3 and add the lane to the parallel node P, Line 9–10. If no such lanes exist the while-loop aborts, see Line 13, and continues with the next *parallel node*.

6.3.4 Closing the Connections to the Control Unit

Every connection to an electronic control unit (ECU) has to be closed by a specific task (Verbindungsabbau). Only a fixed amount of connections can be open at the same time, to this end, the connection should be closed after use. The close connection tasks are not part of the initial list and are added as a post-processing step after the synthesis. For each sequential node we add a close connection task after the last task of every control unit.

6.3.5 Different Configuration

The process model for the compact executive car will be executed using different configurations $\mathcal{C}$. Each of this configurations $c \in \mathcal{C}$ uses different tasks and skips others. The set of task T_c for the configuration can overlap, i. e., $T_c \cap T_{c'} \neq \emptyset$. It is important that the runtime of each configuration is optimized. To this end, we change the optimum criteria to optimize to the sum of all configurations.

$$\min \sum_{c \in \mathcal{C}} runtime\,(T_c)$$

If a task fails, then the task will be repeated in a different configuration. In the worst case this means that all tasks in the model will be executed. To ensure that the total runtime does not exceed the total cycle time planned for the commissioning we add the constraint:

$$runtime\,(T) \leq \lambda t$$

with t as the cycle time of the factory and λ as the scheduled number of cycles for the commissioning process.

6.4 Evaluation

The evaluation consists of two parts: The general evaluation of the synthesis algorithm using simulation, see Subsection 6.4.1, and the automatic generation of the new compact executive car process models, see Subsection 6.4.2.

6.4.1 Evaluation of the Algorithm with Simulation[2]

Our evaluation uses 21 process models from a car manufacturer that specify the testing and commissioning of middle-class vehicles. Each process model reflects several context characteristics which are attached for the generation. The context characteristics consist of properties of the vehicle project, of the factory and of the components to put in commission. Professional process developers have designed the process models. The tasks to be executed depend on the components built into the vehicle to be tested. In cooperation with those domain experts we have built the specification for the 21 process models, i. e., the ordering relation graphs, automatically using a knowledge base, see [Mra+14]. The process models contain up to 185 tasks and over 3000 dependencies, including transitive ones. The parameter λ defines the maximum size of the process trees. The possible number of trees grows exponentially with the maximum size. Therefore, the correct and optimal tree patterns are harder to find for larger values of λ. Otherwise, a higher value could allow finding a process model with a better processing time. We choose $\lambda = 5$ for our evaluation.

Table 6.2 shows the results for commissioning process models A, B, and C. We have chosen A, B, and C because they are representative for the whole set, ranging from a relatively small one (C) to one of the largest (B). For a summary of all models see Table 6.3. The second row in Table 6.2 shows the processing time measured for the process model created by hand. Table 6.2 then lists the expected processing time of the process (PT) and the time our approach needs to generate the respective model, i. e., computation time (CT), for 10 to 100,000 iterations. In all cases, the algorithm has been able to generate a process model in less than 100 ms that outperforms the reference process model. After 100,000 iterations, in less than 1.5 minutes, it could find process models with processing time 34%, 37%, and 50% lower than their manually generated counterparts.

For all 21 process models, Table 6.3 shows the minimum, maximum, and the quartile for 7 values of the evaluation. The process models contain between 98

[2]Parts of this Subsection are published in [MMB15] and the extended version in [MMB14b]

Table 6.2: Computation Time (CT) and Processing Time (PT) of our Approach

	Process A		Process B		Process C	
No. of Tasks	171		185		116	
Ref. process time	171 780 ms		169 606 ms		148 014 ms	
	CT in ms	PT in ms	CT in ms	PT in ms	CT in ms	PT in ms
10 Iteration	35	188 420	34	227 260	32	132 998
50 Iteration	69	127 687	71	131 121	66	103 234
100 Iteration	113	127 687	113	131 121	104	103 234
1 000 Iteration	823	127 687	964	116 155	788	97 264
10 000 Iteration	8 207	112 918	8 298	113 874	7 817	71 513
100 000 Iteration	78 409	112 624	86 594	106 216	77 335	65 892
PT reduction	34.437 %		37.375 %		50.456 %	

and 185 tasks, and need up to 178s to perform. Our approach requires $\approx 30s$ and $\approx 37\,000$ iterations on average to generate the best result found. For all instances our approach has identified a solution that is better than the manually generated one in less than 100 ms. Our approach needs less than 3 iterations to do so in most cases. On average, it nearly halves the processing time of the commissioning process models (47.47%) compared to the reference points.

6.4.2 Use Case: Compact Executive Car Process Models

Besides the general analysis and performance gain of the vehicle-individual generation shown in Subsection 6.4.1 we applied our synthesis to a large use case and testing it in reality. The use case was the commissioning of the new vehicle series for a compact executive car introduced in 2015. The process model is large in size and consists of 496 tasks. The process model consists of regions that need to be exactly described and two large regions that we try to optimize. One region describe the installment of the software on the ECUs, called *Bedatung*. The other region controls and test several parts of the vehicle, called *Funktionstest*. Figure 6.21 shows the manual generated process model on the left hand side, and the one of our synthesis on right hand side, with the two regions, i. e., *Bedatung* and *Funktionstest*. We used the heuristics of Section 6.3 for the synthesis with $\lambda = 0.1$ and $\epsilon = 0.1$. 96 human tasks existed, see also Figure 6.21.

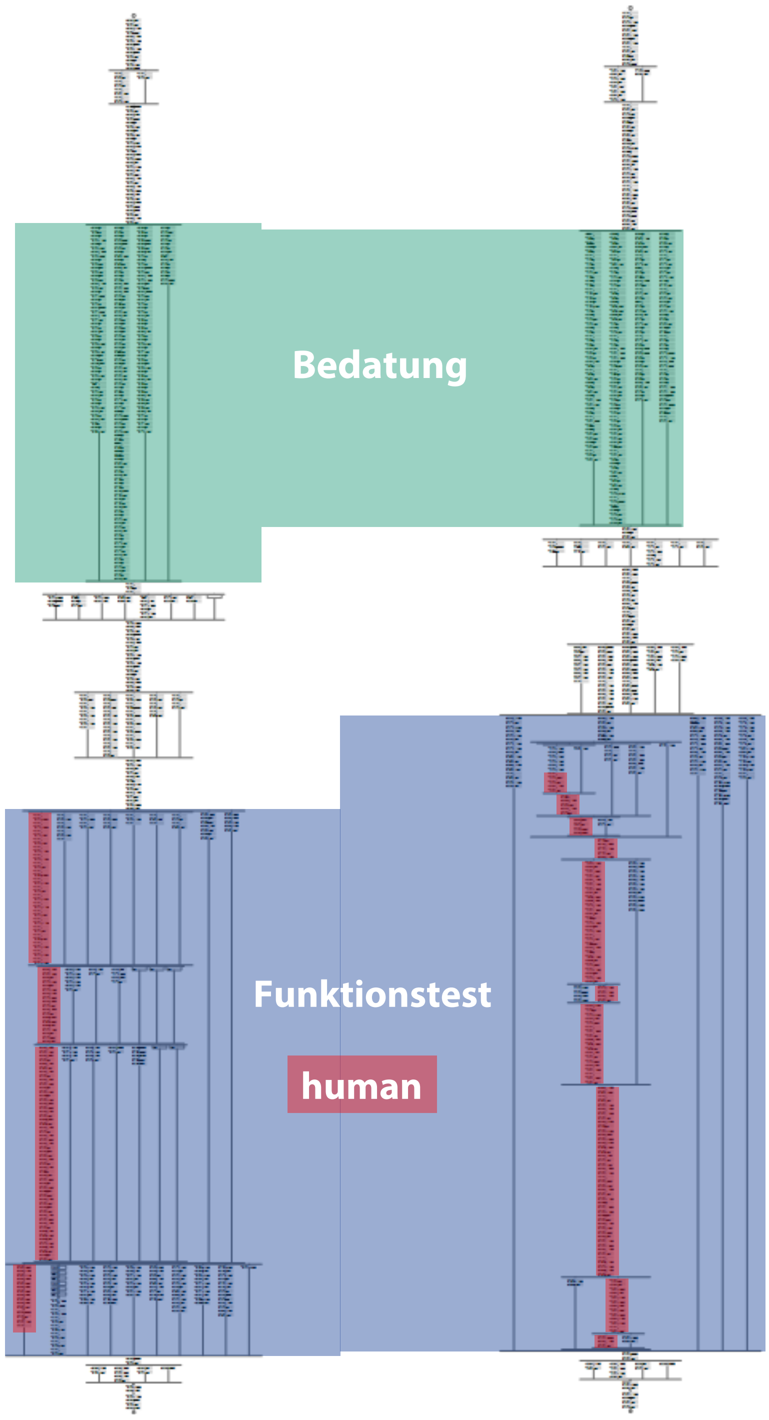


Figure 6.21: The Original Commissioning Process Model for the Compact Executive Car on the Left Hand Side and the Synthesized One on the Right Hand Side.

Table 6.3: The Minimum, Maximum, and Quartile for the Evaluation of 21 Commissioning Process Models

	Minimum		Median		Maximum
	$Q_{0.00}$	$Q_{0.25}$	$Q_{0.5}$	$Q_{0.75}$	$Q_{1.00}$
Nr. of Tasks	98	123	133	146	185
Ref. Process Time in s	144.232	151.623	157.513	166.138	178.606
Best Found pt in s	64.643	72.637	84.529	93.487	108.496
iterations (it)	5 090	15 523	37 733	76 035	94 271
Calculation Time (ct) in s	4.641	12.356	30.284	63.392	77.480
pt reduction in %	33.39	40.62	47.54	53.62	58.03

To compare the performance of the generated process model to the original one we simulated several commissioning processes on both process models with real world runtime data. In total we used 193 runs of the process model. Figure 6.22 shows the runtime of those runs for the original and the synthesized process model. The runs could have one of two configurations C1 (140 runs) or C2 (53 runs). For C1 we could achieve a average reduction of 5.92s or 5.5%, for C2 the average reduction is 209.13s or 50.39% compared to the original manual process model. The lesser reduction in the C1 configuration can be explained by the critical path of the reduction that consists mainly of human worker tasks, i. e., only minor optimization is possible.

6.5 Related Work for Process Synthesis

In this subsection we want to discuss related work in the field of process synthesis, in particular synthesis approaches as [Yu+08; Awa+11], process discovery [Aal11; LFA13b; LFA14], approaches using process similarity, see [Chi+09; Koo+08; Lau+09], approaches to restructure process model [Pol12], approaches using Artificial Intelligence approaches [VM91; MR02; AHK05; AK07], and declarative process modeling [Mon+10; PBA10].

[Yu+08] synthesizes a process model from specification in the form of state machines. First, [Yu+08] combines all state machines of the specification and enlists the possible execution paths. Next, an algorithm similar to the α-algorithm [AWM04] is applied to synthesize a process model from its set of paths. [Yu+08] can only be applied if the specification, i. e., the number of state machines, is small ($\approx$ 6). To this end, [Yu+08] divides the specification into small groups,

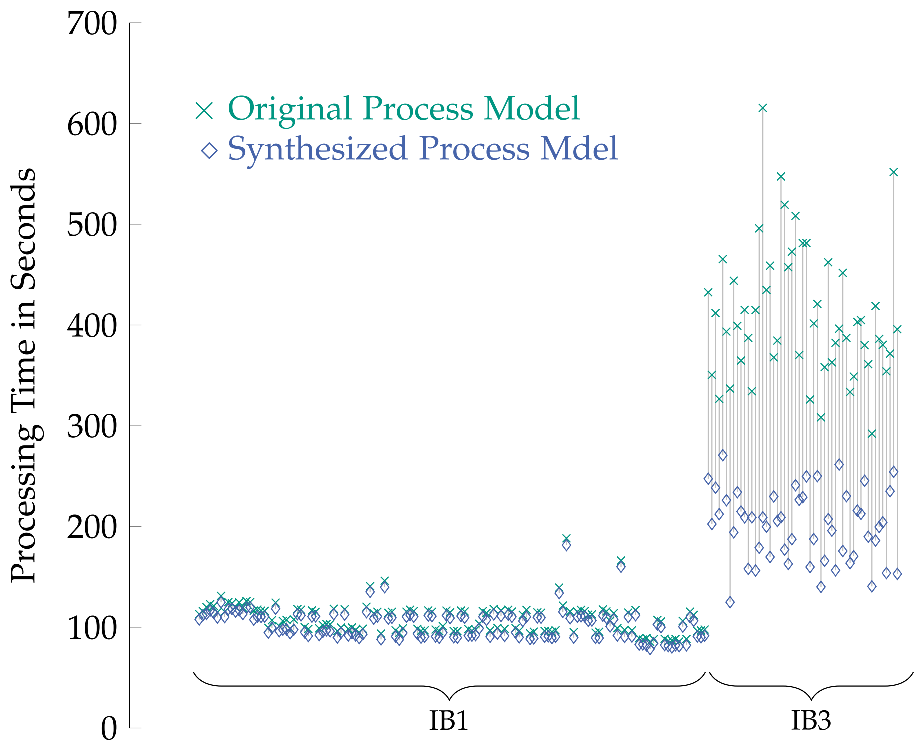


Figure 6.22: Processing Time of the Original and the Synthesized Process Model

synthesizes a process fragment for each group and manually combines the fragments. For our use case, this approach would require over a hundred state machines for each commissioning process model, and the manual combination would not be feasible. [Awa+11] has specifications with LTL as starting point. Next, they generate a pseudo model from the specification. This model lists all paths that fulfill the LTL formula. [Awa+11] generates an ordering relation graph from the set of paths and uses it to synthesize a process tree. For our use case the generation of all paths would not be feasible. This is because the number of paths grows exponentially with the size of the specification. Even for the smallest process model we have evaluated calculating all paths that have not been possible.

Process discovery means finding a process model that can reproduce the behavior given in a log see [Aal11]. [LFA13b] rediscovers a process model in the process-tree notation. It generates a graph, called directly-follows graph, from the log and tries to find different kinds of cuts in the graph. Each kind of cut refers to a control structure in the process tree, i. e., SEQ, AND, XOR, LOOP. The cuts partitions the graph and allow to hierarchically find a process tree for the log. In contrast to an ORG, a directly-follow graph is not transitive, and if two nodes are in parallel they share a two-way edge. In contrast to no edge in the ORG. It is not possible to find a cut for a prime component, thus the approach of [LFA13b] does not help in case the specification is under-specified. Put differently, the problem statement in [LFA13b] is different from ours; the neighborhood graph of the complete log of a process tree never contains a prime component. For an incomplete log, a prime component can occur. [LFA14] proposes to use probabilistic activity relations in the case of an incomplete log. The cut with the highest probability is chosen. This means that their algorithm generalizes from the incomplete log and assumes relationships that are not present. An ORG is an upper bound of the possible behavior. Assuming an additional relation would result in a violation of a constraint. The approaches of the process discovery are related to the synthesis of a process model. Both approaches start with a graph-based source and use it in order to find a process model. Important differences exist in the semantics of the graph. In the case of the process synthesis the graph describes the allowed behavior. Our goal is to find a suitable process model by further constraining the specification. Figure 6.23(a) illustrates the process synthesis. Our approach finds a suitable process model inside the allowed behavior, i. e., we work inwards from the graph-presentation. In the case of process discovery the graph is generated from the log. The graph therefore describes a subset of the behavior of process model to rediscover. In general, the process discovery generalizes from the observed behavior in order to find a process model. Therefore, the process discovery works outwards from the graph, see Figure 6.23(b).

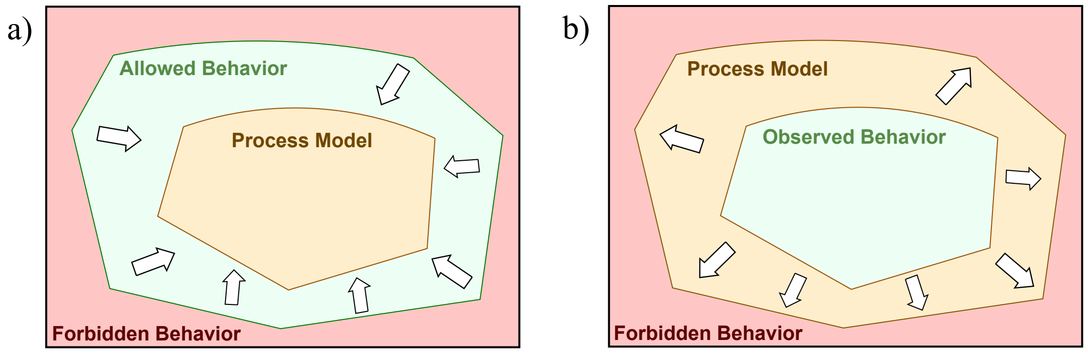


Figure 6.23: Illustration of Process Synthesis (a) and Process Discovery (b)

An approach different from generating the process model from scratch is to extract information from process models already specified and to create a similar process. [Chi+09] uses a CBR-based method to this end. The search is based on keywords that are annotations of the workflows. [Koo+08] guides the process designer with suggestions on how to complete data-oriented visualization models. The suggestions are generated from paths of existing visualization process models stored in a repository. [Koo+08] does not allow building a process model with an AND-Split and therefore is not sufficient in our case. [Lau+09] predicts which activity pattern, i. e., generic process fragment, will follow the partly modeled process. The paths of existing process models are extracted and analyzed with association rule mining. [Koo+08; Lau+09] extend an existing process model, while our approach generates one from a declarative specification. [Chi+09] requires annotations of the existing process models. None of the approaches mentioned optimize the runtime or consider constraints.

[Pol12; PGD12] transform an unstructured model without cycles into a behaviorally equivalent structured process model. By *structured* we mean that for each Split-Gateway there is a corresponding Join-Gateway. Structured processes allow an effective verification, see [MMB14a] and are easy to understand cf. [RM11]. [PGD12] determines relationships between the tasks of a process model and generates an ORG using these relationships. Next, [PGD12] decomposes the ORG into a Modular Decomposition Tree (MDT). In contrast to our approach, [PGD12] generates the ORG from the behavior of an existing process model and not from a set of compliance rules. The behavior is definite. The result therefore is an unique process model. In our approach in turn, the behavior is under-specified, and several process models are possible.

AI planning is the task of defining a set of actions that achieve a specified aim [HTD90]. In a nutshell, it is the search for an applicable plan in the solution space. [VM91] uses a genetic algorithm to find a manufacturing plan. [MR02] uses an AI planning approach to synthesize service compositions. Without calling it AI planning, [AHK05] uses a similar approach for configuration-based workflow composition. [AK07] introduces a planning algorithm to compose data workflows. None of these studies focus on optimizing the runtime of the process or considers requirements similar to ours. These approaches are not applicable to our problem statement.

In contrast to imperative process models, declarative workflows allow for any behavior fulfilling the declarative specification, see [Mon+10]. Thus, declarative workflows provide maximum flexibility not limited by a process model. In comparison to our approach, and other like [Yu+08; Awa+11], that generates an imperative process model from the declarative specification. The enactment of declarative workflows is not trivial, cf. [PBA10], and tool support by major vendors is missing. To our knowledge, there is no tool that executes declarative process models comparable to the commissioning of vehicles.

Chapter 7

Conclusion

In summary this thesis displays novel and important contributions to three interconnected fields in the process model research. Those three fields are namely the specifications of declarative properties, the verification of those properties on a given process model, and the synthesis of a process model from a list of properties. Subsection 7.1 gives a summary over the contributions and results of those fields. We mainly focus in this thesis on our own use case, the commissioning of vehicles. But in fact we claim that our achievements regarding to our research are applicable and more general. Subsection 7.2 sketches possible future applications and show work that has been inspired by our research.

7.1 Summary

First, at the beginning of the thesis we introduced our scenario, the *commissioning of vehicles*, see Chapter 2. This Chapter revealed details about the diagnostic system used for the commissioning, the different communication protocols, and the processes model notations. Next, Chapter 3 gave an overview over the related fields in the business process research. A short description about those research fields was given and the differences and similarities to our own work highlighted.

Chapter 4 described our contributions and results in the field of the declarative specification of properties that a process model should fulfill. Our use case revealed several challenges regarding to the specification. For instance, the knowledge about the properties a process model needs to adhere to is typically distributed between different employees and departments. The properties are often context sensitive, i. e., only hold for a specific process place or vehicle series. To allow an automatic verification and synthesis the properties need to be available in a formal form, e. g., temporal logic. Additionally it is often not possible for a domain expert to give a succinct list of all properties by interview. We addressed those challenges within two systems: Our automatic specification framework and our framework for property candidate detection. To this end, we systematically collected all properties and the relevant information for their specification by a series of interviews. We could map those properties to a distinct set of classes, called property patterns. Next we built a database for the storage of those patterns and with help of a template of the respective temporal logic formula. Before the verification procedure our framework queries the database using the process context, e. g., the process place and vehicle series. Next, the framework automatically instantiate the process patterns to concrete properties. A functional evaluation revealed that this system is able to generate the properties for the verification of a process model. An expert interview showed the usability of our approach. To address the challenge that the expert cannot give a succinct list of all properties out of their head we developed a second system. This system analyses existing process models and tries to extract likely candidates for properties using statistical measures, similar to the frequent item set mining. Our evaluation showed that we indeed could detect properties with this approach and additionally revealed more knowledge about the process models.

Chapter 5 displayed our contributions in the field of the verification, i. e., the test if a process model fulfills all required properties. We could reveal several challenges for the verification of commissioning process models. First, automatic verification techniques, e. g., model checking, require an interpretation of the execution semantic. Second, commissioning processes are typically highly concurrent and large in size, leading to a large state space. Third, it is often not trivial for domain experts to interpret the result of the verification and to detect the cause of the property violation. We addressed those challenges by developing an interpretation of the semantic of commissioning processes by a Petri net representation. In order to address the growing state space we used a reduction algorithm that limits the verification to its parts of the process model relevant for the property. Lastly we give a reporting schema, parsing the verification result and guiding the user in order to correct the property violation. The evaluation revealed that our verification is functional, e. g., it can detect property violations and is efficient for large commissioning processes.

Chapter 6 presented our contribution in the field of process synthesis, i. e., generating a process model from a declarative description. Our use case revealed several challenges within this approach. First, we needed to synthesize a usable process model for our scenario. This means that the process model needed to fulfill several requirements, e. g., block-structured. Second, the declarative description was in general not complete, i. e., more than one process model was possible. We gave two different algorithms for the synthesis. The first algorithm used a resource-constraint scheduling as basis. We showed that algorithms with a scheduling basis lack important characteristics, e. g., structuredness. This led us to our second algorithm that uses a modular decomposition of the dependency graph as its core. In the case of under-specification the decomposition helped us to detect the under-specified parts of the process model. We employed a probabilistic algorithm in order to detect the best solution for those parts. Our evaluation showed that by using this algorithm we could greatly improve the process performance by generating vehicle specific models. We also applied the algorithm to the real use case, the commissioning of a new compact executive car series. This revealed several new requirements that we addressed with heuristic improvements to our synthesis algorithm.

7.2 Impact and Future Research

Overall, the techniques presented in this thesis have been well received by the research community, especially the fact that our work successfully combines requirements of the real world industry with theoretical concepts, that means our research targets real problems. At the same time our main contribution was generic and could be applied to different use cases.

Furthermore, our work on relevant optimization has been the inspiration for other researchers work, e. g., in [TMB16]. [Sta+14] presented an approach for the detection of data-flow errors in BPMN 2.0 process models. Data-flow errors in BPMN 2.0 process models, such as missing or unused data, led to undesired process executions. To this end, [Sta+14] detected anti-patterns in process models. The detection of anti-patterns is symmetrical to the verification, i. e., detecting an anti-pattern X is equivalent to verify the property: *Absence of* X. The work of [TMB16] increased the performance of those techniques by applying a relevance algorithm, similar to ours, in order to detect the relevant regions.

Another important research field of the recent years was the compliance checking of processes. Business process compliance is the execution of a process according to regulations and legal norms. The compliance checking comprises several similarities to the problems faced in our use case. Most compliance rules are written as textual regulations. Those cannot be tested automatically on process models similar to our technical rules. [Rat+15] motivated these problems. Our work regarding to the property pattern instantiation can be applied to this end. Most of the compliance checking approaches does not consider the state space explosion. One exception is the work of [Knu+10] that focused on the state space growth caused by data constraints. For highly concurrent process models the state space is going to growth exponentially. Our relevant optimization could be leveraged to support the compliance checking of highly concurrent models. [Awa+11] used a synthesis approach in order to generate process models from compliance rules. It was limited to fully-specified rule sets. On this respect our work regarding synthesis field could be employed in the case of under-specified specification.

In summary, our work has produced many novel ideas, approaches, and evaluation results for the specification, verification, and synthesis of process models. We could solve real problems from our industry partner and deliver novel concepts regarding to those fields. Therefore, we hope that our contributions can provide a sound basis for future research.

Bibliography

[AAB97] Wil M. P. van der Aalst, Pierre Azéma, and Gianfranco Balbo. »Verification of workflow nets«. In: *Application and Theory of Petri Nets*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 1997, pp. 407–426. ISBN: 978-3-540-63139-2 978-3-540-69187-7.

[AAD12] Wil M. P. van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. »Replaying history on process models for conformance checking and performance analysis«. en. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2.2 (2012), pp. 182–192. ISSN: 1942-4795. DOI: `10.1002/widm.1045`.

[Aal+03a] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. »Workflow mining: A survey of issues and approaches«. In: *Data & Knowledge Engineering* 47.2 (Nov. 2003), pp. 237–267. ISSN: 0169-023X. DOI: `10.1016/S0169-023X(03)00066-1`.

[Aal+03b] Wil M. P. van der Aalst, A. H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. »Workflow Patterns«. en. In: *Distributed and Parallel Databases* 14.1 (July 2003), pp. 5–51. ISSN: 0926-8782, 1573-7578. DOI: `10.1023/A:1022883727209`.

[Aal+09] Wil M. P. van der Aalst, M. Adams, A. H. M. ter Hofstede, Maja Pešić, and Helen Schonenberg. »Flexibility as a Service«. en. In: *Database Systems for Advanced Applications*. Ed. by Lei Chen, Chengfei Liu, Qing Liu, and Ke Deng. Lecture Notes in Computer Science 5667. Springer Berlin Heidelberg, 2009, pp. 319–333. ISBN: 978-3-642-04204-1 978-3-642-04205-8.

[Aal+10] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe T. K. W. Wynn. »Soundness of workflow nets: classification, decidability, and analysis«. en. In: *Formal Aspects of Computing* 23.3 (Aug. 2010), pp. 333–363. ISSN: 0934-5043, 1433-299X. DOI: `10.1007/s00165-010-0161-4`.

[Aal03] Wil M. P. van der Aalst. *Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype"*. 2003.

[Aal11] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. 1st. Springer Berlin Heidelberg, 2011. ISBN: 3-642-19344-7 978-3-642-19344-6.

[Aal98] Wil M. P. van der Aalst. »The Application of Petri Nets to Workflow Management«. In: *Journal of Circuits, Systems and Computers* 08.01 (Feb. 1998), pp. 21–66. ISSN: 0218-1266.

[Aal99] Wil M. P. van der Aalst. »On the automatic generation of workflow processes based on product structures«. In: *Computers in Industry* 39.2 (July 1999), pp. 97–111. ISSN: 0166-3615. DOI: 10.1016/S0166-3615(99)00007-X.

[ABD05] Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. »Process Mining and Verification of Properties: An Approach Based on Temporal Logic«. en. In: *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*. Ed. by Robert Meersman and Zahir Tari. Lecture Notes in Computer Science 3760. Springer Berlin Heidelberg, 2005, pp. 130–147. ISBN: 978-3-540-29736-9 978-3-540-32116-3.

[ADW08] Ahmed Awad, Gero Decker, and Mathias Weske. »Efficient Compliance Checking Using BPMN-Q and Temporal Logic«. In: *Business Process Management*. Ed. by Marlon Dumas, Manfred Reichert, and Ming-Chien Shan. Lecture Notes in Computer Science 5240. Springer Berlin Heidelberg, Jan. 2008, pp. 326–341. ISBN: 978-3-540-85757-0 978-3-540-85758-7.

[AH04] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. en. MIT Press, Jan. 2004. ISBN: 978-0-262-72046-5.

[AH05] Wil M. P. van der Aalst and A. H. M. ter Hofstede. »YAWL: yet another workflow language«. In: *Information Systems* 30.4 (June 2005), pp. 245–275. ISSN: 0306-4379. DOI: 10.1016/j.is.2004.02.002.

[AHK05] Patrick Albert, Laurent Henocque, and Mathias Kleiner. »Configuration based workflow composition«. In: *IEEE International Conference on Web Services*. July 2005, 285–292 vol.1. DOI: 10.1109/ICWS.2005.38.

[AHV02] Wil M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek. »An Alternative Way to Analyze Workflow Graphs«. en. In: *Advanced Information Systems Engineering*. Ed. by Anne Banks Pidduck, M. Tamer Ozsu, John Mylopoulos, and Carson C. Woo. Lecture Notes in Computer Science 2348. Springer Berlin Heidelberg, 2002, pp. 535–552. ISBN: 978-3-540-43738-3 978-3-540-47961-1.

[AK07] José Luis Ambite and Dipsy Kapoor. »Automatically Composing Data Workflows with Relational Descriptions and Shim Services«. In: *The Semantic Web*. Ed. by Karl Aberer et al. Lecture Notes in Computer Science 4825. Springer Berlin Heidelberg, Jan. 2007, pp. 15–29. ISBN: 978-3-540-76297-3 978-3-540-76298-0.

[AMW05] Wil M. P. van der Aalst, A. K. Alves de Medeiros, and A. J. M. M. Weijters. »Genetic Process Mining«. en. In: *Applications and Theory of Petri Nets 2005*. Ed. by Gianfranco Ciardo and Philippe Darondeau. Lecture Notes in Computer Science 3536. Springer Berlin Heidelberg, 2005, pp. 48–69. ISBN: 978-3-540-26301-2 978-3-540-31559-9.

[AP06] Wil M. P. van der Aalst and Maja Pešić. »DecSerFlow: Towards a Truly Declarative Service Flow Language«. In: *Web Services and Formal Methods*. Ed. by Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro. Lecture Notes in Computer Science 4184. Springer Berlin Heidelberg, Jan. 2006, pp. 1–23. ISBN: 978-3-540-38862-3 978-3-540-38865-4.

[Apt03] Krzysztof Apt. *Principles of Constraint Programming*. en. Cambridge University Press, Aug. 2003. ISBN: 978-0-521-82583-2.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. »Fast Algorithms for Mining Association Rules in Large Databases«. In: *Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. ISBN: 1-55860-153-8.

[ASI12] Norris Syed Abdullah, Shazia Sadiq, and Marta Indulska. »A Compliance Management Ontology: Developing Shared Understanding through Models«. en. In: *Advanced Information Systems Engineering*. Ed. by Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza. Lecture Notes in Computer Science 7328. Springer Berlin Heidelberg, Jan. 2012, pp. 429–444. ISBN: 978-3-642-31094-2 978-3-642-31095-9.

[Awa+11] Ahmed Awad, Rajeev Goré, James Thomson, and Matthias Weidlich. »An Iterative Approach for Business Process Template Synthesis from Compliance Rules«. In: *Advanced Information Systems Engineering*. Ed. by Haralambos Mouratidis and Colette Rolland. Lecture Notes in Computer Science 6741. Springer Berlin Heidelberg, Jan. 2011, pp. 406–421. ISBN: 978-3-642-21639-8 978-3-642-21640-4.

[AWM04] Wil M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster. »Workflow mining: discovering process models from event logs«. In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (Sept. 2004), pp. 1128–1142. ISSN: 1041-4347. DOI: `10.1109/TKDE.2004.47`.

[Bar+07] Kamel Barkaoui, Rahma Ben Ayed, Zohra Sbaï, Kamel Barkaoui, Rahma Ben Ayed, and Zohra Sbaï. »Workflow Soundness Verification based on Structure Theory of Petri Nets«. In: *International Journal of Computing and Information Sciences* 5.1 (2007), pp. 51–61.

[BDA12] Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. »A genetic algorithm for discovering process trees«. In: *IEEE Congress on Evolutionary Computation*. June 2012, pp. 1–8. DOI: `10.1109/CEC.2012.6256458`.

[BG12] Bernhard Beckert and Sarah Grebing. »Evaluating the Usability of Interactive Verification System«. In: *Proceedings, 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE), Manchester, UK, June 30, 2012*. Vol. 873. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 3–17.

[BGB14] Bernhard Beckert, Sarah Grebing, and Florian Böhl. »How to Put Usability into Focus: Using Focus Groups to Evaluate the Usability of Interactive Theorem Provers«. In: *Proceedings, Workshop on User Interfaces for Theorem Provers (UITP), Vienna, July 2014*. Ed. by Christoph Benzmüller and Bruno Woltzenlogel Paleo. EPTCS. To appear. 2014.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 978-3-540-68977-5.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. »Bounded Model Checking«. In: *Advances in computers*. Vol. 58. Elsevier, 2003, pp. 117–148.

[BKM08] Aaron Bangor, Philip T. Kortum, and James T. Miller. »An Empirical Evaluation of the System Usability Scale«. In: *International Journal of Human-Computer Interaction* 24.6 (2008), pp. 574–594. ISSN: 1044-7318. DOI: 10.1080/10447310802205776.

[BLK83] Jacek Błażewicz, Jan Karel Lenstra, and Alexander H. G. Rinnooy Kan. »Scheduling subject to resource constraints: classification and complexity«. In: *Discrete Applied Mathematics* 5.1 (Jan. 1983), pp. 11–24. ISSN: 0166-218X. DOI: 10.1016/0166-218X(83)90012-4.

[Bra+05] Marco Brambilla, Alin Deutsch, Liying Sui, and Victor Vianu. »The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae«. en. In: *Web Engineering*. Ed. by David Lowe and Martin Gaedke. Lecture Notes in Computer Science 3579. Springer Berlin Heidelberg, Jan. 2005, pp. 557–568. ISBN: 978-3-540-27996-9 978-3-540-31484-4.

[Bro13] John Brooke. »SUS: A Retrospective«. In: *JUS - The Journal of Usability Studies* (2013).

[CAC06] Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. »User Guidance for Creating Precise and Accessible Property Specifications«. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 208–218. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181801.

[CES83] Edmund M. Clarke, Ernest A. Emerson, and A. Prasad Sistla. »Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: A Practical Approach«. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. New York, NY, USA: ACM, 1983, pp. 117–126. ISBN: 978-0-89791-090-3. DOI: 10.1145/567067.567080.

[CES86] Edmund M. Clarke, Ernest A. Emerson, and A. Prasad Sistla. »Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications«. In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[CGJ78] Edward G. Jr. Coffman, M. Garey, and David S. Johnson. »An Application of Bin-Packing to Multiprocessor Scheduling«. In: *SIAM Journal on Computing* 7.1 (Feb. 1978), pp. 1–17. ISSN: 0097-5397. DOI: 10.1137/0207001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. en. MIT Press, 1999. ISBN: 978-0-262-03270-4.

[CH94] Alain Cournier and Michel Habib. »A new linear algorithm for Modular Decomposition«. In: *Trees in Algebra and Programming — CAAP'94*. Ed. by Sophie Tison. Lecture Notes in Computer Science 787. Springer Berlin Heidelberg, Jan. 1994, pp. 68–84. ISBN: 978-3-540-57879-6 978-3-540-48373-1.

[Che+09] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. »Exploiting Inductive Logic Programming Techniques for Declarative Process Mining«. en. In: *Transactions on Petri Nets and Other Models of Concurrency II*. Ed. by Kurt Jensen and Wil M. P. van der Aalst. Lecture Notes in Computer Science 5460. Springer Berlin Heidelberg, Jan. 2009, pp. 278–295. ISBN: 978-3-642-00898-6 978-3-642-00899-3.

[Chi+09] Eran Chinthaka, Jaliya Ekanayake, David B. Leake, and Beth Plale. »CBR Based Workflow Composition Assistant«. In: *IEEE World Conference on Services*. July 2009, pp. 352–355. DOI: 10.1109/SERVICES-I.2009.51.

[Cla+01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. »Progress on the State Explosion Problem in Model Checking«. en. In: *Informatics*. Ed. by Reinhard Wilhelm. Lecture Notes in Computer Science 2000. Springer Berlin Heidelberg, Jan. 2001, pp. 176–194. ISBN: 978-3-540-41635-7 978-3-540-44577-7.

[CM12] Claudio Di Ciccio and Massimo Mecella. »Mining Constraints for Artful Processes«. In: *Business Information Systems*. Ed. by Witold Abramowicz, Dalia Kriksciuniene, and Virgilijus Sakalauskas. Lecture Notes in Business Information Processing 117. Springer Berlin Heidelberg, 2012, pp. 11–23. ISBN: 978-3-642-30358-6 978-3-642-30359-3.

[CM13] Claudio Di Ciccio and Massimo Mecella. »A two-step fast algorithm for the automated discovery of declarative workflows«. In: *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Apr. 2013, pp. 135–142. DOI: 10.1109/CIDM.2013.6597228.

[CMM14] Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. »Discovering Target-Branched Declare Constraints«. en. In: *Business Process Management*. Ed. by Shazia Sadiq, Pnina Soffer, and Hagen Völzer. Lecture Notes in Computer Science 8659. Springer

Berlin Heidelberg, Jan. 2014, pp. 34–50. ISBN: 978-3-319-10171-2 978-3-319-10172-9.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. »Property Specification Patterns for Finite-state Verification«. In: *Second Workshop on Formal Methods in Software Practice*. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7–15. ISBN: 0-89791-954-8. DOI: 10.1145/298595.298598.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. »Patterns in property specifications for finite-state verification«. In: *International Conference on Software Engineering*. 1999, pp. 411–420.

[DAV05] Boudewijn F. van Dongen, Wil M. P. van der Aalst, and H. M. W. Verbeek. »Verification of EPCs: Using Reduction Rules and Petri Nets«. In: *Advanced Information Systems Engineering*. Ed. by Oscar Pastor and João Falcão e Cunha. Lecture Notes in Computer Science 3520. Springer Berlin Heidelberg, Jan. 2005, pp. 372–386. ISBN: 978-3-540-26095-0 978-3-540-32127-9.

[DDM08] Boudewijn F. van Dongen, Remco Dijkman, and Jan Mendling. »Measuring Similarity between Business Process Models«. en. In: *Advanced Information Systems Engineering*. Ed. by Zohra Bellahsène and Michel Léonard. Lecture Notes in Computer Science 5074. Springer Berlin Heidelberg, Jan. 2008, pp. 450–464. ISBN: 978-3-540-69533-2 978-3-540-69534-9.

[Dij+11] Remco Dijkman, Marlon Dumas, Boudewijn van Dongen, Reina Käärik, and Jan Mendling. »Similarity of business process models: Metrics and evaluation«. In: *Information Systems*. Special Issue: Semantic Integration of Data, Multimedia, and Services 36.2 (Apr. 2011), pp. 498–516. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.09.006.

[EH86] Ernest Allen Emerson and Joseph Y. Halpern. »"Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic«. In: *J. ACM* 33.1 (Jan. 1986), pp. 151–178. ISSN: 0004-5411. DOI: 10.1145/4904.4999.

[EKO07] Marc Ehrig, Agnes Koschmider, and Andreas Oberweis. »Measuring Similarity Between Semantic Business Process Models«. In: *Proceedings of the Fourth Asia-Pacific Conference on Comceptual Modelling - Volume 67*. APCCM '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 71–80. ISBN: 1-920-68285-X.

[Eme97] Ernest A. Emerson. »Model checking and the Mu-calculus«. In: *DIMACS Series in Discrete Mathematics*. American Mathematical Society, 1997, pp. 185–214.

[Fah+09a] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. »Instantaneous Soundness Checking of Industrial Business Process Models«. In: *Business Process Management*. Ed. by Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers. Lecture Notes in Computer Science LNCS 5701. Springer Berlin Heidelberg, Jan. 2009, pp. 278–293. ISBN: 978-3-642-03847-1 978-3-642-03848-8.

[Fah+09b] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. »Declarative versus Imperative Process Modeling Languages: The Issue of Understandability«. en. In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor. Lecture Notes in Business Information Processing 29. Springer Berlin Heidelberg, 2009, pp. 353–366. ISBN: 978-3-642-01861-9 978-3-642-01862-6.

[Fah+10] Dirk Fahland, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. »Declarative versus Imperative Process Modeling Languages: The Issue of Maintainability«. en. In: *Business Process Management Workshops*. Ed. by Stefanie Rinderle-Ma, Shazia Sadiq, and Frank Leymann. Lecture Notes in Business Information Processing 43. Springer Berlin Heidelberg, 2010, pp. 477–488. ISBN: 978-3-642-12185-2 978-3-642-12186-9.

[För+07] Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild van der Straeten. »Verification of Business Process Quality Constraints Based on Visual Process Patterns«. In: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, 2007. TASE '07*. June 2007, pp. 197–208. DOI: `10.1109/TASE.2007.56`.

[FRM13] Kathrin Figl, Jan Recker, and Jan Mendling. »A study on the effects of routing symbol design on process model comprehension«. In: *Decision Support Systems* 54.2 (Jan. 2013), pp. 1104–1118. ISSN: 0167-9236. DOI: `10.1016/j.dss.2012.10.037`.

[Ger+95] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. »A partial order approach to branching time logic model checking«. In: *Theory of Computing and Systems, 1995. Proceedings., Third Israel*

Symposium on the. Jan. 1995, pp. 130–139. DOI: 10.1109/ISTCS.1995.377038.

[Gia+15] Giuseppe De Giacomo, Marlon Dumas, Fabrizio Maria Maggi, and Marco Montali. »Declarative Process Modeling in BPMN«. en. In: *Advanced Information Systems Engineering*. Ed. by Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson. Lecture Notes in Computer Science 9097. Springer Berlin Heidelberg, June 2015, pp. 84–100. ISBN: 978-3-319-19068-6 978-3-319-19069-3.

[GL06] Volker Gruhn and Ralf Laue. »Patterns for Timed Property Specifications«. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005) Quantitative Aspects of Programming Languages 2005 153.2 (2006), pp. 117–133. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.035.

[Göd29] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. Doctoral dissertation. University Of Vienna, 1929.

[Gre+05] Gianluigi Greco, Antonella Guzzo, Giuseppe Manco, and Domenico Saccà. »Mining and reasoning on workflows«. In: *IEEE Transactions on Knowledge and Data Engineering* 17.4 (Apr. 2005), pp. 519–534. ISSN: 1041-4347. DOI: 10.1109/TKDE.2005.63.

[GV06] Stijn Goedertier and Jan Vanthienen. »Designing Compliant Business Processes with Obligations and Permissions«. In: *Business Process Management Workshops*. Ed. by Johann Eder and Schahram Dustdar. Lecture Notes in Computer Science 4103. Springer Berlin Heidelberg, Jan. 2006, pp. 5–14. ISBN: 978-3-540-38444-1 978-3-540-38445-8.

[Har09] Richter Harald. *Elektronik und Datenkommunikation im Automobil*. Tech. rep. Ifl-09-05. Clausthal: TU Clausthahl, 2009.

[HMS12] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. »Nested Dynamic Condition Response Graphs«. en. In: *Fundamentals of Software Engineering*. Ed. by Farhad Arbab and Marjan Sirjani. Lecture Notes in Computer Science 7141. Springer Berlin Heidelberg, 2012, pp. 343–350. ISBN: 978-3-642-29319-1 978-3-642-29320-7.

[Hol97] Gerard J. Holzmann. »The Model Checker SPIN«. In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521.

[HSK04] David Hollingsworth, Fujitsu Services, and United Kingdom. »The Workflow Reference Model: 10 Years On«. In: *Fujitsu Services, UK; Technical Committee Chair of WfMC*. 2004, pp. 295–312.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. »Transforming BPEL to Petri Nets«. en. In: *Business Process Management*. Ed. by Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera. Lecture Notes in Computer Science 3649. Springer Berlin Heidelberg, Jan. 2005, pp. 220–235. ISBN: 978-3-540-28238-9 978-3-540-31929-0.

[HTD90] James Hendler, Austin Tate, and Mark Drummond. *AI Planning: Systems and Techniques*. Tech. rep. College Park, MD, USA: University of Maryland at College Park, 1990.

[IEE11] IEEE. *Adoption of the Project Management Institute (PMI(R)) Standard A Guide to the Project Management Body of Knowledge (PMBOK(R) Guide)*. 2011.

[ISO12] ISO. *13209-12, Road vehicles – Open Test sequence eXchange format (OTX)*. International Organization for Standardization, Geneva, Switzerland, 2012.

[ISO98] ISO. *9241-11, Ergonomics of Human-Computer Interaction - Part 11: Guidance on Useability*. International Organization for Standardization, Geneva, Switzerland, 1998.

[JCZ13] Chuntao Jiang, Frans Coenen, and Michele Zito. »A survey of frequent subgraph mining algorithms«. In: *The Knowledge Engineering Review* 28.01 (2013), pp. 75–105.

[Kar+00] Christos T. Karamanolis, Dimitra Giannakopoulou, Jeff Magee, and Stuart M. Wheater. »Model checking of workflow schemas«. In: *Proceedings of the Fourth International Conference on Enterprise Distributed Object Computing EDOC*. 2000, pp. 170–179. DOI: `10.1109/EDOC.2000.882357`.

[Kli+13] Christopher Klinkmüller, Ingo Weber, Jan Mendling, Henrik Leopold, and André Ludwig. »Increasing Recall of Process Model Matching by Improved Activity Label Matching«. In: *Business Process Management*. Ed. by Florian Daniel, Jianmin Wang, and Barbara Weber. Lecture Notes in Computer Science 8094. Springer Berlin Heidelberg, Jan. 2013, pp. 211–218. ISBN: 978-3-642-40175-6 978-3-642-40176-3.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. »PRISM: Probabilistic Symbolic Model Checker«. en. In: *Computer Performance Evaluation: Modelling Techniques and Tools*. Ed. by Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder. Lecture Notes in Computer Science 2324. Springer Berlin Heidelberg, Jan. 2002, pp. 200–204. ISBN: 978-3-540-43539-6 978-3-540-46029-9.

[Knu+10] David Knuplesch, Linh Thao Ly, Stefanie Rinderle-Ma, Holger Pfeifer, and Peter Dadam. »On Enabling Data-Aware Compliance Checking of Business Process Models«. In: *Conceptual Modeling – ER 2010*. Ed. by Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand. Lecture Notes in Computer Science 6412. Springer Berlin Heidelberg, Jan. 2010, pp. 332–346. ISBN: 978-3-642-16372-2 978-3-642-16373-9.

[Kol96] Rainer Kolisch. »Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation«. In: *European Journal of Operational Research* 90.2 (Apr. 1996), pp. 320–333. ISSN: 0377-2217. DOI: `10.1016/0377-2217(95)00357-6`.

[Koo+08] David Koop, Carlos Eduardo Scheidegger, Steven P. Callahan, Juliana Freire, and Cláudio T. Silva. »VisComplete: Automating Suggestions for Visualization Pipelines«. In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (Nov. 2008), pp. 1691–1698. ISSN: 1077-2626. DOI: `10.1109/TVCG.2008.174`.

[Kop+09] Oliver Kopp, Daniel Martin, Daniel Wutke, and Frank Leymann. »The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages«. In: *Enterprise Modelling and Information Systems Architecture* 4.1 (2009), pp. 3–13.

[KRG07] Jochen M. Küster, Ksenia Ryndina, and Harald Gall. »Generation of Business Process Models for Object Life Cycle Compliance«. In: *Business Process Management*. Ed. by Gustavo Alonso, Peter Dadam, and Michael Rosemann. Lecture Notes in Computer Science 4714. Springer Berlin Heidelberg, Jan. 2007, pp. 165–181. ISBN: 978-3-540-75182-3 978-3-540-75183-0.

[KRL11] Sonja Kabicher, Stefanie Rinderle-Ma, and Linh Thao Ly. »Activity Oriented Clustering Techniques in Large Process and Compliance Rule Repositories«. In: *Proc. BPM'11 Workshops*. Clermont-Ferrand, France: Springer, 2011, pp. 14–25.

[KWW11] Matthias Kunze, Matthias Weidlich, and Mathias Weske. »Behavioral Similarity – A Proper Metric«. en. In: *Business Process Management*. Ed. by Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf. Lecture Notes in Computer Science 6896. Springer Berlin Heidelberg, Jan. 2011, pp. 166–181. ISBN: 978-3-642-23058-5 978-3-642-23059-2.

[Lam+08] Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari. »Applying Inductive Logic Programming to Process Mining«. en. In: *Inductive Logic Programming*. Ed. by Hendrik Blockeel, Jan Ramon, Jude Shavlik, and Prasad Tadepalli. Lecture Notes in Computer Science 4894. Springer Berlin Heidelberg, 2008, pp. 132–146. ISBN: 978-3-540-78468-5 978-3-540-78469-2.

[Lam00] Axel van Lamsweerde. »Formal Specification: A Roadmap«. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. New York, NY, USA: ACM, 2000, pp. 147–159. ISBN: 978-1-58113-253-3. DOI: 10.1145/336512.336546.

[Lau+09] Jean Michel Lau, Cirano Iochpe, Lucineia Thom, and Manfred Reichert. »Discovery and Analysis of Activity Pattern Cooccurrences in Business Process Models«. In: *Int'l Conf. on Enterprise Information Systems*. Milan, Italy, May 2009, pp. 83–88.

[Leo+15] Henrik Leopold, Christian Meilicke, Michael Fellmann, Fabian Pittke, Heiner Stuckenschmidt, and Jan Mendling. »Towards the Automated Annotation of Process Models«. en. In: *Advanced Information Systems Engineering*. Ed. by Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson. Lecture Notes in Computer Science 9097. Springer Berlin Heidelberg, June 2015, pp. 401–416. ISBN: 978-3-319-19068-6 978-3-319-19069-3.

[Lev66] Vladimir Iosifovič Levenštejn. »Binary Codes Capable of Correcting Deletions, Insertions and Reversals«. en. In: *Soviet Physics Doklady* 10 (Feb. 1966), p. 707.

[LF14] Niels Lohmann and Dirk Fahland. »Where Did I Go Wrong?« en. In: *Business Process Management*. Ed. by Shazia Sadiq, Pnina Soffer, and Hagen Völzer. Lecture Notes in Computer Science 8659. Springer Berlin Heidelberg, Jan. 2014, pp. 283–300. ISBN: 978-3-319-10171-2 978-3-319-10172-9.

[LFA13a] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. »Discovering Block-Structured Process Models from Event Logs - A Constructive Approach«. en. In: *Application and Theory of Petri*

Nets and Concurrency. Ed. by José-Manuel Colom and Jörg Desel. Lecture Notes in Computer Science 7927. Springer, Jan. 2013, pp. 311–329. ISBN: 978-3-642-38696-1 978-3-642-38697-8.

[LFA13b] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. »Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour«. en. In: *Business Process Management Workshops*. Ed. by Niels Lohmann, Minseok Song, and Petia Wohed. Lecture Notes in Business Information Processing 171. Springer Berlin Heidelberg, Aug. 2013, pp. 66–78. ISBN: 978-3-319-06256-3 978-3-319-06257-0.

[LFA14] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. »Discovering Block-Structured Process Models from Incomplete Event Logs«. en. In: *Application and Theory of Petri Nets and Concurrency*. Ed. by Gianfranco Ciardo and Ekkart Kindler. Lecture Notes in Computer Science 8489. Springer Berlin Heidelberg, Jan. 2014, pp. 91–110. ISBN: 978-3-319-07733-8 978-3-319-07734-5.

[Lin+02] Hao Lin, Zhibiao Zhao, Hongchen Li, and Zhiguo Chen. »A novel graph reduction algorithm to identify structural conflicts«. In: *Proceedings of the 35th Annual Hawaii International Conference on System Sciences, 2002. HICSS*. Jan. 2002, pp. 1–10. DOI: 10.1109/HICSS.2002.994506.

[LMX07] Ying Liu, Samuel Müller, and Ke Xu. »A static compliance-checking framework for business process models«. In: *IBM Systems Journal* 46.2 (2007), pp. 335–361. ISSN: 0018-8670. DOI: 10.1147/sj.462.0335.

[Loh13] Niels Lohmann. »Compliance by design for artifact-centric business processes«. In: *Information Systems*. Special section on BPM 2011 conference 38.4 (June 2013), pp. 606–618. ISSN: 0306-4379. DOI: 10.1016/j.is.2012.07.003.

[LP85] Orna Lichtenstein and Amir Pnueli. »Checking That Finite State Concurrent Programs Satisfy Their Linear Specification«. In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '85. New York, NY, USA: ACM, 1985, pp. 97–107. ISBN: 0-89791-147-4. DOI: 10.1145/318593.318622.

[LPM15] Henrik Leopold, Fabian Pittke, and Jan Mendling. »Automatic service derivation from business process model repositories via semantic technology«. In: *Journal of Systems and Software* 108 (2015), pp. 134–147. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.06.007.

[LRD10] Linh Thao Ly, Stefanie Rinderle-Ma, and Peter Dadam. »Design and Verification of Instantiable Compliance Rule Graphs in Process-Aware Information Systems«. en. In: *Advanced Information Systems Engineering*. Ed. by Barbara Pernici. Lecture Notes in Computer Science 6051. Springer Berlin Heidelberg, 2010, pp. 9–23. ISBN: 978-3-642-13093-9 978-3-642-13094-6.

[LRW09] Chen Li, Manfred Reichert, and Andreas Wombacher. »Discovering Reference Models by Mining Process Variants Using a Heuristic Approach«. en. In: *Business Process Management*. Ed. by Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers. Lecture Notes in Computer Science 5701. Springer Berlin Heidelberg, Jan. 2009, pp. 344–362. ISBN: 978-3-642-03847-1 978-3-642-03848-8.

[LS95] François Laroussinie and Philippe Schnoebelen. »A hierarchy of temporal logics with past«. In: *Theoretical Computer Science*. Selected Papers of the Eleventh Symposium on Theoretical Aspects of Computer Science 148.2 (Sept. 1995), pp. 303–324. ISSN: 0304-3975. DOI: 10.1016/0304-3975(95)00035-U.

[LVD09] Niels Lohmann, Eric Verbeek, and Remco Dijkman. »Petri Net Transformations for Business Processes – A Survey«. In: *Transactions on Petri Nets and Other Models of Concurrency II*. Ed. by Kurt Jensen and Wil M. P. van der Aalst. Lecture Notes in Computer Science 5460. Springer Berlin Heidelberg, Jan. 2009, pp. 46–63. ISBN: 978-3-642-00898-6 978-3-642-00899-3.

[Ly+08] Linh Thao Ly, Kevin Göser, Stefanie Rinderle-Ma, and Peter Dadam. »Compliance of Semantic Constraints - A Requirements Analysis for Process Management Systems«. In: Montpellier, France, June 2008.

[Ly+11a] Linh Thao Ly, David Knuplesch, Stefanie Rinderle-Ma, Kevin Göser, Holger Pfeifer, Manfred Reichert, and Peter Dadam. »SeaFlows Toolset – Compliance Verification Made Easy for Process-Aware Information Systems«. In: *Information Systems Evolution*. Ed. by Pnina Soffer and Erik Proper. Lecture Notes in Business Information Processing 72. Springer Berlin Heidelberg, Jan. 2011, pp. 76–91. ISBN: 978-3-642-17721-7 978-3-642-17722-4.

[Ly+11b] Linh Thao Ly, Stefanie Rinderle-Ma, David Knuplesch, and Peter Dadam. »Monitoring Business Process Compliance Using Compliance Rule Graphs«. en. In: *On the Move to Meaningful Internet Systems: OTM 2011*. Ed. by Robert Meersman et al. Lecture Notes in

Computer Science 7044. Springer Berlin Heidelberg, 2011, pp. 82–99. ISBN: 978-3-642-25108-5 978-3-642-25109-2.

[Ly13] Linh Thao Ly. *SeaFlows - a compliance checking framework for supporting the process lifecycle [Elektronische Ressource] / Linh Thao Ly*. Ulm: Universität Ulm. Fakultät für Ingenieurwissenschaften und Informatik, 2013.

[Mag+11] Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. »Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata«. en. In: *Business Process Management*. Ed. by Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf. Lecture Notes in Computer Science 6896. Springer Berlin Heidelberg, 2011, pp. 132–147. ISBN: 978-3-642-23058-5 978-3-642-23059-2.

[Mag+12] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. »Runtime Verification of LTL-Based Declarative Process Models«. en. In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. LNCS 7186. Springer Berlin Heidelberg, 2012, pp. 131–146. ISBN: 978-3-642-29859-2.

[Mag+13] Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and Marco Montali. »Discovering Data-Aware Declarative Process Models from Event Logs«. en. In: *Business Process Management*. Ed. by Florian Daniel, Jianmin Wang, and Barbara Weber. Lecture Notes in Computer Science 8094. Springer Berlin Heidelberg, 2013, pp. 81–96. ISBN: 978-3-642-40175-6 978-3-642-40176-3.

[MBA12] Fabrizio Maria Maggi, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. »Efficient Discovery of Understandable Declarative Process Models from Event Logs«. en. In: *Advanced Information Systems Engineering*. Ed. by Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza. Lecture Notes in Computer Science 7328. Springer Berlin Heidelberg, 2012, pp. 270–285. ISBN: 978-3-642-31094-2 978-3-642-31095-9.

[MBG14] Mirjam Minor, Ralph Bergmann, and Sebastian Görg. »Case-based adaptation of workflows«. In: *Elsevier Information Systems* 40 (2014), pp. 142–152. ISSN: 0306-4379. DOI: 10.1016/j.is.2012.11.011.

[MCA13a] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. »Conformance Checking in the Large: Partitioning and Topology«. en. In: *Business Process Management*. Ed. by Florian Daniel, Jianmin Wang, and Barbara Weber. Lecture Notes in Computer Science

8094. Springer Berlin Heidelberg, 2013, pp. 130–145. ISBN: 978-3-642-40175-6 978-3-642-40176-3.

[MCA13b] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. »Hierarchical Conformance Checking of Process Models Based on Event Logs«. en. In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Lecture Notes in Computer Science 7927. Springer Berlin Heidelberg, 2013, pp. 291–310. ISBN: 978-3-642-38696-1 978-3-642-38697-8.

[Men+08] Jan Mendling, H. M. W. Verbeek, Boudewijn F. van Dongen, Wil M. P. van der Aalst, and G. Neumann. »Detection and prediction of errors in EPCs of the SAP reference model«. In: *Data & Knowledge Engineering*. Fourth International Conference on Business Process Management (BPM 2006) Four selected and extended papers 8th International Conference on Enterprise Information Systems (ICEIS' 2006) Three selected and extended papers 64.1 (Jan. 2008), pp. 312–329. ISSN: 0169-023X. DOI: 10.1016/j.datak.2007.06.019.

[Men09] Jan Mendling. »Empirical Studies in Process Model Verification«. In: *Transactions on Petri Nets and Other Models of Concurrency II*. Ed. by Kurt Jensen and Wil M. P. van der Aalst. Lecture Notes in Computer Science 5460. Springer Berlin Heidelberg, Jan. 2009, pp. 208–224. ISBN: 978-3-642-00898-6 978-3-642-00899-3.

[MGP15] Steven Mertens, Frederik Gailly, and Geert Poels. »Enhancing Declarative Process Models with DMN Decision Logic«. en. In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma. Lecture Notes in Business Information Processing 214. Springer Berlin Heidelberg, June 2015, pp. 151–165. ISBN: 978-3-319-19236-9 978-3-319-19237-6.

[Min11] Clemente Minonne. *Business-Process-Management 2011 - Status quo und Zukunft: eine empirische Studie im deutschsprachigen Raum*. de. vdf Hochschulverlag AG, 2011. ISBN: 978-3-7281-3402-8.

[MM05] Ross M. McConnell and Fabien de Montgolfier. »Linear-time modular decomposition of directed graphs«. In: *Discrete Applied Mathematics*. Structural Decompositions, Width Parameters, and Graph Labelings 145.2 (Jan. 2005), pp. 198–209. ISSN: 0166-218X. DOI: 10.1016/j.dam.2004.02.017.

[MMA11] Fabrizio Maria Maggi, Arjan J. Mooij, and Wil M. P. van der Aalst. »User-guided discovery of declarative process models«. In: *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Apr. 2011, pp. 192–199. DOI: 10.1109/CIDM.2011.5949297.

[MMB14a] Richard Mrasek, Jutta Mülle, and Klemens Böhm. »A new verification technique for large processes based on identification of relevant tasks«. In: *Information Systems* (2014). ISSN: 0306-4379. DOI: 10.1016/j.is.2014.07.001.

[MMB14b] Richard Mrasek, Jutta Mülle, and Klemens Böhm. *Automatic Generation of Optimized Process Models from Declarative Specifications*. en. Technical Report 2014-15. Karlsruhe: KIT Scientific Publishing, Nov. 2014.

[MMB15] Richard Mrasek, Jutta Mülle, and Klemens Böhm. »Automatic Generation of Optimized Process Models from Declarative Specifications«. In: *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2015, pp. 382–397.

[MMS02] Daniel Merkle, Martin Middendorf, and Hartmut Schmeck. »Ant colony optimization for resource-constrained project scheduling«. In: *IEEE Transactions on Evolutionary Computation* 6.4 (Aug. 2002), pp. 333–346. ISSN: 1089-778X. DOI: 10.1109/TEVC.2002.802450.

[MNA07] Jan Mendling, Gustaf Neumann, and Wil M. P. van der Aalst. »Understanding the Occurrence of Errors in Process Models Based on Metrics«. In: *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*. Ed. by Robert Meersman and Zahir Tari. Lecture Notes in Computer Science 4803. Springer Berlin Heidelberg, Jan. 2007, pp. 113–130. ISBN: 978-3-540-76846-3 978-3-540-76848-7.

[Mon+08] Marco Montali, Paolo Torroni, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello. »Verification from Declarative Specifications Using Logic Programming«. en. In: *Logic Programming*. Ed. by Maria Garcia de la Banda and Enrico Pontelli. Lecture Notes in Computer Science 5366. Springer Berlin Heidelberg, 2008, pp. 440–454. ISBN: 978-3-540-89981-5 978-3-540-89982-2.

[Mon+10] Marco Montali, Maja Pešić, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. »Declarative Specification and Verification of Service Choreographies«. In: *ACM Trans. Web*

4.1 (Jan. 2010), 3:1–3:62. ISSN: 1559-1131. DOI: `10.1145/1658373.1658376`.

[MR02] Mihhail Matskin and Jinghai Rao. »Value-Added Web Services Composition Using Automatic Program Synthesis«. en. In: *Web Services, E-Business, and the Semantic Web*. Ed. by Christoph Bussler, Richard Hull, Sheila McIlraith, Maria E. Orlowska, Barbara Pernici, and Jian Yang. Lecture Notes in Computer Science 2512. Springer, Jan. 2002, pp. 213–224. ISBN: 978-3-540-00198-0 978-3-540-36189-3.

[Mra+14] Richard Mrasek, Jutta Mülle, Klemens Böhm, Christian Allmann, and Michael Becker. »User-Friendly Property Specification and Process Verification - a Case Study with Vehicle-Commissioning Processes«. In: *Business Process Management*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 326–341. ISBN: 978-3-540-85757-0 978-3-540-85758-7.

[Mra+15] Richard Mrasek, Jutta Mülle, Klemens Böhm, Michael Becker, and Christian Allmann. »Property specification, process verification, and reporting – a case study with vehicle-commissioning processes«. In: *Information Systems* (2015). ISSN: 0306-4379. DOI: `10.1016/j.is.2015.09.005`.

[MRA10] Jan Mendling, Hajo A. Reijers, and W. M. P. van der Aalst. »Seven process modeling guidelines (7PMG)«. In: *Information and Software Technology* 52.2 (Feb. 2010), pp. 127–136. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2009.08.004`.

[MRC07] Jan Mendling, Hajo A. Reijers, and Jorge Cardoso. »What Makes Process Models Understandable?« In: *Business Process Management*. Ed. by Gustavo Alonso, Peter Dadam, and Michael Rosemann. Lecture Notes in Computer Science 4714. Springer Berlin Heidelberg, Jan. 2007, pp. 48–63. ISBN: 978-3-540-75182-3 978-3-540-75183-0.

[MSR14] Fabrizio Maria Maggi, Tijs Slaats, and Hajo A. Reijers. »The Automated Discovery of Hybrid Processes«. en. In: *Business Process Management*. Ed. by Shazia Sadiq, Pnina Soffer, and Hagen Völzer. Lecture Notes in Computer Science 8659. Springer Berlin Heidelberg, Sept. 2014, pp. 392–399. ISBN: 978-3-319-10171-2 978-3-319-10172-9.

[MWA07] A. K. Alves de Medeiros, A. J. M. M. Weijters, and Wil M. P. van der Aalst. »Genetic process mining: an experimental evaluation«. en. In: *Data Mining and Knowledge Discovery* 14.2 (Apr. 2007), pp. 245–304. ISSN: 1384-5810, 1573-756X. DOI: `10.1007/s10618-006-0061-7`.

[OMG11] Object Management Group (OMG). *Business Process Model and Notation (BPMN)*. Standard document. Jan. 2011.

[PA06] Maja Pešić and Wil M. P. van der Aalst. »A Declarative Approach for Flexible Business Processes Management«. en. In: *Business Process Management Workshops*. Ed. by Johann Eder and Schahram Dustdar. Lecture Notes in Computer Science 4103. Springer Berlin Heidelberg, Jan. 2006, pp. 169–180. ISBN: 978-3-540-38444-1 978-3-540-38445-8.

[PBA10] Maja Pešić, Dragan Bošnački, and Wil M. P. van der Aalst. »Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance«. In: *Model Checking Software*. Ed. by Jaco van de Pol and Michael Weber. Lecture Notes in Computer Science 6349. Springer Berlin Heidelberg, Jan. 2010, pp. 146–161. ISBN: 978-3-642-16163-6 978-3-642-16164-3.

[Pel09] Radek Pelánek. »Fighting State Space Explosion: Review and Evaluation«. en. In: *Formal Methods for Industrial Critical Systems*. Ed. by Darren Cofer and Alessandro Fantechi. Lecture Notes in Computer Science 5596. Springer Berlin Heidelberg, Jan. 2009, pp. 37–52. ISBN: 978-3-642-03239-4 978-3-642-03240-0.

[PGD12] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. »Structuring acyclic process models«. In: *Information Systems*. BPM 2010 37.6 (Sept. 2012), pp. 518–538. ISSN: 0306-4379. DOI: `10.1016/j.is.2011.10.005`.

[Pic+12] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. »Imperative versus Declarative Process Modeling Languages: An Empirical Investigation«. en. In: *Business Process Management Workshops*. Ed. by Florian Daniel, Kamel Barkaoui, and Schahram Dustdar. Lecture Notes in Business Information Processing 99. Springer Berlin Heidelberg, 2012, pp. 383–394. ISBN: 978-3-642-28107-5 978-3-642-28108-2.

[Pnu77] Amir Pnueli. »The temporal logic of programs«. In: , *18th Annual Symposium on Foundations of Computer Science, 1977*. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[Pol+14] Artem Polyvyanyy, Matthias Weidlich, Raffaele Conforti, Marcello La Rosa, and Arthur H. M. ter Hofstede. »The 4C Spectrum of Fundamental Behavioral Relations for Concurrent Systems«. en. In: *Application and Theory of Petri Nets and Concurrency*. Ed. by Gianfranco Ciardo and Ekkart Kindler. Lecture Notes in Computer

Science 8489. Springer International Publishing, June 2014, pp. 210–232. ISBN: 978-3-319-07733-8 978-3-319-07734-5.

[Pol12] Artem Polyvyanyy. *Structuring Process Models*. Doctoral dissertation. Potsdam: University of Potsdam, Jan. 2012.

[PSA07] Maja Pešić, Helen Schonenberg, and Wil M. P. van der Aalst. »DECLARE: Full Support for Loosely-Structured Processes«. In: *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. 2007, pp. 287–287. DOI: 10.1109/EDOC.2007.14.

[Rae+07] Ivo Raedts, Marija Petković, Yaroslav S. Usenko, Jan Martijn Van Der Werf, Jan Friso Groote, and Lou Somers. *Transformation of BPMN models for Behaviour Analysis*. 2007.

[Rat+15] Michael Rathmair, Ralph Hoch, Hermann Kaindl, and Roman Popp. »Consistently Formalizing a Business Process and its Properties for Verification: A Case Study«. en. In: *The Practice of Enterprise Modeling*. Ed. by Jolita Ralyté, Sergio España, and Óscar Pastor. Lecture Notes in Business Information Processing 235. DOI: 10.1007/978-3-319-25897-3_9. Springer International Publishing, Nov. 2015, pp. 126–140. ISBN: 978-3-319-25896-6 978-3-319-25897-3.

[RDC14] Carlos Rodriguez, Florian Daniel, and Fabio Casati. »Crowd-Based Mining of Reusable Process Model Patterns«. In: *Business Process Management*. 2014.

[Rei10] Alexander W. Reichhuber. »Grundlagen und Herausforderungen der Automobilindustrie«. de. In: *Strategie und Struktur in der Automobilindustrie*. DOI: 10.1007/978-3-8349-8496-8_2. Gabler, 2010, pp. 15–64. ISBN: 978-3-8349-2218-2 978-3-8349-8496-8.

[RFA12] Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. »Where Did I Misbehave? Diagnostic Information in Compliance Checking«. en. In: *Business Process Management*. Ed. by Alistair P. Barros, Avigdor Gal, and Ekkart Kindler. Lecture Notes in Computer Science 7481. Springer Berlin Heidelberg, Jan. 2012, pp. 262–278. ISBN: 978-3-642-32884-8 978-3-642-32885-5.

[RM11] Hajo A. Reijers and Jan Mendling. »A Study Into the Factors That Influence the Understandability of Business Process Models«. In: *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 41.3 (2011), pp. 449–462. ISSN: 1083-4427. DOI: 10.1109/TSMCA.2010.2087017.

[Ros+06] Michael Rosemann, Jan C. Recker, Christian Flender, and Peter D. Ansell. »Understanding context-awareness in business process design«. In: *Faculty of Science and Technology; Institute for Creative Industries and Innovation*. Ed. by Su Spencer and Adam Jenkins. Adelaide, Australia: Australasian Association for Information Systems, 2006. ISBN: 978-0-9758417-1-6.

[RWM10] Fazle Rabbi, Hao Wang, and Wendy MacCaull. »YAWL2DVE: An Automated Translator for Workflow Verification«. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*. June 2010, pp. 53–59. DOI: 10.1109/SSIRI.2010.31.

[Sau11] Jeff Sauro. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. en. CreateSpace Independent Publishing Platform, 2011. ISBN: 978-1-4610-6270-7.

[Sch+10a] David Schumm, Frank Leymann, Zhilei Ma, Thorsten Scheibler, and Steve Strauch. »Integrating Compliance into Business Processes«. In: *Multikonferenz Wirtschaftsinformatik*. 2010.

[Sch+10b] David Schumm, Oktay Turetken, Natallia Kokash, Amal Elgammal, Frank Leymann, and Willem-Jan van den Heuvel. »Business Process Compliance through Reusable Units of Compliant Processes«. In: *Current Trends in Web Engineering*. Ed. by Florian Daniel and Federico Michele Facca. Lecture Notes in Computer Science 6385. Springer Berlin Heidelberg, Jan. 2010, pp. 325–337. ISBN: 978-3-642-16984-7 978-3-642-16985-4.

[Sch+11] Daniel Schleicher, Stefan Grohe, Frank Leymann, Patrick Schneider, David Schumm, and Tamara Wolf. »An approach to combine data-related and control-flow-related compliance rules«. In: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 2011, pp. 1–8. DOI: 10.1109/SOCA.2011.6166212.

[Sch00a] Karsten Schmidt. »LoLA A Low Level Analyser«. en. In: *Application and Theory of Petri Nets 2000*. Ed. by Mogens Nielsen and Dan Simpson. Lecture Notes in Computer Science 1825. Springer Berlin Heidelberg, Jan. 2000, pp. 465–474. ISBN: 978-3-540-67693-5 978-3-540-44988-1.

[Sch00b] Karsten Schmidt. »Stubborn Sets for Model Checking the EF/AG Fragment of CTL«. In: *Fundam. Inf.* 43.1-4 (Aug. 2000), pp. 331–341. ISSN: 0169-2968.

[Sch12] Tobias Schneider. *Specification of Testing Workflows for Vehicles and Validation of Manually Created Testing Processes*. German. Karlsruhe Insititute of Technology, Master Thesis, May 2012.

[Sch99] Karsten Schmidt. »Stubborn Sets for Standard Properties«. en. In: *Application and Theory of Petri Nets 1999*. Ed. by Susanna Donatelli and Jetty Kleijn. Lecture Notes in Computer Science 1639. Springer Berlin Heidelberg, Jan. 1999, pp. 46–65. ISBN: 978-3-540-66132-0 978-3-540-48745-6.

[Sen+15] Arik Senderovich, Matthias Weidlich, Avigdor Gal, Avishai Mandelbaum, Sarah Kadish, and Craig A. Bunnell. »Discovery and Validation of Queueing Networks in Scheduled Processes«. en. In: *Advanced Information Systems Engineering*. Ed. by Jelena Zdravkovic, Marite Kirikova, and Paul Johannesson. Lecture Notes in Computer Science 9097. Springer International Publishing, June 2015, pp. 417–433. ISBN: 978-3-319-19068-6 978-3-319-19069-3.

[SGN07] Shazia Sadiq, Guido Governatori, and Kioumars Namiri. »Modeling Control Objectives for Business Process Compliance«. en. In: *Business Process Management*. Ed. by Gustavo Alonso, Peter Dadam, and Michael Rosemann. Lecture Notes in Computer Science 4714. Springer Berlin Heidelberg, 2007, pp. 149–164. ISBN: 978-3-540-75182-3 978-3-540-75183-0.

[Smi+02] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. »PROPEL: An Approach Supporting Property Elucidation«. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. New York, NY, USA: ACM, 2002, pp. 11–21. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581345.

[Smi+12] Sergey Smirnov, Matthias Weidlich, Jan Mendling, and Mathias Weske. »Action patterns in business process model repositories«. In: *Computers in Industry*. Managing Large Collections of Business Process Models Managing Large Collections of Business Process Models 63.2 (Feb. 2012), pp. 98–111. ISSN: 0166-3615. DOI: 10.1016/j.compind.2011.11.001.

[SMS05] Holger Schlingloff, Axel Martens, and Karsten Schmidt. »Modeling and Model Checking Web Services«. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 2nd International Workshop on Logic and Communication in Multi-Agent Systems (2004) Logic and Communication in Multi-Agent Systems 2004 126 (2005), pp. 3–26. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.11.011.

[SO00] Wasim Sadiq and Maria E. Orlowska. »Analyzing process models using graph reduction techniques«. In: *Information Systems*. The 11th International Conference on Advanced Information System Engineering 25.2 (Apr. 2000), pp. 117–134. ISSN: 0306-4379. DOI: `10.1016/S0306-4379(00)00012-0`.

[SO99] Wasim Sadiq and Maria E. Orlowska. »Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models«. en. In: *Advanced Information Systems Engineering*. Ed. by Matthias Jarke and Andreas Oberweis. Lecture Notes in Computer Science 1626. Springer Berlin Heidelberg, 1999, pp. 195–209. ISBN: 978-3-540-66157-3 978-3-540-48738-8.

[SSO01] Shazia Sadiq, Wasim Sadiq, and Maria Orlowska. »Pockets of Flexibility in Workflow Specification«. en. In: *Conceptual Modeling — ER 2001*. Ed. by Hideko S.Kunii, Sushil Jajodia, and Arne Sølvberg. Lecture Notes in Computer Science 2224. Springer Berlin Heidelberg, 2001, pp. 513–526. ISBN: 978-3-540-42866-4 978-3-540-45581-3.

[Sta+14] Silvia von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm. »Detecting Data-Flow Errors in BPMN 2.0«. In: *Open Journal of Information Systems (OJIS)* 2.1 (2014), pp. 1–19. ISSN: 2198-9281.

[Sta05] Christian Stahl. *A Petri Net Semantics for BPEL*. Tech. rep. 188. Humboldt-Universität zu Berlin, Jan. 2005.

[Tag+13] Elham Ramezani Taghiabadi, Dirk Fahland, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. »Diagnostic Information for Compliance Checking of Temporal Compliance Requirements«. en. In: *Advanced Information Systems Engineering*. Ed. by Camille Salinesi, Moira Norrie, and Óscar Pastor. Lecture Notes in Computer Science 7908. Springer Berlin Heidelberg, Jan. 2013, pp. 304–320. ISBN: 978-3-642-38708-1 978-3-642-38709-8.

[TAS09] Nikola Trčka, Wil M. P. van der Aalst, and Natalia Sidorova. »Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows«. In: *Advanced Information Systems Engineering*. Ed. by Pascal van Eck, Jaap Gordijn, and Roel Wieringa. Lecture Notes in Computer Science 5565. Springer Berlin Heidelberg, Jan. 2009, pp. 425–439. ISBN: 978-3-642-02143-5 978-3-642-02144-2.

[TMB16] Christine Tex, Jutta Mülle, and Klemens Böhm. »A Practical Data-Flow Verification Scheme for Business Processes«. In: *Karlsruhe Reports in Informatics* (2016).

[TRI09] Lucineia Heloisa Thom, Manfred Reichert, and Cirano Iochpe. »Activity patterns in process-aware information systems: basic concepts and empirical evidence«. In: *International Journal of Business Process Integration and Management* 4.2 (Jan. 2009), pp. 93–110. DOI: 10.1504/IJBPIM.2009.027778.

[VA14] H. M. W. Verbeek and Wil M. P. van der Aalst. »Decomposed Process Mining: The ILP Case«. en. In: *Business Process Management Workshops*. Ed. by Fabiana Fournier and Jan Mendling. Lecture Notes in Business Information Processing 202. Springer Berlin Heidelberg, Sept. 2014, pp. 264–276. ISBN: 978-3-319-15894-5 978-3-319-15895-2.

[VBA01] H. M. W. Verbeek, T. Basten, and Wil M. P. van der Aalst. »Diagnosing Workflow Processes using Woflan«. en. In: *The Computer Journal* 44.4 (Jan. 2001), pp. 246–279. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/44.4.246.

[VL93] Bart Vergauwen and Johan Lewi. »A linear local model checking algorithm for CTL«. In: *CONCUR'93*. Ed. by Eike Best. Lecture Notes in Computer Science 715. Springer Berlin Heidelberg, Jan. 1993, pp. 447–461. ISBN: 978-3-540-57208-4 978-3-540-47968-0.

[VM91] József Váncza and András Márkus. »Genetic algorithms in process planning«. In: *Computers in Industry*. Special Issue IMS '91-Learning in IMS 17.2–3 (Nov. 1991), pp. 181–194. ISSN: 0166-3615. DOI: 10.1016/0166-3615(91)90031-4.

[VVK09] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. »The refined process structure tree«. In: *Data & Knowledge Engineering*. Sixth International Conference on Business Process Management (BPM 2008) – Five selected and extended papers 68.9 (Sept. 2009), pp. 793–818. ISSN: 0169-023X. DOI: 10.1016/j.datak.2009.02.015.

[WA03] A. J. M. M. Weijters and Wil M. P. van der Aalst. »Rediscovering workflow models from event-based data using little thumb«. In: *Integrated Computer-Aided Engineering* 10.2 (Jan. 2003), pp. 151–162.

[Wei+11a] Matthias Weidlich, Artem Polyvyanyy, Nirmit Desai, Jan Mendling, and Mathias Weske. »Process compliance analysis based on behavioural profiles«. In: *Information Systems*. Special Issue: Advanced Information Systems Engineering (CAiSE'10) 36.7 (Nov. 2011), pp. 1009–1025. ISSN: 0306-4379. DOI: 10.1016/j.is.2011.04.002.

[Wei+11b] Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. »Causal Behavioural Profiles - Efficient Computation, Applications, and Evaluation«. In: *Fundam. Inf.* 113.3-4 (Aug. 2011), pp. 399–435. ISSN: 0169-2968.

[Wer+08] Jan Martijn van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens, and Alexander Serebrenik. »Process Discovery Using Integer Linear Programming«. en. In: *Applications and Theory of Petri Nets*. Ed. by Kees M. van Hee and Rüdiger Valk. Lecture Notes in Computer Science 5062. Springer Berlin Heidelberg, 2008, pp. 368–387. ISBN: 978-3-540-68745-0 978-3-540-68746-7.

[WMW11] Matthias Weidlich, Jan Mendling, and Mathias Weske. »Efficient Consistency Measurement Based on Behavioral Profiles of Process Models«. In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 410–429. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.96.

[Wol07] Karsten Wolf. »Generating Petri Net State Spaces«. In: *Petri Nets and Other Models of Concurrency – ICATPN 2007*. Ed. by Jetty Kleijn and Alex Yakovlev. Lecture Notes in Computer Science 4546. Springer Berlin Heidelberg, Jan. 2007, pp. 29–42. ISBN: 978-3-540-73093-4 978-3-540-73094-1.

[Wom+07] James P. Womack, Daniel T. Jones, Daniel Roos, and Massachusetts Institute of Technology. *Machine that Changed the World*. en. London: Free Press, 2007. ISBN: 978-0-89256-350-0.

[Wyn+09a] Moe T. K. W. Wynn, H. M. W. Verbeek, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. »Reduction rules for YAWL workflows with cancellation regions and OR-joins«. In: *Information and Software Technology* 51.6 (June 2009), pp. 1010–1020. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.12.002.

[Wyn+09b] Moe T. K. W. Wynn, H. M. W. Verbeek, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. »Soundness-preserving reduction rules for reset workflow nets«. In: *Information Sciences* 179.6 (2009), pp. 769–790. ISSN: 0020-0255. DOI: 10.1016/j.ins.2008.10.033.

[Yan+12] Yong Yang, Marlon Dumas, Luciano García-Bañuelos, Artem Polyvyanyy, and Liang Zhang. »Generalized aggregate Quality of Service computation for composite services«. In: *Journal of Systems and Software* 85.8 (Aug. 2012), pp. 1818–1830. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.03.005.

[Yu+06] Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. »Pattern Based Property Specification and Verification for Service Composition«. en. In: *Web Information Systems – WISE 2006*. Ed. by Karl Aberer, Zhiyong Peng, Elke A. Rundensteiner, Yanchun Zhang, and Xuhui Li. Lecture Notes in Computer Science 4255. Springer Berlin Heidelberg, 2006, pp. 156–168. ISBN: 978-3-540-48105-8 978-3-540-48107-2.

[Yu+08] Jian Yu, Yan-Bo Han, Jun Han, Yan Jin, Paolo Falcarin, and Maurizio Morisio. »Synthesizing Service Composition Models on the Basis of Temporal Business Rules«. en. In: *Journal of Computer Science and Technology* 23.6 (Nov. 2008), pp. 885–894. ISSN: 1000-9000, 1860-4749. DOI: `10.1007/s11390-008-9196-x`.

[ZS11] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur*. German. Springer Berlin Heidelberg, 2011. ISBN: ISBN: 978-3-658-02418-5.

List of Figures

List of Tables

Glossary

BPMN : **B**usiness **P**rocess **M**odel **N**otation (BPMN) is a graphical graph-based notation, used to model business process models. 4, 21, 37, 69, 97, 119, 120, 134, 140

CAN : **C**ontroller **A**rea **N**etwork (CAN) is a serial communication bus system. CAN is developed by BOSCH in 1983 for the communication of controlling units in a vehicle, and later becomes several ISO standards. The two most common implementation are the Highspeed-CAN and the Lowspeed-CAN. In 2012 BOSCH present the extension CAN FD (**F**lexible **Da**ta Rate) allowing a higher bandwidth. 9

CTL : **C**omputation **T**ree **L**ogic (CTL) is a temporal logic with a branching time concept, i. e., more than one possible futures. Ctl is used to specify properties for a formal verification, e. g., model checking. 37, 39–41, 48–51, 69, 86, 87, 90, 91, 94, 96, 119, 121, 122, 197, 201

CTL* : **C**omputation **T**ree **L**ogic Star (CTL*) is a temporal logic with a branching time concept, i. e., more than one possible futures. Ctl* is a superset of both CTL and LTL. It is used to specify properties for a formal verification, e. g., model checking. 39, 40, 48, 49

Declare : Declare is a framework and process notation for the declarative specification of a process model in contrast to imperative languages like BPMN, YAWL, or Petri nets.. 135–137, 199

ECU : **E**lectronic **C**ontrol **U**nit (ECU) in the automobile context is an embedded system, controlling a specific system in a vehicle, e.g., the engine electronic.. 10, 13, 15, 46–48, 50, 65, 82, 86, 87, 89–91, 100, 101, 109, 201

EPC : **E**vent-driven **P**rocess **C**hains (EPC) are a graphical graph-based process modeling language. EPCs have been developed by the University of Saarland together with the SAP AG in 1992.. 119, 120

KWP2000: **K**ey**W**ord **P**rotocol 2000 (KWP2000) is a communication protocol for vehicle diagnosis. 10, 44, 46, 73, 87

LIN: **L**ocal **I**nterconnect **N**etwork (LIN) is a serial communication protocol. It is a cost-efficient protocol for the connections of sensors and actors in a vehicle. 9

LTL: **L**inear **T**emporal **L**ogic (LTL) is a temporal logic with a linear time concept, i. e., every event has one possible future. Ltl is used to specify properties for a formal verification, e. g., model checking. 39–41, 48, 49, 69, 70, 121, 161

MOST: **M**edia **O**riented **S**ystems **T**ransport (MOST) is a serial bus system for the transport of audio, video, language, and data in a vehicle. 9

MPS: The **M**obile **P**rüf**S**tation (mobile testing station) is a mobile computer system with dedicated software for the commissioning of vehicles. It connects to the vehicle over a cable and communicates to the factory worker over wireless hand terminals.. 1

ODX: **O**pen **D**iagnostic Data e**X**change (ODX) is a XML-based formal description language for the diagnosis of vehicles.. 13

ORG: **O**rdering **R**elation **G**raph is a graph structure with several edge types that represent the allowed ordering of tasks in a process model, i. e., the ORG is a declarative specification of a process model.. 133, 135–137, 199

OTX: The **O**pen **T**est sequence e**X**change (OTX) is a formal process notation for commissioning processes in the automobile industry. 4–6, 8, 12, 14–16, 21, 28, 37, 43, 45, 74–77, 81–83, 86, 98, 99, 120, 128, 134

RCSP: The **R**essource **C**onstraint **S**cheduling **P**roblem (RCSP) is an optimizing problem to find for a given set of activities an optimal execution. To consider are predecessor relations between the activities, and capacity dependencies for resources. 124

RPST: **R**efined **P**rocess **S**tructure **T**ree) (RPST) is a decomposition of a graph-based notation into a block-based hierarchy. 21

UDS: **U**nified **D**iagnostic **S**ervices (UDS) is a communication protocol to communicate with ECUs in an automobile. 10, 44, 46

WS-BPEL : **W**eb **S**ervices **B**usiness **P**rocess **E**xecution **L**anguage (WS-BPEL) is an executable block-based process notation for business processes containing web services. 21, 37, 120, 134, 140

XML : E**X**tensible **M**arkup **L**anguage (XML) is an markup language for documents that are both human-readable and machine-readable.. 13, 14, 45

YAWL : **Y**et **A**nother **W**orkflow **L**anguage (YAWL) is a graph based process notation based on the workflow patterns. 4, 21, 37, 69, 117