Property Specification, Process Verification, and Reporting – a Case Study with Vehicle-Commissioning Processes

Richard Mrasek[a], Jutta Mülle[a], Klemens Böhm[a], Michael Becker[b], Christian Allmann[b]

[a] *Karlsruhe Institute of Technology (KIT)*
Institute for Program Structures and Data Organization
76131 Karlsruhe, Germany
[b] *AUDI AG*
85045 Ingolstadt, Germany

Abstract

Testing in the automotive industry is supposed to guarantee that vehicles are shipped without any flaw. Respective processes are complex, due to the variety of components and electronic devices in modern vehicles. To achieve error-free processes, their formal analysis is required. Specifying and maintaining properties the processes must satisfy in a user-friendly way is a core requirement on any verification system. We have observed that there are few property templates that testing processes must adhere to, and we describe these templates. They depend on the context of the processes, e. g., the components of the vehicle or testing stations. We have developed a framework that instantiates the templates of properties at verification time and then verifies the process against these instances. To allow an automatic verification we develop a transformation of the commissioning process to a Petri net. Using a novel approach, we are able to report the found violations to the user in a user-friendly way. Our empirical evaluation with the industrial partner has shown that our framework does detect property violations in processes. From expert interviews we conclude that our framework is user-friendly and well suited to operate in a real production environment.

Keywords: Property Specification, Business Process Management, Workflow Management, Verification, Model Checking, Petri Net, Industrial Processes, Vehicle Commissioning Processes

Email addresses: `richard.mrasek@kit.edu` (Richard Mrasek), `jutta.muelle@kit.edu` (Jutta Mülle), `klemens.boehm@kit.edu` (Klemens Böhm), `michael1.becker@audi.de` (Michael Becker), `christian.allmann@audi.de` (Christian Allmann)

1. Introduction

The systematic testing and configuration of complex products, e. g., vehicles, is an important part of a production process. For testing and configuring, certain tasks need to be executed, automatically or with the help of a human. So-called commissioning tasks test a component or put it into service, e. g., configure the software [1]. Workflows called commissioning processes describe the arrangement of these tasks. The scenario dealt with in this article is one where domain experts from industry develop the commissioning processes. Workflow management systems (WfMS) in the production domain that coordinate the testing and end-of-line manufacturing of the items produced are referred to as diagnostic frameworks.

Our overall goal is to verify if a given commissioning process is correct. This means checking that it 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. As just mentioned, for verification it is necessary to specify properties. We have collected such properties in cooperation with domain experts from industry by analyzing existing processes, and by closely observing these experts when designing processes.

Example 1. *Some tasks use a specific resource with a limited capacity. A property is that a resource must not be accessed more often than the maximum capacity. The consequence of the violation would be that the process must halt. In this case process execution time is unnecessarily long.*

A common definition of correctness of a process is that it fulfills all properties required. Recent research [2, 3] has shown, and we have observed this as well, that process models do not always comply with all properties required. Properties typically are formulated as property rules, which are similar to compliance rules [4, 5]. For example, a property rule states that before executing Task `X` another Task `Y` has to be executed.

Verification is itself 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. This gives way to the following questions. First, how must processes as well as the properties be specified to facilitate the deployment of verification techniques? Second, how to utilize domain information to support the users specifying the formal properties? Finally, how user-friendly are respective solutions? To verify process models given in a formal representation like Petri nets against properties, there already exist efficient model checking approaches [6, 7]. However, deriving and specifying the properties the model must satisfy is another issue. A core question is how a user-friendly framework for process verification should look like.

Designing such a framework gives way to several challenges: First, the knowledge on which characteristics an industrial process should fulfill is typically

distributed among several employees in different departments. Often a documentation is missing, and properties merely exist in the minds of the process modelers. Second, 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 [8]. For instance, an average process model from our use case has to comply with 39 properties. The properties and process model are constantly being revised. This leads to serious maintenance problems. Third, to apply an automatic verification technique like model checking, it is necessary to specify the properties in a formal language such as a temporal logic [9]. With vehicle-commissioning processes as well as in other domains, see, for instance [10], [11], specifying the properties in this way is error-prone and generally infeasible for domain experts who are not used to formal specification. To facilitate an automatic verification, the process must be formalized in a notation that allows to directly construct its state space. To this end, it must be easy to let the properties refer to the processes modeled. Fourth, it is challenging to present the violations found to the user in a way that is both succinct and understandable. Fifth, evaluating an approach such as the one envisioned is difficult. One issue is that the evaluation criteria must be specified.

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 templates these properties adhere to. We propose to explicitly represent these templates, rather than each individual property. Next, we develop a model of the context knowledge regarding vehicle-commissioning processes. Here *context*, is 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 templates, transforms the process models to a Petri net, and verifies the models against these properties, see Figure 1. For the verification we rely on a transformation of the notation OTX for commissioning processes to Petri nets which we have developed ourselves. Our framework is able to interpret the verification results and the characteristics of the process that most likely are responsible for a property violation. The tool uses the verification result to highlight the important elements in a visualization of the process. We use a template-specific approach whose output is more concise than the one of a generic solution. Our evaluation has shown that the framework as a whole does detect rule violations in actual real-world commissioning processes. Further, we have evaluated whether our model of the context together with the rules is

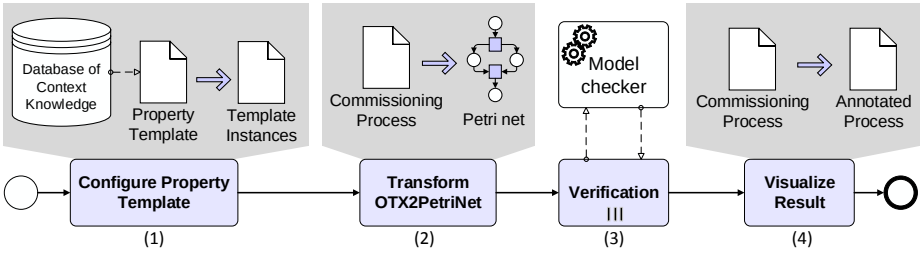


Figure 1: Steps of the Verification Framework

expressive enough for our domain, in two steps. First, we have evaluated whether our framework can indeed find property violations in real-world commissioning processes (with a median of 125 tasks per process model). Second, we have evaluated the non-functional requirements on our framework by means of expert interviews, as part of an established test. We observe that our framework is operational, sufficiently general and usable in a production environment.

This current article is a study that addresses the installment and usage of advanced process-verification techniques in a real use case. It covers the complete verification procedure for general properties, including their specification, verification and visualization. Despite the fact that this article is titled 'case study', we go beyond just describing one specific use case, in two respects: First, the paper describes certain innovations that we have not encountered anywhere in the scientific literature, including the usage of property templates and their instantiation with a carefully designed database for context, as well as the template-specific visualization of constraint violations. Second, the paper features a thorough evaluation which also takes feedback from domain experts into account.

Section 2 describes our scenario *commissioning processes*. Section 3 introduces our notation. Section 4 explains how to specify the properties required, Section 5 features a transformation of the commissioning notation OTX to Petri nets, and Section 6 describes one way how the verification can be done. Section 7 says how we present the verification result to the user. Section 8 describes the implementation of our framework. Section 9 features our evaluation. Section 10 reviews related work, Section 11 concludes.

This article is an extended version of [12]. The new contributions are the transformation of OTX to Petri nets (Section 5), the reporting of property violations to the user (Section 7), a significantly extended description of the implementation (Section 8), and more details in the remaining sections.

2. Scenario and Requirements

Commissioning processes describe the end-of-line manufacturing and testing of vehicles. Process developers define these processes with development tools, in

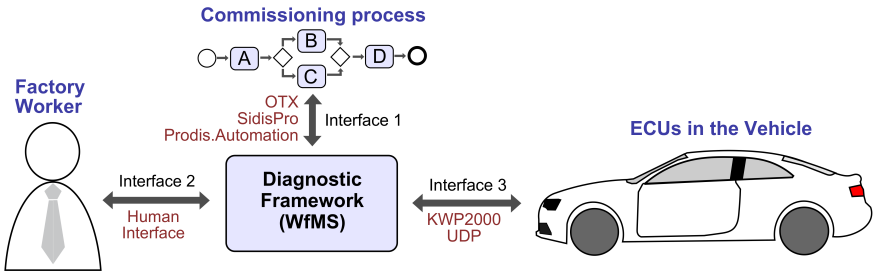


Figure 2: The Architecture of a Diagnostic System

several notations (OTX, SidisPro, Prodis.Automation). Workflow Management Systems (WfMS), here referred to as Diagnostic Frameworks, execute these processes, see Interface 1 in Figure 2. Vehicle commissioning includes, say, to check for each vehicle produced if all its Electronic Control Units (ECU) are integrated correctly and to put the ECUs into service. ECUs are components built into the vehicle which control specific functionality of the car, e. g., the ECU MOT controls the engine electronics. Each ECU needs to be tested and put into action, e. g., by installing certain software. To this end, the WfMS executes several tasks for each ECU. The tasks can be automatic like the configuration, or they may require a factory worker equipped with a handheld terminal. There typically are hundreds of tasks for each vehicle.

For example, for an executive-car series there are more than 1650 tasks in 13 processes altogether, and for some vehicle variation it is necessary to check each of the tasks. Most, but not all tasks communicate with at least one ECU. For instance, a human task tests if the light in the glove compartment functions correctly. This task does not need to communicate with an ECU. To communicate with an ECU in the vehicle, the Diagnostic Framework uses one of two different communication protocols, the Keyword 2000 protocol (KWP2000) or the Unified Diagnostic Services (UDS) [1], see Interface 3 in Figure 2. The protocol of an ECU depends on the vehicle series and its generation. For manual commissioning tasks, the WfMS presents the task to a factory worker on a handheld device. For instance, the worker has to check the light bulbs and to either confirm that they function properly or mark them for post processing, see Interface 2 in Figure 2. Diagnostic Frameworks execute the commissioning processes at several specific physical stations in the factory called *process places*. For each vehicle project and each process place, at least one process has been designed by process developers. The processes execute in a pipelined fashion. For instance, the first testing station tests a vehicle A. After a production cycle, a worker drives A to the second station, and another vehicle B moves to the first station. If a process execution takes longer than expected, e. g., because of a deadlock, the entire commissioning must stop, and this can result in high costs.

Example 2. *A vehicle of the executive-car series (M3) is tested at the process place VP2, next to other places. To this end, the Diagnostic Framework executes*

the process (M3_VP2). The Diagnostic Framework activates tasks that an ECU *executes automatically, other tasks are allocated to workers. One task checks if the injection system works properly. For this purpose, the task communicates with the* ECU *of the engine of the automobile.*

Our framework should be able to detect property violations in commissioning processes. Additionally to this functional criterion, the framework must meet further real needs of the process developers: The number of *false positives*, i. e., the number of reported violations that are not problematic, and the number of *false negatives*, i. e., the number of undetected rule violations in the processes, should be small. The framework should be general enough to be used in another factory. The handling of the framework should be intuitive and not require the help of a technical person.

3. Notation

In this section we introduce the notation used in this paper, i. e., Petri nets as formal representation of a process to be verified, and CTL (**C**omputation **T**ree **L**ogic) as the language to specify properties. Our framework aims to verify whether commissioning process given fulfill certain rules regarding the commissioning of vehicles, i. e., properties. We transform our processes to Petri nets because their execution semantics is unambiguously defined, and established verification techniques for Petri nets exist. We use CTL because it can express general properties, and efficient model checking algorithms for CTL exist. For a more detailed introduction, see the standard literature, e. g., [13] and [14].

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.

Definition 1 (Petri net)**.** *A Petri net is a triple* (P, T, F)

- 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*

$p \in P$ 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 (M) represents a state of the Petri net. 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:

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

The set of states reachable from a start state M_0 of a Petri net is its state space.

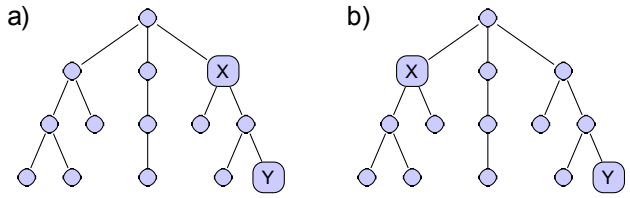


Figure 3: Two Computation Trees, (a) fulfills the CTL Formula $\mathsf{AG}(X \rightarrow \mathsf{EF}(Y))$, (b) does not

CTL is a temporal logic to specify properties. Model checking algorithms exist to efficiently verify CTL properties [15]. The CTL syntax is as follows:

Definition 2 (**C**omputation **T**ree **L**ogic:)**.** *Every atomic proposition $p \in AP$ is a* CTL *formula. If ϕ_1 and ϕ_2 are* CTL *formulas then $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\mathsf{AX}\phi_1$, $\mathsf{EX}\phi_1$, $\mathsf{AG}\phi_1$, $\mathsf{EG}\phi_1$, $\mathsf{AF}\phi_1$, $\mathsf{EF}\phi_1$, $\mathsf{A}[\phi_1\ \mathsf{U}\ \phi_2]$, $\mathsf{E}[\phi_1\ \mathsf{U}\ \phi_2]$ are* CTL *formulas.*

In our domain, AP is a state M of a Petri net. 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 ($\mathsf{AG}(X \rightarrow \mathsf{EF}(Y))$). 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. A means that the formula holds in *all* succeeding execution paths, i. e., for all child nodes in the computation tree, E means that at least one execution path *exists*, i. e., for at least one child node. The temporal operators argue about states. X means that the formula holds in the next state, G means that it holds in all succeeding states, F means that it holds in at least one succeeding state, and $[\phi_1\ \mathsf{U}\ \phi_2]$ means that ϕ_1 holds until ϕ_2 is reached.

Example 3. *The computation tree in Figure 3(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 3(b) does not fulfill the formula. This is because after the event X there is not any path that leads to the event Y.*

4. Property Specification

Our overall goal is to develop a verification framework for vehicle commissioning processes which is easy to use, easily adaptable to new vehicle variants, and adequate for flexible commissioning process execution. Before verification takes place, it is necessary to specify the properties for a process. To support this step, we have collected so-called property templates, together with engineers who develop diagnostic programs, see Section 4.1. As part of the verification, our framework determines the context of the process first. For

instance, the context consists of the process place, the vehicle project and the list of tasks and ECUs used. This process context is used to query a database for the information required to dynamically generate instances of the property templates. Section 4.3 identifies recurring characteristics of such templates and proposes a respective database representation. Section 4.4 says how to generate process-specific instances of the templates.

4.1. Properties and Property Templates for Commissioning Processes

We have identified typical properties as follows.

P1 Syntactical Correctness
: The commissioning process must be syntactically correct and comply with the naming conventions of the company for tasks.

P2 Resources of the ecus
: Some ECUs require specific resources at the process place for their testing. When a task requires a resource not available at the current process place the process is blocked.

P3 Connections of the ecus
: Each ECU opens a connection to one of two transport protocols supported (UDS or KWP2000). Each transport protocol can handle a certain number of open connections, in our environment 10 at the same time. In total, 14 connections altogether can be open at the same time. To avoid blocking of a process, the process must not open more connections. Table 1 shows the respective property templates.

P4 Task Conditions
: Some tasks depend on the occurrence of other tasks in the process, e.g., they cannot run in parallel or need to occur in a certain sequential order. Table 1 contains the different property templates for commissioning processes. They are the result of a comprehensive survey of ours to detect all dependencies that are conceivable in the commissioning context.

P5 ecu Conditions
: Additionally to the conditions on tasks, conditions specific to certain ECUs exist, see Table 1. These conditions hold for any task that communicates with the respective ECU.

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 be verified by model checking.

Violations of the control flow can result in undesirable characteristics of the process execution, referred to as *major disturbances*. An example is that it may

Table 1: Property Templates for Task and ECU Conditions

PROP.	NAME	DESCRIPTION
P3.1	Maximal UDS Connections	The number of connections to UDS should not exceed 10.
P3.2	Maximal KWP-2000 Connections	The number of connections to KWP2000 should not exceed 10.
P3.3	Maximal Connections	The number of connections UDS and KWP2000 should not exceed 14.
P4.1	Sequential before (Precedence)	If a task A is 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/test 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.

block the execution of the process. This holds for properties P3, P4 and P5. As mentioned, our approach is to define templates for these properties, see Table 1.

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. To check Property P2, our framework queries the database of context knowledge, see Subsection 4.3. It does so 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 [16, 17]. Such properties would require including the handling of resources at the Petri net level.

4.2. Choice of Temporal Logic

The properties in Table 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 (Linear Temporal Logic), CTL (Computational Tree Logic), CTL*, and μ-calculus [18]. CTL* is an extension of CTL without the limitation that the path operators and temporal operators 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 ($\bigcirc, \Box, \Diamond, \mathcal{U}$). 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 μ 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. Figure 4 shows the expressiveness of the four temporal logics. The property templates of Table 1 lie in LTL $\cap$ CTL, i. e., can be expressed in all temporal logics. Related research has described templates expressible in CTL but not in LTL, e. g., the weak data flow patterns of [16]. To allow future templates 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.

In conclusion, CTL is most appropriate in our framework. It allows to express 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.

4.3. Database of Context Knowledge

Our goal is to generate properties for checking commissioning processes automatically, based on the information collected a priori. To this end, we have developed a model of the context knowledge on commissioning processes in the automotive industry which supports generating the properties. By definition, process context is any information that influences the process flow, and that is not defined by the process model. [19] 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*

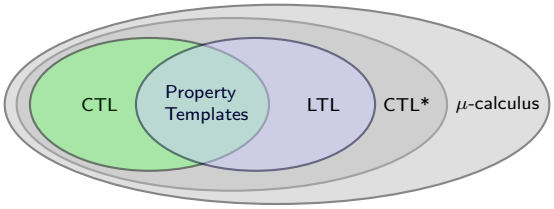


Figure 4: The Expressiveness of the four Temporal Logics.

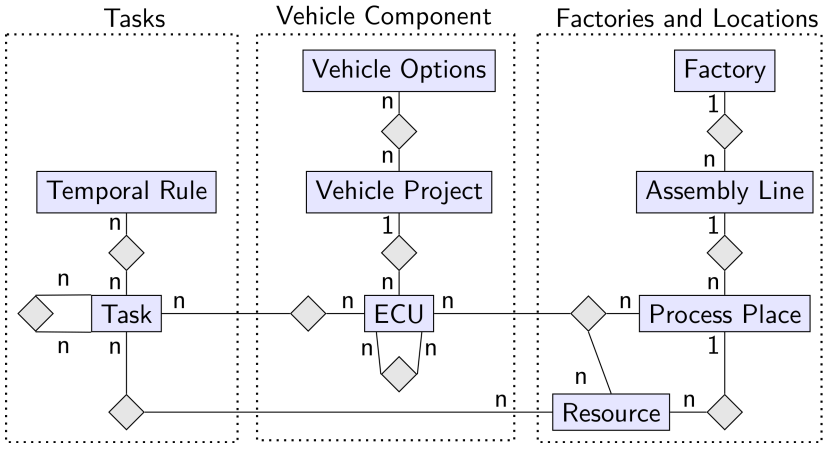


Figure 5: Excerpt of the Database Schema for Context Knowledge

or *internal context*). As argued in [19], 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 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., on the components built into it which have to be tested, mostly ECUs. 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 4.1.

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 5 shows an excerpt of our database model illustrating the overall structure, see [20] 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. We use it 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*.

4.4. Template Instances

As part of the verification, our framework determines the context of the process in a first step. It is used to query the database for the information required to dynamically generate instances of the property templates of Table 1. To this end, we specify for each template a CTL-formula in the database. These CTL-formulas are under-specified, i. e., they contain placeholders filled with the respective places of the context information. See Subsection 5.5 for details.

Our approach for the property specification is as follows. We start by extracting the immediate context of the process model to verify, e. g., the process place, the ECUs used, and the vehicle project. Our framework uses the context to automatically specify the relevant properties for the process model. For instance, two tasks should not be in parallel if the process model is for a vehicle project X. For the vehicle project Y the property does not need to hold, due to differently used communication protocols.

The dynamic generation of properties from the database has several benefits compared to their direct specification in, say, CTL. First, for a process given we only consider the properties relevant for it. 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 revise the specification of several hundred properties. The database stores the contextual knowledge in a centralized and non-redundant form, instead of managing all properties specified in CTL.

For example, the template "A leads to B" has a few hundred instances – A and B are variables/placeholders which an instantiation replaces with concrete tasks. If, say, the need to change the template to "The first occurrence of A leads to B" arose, updating all property instances would be avoided. Third, 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.

5. Transformation

To allow an automatic verification, e. g., model checking, the process representation has to allow search in 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 has two concurrent proprietary systems in use for the commissioning of a vehicle series: Sidis Pro by SIEMENS AG and Prodis.Automation by DSA. Each of these systems has 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 will replace those proprietary notations in the near future. Therefore, we support three notations Sidis Pro, Prodis.Automation and OTX at the moment. To this end, we have specified and implemented a transformation of Sidis Pro and Prodis.Automation to OTX containing the structural properties to verify (Step 1 in Figure 6). It is not possible to generate the state space for the analysis efficiently in a direct way from an OTX process. Thus, we transform the OTX process to a Petri net beforehand (Step 2 in Figure 6). In Subsection 5.1 we discuss our choice of the formal language. [7][6] 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 this task that produces the state space in the form of a graph (Step 3 in Figure 6).

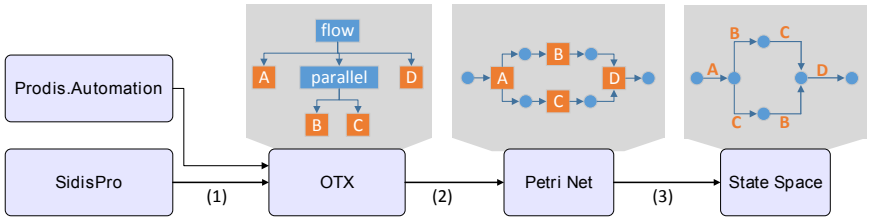


Figure 6: The Transformation Steps

5.1. Choice of the Formal Language

Formal verification techniques require an explicit representation of the execution semantics. Unfortunately, the ISO Standard of OTX does not include a formal description of this. Thus, we have to 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 to analyze 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.

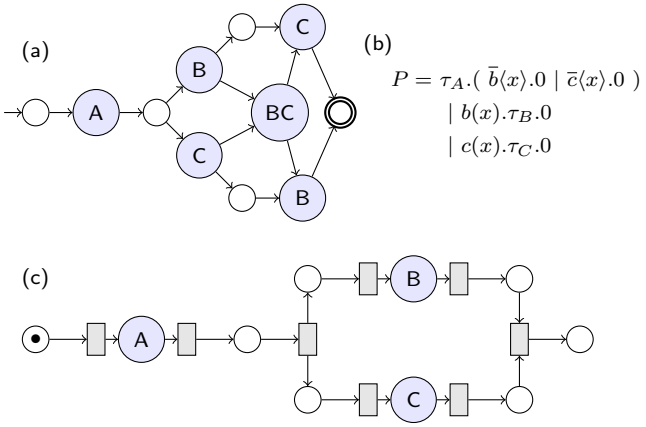


Figure 7: A Process as a Kripke-Structure, in the π-Calculus, and as a Petri Net

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 to a set of atomic propositions that are true in the state [14]. A plethora of tools exist for model checking Kripke structures, e. g., the SPIN framework [21]. However, commissioning processes contain a lot of parallel structures. It is not possible to represent these concurrent structures directly as a Kripke structure. It would be possible to list all combinations of executions each as a new state. But this would lead to a Kripke structure too complex to handle, in our case 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 [22]. The π-calculus is based on a textual (i. e., rather a linear) description [23]. 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 3. They allow for a direct representation of concurrent systems. Tool support exists for model checking, e. g., the LoLA-Framework [24]. Petri nets are structurally similar to most process notations and allow for a simple mapping of subnets to constructs of other notations.

Example 4. *Figure 7 shows a process in the three notations. The process consists of three tasks. First, Task A is executed, followed by a parallel execution of B and C. Figure 7(a) shows the process as a Kripke structure. The labels in the states describe the execution of tasks. Figure 7(b) shows the process in π-calculus. The process model consists of three sub models, to be executed in*

Table 2: Core Requirements for the three Formal Languages

	Kripke-structure	π-Calculus	Petri Net
Tool Support	×	×	×
Direct Concurrency		×	×
Graph-based	×		×

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 7(c) shows the process as a Petri net.

We decided to use Petri nets as our formal representation. Tool support exists, it allows to represent concurrency directly and is graph-based. So a direct mapping of constructs is possible. Table 2 shows a summary of the core requirements for Kripke-structures, π-calculus, and Petri nets. We do not see any problem when using our approach with other formal languages if a respective transformation of OTX to the formal language is given.

5.2. OTX2PetriNet

OTX (**O**pen **T**est e**X**change) is an XML-based process notation for commissioning processes. It only allows the definition of structured process models, i. e., an OTX process model can be represented in a notation similar to process trees [25]. OTX defines a commissioning process as nodes arranged as a tree. In this sense it is similar to WS-BPEL [26]. The nodes fall into 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 we define a template, specifying a Petri net representation of the routing behavior of the node, see Subsection 5.3. Additionally, we will refer to the respective WS-BPEL construct in parentheses. Observe that the *flow* node in OTX is used for a sequential execution. In WS-BPEL in turn, *flow* is used for concurrent execution. The *handler* node does not have a direct representation in WS-BPEL.

5.3. Petri Net Templates for OTX

For each node type of OTX we define at least one Petri net template (Figures 8 to 11). A Petri net template is a Petri net subnet with a certain input place *In* and an Output Place *Out*. If a node type allows for child nodes, the template contains specific regions for the insertion of the templates of the child nodes. Figures 8 to 11 show the insertion regions as dotted boxes.

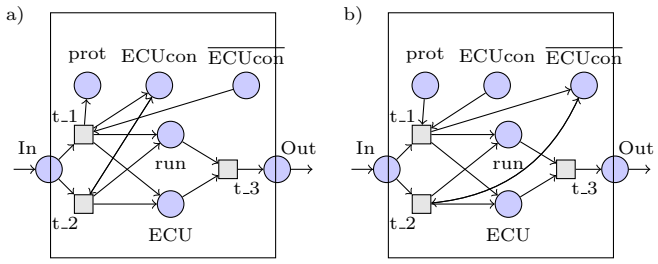


Figure 8: Template of the *action node* (a) and the Close Connection *action node* (b).

5.3.1. Action Node (Basic Activity)

Action nodes are used to represent atomic operations in the commissioning process, i. e., the commissioning tasks. In the Petri net representation (see figure 8) we represent the fact that a task is running as a state *run*. As stated earlier, each task communicates with an electronic control unit ECU in the vehicle. We present the ECU with three places. *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 UDP 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 8(b), and one for the tasks that can implicitly open the connection, see Figure 8(a). At the beginning of the process execution, each place $\overline{ECUcon}$ contains a token, meaning that all connections are closed. Figure 8(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 run 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*, *run*, and *ECU*. The transition t_3 ends the execution of the task, removing the token in *run* and *ECU*. The places *prot*, *ECU*, *ECUcon* and $\overline{ECUcon}$ are shared between tasks. Each task has its own *run*, *In*, and *Out* Place.

5.3.2. Loop Node (While)

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 [27]:

> 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.

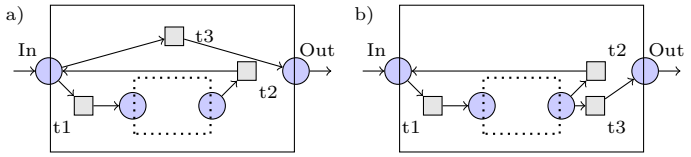


Figure 9: Template of the *loop node* with Checks Before the Execution (a) and After the Execution (b).

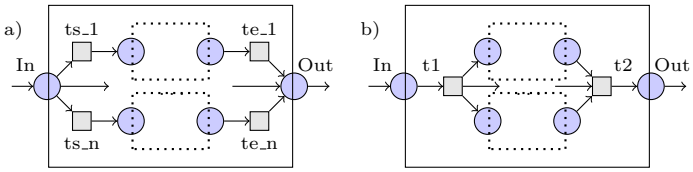


Figure 10: Template of the *branch node* (a) and the *parallel node* (b).

This is equivalent to the structured iteration control flow pattern of [28]. We use two templates for the *loop node* depending on whether the condition is checked before or after the execution. Figure 9 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 subnet for the unique child node.

5.3.3. Branch Node (Switch)

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 [27]:

> 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.

The *branch node* is equivalent to the exclusive choice control flow pattern of [28]. Figure 10(a) shows the template for the branch node. The dotted box is the region for the subnets corresponding to the child nodes.

5.3.4. Parallel Node (Flow)

The *parallel node* is used to model flows that are executed in parallel. ISO defines the *parallel node* as [27]:

> A Parallel node consists of one or more flows that shall be executed at the same time.

This is equivalent to a combination of the *Parallel Split* and *Join* control flow patterns of [28]. Figure 10(b) shows the template for the *parallel node*. The dotted region identifies where the subnets for the child nodes are inserted.

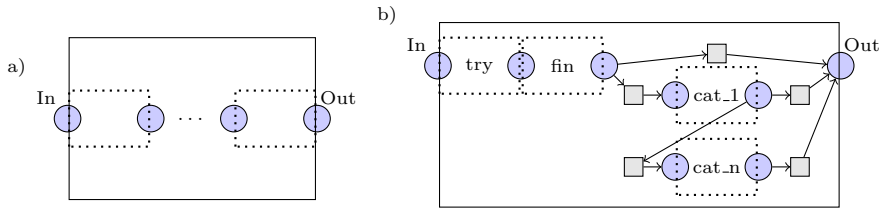


Figure 11: Template of the *flow node* (a) and the *handler node* (b).

5.3.5. Flow Node (Sequence)

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 [27]:

> 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.

The *flow node* can be used as the *sequence* control flow pattern of [28]. Figure 11(a) shows the templates for the *flow node*. The dotted region is the place where to insert the subnets for the child nodes.

5.3.6. 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 [27]:

> 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, [...]

Figure 11(b) shows the templates for the *handler node*. The dotted regions specify the subnets for the try, final and the catch subprocesses.

5.4. Transformation Algorithm

To transform an OTX process model into a Petri net representation, we define a function transformNode(Node n). The function first determines the type of the node n. The possible types are *flow*, *branch*, *parallel*, *loop*, *handler*, and *action*. Next, the function adds the places and transitions as defined by means of the template of the type, i. e., a Petri net subnet with a certain input place and a certain output place. If the node is a compound node, the function calls itself recursively for each child node. Our transformation algorithm starts by calling transformNode() on the root node.

Table 3: CTL-Formula for the Property Templates

PROP.	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	$\mathsf{A}\ [\ (A_{run}=0)\ \vee\ \mathsf{AG}(\ B_{run}=0\ )\ )\ \mathsf{W}\ (\ B_{run}>0\ )\ ]$
P4.3	Sequential After	$\mathsf{AG}\ (\ (A_{run}>0)\ \rightarrow\ \mathsf{AF}\ (B_{run}>0)\ )$
P4.4	Non-Parallel	$\mathsf{AG}\ (\neg(\ (A_{run}>0)\ \wedge\ (B_{run}>0)\ ))$
P5.1	Restricted Access	$\mathsf{AG}\ \ (X_{\text{ECU}} \leq 1)$
P5.2	Non-Parallel	$\mathsf{AG}\ (\ \neg\ (\ (X_{\text{CON}}>0)\ \wedge\ (Y_{\text{CON}}>0)\ )\ )$
P5.3	Close Connection	$\mathsf{AG}\ (\ (\text{END}>0)\ \rightarrow\ (X_{\text{CON}}=0)\ )$

5.5. Property Instantiation

As explained in Section 4.4, 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 which our framework replaces with concrete places in the Petri net of the commissioning process during instantiation. These placeholders are of five types, see Figure 8:

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 (the place *Out* of the root element in OTX).

Table 3 shows the under-specified CTL-formula for each template of Table 1.

Example 5. *The process to be verified contains the* ECU*s* $=$ [GWA, KEL, FBE]. *For the process place* VP2 *and the vehicle series M3, an* ECU *dependency exists that* KEL *and* FBE *must not be used in parallel. For Property Template P5.3 one of the properties our framework generates is as follows:*

$$\mathsf{AG}(\ \neg\ (\ (\text{KEL}_{\text{CON}}>0)\ \wedge\ (\text{FBE}_{\text{CON}}>0)\ )\ )$$

6. Verification

We now describe the architecture of our verification framework and how it verifies whether a commissioning process fulfills a set of property instances. A

preprocessing step transforms a process file in another format into OTX (Figure 12, Step 1). Next, the context information regarding the process place and the vehicle project are extracted from the commissioning process (Figure 12, Step 2). Not all properties can be verified efficiently with one paradigm, i.e., model checking. Therefore, our verification component A^{3}FT (**A**utomatic **A**rrangement of Working Steps in Production and Testing) consists of two modules: the *Data Reconciliation module* (Figure 12, Step 3) and the *Model Checker* (Figure 12, Step 4) to (Figure 12, Step 6). In the past, researchers have developed efficient tools for model checking with Petri nets [24][29]. Hence, model checking in the narrow sense of the word is assumed as given and is not a topic of this article. Our framework contains an established framework for model checking, the Low Level Petri Net Analyzer (LoLA) [24].

6.1. Data Reconciliation

First, our framework tests the syntactical correctness of the OTX process. To do so, the module validates the commissioning process against the XML schema of OTX. Additionally, we check for each task if it complies with the naming conventions of the company. The module then checks if the resources are available at the process place of the commissioning process (P2). To this end, our framework queries the database to evaluate if the resources at the process place match the resources used in the process.

6.2. Model Checking

"Model checking is a technique for verifying finite state transition systems. [...] Model checking normally uses an exhausting search of the state space of the system to determine if some specification is true or not." [14, p. 3] The finite state transition system (M) in our case is the state space of the commissioning process, and the specification (ϕ) is the CTL formula of the property instances. Formally, we test if $M \models \phi$. For state space generation and the subsequent model checking our framework includes the LoLA-Framework [24]. It takes a Petri net pn and the property as a CTL-Formula ϕ as input. LoLA then generates the state space of pn and subsequently uses the ALMC-Algorithm to check if a state is found that violates property ϕ. Two possible outcomes of the verification can occur. First, the verification has not detected a violating state. This means that pn fulfills the property. The alternative is that the algorithm detects a violating state and aborts. In the case of a property violation, LoLA reports a sequence of transitions fired, i. e., a counter example. In general, the search space grows exponentially with the size of the Petri net (state space explosion

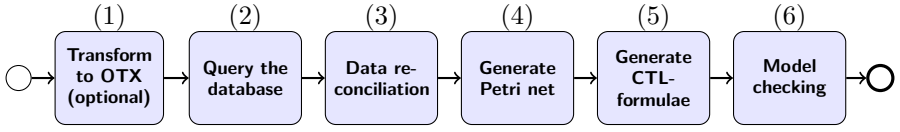


Figure 12: The Verification Steps

problem). [24] includes techniques like stubborn set reduction and invariants in the LoLA-Framework, to reduce the state space. These reductions alone have not been sufficient in all of our cases, due to the sometimes large number of parallel sequences, which are common in commissioning processes. Thus, we have applied our algorithm of [30] to generate a reduced Petri net for each property, tailored to the specific tasks referred to in the property. Our framework is not specifically tailored to that concrete model checker. We do not foresee any difficulties when including other frameworks for state-space generation or model checking.

7. Reporting

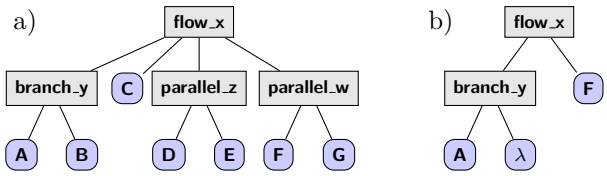
As stated in Section 6, the LoLA-Framework returns a sequence of transitions fired, leading to a violating state in the case of a property violation. Our goal is to highlight the important aspects in a graphical interface, supporting the user to understand the property violation. Reporting all firing transitions 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 template 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 7.1. Second, we collect the important information in the remaining path for each template and report it to the user, see Subsection 7.2. Subsection 7.3 gives an example of a commissioning process and the tasks highlighted.

7.1. Reduction of the Counter Example

[30] shows 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 reduces the size of the counter example significantly.

Example 6. *Consider the* OTX *process tree in Figure 13(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, the execution of the second branch of y, is hard to perceive, due to these unimportant events listed. Figure 13(b) shows the process tree reduced with the algorithm described in [30]. This reduction leads to a much smaller counter example, which only contains events directly related to the property. This example contains only three events that do show the cause of the violation.*



7.2. Property Template 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 the Response-template (cf. Table 1) for instance, the information of the sequential arrangement is important, while to understand a violation of the maximal connection template one needs the parallel arrangement. To this end, we propose a specific reporting for each template. We will describe how we extract the information from the counter example, and which elements our visualization should highlight. Elements are nodes in the graph, i. e., control nodes or tasks. We will propose reporting schemes for each template. Their input is the counter example, i. e., a sequence of transitions fired that leads to a violating state. We refer to this sequence as L. The reporting schemes return 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 5.3.

P3.1-P3.3 For the template *Maximal Connection* we are only interested in elements that open or close a connection, i. e., the transitions belonging to tasks. Our reporting scheme further reduces the event log L to the tasks which open or close a connection to the respective protocol (UDS for P3.1, KWP2000 for P3.2, both for P3.3). Figure 8 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". Last, the reporting scheme highlights the remaining tasks in the visualization.

P4.1 The Sequential Before (or precedence template) 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. For this template we highlight A and a non-existing instance of B.

P4.2/P4.4 The Optional Sequential Before and Not-Parallel Tasks templates 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.

P4.3 The Sequential After template (also known as response-to or leads-to) describes a Task A that requires the subsequent execution of another Task B. The template is symmetric to the Sequential-Before Template, and its reporting is analogous.

P5.1 The template for the tasks has a place for the ECU used, see Subsection 5.3.1. This place represents the access to the ECU. The Restricted Access template restricts the access to one task at a time. In other words, the place ECU is restricted to 1. For the template, we identify two tasks that access the ECU at the same time. First, we reduce the event log to tasks for the ECU we are looking for. Next, we are only interested in tasks that are active, i. e., the start event is in L but not the end event. Exactly two tasks A and B are active for a counter example for P5.1. We highlight these tasks A and B.

P5.2 We handle the Non-Parallel template 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 tasks in the log (thus t_3 has not fired), 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.

P5.3 For the Close Connection Template, the model checking aborts at the first task X that does not get its connection closed on at least one subsequent execution path. We highlight this task X. Additionally, we look for all tasks in the process that could close the connection for the tasks and highlight them as well. Remember that the situation is not 'symmetric', i. e., while a connection may be opened implicitly within a task, there always must exist a designated activity whose only responsibility is to close the connection.

7.3. Example for a Commissioning Process with Highlighted Violations

Consider the commissioning process in Figure 14. 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 '*Verbaupruefung*' precedes task '*WFS-Anfrage*' (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 '*Verbindungsabbau*'. Our framework has verified the commissioning process and has been able to detect all three errors. Using the reporting just described, the tool finds the tasks relevant for the errors and highlights them, see Figure 15. The large red boxes with exclamation points contain a description of the violation as well, visible when the mouse hovers over them.

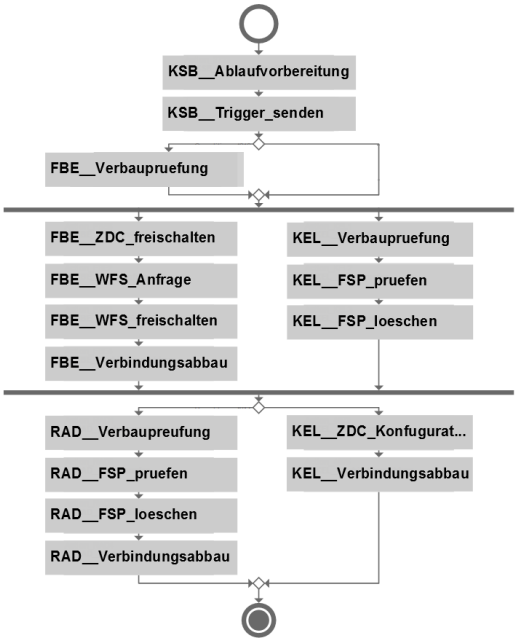


Figure 14: A Commissioning Process

8. Implementation

Our framework consists of several components, see Figure 16. The main components are the contextual knowledge base, the verification system A^{3}FT, the LoLA-Framework and the visualization component CoVA (**Co**mmissioning Process **V**erification and **A**nalysis).

The A^{3}FT-component transforms, specifies and verifies a commissioning process. It itself is a standalone command-line tool and can be used without the visualization. The A^{3}FT-component receives the file-system path to a commissioning process and detects the notation automatically (Prodis.Automation, Sidis Pro, or OTX). It transforms the process first to OTX if necessary. The component verifies the tool against the OTX schema to detect syntax errors and to check the data reconciliation, cf. Subsection 6.1. Next, the A^{3}FT-component generates the property instances, see Section 4. To this end, it sends queries to the database of context knowledge. The component transforms the OTX process tree to a Petri net using the algorithm of Section 5. The A^{3}FT-System calls the LoLA-component for each property, with the Petri net process model and the property as CTL formula as parameters. Using the results, the A^{3}FT-component presents detailed information on the property violation to the user with the reporting tool.

The database of context knowledge is implemented as a MySQL database. The full database contains 25 different relations with approximately 2 500 data

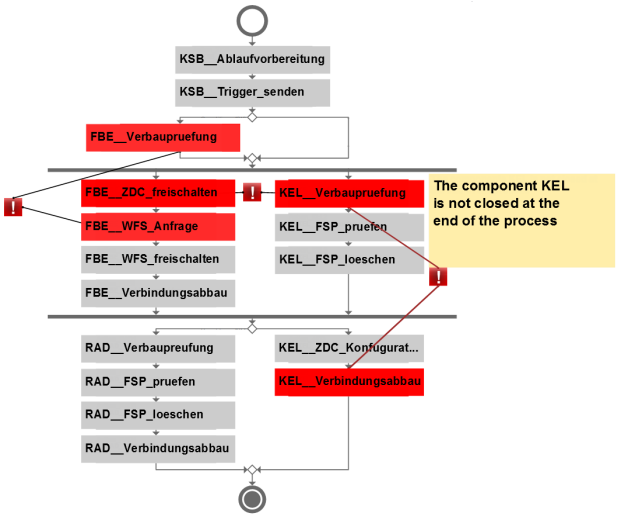


Figure 15: The Highlighted Violations in the Commissioning Process

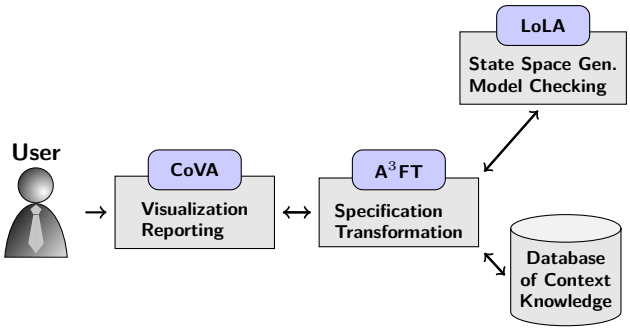


Figure 16: The Components of our Framework

sets. Using index structures, the query time for the property verification is negligible in comparison to the verification time. The LoLA component has been developed elsewhere [24]. We use its version 1.16, to generate the state space and to verify the properties. We use a depth-first search and apply all the possible reduction techniques for model checking.

The CoVA component visualizes the commissioning process and presents the verification result to the user. The component loads the commissioning process of the three notations mentioned and graphically presents the process to the user. We present the process in a notation similar to UML sequence diagrams, with bars for parallel splits/joins and diamonds for XOR-splits/joins. The visualization is able to expand and collapse regions of the process for a more focused view on the commissioning process. A click on the verify button of the CoVA-component leads to a call of the A^{3}FT-component. The textual results of the verification

can be seen in a tab *Verification*. The tool uses the algorithm of Section 7 to highlight the property violations in the process and additionally lists them on the left-hand side.

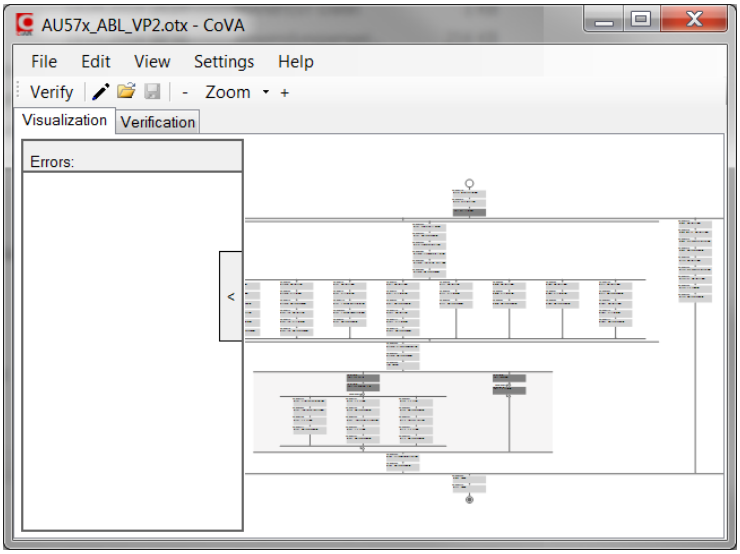


Figure 17: Screenshots of the Verification Framework

Example 7. *In the screenshot Figure 17 on the left-hand side there is a visualization of a commissioning process in* OTX *notation. The darker tasks stand for compressed regions. A region can be easily compressed or extended with a click. When clicking on the verify button, the CoVA component calls the* A^3FT *component. Within a few seconds, the tool has found several property violations in the commissioning process. Our framework lists them on the left-hand side, see Figure 18. When clicking on the property violation, our tool zooms into the error, see Figure 19. The red box is a message box describing the error when hovered over. Additionally, a second tab "verification" presents the detailed textual output of the* A^3FT*-component to the user. This output contains technical information on the configuration and the verification, see Figure 20.*

9. Evaluation

When evaluating our framework for the verification of commissioning processes, we focus on two points, namely quality of verification in our specific application domain (Subsection 9.1) and usability of our tool (Subsections 9.2 and 9.3). According to ISO 9241-11 [31], usability has three different aspects, to be evaluated separately:

Efficiency The amount of the resource usage to achieve the goals (Subsection 9.2).

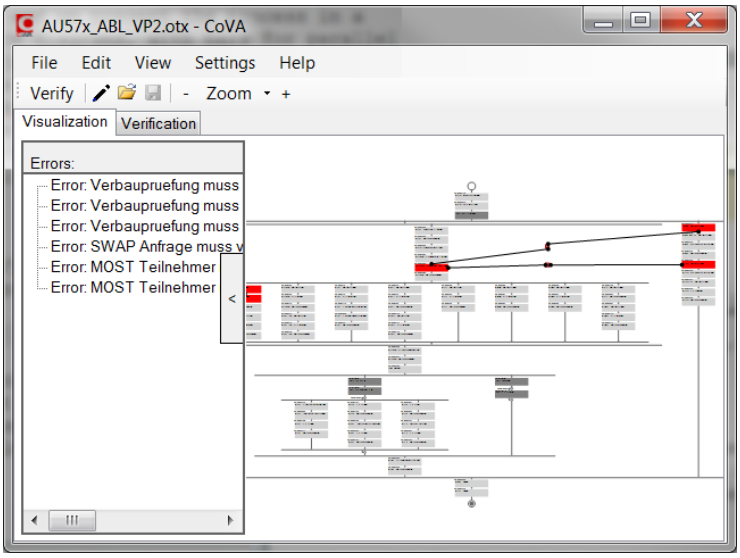


Figure 18: Screenshots of the Verification Framework with Highlighted Violations

Effectiveness Whether the user can complete his tasks and achieve the goals (Subsection 9.3).

Satisfaction The level of comfort the users experience achieving the goals (Subsection 9.3).

9.1. Quality of Verification in the Commissioning Context

Regarding the quality of verification, we confine ourselves to the specifics of the commissioning scenario in this subsection; we have examined more generic aspects of verification quality elsewhere already [30]. We have used our prototype 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 inspected the verification results and have categorized the processes into three categories: correct, with minor process disturbance and with major process disturbance. Figure 21 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.

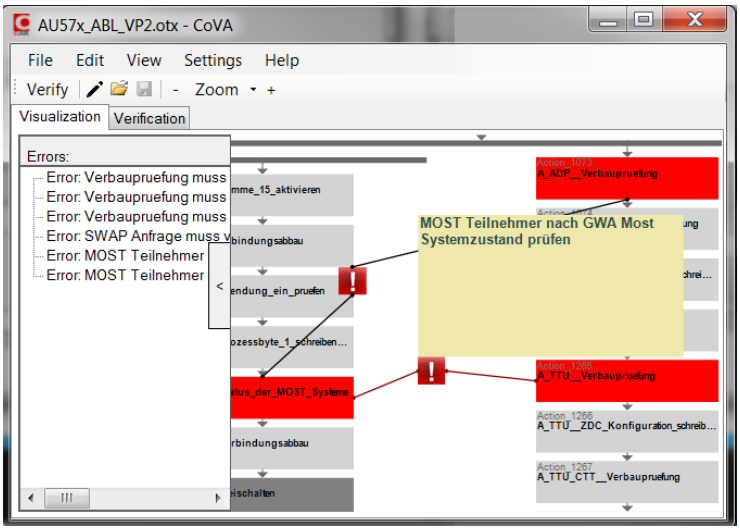


Figure 19: The Graphical Reporting

9.2. Efficiency

We now study characteristics of the verification process and of the visualization. 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, 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 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

Table 4 shows the five-number summary of these characteristics and the respective boxplots. The five-number summary consists of the five most important percentile: the minimum value found (Minimum), the first Quartile Q_1, the median (or second Quartile) Median, the third Quartile Q_3 and the maximum value found (Maximum). Even for the most complex commissioning process with

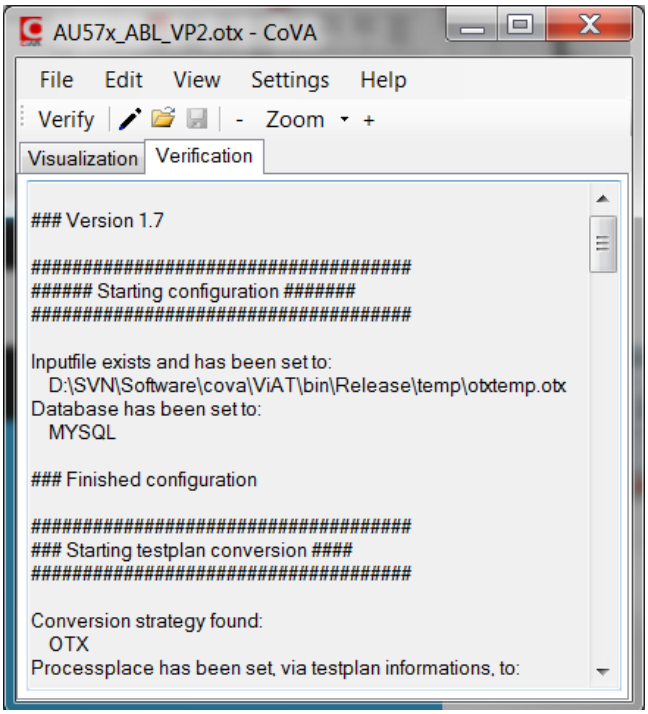


Figure 20: The Textual Description of the Reporting

870 tasks, 103 ECU, and 280 different property instances our framework is able to do the verification in 10.5s. In 75% of the cases the verification needs under 3s in total. The runtime is strongly correlated with the size of the process (.926 for #tasks and .904 for #ECU) and even stronger with the number of property instances (.973 for #PropIns). We have generated and verified 2801 property instance and verified them in 104.1s, i. e., an individual property instance takes only 37.2ms to be verified on average. The boxplots in Figure 21 show that most of the commissioning processes contain between 50 and 200 tasks and have between 15 and 100 properties and take between 0.5s and 3s to verify.

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 7.1, 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 template-specific reporting, see Subsection 7.2. 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 templates: 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

	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

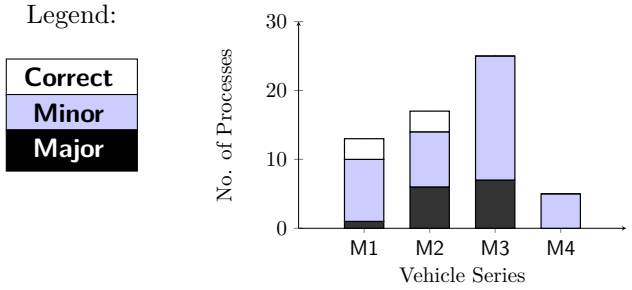


Figure 21: Process Disturbances Found in the Evaluated Processes

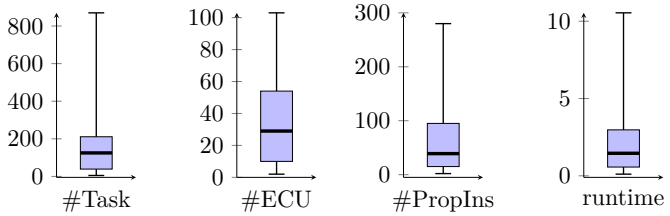


Figure 22: The Boxplots for the #Task, #ECU, #PropIns, and *runtime*

their complexity is representative of the violations found.

Table 5 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.

9.3. Effectiveness and Satisfaction

Effectiveness: Regarding effectiveness, we ask two important questions, as follows:

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*.

Table 5: The Number of Highlighted Elements for our Approach and the two Baselines

Violation	A	B	C	D	E	F	G	H	I
Template	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

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? We have also asked how well the framework can be integrated into the tool chain.

Satisfaction: *Can the framework be used in an intuitive way? Is the help of a technical person needed to use the framework?*

To evaluate effectiveness and satisfaction, we have held semi-structured interviews with domain experts. The interview guide is available on our website: `http://dbis.ipd.kit.edu/2027.php`. Regarding satisfaction, we have used the Standard System Usability Test (SUS) [32]. 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 [32].

9.3.1. 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 were from different factories and departments at the AUDI AG.

9.3.2. Results

Figure 23 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 has detected slightly more false positives than false negatives. The experts see 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. From all this, we conclude that effectiveness is achieved. The SUS score (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

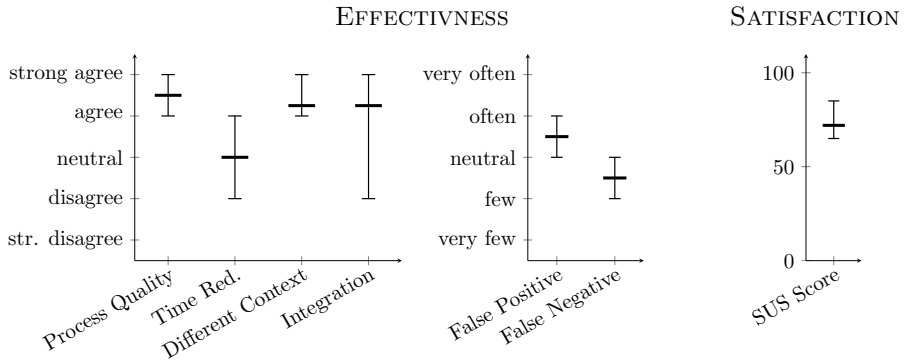


Figure 23: Results of the Empirical Evaluation

score [32]. 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. So we arrive at the conclusion that the satisfaction of users with our verification tool in the context of commissioning is high.

9.3.3. 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 impossible 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. 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 re-implemented our tool in C#, which is used to implement the rest of the tool chain. At the time of this study, our tool was implemented in Java. The implementation in C# allows to easily use data objects from other departments, e. g., the one constructing vehicles. We use these data objects to instantiate the property templates.

10. Related Work

Related work includes the user-friendly specification of properties, their management, the property-specific verification of processes, constraint programming, and declarative process models.

10.1. Specification

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 [33], YAWL [34] or Petri nets [35]. [36] extends the BPMN notation with new elements that directly represent LTL operators. BPMN-Q [37] extends BPMN with new edge types that represent sequential ordering between tasks. Compliance Rule Graphs [11] allow a specification of requirements in a graph-based formal language. Another approach is specification templates. [10] introduces the property patterns to specify concurrent systems. [38] extends the pattern system to cover variations (PROPEL, PROPerty ELucidation). [39] uses a question tree to allow specifying PROPEL patterns. In our domain, only a few different property templates exist. Dependent on the context, many instances are generated. Because of the small number of templates but many similar instances, we have not found any of the approaches to be helpful in our specific case.

10.2. Management of Properties

[40] builds an ontology for the domain of compliance management. However, it is not sufficient to capture the domain-specific information needed for the instantiation of our templates. Managing compliance properties includes allocating the properties to the business processes. [8] allocates the compliance properties to the processes using potentially relevant activities. We in turn dynamically generate only the properties relevant for the commissioning process using the context knowledge directly before verification.

10.3. Verification

We aim to check if a business process complies with the properties given. [41] uses an approach that checks if the event log L (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 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. [42] gives an overview of transformations from BPMN, YAWL and WS-BPEL to Petri nets. Regarding the transformation aspect, our approach is similar to [43]. [2] empirically evaluates different approaches for soundness verification. The criteria include error rate, process size and verification time. [3] evaluates

three techniques (Partial-Order-Reduction, Woflan and SESE-Decomposition) to verify the soundness of over 700 industrial processes. Our verification technique is more general than just verifying soundness as in [2] and [3]. 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.

10.4. Reporting

[44] reduces the counter examples to give the end user a compact report of the property violation found. To this end, [44] 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 7.1, which is easier to compute, performs well. In consequence, we do not have to apply [44] as a preprocessing step for the reporting templates. [44] could be used for the reporting of general properties not covered by the templates.

10.5. Constraint Programming

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 $\mathcal{X}$, to describe the problem. A solver then finds an allocation for all $x \in \mathcal{X}$ that comply with all $c \in \mathcal{C}$ [45]. Constraint programming therefore consists of 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 directly express 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 [46]. In our use case however, this is not practical. This is because the set of equations grows exponentially, and finding a solution is in PSPACE, which has been proven formally [47].

10.6. Declarative Process Models

The goal of our work is to check if an imperative process model complies with declarative properties. This is 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 [48]. 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 [49], 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, not all possible executions of a declarative process model tend to be obvious at specification time. This may lead to an unexpected and possibly unwanted 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.

11. Conclusions

The concern of this article has been the detection of property violations in processes. To this end, we have described and evaluated a respective tool that covers the complete verification procedure. Our tool is able to detect the property violations in a process. Detecting such violations is an important step of quality assurance. Our approach has several important features, in particular:

- distinction between property templates and concrete properties,
- automated instantiation of property templates by means of a context database; careful investigation of what 'context' exactly means here and design of a respective database,
- integration of a model checker that is suited for our purposes,
- visualization of property violations that is adequate, according to user feedback collected objectively.

It is this bundle of measures that renders our use case manageable in the first place. The evaluation has shown that this is indeed the case. Our conclusion is that our approach is capable of improving processes: In particular, by being able to cope with many more properties than a naive solution, we do arrive at better processes.

While our study has focused on the commissioning scenario, we now conclude with some insights which might be of interest to individuals facing similar problems in other scenarios:

- Model checking per se only covers a relatively small part of an infrastructure needed in realistic settings to check whether processes have certain characteristics.
- Differentiating between property templates on the one hand and actual properties on the other hand has proved of value, in order to cope with a large number of different constraints. In particular, this holds true when it comes to the visualization of constraint violations. We for our part have tailored the visualization to the different templates, which has given way to a much more compact representation, at least in some cases.
- Property templates and the accompanying question what exactly relevant context is, are application-specific. Not only in our specific scenario, but foreseeably in other settings as well properties come from different

sources (wisdom gained from experience of individuals, concrete incidents which have not been conceivable earlier, requirements coming from the construction department etc.). Hence, this (fairly laborious) discovery of the relevant kinds of properties and of the structure of the background knowledge must take place anew every time, at least for the time being, and respective projects should be planned accordingly.

- The answer to the question how well one can transfer our approach to other use cases, i.e., generalize it, is rather differentiated. A study that is confined to one specific scenario all along naturally cannot give answers that are final. Nevertheless, we have made some important observations. Our approach caters to certain specifics of the underlying use case that gives way to scenarios which could also profit from our approach, in particular:
  - Two commissioning processes hardly ever are identical. One reason is that each vehicle is furnished individually, because of different customer preferences. Next, our car manufacturer has different production sites with different distinctive features, i. e., commissioning processes even for vehicles that are identical would look different at two sites.
  - In the context of vehicle manufacturing, just restarting a process already is a significant issue. This is because slight delays typically lead to substantial costs and are not acceptable. Put differently, car manufacturers (at least those that are aware of quality) are ready to take significant effort (the one described in this article) to ensure that such delays take place as rarely as possible. If starting a process anew was not critical, the number of problems to be ruled out would be much smaller.

As future work, we are going to extend the usage of our tools to other production lines and factories. We also intend to include additional information sources of the company for the property specification. Finally, we are leveraging process properties to support the design of processes. [50] is our first advance in this direction.

References

[1] W. Zimmermann, R. Schmidgall, Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur, 2011.

[2] J. Mendling, Empirical Studies in Process Model Verification, in: K. Jensen, W. M. P. v. d. Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, number 5460 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 208–224. URL: `http://link.springer.com/chapter/10.1007/978-3-642-00899-3_12`.

[3] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Instantaneous Soundness Checking of Industrial Business Process Models, in: U. Dayal, J. Eder, J. Koehler, H. A. Reijers (Eds.), Business Process Management, number 5701 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 278–293.

[4] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, P. Dadam, On Enabling Data-Aware Compliance Checking of Business Process Models, in: J. Parsons, M. Saeki, P. Shoval, C. Woo, Y. Wand (Eds.), Conceptual Modeling – ER 2010, number 6412 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 332–346.

[5] Y. Liu, S. Müller, K. Xu, A static compliance-checking framework for business process models, IBM Systems Journal 46 (2007) 335–361. doi:10.1147/sj.462.0335.

[6] K. Schmidt, Stubborn Sets for Model Checking the EF/AG Fragment of CTL, Fundam. Inf. 43 (2000) 331–341.

[7] K. Schmidt, Stubborn Sets for Standard Properties, in: S. Donatelli, J. Kleijn (Eds.), Application and Theory of Petri Nets 1999, number 1639 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 46–65.

[8] S. Kabicher, S. Rinderle-Ma, L. T. Ly, Activity-Oriented Clustering Techniques in Large Process and Compliance Rule Repositories, in: Proc. BPM'11 Workshops, Springer, Clermont-Ferrand, France, 2011, pp. 14–25.

[9] H. Schlingloff, A. Martens, K. Schmidt, Modeling and Model Checking Web Services, Electronic Notes in Theoretical Computer Science 126 (2005) 3–26. doi:10.1016/j.entcs.2004.11.011.

[10] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Property Specification Patterns for Finite-state Verification, in: Proceedings of the Second Workshop on Formal Methods in Software Practice, FMSP '98, ACM, New York, NY, USA, 1998, pp. 7–15. doi:10.1145/298595.298598.

[11] L. T. Ly, D. Knuplesch, S. Rinderle-Ma, K. Göser, H. Pfeifer, M. Reichert, P. Dadam, SeaFlows Toolset – Compliance Verification Made Easy for Process-Aware Information Systems, in: P. Soffer, E. Proper (Eds.), Information Systems Evolution, number 72 in Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2011, pp. 76–91.

[12] R. Mrasek, J. Mülle, K. Böhm, C. Allmann, M. 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.

[13] W. M. P. van der Aalst, K. M. van Hee, Workflow Management: Models, Methods, and Systems, MIT Press, 2004.

[14] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.

[15] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, ACM Trans. Program. Lang. Syst. 8 (1986) 244–263. URL: http://doi.acm.org/10.1145/5397.5399. doi:10.1145/5397.5399.

[16] N. Trčka, W. M. P. van der Aalst, N. Sidorova, Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows, in: P. v. Eck, J. Gordijn, R. Wieringa (Eds.), Advanced Information Systems Engineering, number 5565 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 425–439.

[17] S. von Stackelberg, S. Putze, J. Mülle, K. Böhm, Detecting Data-Flow Errors in BPMN 2.0, Open Journal of Information Systems (OJIS) 2 (2014) 1–19.

[18] E. A. Emerson, Model checking and the Mu-calculus, in: DIMACS Series in Discrete Mathematics, American Mathematical Society, 1997, pp. 185–214.

[19] M. Rosemann, J. C. Recker, C. Flender, P. D. Ansell, Understanding context-awareness in business process design, in: S. Spencer, A. Jenkins (Eds.), Faculty of Science and Technology; Institute for Creative Industries and Innovation, Australasian Association for Information Systems, Adelaide, Australia, 2006. URL: http://eprints.qut.edu.au/6160/.

[20] T. Schneider, Specification of Testing Workflows for Vehicles and Validation of Manually Created Testing Processes, Karlsruhe Insititute of Technology, Master Thesis, 2012.

[21] G. J. Holzmann, The Model Checker SPIN, IEEE Transactions on Software Engineering 23 (1997) 279–295. doi:10.1109/32.588521.

[22] M. Kwiatkowska, G. Norman, D. Parker, PRISM: Probabilistic Symbolic Model Checker, in: T. Field, P. G. Harrison, J. Bradley, U. Harder (Eds.), Computer Performance Evaluation: Modelling Techniques and Tools, number 2324 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 200–204. URL: http://link.springer.com/chapter/10.1007/3-540-46029-2_13.

[23] W. M. P. v. d. Aalst, Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype", 2003.

[24] K. Schmidt, LoLA A Low Level Analyser, in: M. Nielsen, D. Simpson (Eds.), Application and Theory of Petri Nets 2000, number 1825 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 465–474.

[25] S. J. J. Leemans, D. Fahland, W. M. P. van der Aalst, Discovering Block-Structured Process Models from Event Logs - A Constructive Approach, in: J.-M. Colom, J. Desel (Eds.), Application and Theory of Petri Nets and

Concurrency, number 7927 in Lecture Notes in Computer Science, Springer, 2013, pp. 311–329.

[26] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, A. H. M. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Science of Computer Programming 67 (2007) 162–198. URL: http://www.sciencedirect.com/science/article/pii/S0167642307000500. doi:10.1016/j.scico.2007.03.002.

[27] ISO 13209, Road vehicles – Open Test sequence eXchange format (OTX), ISO, Geneva, Switzerland, 2012.

[28] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow Patterns, Distributed and Parallel Databases 14 (2003) 5–51. doi:10.1023/A:1022883727209.

[29] K. Jensen, L. M. Kristensen, L. Wells, Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (2007) 213–254. URL: http://link.springer.com/article/10.1007/s10009-007-0038-x. doi:10.1007/s10009-007-0038-x.

[30] R. Mrasek, J. Mülle, K. Böhm, A new verification technique for large processes based on identification of relevant tasks, Information Systems (2014). doi:10.1016/j.is.2014.07.001.

[31] ISO 9241-11, Ergonomics of Human-Computer Interaction - Part 11: Guidance on Useability, ISO, Geneva, Switzerland, 1998.

[32] A. Bangor, P. T. Kortum, J. T. Miller, An Empirical Evaluation of the System Usability Scale, International Journal of Human-Computer Interaction 24 (2008) 574–594. doi:10.1080/10447310802205776.

[33] O. M. Group, Business Process Model and Notation (BPMN), 2011.

[34] W. M. P. van der Aalst, A. H. M. ter Hofstede, YAWL: yet another workflow language, Information Systems 30 (2005) 245–275. doi:10.1016/j.is.2004.02.002.

[35] W. M. P. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits, Systems and Computers 08 (1998) 21–66. doi:10.1142/S0218126698000043.

[36] M. Brambilla, A. Deutsch, L. Sui, V. Vianu, The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae, in: D. Lowe, M. Gaedke (Eds.), Web Engineering, number 3579 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 557–568. URL: http://link.springer.com/chapter/10.1007/11531371_70.

[37] A. Awad, G. Decker, M. Weske, Efficient Compliance Checking Using BPMN-Q and Temporal Logic, in: M. Dumas, M. Reichert, M.-C. Shan (Eds.), Business Process Management, number 5240 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 326–341.

[38] R. L. Smith, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, PROPEL: An Approach Supporting Property Elucidation, in: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, ACM, New York, NY, USA, 2002, pp. 11–21. URL: http://doi.acm.org/10.1145/581339.581345. doi:10.1145/581339.581345.

[39] R. L. Cobleigh, G. S. Avrunin, L. 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, ACM, New York, NY, USA, 2006, pp. 208–218. URL: http://doi.acm.org/10.1145/1181775.1181801. doi:10.1145/1181775.1181801.

[40] N. S. Abdullah, S. Sadiq, M. Indulska, A Compliance Management Ontology: Developing Shared Understanding through Models, in: J. Ralyté, X. Franch, S. Brinkkemper, S. Wrycza (Eds.), Advanced Information Systems Engineering, number 7328 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 429–444. URL: http://link.springer.com/chapter/10.1007/978-3-642-31095-9_28.

[41] E. R. Taghiabadi, D. Fahland, B. F. v. Dongen, W. M. P. van der Aalst, Diagnostic Information for Compliance Checking of Temporal Compliance Requirements, in: C. Salinesi, M. C. Norrie, s. Pastor (Eds.), Advanced Information Systems Engineering, number 7908 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 304–320. URL: http://link.springer.com/chapter/10.1007/978-3-642-38709-8_20.

[42] N. Lohmann, E. Verbeek, R. Dijkman, Petri Net Transformations for Business Processes – A Survey, in: K. Jensen, W. M. P. van der Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, number 5460 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 46–63.

[43] S. Hinz, K. Schmidt, C. Stahl, Transforming BPEL to Petri Nets, in: W. M. P. v. d. Aalst, B. Benatallah, F. Casati, F. Curbera (Eds.), Business Process Management, number 3649 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 220–235. URL: http://link.springer.com/chapter/10.1007/11538394_15.

[44] N. Lohmann, D. Fahland, Where Did I Go Wrong?, in: S. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management, number 8659 in Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 283–300. URL: http://link.springer.com/chapter/10.1007/978-3-319-10172-9_18.

[45] K. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.

[46] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, Bounded Model Checking, volume 58, Elsevier, 2003, pp. 117–148. URL: http://www.sciencedirect.com/science/article/pii/S0065245803580032.

[47] O. Lichtenstein, A. 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, ACM, New York, NY, USA, 1985, pp. 97–107. URL: http://doi.acm.org/10.1145/318593.318622. doi:10.1145/318593.318622.

[48] M. Montali, M. Pešić, W. M. P. van der Aalst, F. Chesani, P. Mello, S. Storari, Declarative Specification and Verification of Service Choreographies, ACM Trans. Web 4 (2010) 3:1–3:62. doi:10.1145/1658373.1658376.

[49] M. Pešić, D. Bošnački, W. M. P. van der Aalst, Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance, in: J. v. d. Pol, M. Weber (Eds.), Model Checking Software, number 6349 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 146–161.

[50] R. Mrasek, J. Mülle, K. Böhm, Automatic Generation of Optimized Process Models from Declarative Specifications, in: Advanced Information Systems Engineering, Springer Berlin Heidelberg, Stockholm Schweden, 2015.