User-Friendly Property Specification and Process Verification – a Case Study with Vehicle-Commissioning Processes

Richard Mrasek[1], Jutta Mülle[1], Klemens Böhm[1], Michael Becker[2], and Christian Allmann[2]

[1] Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany
{ richard.mrasek | jutta.muelle | klemens.boehm }@kit.edu
[2] AUDI AG, 85045 Ingolstadt, Germany
{ christian.allmann | michael1.becker }@audi.de

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 patterns properties of testing processes adhere to, and we describe these patterns. They depend on the context of the processes, e.g., the components of the vehicle or testing stations. We have developed a framework that instantiate the property patterns at verification time and then verifies the process against these instances. 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.

1 Introduction

The systematic testing and configuration of complex products, e.g., vehicles, is an important step of any production process. To this end, 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 [33]. Workflows called commissioning processes describe the arrangement of these tasks. Domain experts of the industrial partner develop the commissioning processes. Workflow management systems (WfMS) in the production domain that plan and coordinate the testing and end-of-line manufacturing are referred to as diagnostic frameworks.

Our overall goal is to verify if a commissioning process given is correct. To verify a process, one must specify which properties the process must fulfill. We have collected such properties in cooperation with domain experts by analyzing existing processes, and by closely observing these experts when designing processes. To illustrate, if a process uses more connections than available, the process must halt, i.e., process execution time is unnecessarily long. A common definition of correctness of a process is that it observes all properties required. Properties typically are formulated as property rules, similarly to compliance rules [16][18]. For example, a property rule states that before executing Task `X` another Task `Y` has to be executed.

Verification itself is a process that consists of several phases, namely specifying the properties of the commissioning process, verifying them, and presenting the results to the users. Our concern is the design and realization of a framework supporting users throughout this entire process. 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 [25][26]. However, deriving and specifying the properties the model must satisfy is a separate 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 a 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 [15]. 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 [23]. With vehicle-commissioning processes as well as in other domains, see, e.g., [10], [20], specifying the properties in this way is error-prone and generally infeasible for domain experts. To allow for 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, 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 patterns these properties adhere to. We propose to explicitly represent these patterns, rather than each individual property. Next, we develop a model of the context knowledge regarding vehicle-commissioning processes. Here *context* 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

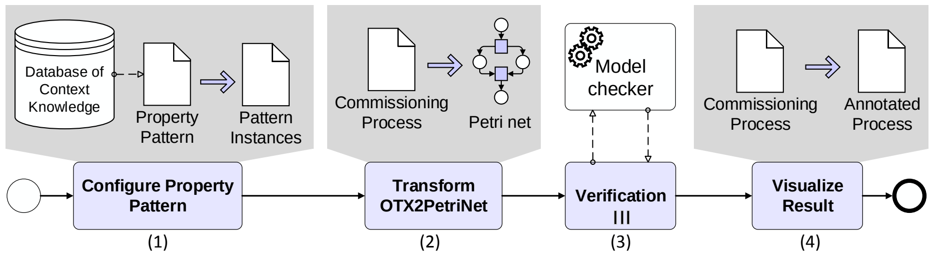


Fig. 1. Steps of the Verification Framework

manage the context information. To populate it, we use several sources, e.g., information on the vehicle components from production planning, constraints from existing commissioning processes, and information provided by the process designers themselves. Our framework uses this information to generate process-specific instances of the property patterns, transforms the process to a Petri net, and verifies it against these properties, see Figure 1. Our evaluation has shown that the framework 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 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. Second, we have evaluated the non-functional requirements on our framework by means of expert interviews, as part of an established test. Our evaluation is one of only few studies that collect feedback from domain experts systematically. We conclude that our framework is operational, sufficiently general and usable in a real production environment.

Section 2 describes our scenario *commissioning processes*. Section 3 introduces our notation. Section 4 explains how to specify the properties required, and Section 5 says how to verify them. Section 6 describes our framework. Section 7 features our evaluation. Section 8 reviews related work, Section 9 concludes.

2 Scenario and Requirements

Commissioning processes describe the end-of-line manufacturing and testing of vehicles. This 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. To check an ECU, several tasks have to be executed. 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 it is necessary to check each of them. 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. Diagnostic Frameworks, i.e., respective workflow management systems, 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 exists.

Example 1. 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, otherwise the task is allocated to a worker. 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 criteria, the framework must meet the needs of the process developers in practice: 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 – we have categorized these non-functional requirements into three categories, namely quality, generality and usability.

3 Notation

In this section we introduce the notations 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 processes 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., [2] and [8].
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 $M : P \rightarrow \mathbb{N}_0$ 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.

CTL is a temporal logic to specify properties. Model checking algorithms exist to efficiently verify CTL properties [7]. The CTL syntax is as follows:

Definition 2 (Computation Tree Logic:). *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. The operators always occur in pairs: a path operator (A or E) and a temporal operator (X,G,F or U). A means that the formula holds in *all* succeeding execution paths, E means that at least one execution path *exists*. 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.

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 usually required to specify the properties for a process. To support this step, we have collected so-called property patterns 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 concrete process context is used to query a database for the information required to dynamically generate instances of the property patterns. Section 4.2 identifies recurring characteristics of such patterns and proposes a respective database representation. Section 4.3 says how to use the patterns to generate process-specific instances of the patterns.

4.1 Properties and Property Patterns for Commissioning Processes

We have identified typical properties of commissioning processes and characteristics of processes, 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 patterns.

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 patterns for commissioning processes. They are the result of a comprehensive survey of ours to detect all dependencies that are conceivable.

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. Properties that refer to structural constraints like the occurrence and arrangement of tasks can be expressed in temporal logic and thus call for a model-checking verification. Violations of those properties can result in undesirable characteristics of the process execution, subsequently referred to as *major disturbance*. An example is that it may block the execution of the process. This holds for properties P3, P4 and P5. Our approach is to define patterns for these properties. Table 1 shows the patterns and the respective CTL formulas. The atomic propositions are inequations referring to states of a Petri net. For instance, ($\textbf{run-A} > 0$) refers to all states where the place **run-A** contains more than zero tokens, i.e., A is currently running. We use the term *minor disturbance* accordingly. This holds for properties P1 and P2. They are on a representational level, i.e., the syntax and the environment of the processes. Examples are violations of conventions or deviation from best practice or from guidelines. We use a syntax check and a query-based verification to check these properties, see *Data Reconciliation* in Section 5.

4.2 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. We then 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:

Table 1. Property Patterns for Task and ECU Conditions

Prop.	Name	Description	CTL
P3.1	Maximal **UDS** Connections	The number of connections to **UDS** should not exceed 10.	$\mathsf{AG}(\neg(\mathbf{UDS} > 10))$
P3.2	Maximal **KWP-2000** Connections	The number of connections to **KWP2000** should not exceed 10.	$\mathsf{AG}(\neg(\mathbf{KWP2000} > 10))$
P3.3	Maximal Connections	The number of connections **UDS** and **KWP2000** should not exceed 14.	$\mathsf{AG}(\neg(\mathbf{UDS} + \mathbf{KWP2000} > 14))$
P4.1	Sequential before	If a task A is in the commissioning process a task B has to occur before A.	$\mathsf{A}\ [\neg(\textbf{run-A} > 0)\ \mathsf{W}\ (\textbf{run-B} > 0)]$
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.	$\mathsf{A}\ [\neg(\textbf{run-A} > 0) \vee \mathsf{AG}\ (\neg\textbf{run-B} > 0))\ \mathsf{W}\ \textbf{run-B} > 0]$
P4.3	Sequential after	The occurrence of task A leads to the occurrence of task B.	$\mathsf{AG}\ (\textbf{run-A} > 0 \rightarrow \mathsf{AF}\ (\textbf{run-B} > 0))$
P4.4	Non-Parallel	Tasks A and B are not allowed to occur in parallel.	$\mathsf{AG}\ (\neg((\textbf{run-A} > 0) \wedge (\textbf{run-B} > 0)))$
P5.1	Restricted access	Only one task at the same time can access/test each ECU C.	$\mathsf{AG}\ (\neg(\mathbf{C} > 1))$
P5.3	Non-Parallel	Some ECU C must never be tested in parallel with an ECU C_2.	$\mathsf{AG}\ (\neg((\mathbf{C} > 0) \wedge(\mathbf{C}_2 > 0)))$
P5.4	Close Connection	Task $\mathsf{close\text{-}C}$ must close the connection to an ECU C.	$\mathsf{AG}\ ((\mathbf{C} > 0) \rightarrow \mathsf{AF}\ (\textbf{close-C} > 0))$

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 2 shows an excerpt of our database model illustrating the overall structure. Appendix A shows the complete diagram, see [27] for more details. Our

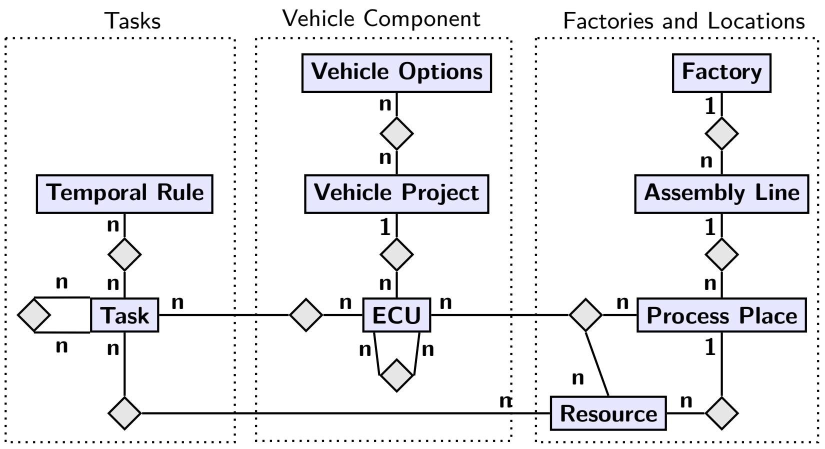


Fig. 2. Excerpt of the Database Schema for Context Knowledge

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

4.3 Pattern Instances

As part of the verification, our framework determines the context of the process first. It is used to query the database for the information required to dynamically generate instances of the property patterns of Table 1.

Example 2. The process to be verified contains the ECUs = [*GWA, KEL, FBE*]. For the process place *VP2* and vehicle series M3, an ECU dependency exists that *KEL* and *FBE* must not be used in parallel. For Property Pattern P5.3 our framework generates the following property: $\mathsf{AG}(\neg((KEL{>}0) \wedge (FBE{>}0)))$.

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 specify 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 Pattern "*A* leads to *B*" has a few hundred instances. If, for example, the need to change the pattern to "The first occurrence of *A* leads to *B*" arose, updating would be avoided. 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 Verification

We now describe the architecture of our verification framework and how it verifies if a commissioning process fulfills a set of property instances. The industrial partner uses several different process notations, depending on the factory and vehicle project. OTX is an ISO-Standard [13] that is planned as a vendor-independent standard for commissioning processes. A preprocessing step transforms a process file in another format into OTX (Figure 3.1). Next, the context information regarding the process place and the vehicle project are extracted from the commissioning process (Figure 3.2). Not all properties can be verified with one paradigm. Therefore, our program consists of two modules: the *Data-Reconciliation* (Figure 3, Step 3) and the *Model Checker* (Figure 3 Steps 4-6). In the past, researchers have developed efficient tools for model checking with Petri nets [24][14]. Hence, model checking in the narrow sense of the word is not a topic of this article. Our framework contains an established framework for model checking [24].

5.1 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. Then, the module 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.

5.2 Model Checking

Model Checking is the problem of finding all states s such that the state machine M has a given property ϕ in s. The commissioning processes are given in OTX,

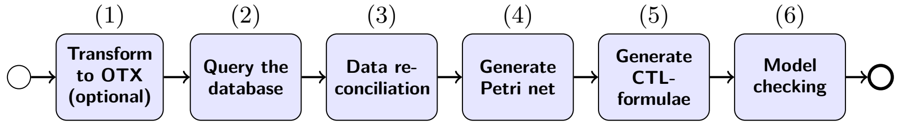


Fig. 3. The Verification Steps

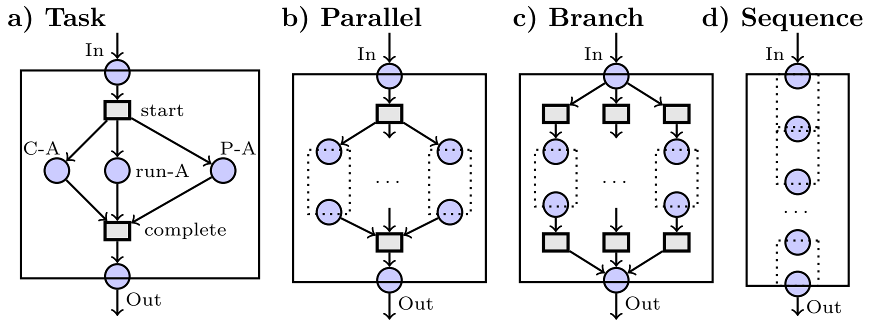
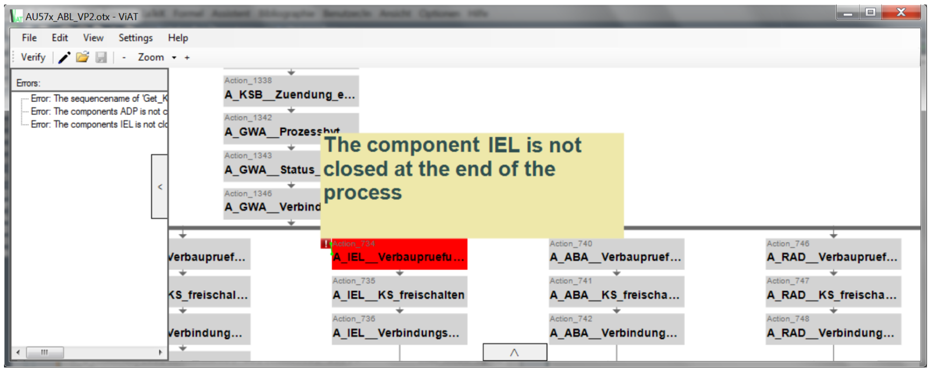


Fig. 4. The Templates for a Task (a), a Parallel-Node (b), a Branch-Node (c) and a Sequence-Node (d).

a block-based language [17] similar to WS-BPEL [33]. OTX does not allow for a direct construction of the state space. Therefore, we transform the OTX process model into a Petri net and can then analyze its state space.

Transformation: OTX describes the process as tree structure (cf. RPST [31]). Each leaf node corresponds to a task, and each inner node represents a control structure, e.g., parallel execution, exclusive execution or sequential execution of the child nodes. For each type of nodes we define a template. A template is a Petri net with an input *In* and an output place *Out*. Figure 4 shows the templates for a task, a parallel node, a branch node and a sequence node. The control structure nodes have specific regions, where to include the child elements (dotted boxes in Figure 4). To transform the OTX process, we parse the process tree in a breadth-first manner and include for each node the respective Petri net template into the net. Our approach is similar to the one of [12] and [29]. Figure 4 a) shows the Petri net template for a task, i.e., a commissioning routine. The place *In* marks that Task *A* is activated and ready for execution. If a task execution starts, the transition *start* fires and creates a token in each of the places *run-A*, *C-A* and *P-A*. *run-A* represents the actual execution of the task. *C-A* is the place of the ECU *A* communicates with, and *P-A* gives the bus protocol that *A* uses, either **UDS** or **KWP2000** [33]. Several tasks use the same place for *C-A* and *P-A*.

Verification: For model checking we have included the LoLA-Framework [24] into our framework. It generates the state space for the Petri net and uses a model-checking algorithm to verify the properties. Note that our framework is not specifically tailored to this concrete model checker. We do not foresee any major difficulties when including other frameworks for state-space generation or model checking.



Error: The sequencename of 'Get_K
Error: The components ADP is not c
Error: The components IEL is not cl

The component IEL is not closed at the end of the process

Fig. 5. Screenshot of the Verification Framework

6 Implementation

We have implemented concepts described so far in our framework called AAAFT (Automatic Arrangement of Working Steps in Production and Testing). The database for context knowledge is a MySQL database. We also have implemented a graphical front-end that can load an OTX process, visualize it, verify the process and highlight any task that relates to the violations detected.

Example 3. Figure 5 shows a screenshot of our framework. The program has loaded an OTX process and has verified it against the database. Dark (red) boxes highlight tasks that cause a rule violation, e.g., the component IEL is not closed at the end of the process. On the left-hand side, the framework lists the violations detected.

7 Empirical Evaluation

In Subsection 7.1 we say how we have evaluated that our framework can identify rule violations in real processes. Additionally to this criterion we want to evaluate that the framework does meet the requirements of the process developers in practice, see Subsection 7.2.

7.1 Functional Evaluation

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 discussed the verification results and have categorized the processes into three categories: correct, with minor process disturbance and with major process disturbance. Figure 6 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 (≈ 23%), we could detect a major disturbance.

7.2 Expert Interviews

To evaluate our approach we have held semi-structured expert interviews. We aim to test the three characteristics: process quality, generality and usability, as explained next. The interview guide is available on our website:
http://dbis.ipd.kit.edu/2027.php.

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

Generality: *Can the framework be used in a different context within the company?* For instance, is the framework general enough to be used in another factory? We have also asked how well the framework can be integrated into the tool chain.

Usability: *Can the framework be used in an intuitive way? Is the help of a technical person needed to use the framework?* For usability we have used the Standard System Usability Test (SUS) [5]. 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 [5].

	M1	M2	M3	M4
Nr. of Processes	13	17	25	5
Correct	3	3	0	0
Minor Disturbance	9	8	18	5
Major Disturbance	1	6	7	0

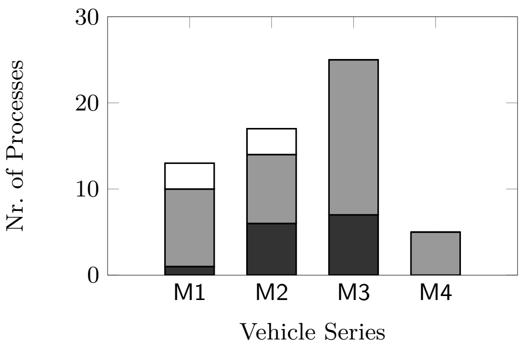


Fig. 6. Process Disturbances Found in the Processes Evaluated

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 range between 1 and 14 years, with an average of 7 years.

Results and Discussion: Figure 7 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 false negatives are acceptable but should be improved. Our framework detected slightly more false positives than false negatives. The experts saw a great potential of our framework to be used in other testing environments as well. The rating of how well the framework can be integrated into the tool chains varies between the experts. The SUS score (a measure for the usability) ranges between 65 and 85 with an average of 71.67. This is slightly above the average (69.69) and median (70.91) of reported studies using the SUS score [5]. All experts see great potential in improving the quality of the commissioning processes by means of our framework.

Conclusion: The evaluation has shown that the experts deem our framework very useful and with high potential to enhance process quality. A minor issue is that they have criticized the amount of information presented by our framework. To this end, 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. To this end, we currently are reimplementing it in C#. The framework currently is implemented in Java.

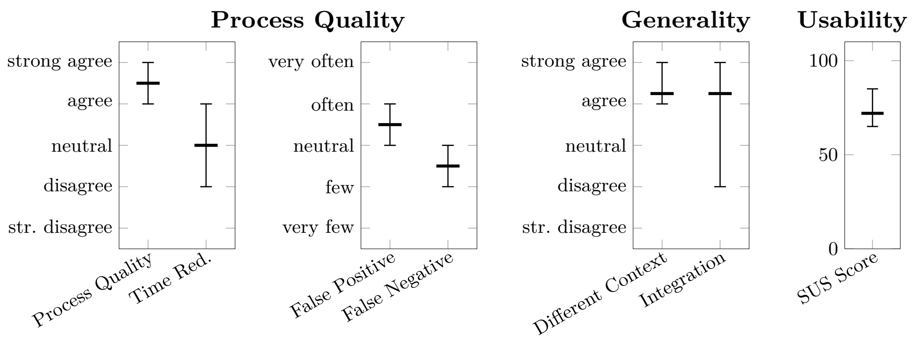


Fig. 7. Results of the Empirical Evaluation

8 Related Work

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

Specification: The direct specification of properties in a formalism like CTL is error-prone and not feasible for a user without experience in formal specification. To this end, different approaches have been developed. Most business processes are modeled in a graph-based modeling language like BPMN [32], YAWL [3] or Petri nets [1]. [6] extends the BPMN notation with new elements that directly represent LTL operators. BPMN-Q [4] extends BPMN with new edge types that represent sequential ordering, between tasks. Compliance Rule Graphs [20] allow a specification of requirements in a graph-based formal language. Another approach is the use of specification patterns. [10] introduces the property patterns to specify concurrent systems. [28] extends the pattern system to PROPEL (PROPerty ELucidation) to cover variations of the property patterns. [9] uses a question tree to allow specifying PROPEL patterns. In our domain, only a few different property patterns exist. Dependent on the context, many instances are generated. Because of the small number of patterns but many similar instances, we have not found any of the approaches to be very helpful in our specific case.

Management of Properties: [30] 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 patterns. Managing compliance properties includes allocating the properties to the business processes. [15] 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.

Verification: We aim to check if a business process complies with the properties given. [22] 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. [19] gives an overview of transformations from BPMN, YAWL and WS-BPEL to Petri nets. Our approach is similar to [12]. [21] empirically evaluates different approaches for soundness verification. The criteria include error rates, process size and verification time. [11] 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 [21] and [11]. 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.

9 Conclusions

To avoid property violations in commissioning processes, a framework to verify if a process is correct clearly is helpful. With verification, an important step is specifying which properties a process must fulfill. Given that verification algorithms already exist, a core question is how to arrive at a user-friendly framework for process verification that supports collecting and maintaining the properties.

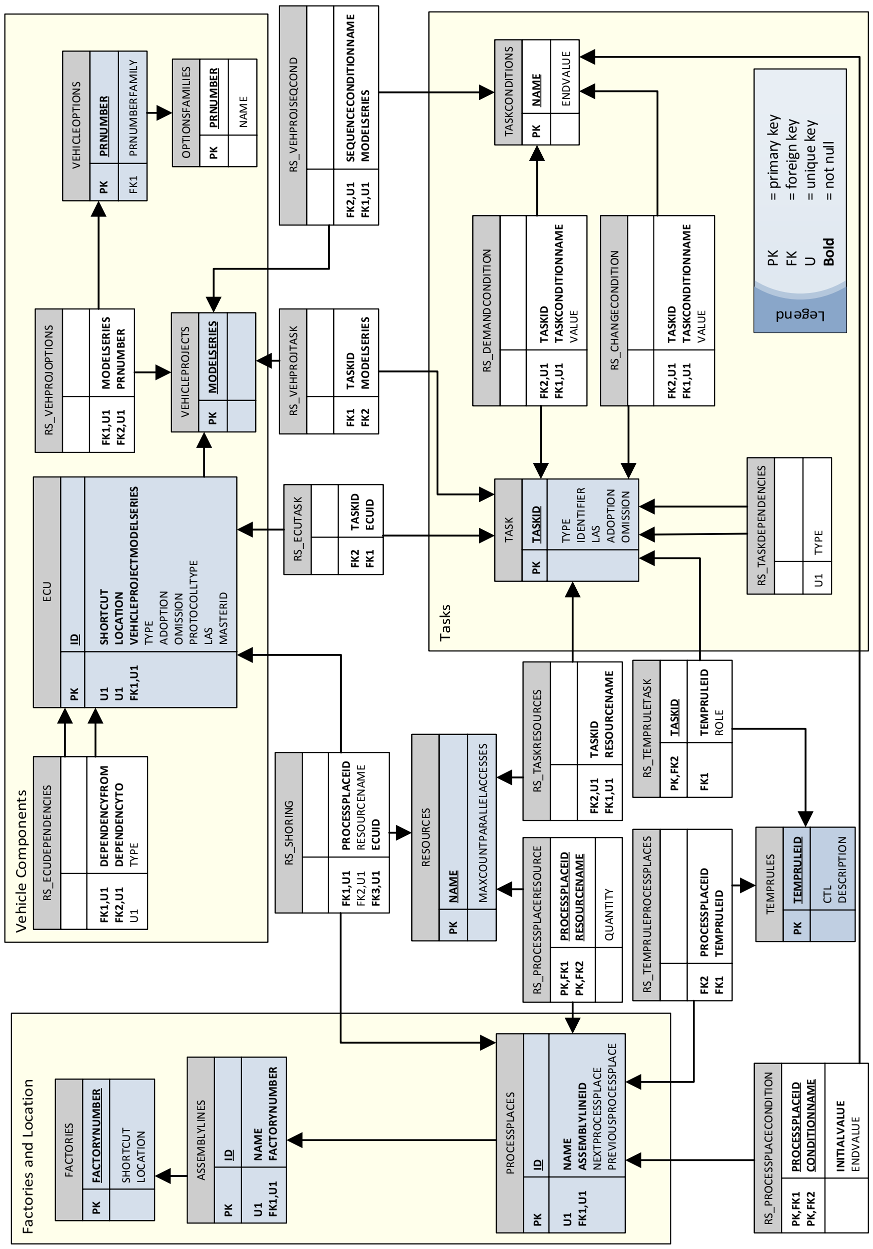
We have analyzed which properties vehicle commissioning processes in the automotive industry must fulfill and have identified the context information relevant for verification. An important insight has been that there exist only a few types of properties, but the number of properties may be very large. Thus, an important design decision has been to develop a database with contextual information and to focus on property patterns covering all properties relevant for vehicle-commissioning processes. Consequently, we have proposed a transformation of patterns to properties tailored to a certain process model. Our framework then verifies these properties on the commissioning processes. An interview-based evaluation together with domain experts has shown that the framework does enhance the process quality. Ongoing work addresses an even tighter integration of the framework and its database into the tool chain.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers (1998)
2. van der Aalst, W.M.P., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press (2004)
3. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems (2005)
4. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. Conf. Business Process Management (2008)
5. Bangor, A., Kortum, P.T., Miller, J.T.: An Empirical Evaluation of the System Usability Scale. International Journal of Human-Computer Interaction (2008)
6. Brambilla, M. et al.: The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae. Conf. Web Engineering (2005)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. (1986)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (1999)
9. Cobleigh, R.L., Avrunin, G.S., Clarke, L.A.: User Guidance for Creating Precise and Accessible Property Specifications. ACM SIGSOFT International Symposium on Foundations of Software Engineering. (2006)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-State Verification. In: 2nd Workshop Formal Methods in Software Practice. (1998)
11. Fahland, D. et al.: Instantaneous Soundness Checking of Industrial Business Process Models. Conf. Business Process Management. (2009)

12. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets. Conf. Business Process Management. (2005)
13. ISO, Geneva, Switzerland: Road vehicles – Open Test sequence eXchange format (OTX). ISO 13209 (2012)
14. Jensen, Kurt, Lars Michael Kristensen, and Lisa Wells. "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems." International Journal on Software Tools for Technology Transfer (2007)
15. Kabicher, S., Rinderle-Ma, S., Ly, L.T.: Activity-Oriented Clustering Techniques in Large Process and Compliance Rule Repositories. Workshop on Process Model Collections (2011)
16. Knuplesch, D. et al.: On Enabling Data-Aware Compliance Checking of Business Process Models. Conf. Conceptual Modeling. (2010)
17. Kopp, O., et al.: The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems Architecture (2009)
18. Liu, Y., Muller, S., Xu, K.: A Static Compliance-Checking Framework for Business Process Models. IBM Systems Journal (2007)
19. Lohmann, N., Verbeek, E., Dijkman, R.: Petri Net Transformations for Business Processes–a Survey. Transactions on Petri Nets and Other Models of Concurrency II (2009)
20. Ly, L. et al.: SeaFlows Toolset – Compliance Verification Made Easy for Process-Aware Information Systems. Conf. Information Systems Evolution. (2011)
21. Mendling, J.: Empirical Studies in Process Model Verification. Transactions on Petri Nets and Other Models of Concurrency II. (2009)
22. Ramezani Taghiabadi, E. et al.: Diagnostic Information for Compliance Checking of Temporal Compliance Requirements. Conf. Adv. Inf. Syst. Engineering. (2013)
23. Schlingloff, H., Martens, A., Schmidt, K.: Modeling and Model Checking Web Services. Electronic Notes in Theoretical Computer Science (2005)
24. Schmidt, K.: LoLA A Low Level Analyser. Conf. Application and Theory of Petri Nets. (2000)
25. Schmidt, K.: Stubborn Sets for Standard Properties. Conf. Application and Theory of Petri Nets (1999)
26. Schmidt, K.: Stubborn Sets for Model Checking the EF/AG Fragment of CTL. Fundamenta Informaticae (2000)
27. Schneider, T.: Specification of Testing Workflows for Vehicles and Validation of Manually Created Testing Processes (in German). Master's thesis, Karlsruhe Institute of Technology (May 2012)
28. Smith, R.L. et al.: PROPEL: An Approach Supporting Property Elucidation. Conference on Software Engineering. (2002)
29. Stahl, C.: A Petri Net Semantics for BPEL, Technical Report 188. Humbold-Universität zu Berlin (2005)
30. Syed Abdullah, N., Sadiq, S., Indulska, M.: A Compliance Management Ontology: Developing Shared Understanding through Models. Conf. Adv. Inf. Syst. Engineering. (2012)
31. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. Conf. Business Process Management. (2008)
32. Wohed, P. et al.: On the Suitability of BPMN for Business Process Modelling. Conf. Business Process Management. (2006)
33. Zimmermann, W., Schmidgall, R.: Bussysteme in der Fahrzeugtechnik – Protokolle, Standards und Softwarearchitektur, Vieweg + Teubner (2010)

A The Complete Database Model


Vehicle Components
VEHICLEOPTIONS
PK PRNUMBER
FK1 PRNUMBERFAMILY
OPTIONSFAMILIES
PK PRNUMBER
NAME
RS_VEHPROJOPTIONS
FK1,U1 MODELSERIES
FK2,U1 PRNUMBER
VEHICLEPROJECTS
PK MODELSERIES
ECU
PK ID
U1 SHORTCUT
U1 LOCATION
FK1,U1 VEHICLEPROJECTMODELSERIES
TYPE
ADOPTION
OMISSION
PROTOCOLLTYPE
LAS
MASTERID
RS_ECUDEPENDENCIES
FK1,U1 DEPENDENCYFROM
FK2,U1 DEPENDENCYTO
U1 TYPE
RS_VEHPROJSEQCOND
FK2,U1 SEQUENCECONDITIONNAME
FK1,U1 MODELSERIES
RS_VEHPROJTASK
FK1 TASKID
FK2 MODELSERIES
RS_ECUTASK
FK2 TASKID
FK1 ECUID
RS_SHORING
FK1,U1 PROCESSPLACEID
FK2,U1 RESOURCENAME
FK3,U1 ECUID
RESOURCES
PK NAME
MAXCOUNTPARALLELACCESSES
Tasks
TASKCONDITIONS
PK NAME
ENDVALUE
RS_DEMANDCONDITION
FK2,U1 TASKID
FK1,U1 TASKCONDITIONNAME
VALUE
RS_CHANGECONDITION
FK2,U1 TASKID
FK1,U1 TASKCONDITIONNAME
VALUE
Legend
PK = primary key
FK = foreign key
U = unique key
Bold = not null
TASK
PK TASKID
TYPE
IDENTIFIER
LAS
ADOPTION
OMISSION
RS_TASKDEPENDENCIES
U1 TYPE
RS_TASKRESOURCES
FK2,U1 TASKID
FK1,U1 RESOURCENAME
RS_TEMPRULETASK
PK,FK2 TASKID
FK1 TEMPRULEID
ROLE
RS_PROCESSPLACERESOURCE
PK,FK1 PROCESSPLACEID
PK,FK2 RESOURCENAME
QUANTITY
RS_TEMPRULEPROCESSPLACES
FK2 PROCESSPLACEID
FK1 TEMPRULEID
TEMPRULES
PK TEMPRULEID
CTL
DESCRIPTION
Factories and Location
FACTORIES
PK FACTORYNUMBER
SHORTCUT
LOCATION
ASSEMBLYLINES
PK ID
U1 NAME
FK1,U1 FACTORYNUMBER
PROCESSPLACES
PK ID
U1 NAME
FK1,U1 ASSEMBLYLINEID
NEXTPROCESSPLACE
PREVIOUSPROCESSPLACE
RS_PROCESSPLACECONDITION
PK,FK1 PROCESSPLACEID
PK,FK2 CONDITIONNAME
INITIALVALUE
ENDVALUE