Correct Formalization of Requirement Specifications: A V-Model for Building Formal Models

Marco Filax(✉), Tim Gonschorek, and Frank Ortmeier

Chair of Software Engineering, Otto-von-Guericke Universität Magdeburg, Magdeburg, Germany
{marco.filax,tim.gonschorek,frank.ortmeier}@ovgu.de

Abstract. In recent years, formal methods have become an important approach to ensure the correct function of complex hardware and software systems. Many standards for safety critical systems recommend or even require the use of formal methods. However, building a formal model for a given specification is challenging. This is, because verification results must be considered with respect to the validity of the model.

This leads to the question: "Did I build the right model?". For system development the analogous question "Did I build the right system?". This is often answered with requirements traceability through the whole development cycle. For formal verification this question often remains unanswered.

The standard model, which is used in development of safety critical applications is the V-model. The core idea is to define tests for each phase during system development. In this paper, we propose an approach - analogously to the V-model for development - which ensures correctness of the formal model with respect to requirements. We will illustrate the approach on a small example from the railways domain.

Keywords: Formal modelling process · Requirements traceability · System verification · Railway system verification

1 Introduction

Reliability, Availability, Maintainability and Safety are important aspects of the development of railway transportation systems. But, the more complex the system gets the harder it becomes to verify, often by hand, whether a system meets its given specification. Since modern systems are too complex to get verified and validated by hand, especially for the safety critical parts, formal verification comes into the focus. Moreover, formal verification techniques are highly recommended in standards, like the EN 50129 [7]. However, developing a formal model which is a sufficient representation of the real system, is a quite challenging task. It is not only to build the model, but to ensure that the formal models meets all requirements, i.e., is a representative projection of the system-to-build – especially in front of some governmental certification authority.

T. Lecomte et al. (Eds.): RSSRail 2016, LNCS 9707, pp. 106–122, 2016.
DOI: 10.1007/978-3-319-33951-1_8

The goal of this paper is to propose a development approach that simplifies the application of formal methods in the development of safety critical systems in general, but in special in the development of railway systems. The whole process had been determined in cooperation with the German Federal Railway Authority[1] and an independent and certified appraiser for the rail domain.

To overcome the difficulty of building the correct formal model, we focus on the traceability of all requirements to their formal realization, i.e., as for the real system, it shall be possible to trace each formal model element to its original requirement. Moreover, we derive additional acceptance and system tests from the different development phases and present how they can be verified with the help of model checking [11] techniques. In the scope of this paper, we concentrate on the verification of system safety requirements. However, with the help of the formal model, other reliability and availability measures, e.g., Fault Tree Analysis or Failure Mode Effects Analysis, can be executed.

Unfortunately, domain experts are often unfamiliar with formal verification languages and techniques. Therefore, we use our Verification Environment for Critical Systems (VECS[2]) [17], aiming to simplify the application of formal methods, which implements an import interface to the more popular Unified Modeling Language (UML). For the life-cycle pattern, we propose a process inspired by the established V-model. Project specific variations of the V-model have already been applied successfully in a range of different software projects and they all share the prominent V-shape. In 1984 Boehm introduced the characteristic V-shape [1]. However, it still symbolizes a linear project progression. It is divided into two branches: the left branch symbolizes exploratory and design tasks whereas the right branch represents verification and validation tasks. The methodology does not require a specific (formal) implementation language. It rather specifies what a product describes and recommends methods for the production [3] and thus can be used to develop a formal model. Further, it helps to ensure the traceability throughout the whole process.

The paper is structured as follows: Sect. 2 gives a short overview of related works from other authors. In Sect. 3 we present the proposed approach ensuring the correct formalization of large requirement sets while preserving traceability. Section 4 reports our experiences made with the proposed process while verifying a real world spot transmission based train breaking system. In the end, we conclude our paper in Sect. 5.

2 Related Work

A lot of work has been done for formalizing a set of requirements using UML as an intermediate language. Typically, these approaches cover the extraction and formalization of a semi-formal given architecture [2,4,16,19]. Some approaches consider the extraction of behavior, denoted as state machines, in order to generate a formal model from UML [12,20–22,24,27]. However, the execution semantics of state machines in UML is ambiguous [14]. Snook and Butler proposed an

[1] http://www.eba.bund.de.
[2] http://cse.cs.ovgu.de/vecs.

approach to translate architectural and behavioral UML entities into B [24]. The authors propose to develop a formal model from classes and their relation. Further, they require a complete behavioral description in UML. Utilizing classes requires the definition of a detailed architecture derived only from the set of requirements. Defining a system in this degree of detail requires a lot of insight in the developed system inappropriate for requirement analysis. The authors also propose to translate contents of a package into a single formal component, which we think is unsuitable for larger requirement specifications.

Brill et al. use the V-model in order to generate a formal model, but developed as state charts, in particular statemate [3]. In contrast to our approach, they propose to use live sequence charts derived in earlier and refined in later phases in order to "aid in debugging" the formal model. Although not proposed, their approach can be extended to be similar to ours. This is, because the methodology to use commonly known behavior descriptions in order to proof the feasibility of the formal model is shared. In contrast to our approach they utilize send and receive events in order to express interfaces between components. Thus, their approach does not feature parameters for operations - vital in almost every state-of-the-art programming language. This means, that their approach cannot be used to validate whether the architecture of the model is feasible.

Other approaches verify the feasibility of scenarios described in a SRS [25]. The authors propose to refine scenarios, manually extracted from the requirements, with sequences. Every sequence shall then be translated into a temporal logic formula. A model checker can compute whether the specification holds for a specific model. The authors do not convincingly demonstrate how architecture and behavior of the formal model are developed nor emphasize the traceability between model and requirements in their approach. Further, the computation of witnesses in order to support reviews through domain experts is not covered. However, we think such witnesses are important for the proposed approach as the formal model can be erroneous and witnesses can serve as provable example paths in the models state space, emphasizing the requirement coherence.

Carnevali et al. proposed a methodology to integrate preemptive time petri nets into a given software development cycle [5]. They utilized a V-model issued by the German Federal Administration for software development, maintenance and modification. However, they did not describe the integration in a semi-formal description language, such that the traceability of requirements through the hole process life cycle is not emphasized. As a result, a reviewing process of domain experts which are typically not familiar with timed petri nets is required.

3 A Process for Building Formal Models "right"

Building a formal model is a challenging task. It typically takes numerous iterations until one is satisfied with the model. In each iteration some unwanted behavior is removed, extra functionality is added and/or inconsistencies are eliminated. In contrast to unstructured modeling, we propose a structured process, inspired by the V-model, which ensures the coherence of the formal model and

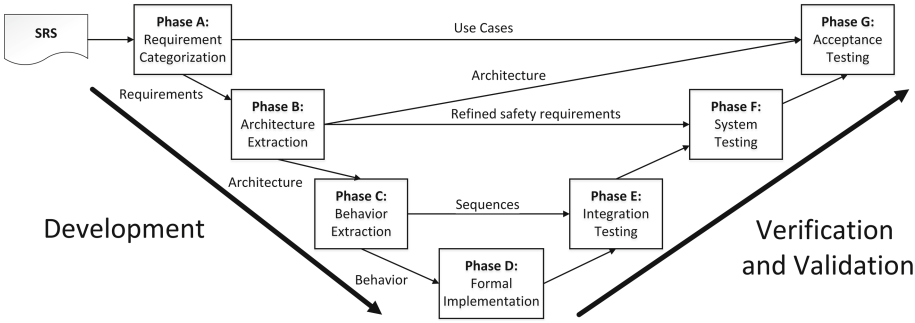

Fig. 1. The adapted V-model for developing a traceable, complete, consistent and correct formal model.

its informal SRS by preserving the traceability between elements in the formal model and their origin. The core idea is – like in the system development model – to define different phases of the modeling process and use state as well as sequence acceptance criteria, test properties or hierarchically analysis questions for validating the model (cf. Sects. 3.5, 3.6 and 3.7).

Figure 1 illustrates the proposed process. All in all, the proposed process consists of seven consecutive phases (Phase A - Phase G). As starting point, we assume some informal specifications to be given, mainly written system requirements or additional documents, e.g., some sketched system architecture, specifications of subcomponents, failure mode specifications or other safety relevant documents.

In the following, we give a more detailed description of the different phases. For a better understanding, we illustrate each phase with a small example from a real world case study, issued by the Federal Railway Authority of Germany. This is a standard protection system in German railroads: the *"Punktförmige Zugbeeinflussung" (PZB*[3]*)* – a spot transmission based train braking system. The informal specification consists of a 46 paged document, containing text, graphics and tables. Altogether, this results in 777 requirements and a formal model with a state space of approx. $5,8 * 10^{24}$ states[4].

3.1 Phase A: Requirement Categorization

The goal of this phase is to prepare the informal text such that it can be better processed by the following phases. Therefore, the SRS is divided into a set of atomic and indexable text fragments. Further, the atomic fragments are categorized to determine their semantics for the following process phases. For example, a requirement categorized as architecture fragment is proposed to be used in connection with the architecture modeling (cf. Sect. 3.2). According to the projects

[3] https://cse.cs.ovgu.de/vecs/index.php/techniques/examples/17-casestudies/25-pzb-achievements.

[4] Worst case approximation by multiplying all possible state variable values.

needs one can choose different requirement patterns. In our developed case studies, we found an adoption of the pattern [15] originally defined by Cimatti et al. [8] most applicable for our process. This pattern defines eight different categories for functional requirement fragments. Every category is defined by a condition rule (shown in Table 1) supporting the mapping of fragment and category. Unfortunately, this categorization has to be done manually. This is the case, because processing informal requirements, written without given rules, is, even for humans, a challenging task. Especially, since the mapping of the fragments is not unique. Of course, it is possible that one fragment can be responsible for the architecture (e.g. defining a required method) as well as for some system state behavior (the method sets a specific value). However, the benefit of having ordered requirements, that can be mapped to the different design stages, make it easier to get an overview of the set of requirements. This preponderates the effort of categorizing each requirement fragment by hand.

Table 1. Functional requirement categories adopted from Cimatti et al. [8].

Category	Condition
Glossary requirement	Does the text fragment define a specific concept of the domain?
Architecture requirement	Does the requirement introduce some system's modules and describe how they interact?
State requirement	Does the requirement describe the steps a particular module performs or the states where a module might be in?
Communication requirement	Does the requirement describe messages modules exchange?
Property requirement	Does the requirement describe expected properties of the domain or constraints of the system-to-be?
User requirement	Does the requirement describe actions or constraints which have to be considered, satisfied or performed by the user?
Safety requirement	Does the requirement describe necessary safety constraints?
Annotation	Is the text fragment a note that does not add any information about the ontology or the behavior of the specified system?

We illustrate this phase with a subset of requirements taken from the PZB (Table 2). Requirement PZB1 does not provide any new information and thus is an annotation. PZB2 and PZB3 introduce two different modules, thus they are architecture fragments. PZB4 defines a message being issued by the system's modules and is a communication requirement. PZB5 gives further information on the methods issued in PZB4. PZB6 describes interaction with a surrounding

Table 2. Excerpt of requirements taken from the case study.

ID	Requirement	Category
PZB1	The PZB is a train protection system developed in Germany.	Annotation
PZB2	The system relies on onboard transmitter coils with different frequiencies.	Architecture requirement
PZB3	On the *trackside* different passive tuned *inductors* are installed.	Architecture requirement
PZB4	If a trackside inductor is passed the active *onboard transmitter coil* induces a voltage.	Communication requirement
PZB5	Three frequencies, which can be induced by the *magnets*, have to be distinguished: 1000 Hz, 500 Hz, 2000 Hz.	Property requirement
PZB6	Trackside inductors can be deactivated or activated depending on the signal.	User requirement
PZB7	If the frequencies match, an oscillation is generated in the trackside inductor resulting in an onboard voltage drop indicating an overrun.	State requirement
PZB8	Depending on the transited inductor's frequency different actions have to be issued.	Safety requirement
PZB9	2. The Indusi	Glossary requirement

system: the signal. Here, we consider the signal as an external actor and thus consider it as an user interaction. PZB7 describes the steps a trackside inductor has to perform and thus is a state requirement. PZB8 states that specific actions have to be triggered – relevant for the overall systems safety – meaning an overrun should not be missed and thus, is a safety requirement. PZB9, a section heading, introduces a concept of the domain and therefore is categorized as a glossary fragment.

3.2 Phase B: Architecture Extraction

Typically, functional requirements contain information about system modules and surrounding systems. Their direct formalization, however, is error-prone, as natural language is ambiguous. Thus, we propose to translate informal text into an intermediate language: UML [23]. This is, because UML, as a de facto standard, has a broad audience ensuring that domain experts not familiar with formal verification methods are able to understand basic architecture and in later phases intended behavior (cf. Sect. 3.3) of the formal model. Further, it offers the possibility to derive the architecture of the actual implementation.

In order to represent an architecture, UML offers a variety of elements. We restrict ourselves to components, ports and interfaces in order to define the

high-level architecture. This is, because elaborate elements like classes are designed to reflect implementational aspects. Formal models typically behave differently. Hence, to be able to define a proper transformation from UML into a more formal representation, we need to define the used subset of UML elements.

Definition 1 (Component). *Components in UML describe hierarchically ordered units within a system or subsystem [23]. C is the set of a all components defined by the given requirements. A single component $c_i \in C$ is a tuple $c_i = \langle n_i, P_i, C_{Sub_i} \rangle$ with an identifier n_i, a set of ports P_i and a set of subcomponents C_{Sub_i}.*

Further, we define that $c_i \notin C_{Sub_i}$. In order support the interaction of different components, we need to define interfaces. I is the set of all interfaces defined by the requirement specification in specific: also derived from architectural requirements.

Definition 2 (Interface). *An interface $i_j \in I$ is a declaration of a set of coherent public features and obligations [23]. It is a tuple $i_j = \langle n_j, A_j, O_j \rangle$ with an identifier n_j, a set of attributes $A_j \in A$ where an attribute $a_k = \langle n_k, t_k \rangle$ is a tuple with an identifier n_k, a type t_k and a set of operations $O_j \in O$. An operation is a tuple $o_l = \langle n_l, t_l, PAR_l \rangle$ with an identifier n_l, a type t_l and a set of parameters PAR_l where every parameter par_l is a tuple $par_m = \langle n_m, t_m \rangle$ with an identifier n_m and a type t_m.*

Components can either provide or require an interfaces, which is exposed through a port.

Definition 3 (Port). *A port $p_l \in P$ is a tuple $p = \langle n, i, type \rangle$ with an identifier n, an interface i and the type $\in \{provides, requires\}$.*

Further, we define a function **provides** : $P \to (I \cup \bot)$ mapping a port to exactly one interface:

$$\langle n, i, type \rangle \mapsto \begin{cases} i & \text{iff } type = provides \\ \bot & \text{iff } type = requires \end{cases}. \tag{1}$$

Analogously, **requires** : $P \to (I \cup \bot)$ is defined as

$$\langle n, i, type \rangle \mapsto \begin{cases} i & \text{iff } type = requires \\ \bot & \text{iff } type = provides \end{cases}. \tag{2}$$

Multiple components shall be assembled using their shared boundary: the interface exposed through a port.

Definition 4 (Assembly). *An assembly is a tuple $assembly = \langle P, P \rangle$ where*

$$\begin{aligned} (p_1, p_2) \in assembly \implies & requires(p_1) = provides(p_2) \wedge \\ & provides(p_1) = requires(p_2) = \bot. \end{aligned} \tag{3}$$

All UML elements must be manually derived from the requirements. The categorization of requirements (cf. Phase A) allows to define components much more systematically. Components are typically derived from architecture or glossary requirements. Interfaces and methods shall be extracted from property requirements. We illustrate the proposed approach with the running example. The architecture in Fig. 2 has been derived from the requirements in Table 2. The three different components are derived from the glossary and architecture requirements PZB2, PZB3 and PZB9 namely *trackside inductors* (ti), *onboard transmitter coils* (otc) and *indusi*. PZB5 was used to derive the interfaces *magnet* and *transit*. Further, ports needed to be specified, i.e. *inductors* requiring *magnet* and *coils* providing *magnet*. Both components *ti* and *otc* interchange information, thus an assembly connector is denoted to visualize this information. Analogously, the information has been modeled for *otc* and *indusi*. In order to ensure traceability all requirements had been linked to their UML element as shown in Fig. 2.

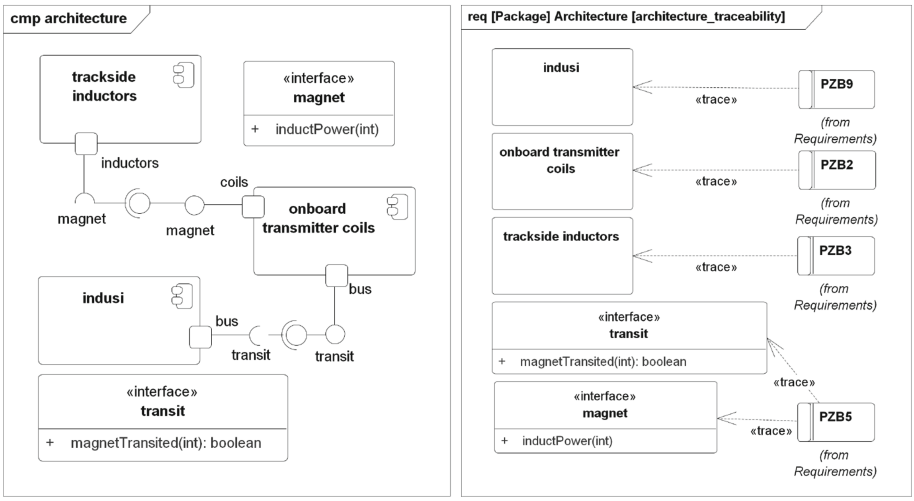

Fig. 2. Example architecture derived from the requirements of the real world case study defined in Table 2.

During this phase, we also refine the requirements that are important for the system testing phase. Safety requirements shall be used in context with the architecture. In order to refine the requirements we propose to use a formal specification language, e.g. linear temporal logic [13]. But the actual formalization depends on the used model checker. An example from the running example, namely for requirement PZB8, is given in Sect. 3.6.

3.3 Phase C: Behavior Extraction

In this step, the intended system behavior is modeled. This is, of course, done with respect to the architecture modeled in the previous step. Often the behavior

in the SRS is not specified completely, i.e., some parts are free to be designed by the modeler in the way that no requirements are not violated. We propose to translate communication requirements into sequences, as a sequence is intended to represent some excerpt of the complete behavior.

Definition 5 (Sequence). *A sequence is a tuple* $seq_m = \langle n_m, C_m, M_m \rangle$ *where* n_m *is its identifier,* $C_m \in C$ *a finite set of components and a finite set of messages* $M_m \in M$*.*

Each sequence describes some communication between different modules specified as messages [23].

Definition 6 (Message). *A message* $m = \langle source, target, o, val, v, index \rangle$ *is a tuple with a* $source \in C_m$ *and a* $target \in C_m$ *representing the origin and destination component of the message* m*. Further, the tuple consists of a operation* o*. Each operation* o *must be discrete, meaning each parameter* $par_o \in o$ *shall be valued. This is expressed through the function* $val : PAR_o \rightarrow V$*, which labels every parameter with a concrete value with respect to its type.* v *is the return value of the operation and index sorts all messages of a sequence* seq*.*

An example sequence is given in Fig. 3, it depicts the behavior described in PZB4. Three actors are imported from the architecture: *ti*, *otc* and *indusi*. *indusi* checks whether a 2000 Hz magnet has been overrun (cf. *magnetTransited(2000)*)

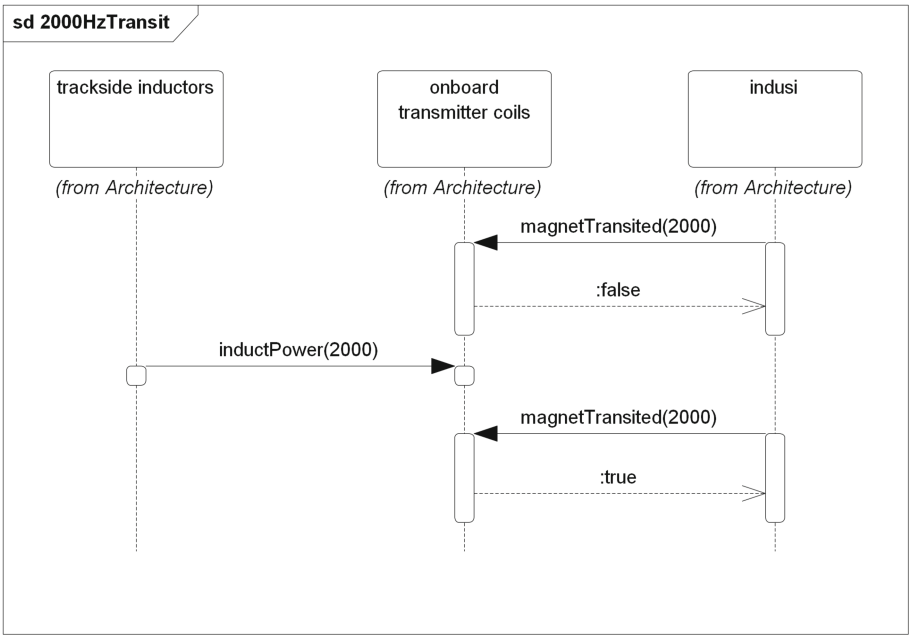

Fig. 3. An example sequence derived from the requirement PZB4: If a trackside inductor is passed the active onboard transmitter coil induces a voltage.

or not. Every time a voltage is inducted from *ti* into *otc* an overrun shall be symbolized. The exchanged messages utilize operation defined in the architecture shown in Fig. 2.

3.4 Phase D: Formal Implementation

During this phase the formal model is implemented. In order to ensure traceability, we propose to generate a formal skeleton from the UML model which had been build from the requirements. Thereby, we retain the option to later use specifications, automatically generated from the corresponding UML sequences, to build integration tests (see Sect. 3.5). Further, it is possible to provide traceability links from the formal implementation tool to the corresponding requirement. However, the modeler has to complete the formal model manually by defining the complete behavior, i.e., modeling all necessary state variables and transition rules.

Definition 7 (Formal Model). *We define a formal model* fm *as a tuple* $fm = \langle FC, SPEC \rangle$ *of a finite set of formal components* FC *and a finite set of formal specifications* $SPEC$*.*

Definition 8 (Formal Component). *A formal component* $fc \in FC$ *is tuple* $fc = \langle n, V, T, F, FC_{Sub} \rangle$ *where* n *is the name,* V *is a finite set of state variables and a set of transition rules* T*.* FC_{Sub_p} *is the set of subcomponents in* fc_p *where* $\forall fc_p \in FC : fc_p \notin FC_{Sub_p}$*.* F *is a set of formulae, whereas a formula* $f = \langle n, t, prop \rangle$ *is a tuple with an identifier* n*, a type* t *and a proposition* $prop$*. Thus, a formula is a stateless, parameter-free, typed and named expression that commonly is an abbreviation for conditions.*

Using formal components and formulae, we are able to define a translation from the previous defined UML subset (cf. Sect. 3.2) into a formal model fm. In order to translate the architecture, every component $c \in C$ has to be transformed into a formal component fc. We define a translation function $transComp : C \rightarrow FC$ with

$$\langle n, P, C_{Sub} \rangle \mapsto \langle n, \bot, \bot, \bot, \{transC(C_{Sub}) \cup transP(P)\} \rangle. \tag{4}$$

It generates for every component $c \in C$ a formal component fc with the same identifier. Further, it recursively invokes the translation of every subcomponent in C_{Sub}. Note that state variables and transition cannot be generated as the architecture does not contain a behavioral definition.

Every port in c has to be translated using $transP : P \rightarrow FC$ with

$$\langle n, i, t \rangle \mapsto \langle n, \bot, \bot, transI(i), \bot \rangle. \tag{5}$$

This generates a new formal component for each port with the same identifier. Further, it invokes the translation $transI$ (cf. Eq. 6) for the interface. It returns the formalization of an interface provided or required by the port. The function $transI : I \mapsto F$ translates an interface into a set of formulae where

$$\langle n, A, O\rangle \mapsto \{transA(A) \cup transO(O)\}. \tag{6}$$

The formalization of an attribute is specified by $transA : A \rightarrow F$, whereas

$$\langle n, t\rangle \mapsto \langle n, t, \bot\rangle. \tag{7}$$

It transforms an attribute with name and type into a formula with the same name and type. The proposition is empty, because its value is not defined by the attribute.

A given operation is translated into a set of formulae with $transO : O \rightarrow F$. With the '.' operator, we access an attribute of the object, e.g., $par_i.type$ access the *type* of parameter par_i.

$$\begin{aligned}\langle n, t, PAR\rangle \mapsto \{&\langle n_{call}, bool, \bot\rangle, \langle n_{par_i}, par_i.type, \bot\rangle, ..., \\ &\langle n_{par_j}, par_j.type, \bot\rangle, \langle n_{return}, t, \bot\rangle\}.\end{aligned} \tag{8}$$

Every operation is translated into a formula representing the method call, a formulae for every parameter and a formula representing the return. Note that the propositions cannot be generated as the complete intended behavior is not defined in UML. Thus, the propositions have to be denoted manually.

Applying these translation rules to the UML model generates a formal model with multiple components and formulae. Until now, assemblies have not been considered. It can be done automatically by linking the call, parameter and return formulae in the code of the formal model. An interface provided by a port exposes methods to other components. Its methods must react on external input and provide feasible return values, determined through the formal implementation of the modeler. Hence, call and parameter formulae are determined through the requiring port. Methods not defined by an interface can not be "invoked". This increases the quality of the formal model as the modeler has to use provided interfaces to "invoke" necessary behavior. In the formal model return values have to be computed with respect to the method call and parameter formulae.

An example is given in Listing 1.1. The example is denoted in the System Analysis Modeling Language (SAML) [18]. A SAML model describes a set of finite state automata. These are executed in a synchronous parallel fashion. An automaton is described as a component that can contain state variables which are updated according to a set of transition rules. An example is shown in line 1. As we described before, the formal skeleton does not contain any states or transition rules, thus they are not shown here. Only formulae can be generated. An example formulae is shown in line 4, typically it is used as an abbreviation for typed expressions. Valid types are bool, integer, floats, and previously specified enumerations. In addition to components and formulae, traceability links are generated. These links are added as structured comments (cf. lines 1, 2 and 8). They are used by the VECS-IDE, to provide one-click tool support, for tracing formal elements to their original requirement (in the requirement specification IDE). Note that we use the abbreviations *ti* and *otc* for the components *trackside inductors* and *onboard transmitter coils*. Further, we abbreviate *inductPower* and *magnetTransited* with *iP* and *mT*.

Listing 1.1. Formal skeleton generated from the architecture given in Fig. 2.

```
component ti //<<trace>> PZB3
 //<<trace>> PZB5
 component inductors
  formula bool iP_call := null;
  formula int iP_freq := null;
 endcomponent
endcomponent
component indusi //<<trace>> PZB9
 //<<trace>> PZB5
 component bus
  formula bool mT_call := null;
  formula int mT_freq := null;
  formula bool mT_return := otc.bus.mT_return;
 endcomponent
endcomponent
component otc //<<trace>> PZB2
 //<<trace>> PZB5
 component bus
  formula bool mT_call := indusi.bus.mT_call;
  formula int mT_freq := indusi.bus.mT_freq;
  formula bool mT_return := null;
 endcomponent
 //<<trace>> PZB5
 component coils
  formula bool iP_call := ti.inductors.iP_call;
  formula int iP_freq := ti.inductors.iP_freq;
 endcomponent
endcomponent
```

Having the formal skeleton generated, the model needs to be manually completed by the modeler. The modeler has to systematically formalize all state requirements manually. If completed, the correct formal implementation shall be validated. Utilizing control data, a formal modeler is able to test single formal components through its behavioral triggers. Depending on overall project structure, the specification of the control data is either indirectly defined through requirements (e.g. it can be inherited from state requirements) or it can be generated through state-of-the-art algorithms (e.g. [3]). The corresponding assertions, representing the test checks, have to be formalized as temporal logic specifications and executed using a model checker.

3.5 Phase E: Integration Testing

The correct integration of every single formal unit is validated during this phase. Having the correct behavioral integration formally validated can help finding requirement specification errors by proving mathematically that the given behavior is underspecified or inconsistent. The modeler can identify erroneous behavior

and trace the corresponding transitions rules through the UML to their origin in the SRS with counterexamples calculated by the model checker.

In order to efficiently ensure the correct behavioral requirement formalization, we utilize the UML behavior descriptions. We defined sequences to rely on the architecture, thus, every message of a sequence is well-defined and invoked from one component to another. This is, because a message symbolizes a method call and/or return with a parameter configuration and a return value.

Under these assumptions, we can automatically formalize every sequence. We propose to generate a specification for every sequence such that it evaluates to true if the correct method call/return with the specific parameters is invoked in the correct order. Applying these semantics to the running example we can generate the branching-time logic specification shown in Listing 1.2.

Listing 1.2. Formal specification generated from Fig. 3.

```
1 SPEC EF(indusi.bus.mT_call = true & indusi.bus.mT_freq = 2000
      ↪ & EF(otc.bus.mT_return = false & EF(ti.inductors.iP_call
      ↪ = true & ti.inductors.iP_freq = 2000 & EF(indusi.bus.
      ↪ mT_call  = true & indusi.bus.mT_freq = 2000 & EF (otc.bus
      ↪ .mT_return = true))));
```

However, this specification does not generate a witness – a proof that the specified path exists in the state space. Thus, we propose to translate every specification into an acceptor automaton [26]. A witness can then be calculated by constructing a specification such that it evaluates to true if there is globally no path to the final valuation of the acceptor. The model checker then computes a counterexample providing a path such that the sequence is fulfilled which can be used during a safety assessment in order to argue about the feasibility of the formal model.

3.6 Phase F: System Testing

The goal of the system testing phase is to ensure that the formal implementation is tested for compliance. To do so, we utilize the safety requirements defined in the SRS. Safety requirements typically state unwanted hazardous behavior or states of system components or the whole system. With the help of the described architecture these requirements can be translated into temporal logic specifications. Further, state-of-the-art model checkers can be used to evaluate these specifications and objectify the assessment.

We use PZB8 to demonstrate the manual extraction of a safety specification from the running example. PZB8 states that a magnet transit triggers different safety relevant action, e.g. an emergency stop of the train. Thus, an overrun shall not be missed. Utilizing the architecture defined in phase B the following linear time logic formula was developed:

$$G((iP_{call} = true \wedge iP_{freq} = 2000) \Rightarrow (mT_{call} = true \wedge mT_{freq} = 2000 \wedge mT_{return} = true)) \ . \tag{9}$$

With respect to the formal architecture depicted in Listing 1.1 the formula can be translated into the specific formal language as follows:

Afterwards, a state-of-the-art model checker computes whether the specification holds or not. If so, the formalization of the requirements is correct. If not, the model checker generates a counterexample indicating where an error is stated. Then, the modeler has to manually locate the error and conclude different solutions. The manual translation of state requirements can be erroneous which has to be resolved. If the formal behavior is correct, the UML model can be faulty. Further, the SRS can be under specified or erroneous. Thus, the proposed approach can either be used to validate if a formal model violates the SRS, the UML model is not consistent or if the SRS is erroneous.

Listing 1.3. Final formal specification generated from Eq. 9.

```
1 //<<trace>> PZB8
2 SPEC G((ti.inductors.iP_call = true & ti.inductors.iP_freq =
      ↪ 2000) => (indusi.bus.mT_call = true & indusi.bus.mT_freq
      ↪ = 2000 & otc.bus.mT_return = true))
```

3.7 Phase G: Acceptance Testing

Acceptance testing aims to validate if the system meets its user requirements. Cimperman defines user acceptance tests as the validation if the system works for the user [10].

In phase A user requirements have been identified and have been modeled in UML as use cases. In order to objectify the test a use case is refined with multiple sequences. This is necessary, because a use case does not specify behavior in terms of the architecture. Using sequences refines a use case with the underlying architecture. Thus, we can use the same mechanism as described phase E (cf. Sect. 3.5) to verify the validity: acceptor automata. This eliminates a manual review of use cases.

4 Using the V-Model to Formalize the PZB

We demonstrate the proposed approach on a real world example from the railways domain: the *"Punktförmige Zugbeeinflussung"*. The PZB is a train protection system which uses three different kind of trackside and onboard magnets to detect speed limitations and signals in order to ensure the safety of the train with automatically breaking actions if speed limits or signals are ignored. The German Federal Railway Authority issued a SRS with over 770 text fragments. These fragments were indexed, inter-linked and categorized resulting in over 500 functional requirements. A domain expert identified 18 use cases from four user requirements refined through 13 sequence diagrams.

An UML model with ten components, 19 ports and eleven interfaces has been derived from 45 glossary and 35 architecture requirements. Further, we identified 62 methods with 17 parameters where 32 were non-void from 129 property requirements. 23 sequences have been exemplary derived from 38 communication requirements.

We developed elaborate tool support integrated in the verification environment VECS [17]. We generated a formal skeleton in SAML [18] with 29 formal components with 226 different formulae. 73 state requirements have been formalized into 76 state variables and 164 transition rules. Further we added 129 components, 228 formulae for readability purposes. In order to test the behavior of single formal units, a variety of individual tests have been developed.

Further, 23 acceptor automata have been generated with 23 state variables and 1037 update rules. Using the IC3 approach [28] of NuXMV [6] a variety of errors have been found. Typical errors found were incorrect value assignments across different components. With the proposed approach these mistakes have been found systematically. At the end of the integration testing phase, each acceptor automaton provided a witness for its corresponding sequence. These witnesses were used to demonstrate feasibility of the formal model and as a proof for the correctness of the process method.

Further, 30 safety specifications have been checked with the k-liveness approach [9] of NuXMV. Especially underspecified behavior in the SRS has been found during this phase. For example, stopping the train directly on an trackside inductor has not been mentioned in the SRS. Using the proposed approach, a counterexample has been generated describing the unintended behavior in detail. With the help of the requirement links the intended behavior was traced to the SRS, where the requirements had been extended to cover this behavior. Afterwards, the faulty behavior definition has been fixed in the formal model.

In order to validate if the developed formal model does fulfill all user stories, the refined use cases have been formalized into additional acceptor automata. Through their formalization and executing using the IC3 approach, we proved the correct implementation of all required use cases.

Following the proposed approach for building a "correct" formal model improved the quality and feasibility of the PZB model. Architecture and behavior for the model had been specified more systematically. Further, traceability of requirements had be achieved by automatically adding structured comments as requirements links. These links helped identifying faulty behavior and getting deeper insight the formal model. Generating automated tests cases also improves the quality of the model. This is, because erroneous behavior had be found much more systematically.

5 Conclusion and Further Work

In this paper, we proposed a process in order to formalize even large system requirement specifications while preserving traceability through all stages. We apply the well known V-model to ensure the correct transformation of requirements. Doing so, we ensure that a formal model is developed which is mathematically proven consistent to a set of requirements.

This was done by categorizing requirement fragments and transforming them into UML. It ensures that domain experts, not familiar with formal verification techniques but with a set of UML artifacts, can review the generated formal verification results and trace them to their origins. Further, we proposed a method

to transform an architecture into a formal representation without the need of generating state variables. We showed how to transform partial UML behavior to ensure a correct a formal model. By utilizing state-of-the-art model checkers manual reviewing of correct formal behavior specification was eliminated. Further, we have shown how to validate mathematically that the formal model fulfills every use case specified in the requirements.

Finally, we successfully applied the proposed process on a real world system requirement specification from the railway domain. We were capable of finding and correcting specification errors in all three formalization stages: we found inconsistencies in the informal requirements, manual transformation errors in UML artifacts and faulty behavior in the formal model.

We plan to apply and extend the proposed methodology in an even larger case study: In cooperation with the German Federal Railway Authority, we will formalize a subset of the European Train Control System requirement specification in order to prove its functional safety. We will develop new mechanisms to support a broader subset of UML artifacts like combined fragments, activity diagrams and state charts. Further, we will extend the process in order to work in cooperation with state-of-the-art code generators and develop new methodologies to ensure the consistency of the generated code and the developed formal model.

Acknowledgment. The work presented in this paper is funded by the German Ministry of Education and Science (BMBF) in the VIP-MoBaSA project (project-Nr. 16V0360).

References

1. Boehm, B.W.: Verifying and validating software requirements and design specifications. IEEE Softw. **1**, 75–88 (1984)
2. Bose, P.: Automated translation of UML models of architectures for verification and simulation using spin. In: ASE, pp. 102–109. IEEE (1999)
3. Brill, M., Buschermöhle, R., Damm, W., Klose, J., Westphal, B., Wittke, H.: Formal verification of LSCs in the development process. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 494–516. Springer, Heidelberg (2004)
4. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In: SVERTS, pp. 1–20. Citeseer (2004)
5. Carnevali, L., Grassi, L., Vicario, E.: A tailored V-model exploiting the theory of preemptive time petri nets. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 87–100. Springer, Heidelberg (2008)
6. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Heidelberg (2014)
7. CENELEC: 50129. Railway Applications: Safety Related Electronic Systems for Signalling (1998)
8. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: From informal requirements to property-driven formal validation. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 166–181. Springer, Heidelberg (2009)

9. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
10. Cimperman, R.: UAT Defined: A Guide to Practical User Acceptance Testing. Addison-Wesley Professional, Upper Saddle River (2006)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. David, A., Möller, M.O., Yi, W.: Formal verification of UML statecharts with real-time extensions. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 218–232. Springer, Heidelberg (2002)
13. Emerson, E.A., Halpern, J.Y.: Sometimes and not never revisited: on branching versus linear time temporal logic. J. ACM **33**, 151–178 (1986)
14. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.-P.: 29 new unclarities in the semantics of UML 2.0 state machines. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 52–65. Springer, Heidelberg (2005)
15. Filax, M., Gonschorek, T., Lipaczewski, M., Ortmeier, F.: On traceability of informal specifications for model-based verification. In: IMBSA 2014: Short & Tutorial Proceedings, pp. 11–18 (2014)
16. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time UML designs. In: ESEC/FSE, pp. 38–47 (2003)
17. Gonschorek, T., Filax, M., Lipaczewski, M., Ortmeier, F.: VECS - verification enviroment for critical systems - tool supported formal modeling an verification. In: IMBSA 2014: Short & Tutorial Proceedings, pp. 63–64 (2014)
18. Güdemann, M.: Qualitative and quantitative formal model-based safety analysis. Ph.D. thesis, Otto-von-Guericke-Universität Magdeburg (2011)
19. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: formal verification of object-oriented models. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 187–206. Springer, Heidelberg (2004)
20. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker. FAC **11**, 637–664 (1999)
21. Lilius, J., Paltor, I.P.: Formalising UML state machines for model checking. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 430–444. Springer, Heidelberg (1999)
22. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: ASE, pp. 255–258. IEEE (1999)
23. OMG: OMG Unified Modeling Language (OMG UML), Superstructure. Object Management Group (2011)
24. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. TOSEM **15**, 92–122 (2006)
25. Tang, W., Ning, B., Xu, T., Zhao, L.H.: Scenario-based modeling and verification of system requirement specification for the european train control system. In: Computers in Railways XII, pp. 759–770 (2010)
26. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 322–331. IEEE (1986)
27. Varró, A.: A formal semantics of UML statecharts by model transition systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 378–392. Springer, Heidelberg (2002)
28. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and sat-based reachability in hardware model checking. In: FMCAD, pp. 173–181. IEEE (2012)