Urban Rail Transit (2022) 8:217–245
https://doi.org/10.1007/s40864-022-00177-8

http://www.urt.cn/

ORIGINAL RESEARCH PAPERS

Formal Modeling and Verification of the Functionality of Electronic Urban Railway Control Systems Through a Case Study

Gábor Lukács[1] · **Tamás Bartha**[1,2]

Received: 4 October 2021 / Revised: 26 August 2022 / Accepted: 16 September 2022 / Published online: 8 November 2022

Abstract This paper presents a formal model-based methodology to support railway engineers in the design of safe electronic urban railway control systems. The purpose of our research is to overcome the deficiencies of existing traditional design methodologies, namely the incompleteness and the potential presence of contradictions in the system specification resulting from non-formal development techniques. We illustrate the application of the methodology via a case study of a tram-road level crossing protection system. It was chosen partly because it has a simple architecture and a small number of elements, thus it fits the scope limitations of this article. At the same time, it is suitable for presenting all essential features of our methodology. The proposed solution provides a specification/verification environment that facilitates the construction of correct, complete, consistent, and verifiable functional specifications during the development, while hiding all the formal method-related details from the railway engineers writing the specifications. Using this formal model-based methodology, a high-quality functional specification can be achieved, which is guaranteed to be more exhaustive and will contain fewer errors than traditional development.

✉ Gábor Lukács
lukacs.gabor@edu.bme.hu

Tamás Bartha
bartha.tamas@kjk.bme.hu

[1] Department of Control for Transportation and Vehicle Systems, Budapest University of Technology and Economics, Budapest, Hungary

[2] Systems and Control Laboratory, Institute for Computer Science and Control (SZTAKI), Budapest, Hungary

Communicated by Bo Ai.

Keywords Requirement specifications · Statechart · Model checking · Safety critical · Urban railway control

1 Introduction

In the current practice of system development, the rigor of requirements and the level of expected quality and safety both are required to be proportional to the risk posed by the potential faulty behavior of the system (more precisely, the necessary risk reduction). One way to meet the high expectations of safety-critical applications is the use of formal methods. Rail transport is a traditionally safety-critical engineering field; thus, it is no surprise that formal methods are also prescribed by railway-related standards (e.g., EN 50129 [1], EN 50128 [2]). These standards classify formal methods as "highly recommended" at Safety Integrity Level (SIL) 3 and 4 (see [1] Annex E, Table E.7, point 4).

Functional safety aims at preventing the unacceptable risk of physical injury or damage to human health caused by the erroneous operation of a system, with the proper implementation of one or more automatic protection functions (also called safety functions). The definition (specification) of the correct functionality of the system is a core concept of safety, which is the task and responsibility of the engineers working in the given application domain. However, domain engineers are reluctant to use formal methods during the system design/development activities, because even though these are claimed to help avoid specification errors, their application often requires significant additional expertise because they are abstract, computer science-based methods. The use of formal methods is much more common in the software design/development activities of the projects, because software

engineers (computer scientists, programmers) receive an adequate level of education to be able to readily use these techniques. Railway engineers can only acquire these skills after years of work.

At the same time, domain engineers are at the heart of the information flow of the development. They help to transform the different needs of the stakeholders into system design and implementation, and they also give feedback about the detected errors, limitations, and deviations to the stakeholders during the life cycle. This two-way activity can be complicated and, in many cases, can lead to many conflicts and many iterations during the development. In practice, user requirement specifications very rarely meet the criteria of a "good" requirement specifications. The work of domain engineers is to overcome these difficulties, which require simple, fast, efficient, and easy-to-apply solutions.

The use of formal modeling [3] is gaining popularity in the development of safety-critical transport applications. This technique is also preferred by railway engineers due to the advantages and power of model-based systems engineering [4]. Formal modeling provides an opportunity to precisely specify the functionality of systems using mathematical/logical rules. The research described in this article aims to present a case study demonstrating the practical utilization of a new methodology that supports railway engineers in the construction and verification of formal specifications. The proposed specification-verification environment aims to decrease the need for mathematical and computer-science background/knowledge at the system development level. Our framework integrates a set of well-known and widely used methods, techniques, and tools such as object-oriented formalisms (e.g., the Unified Modeling Language [5]), model-driven development [6], and model checking [7]. The application of the methodology supports the construction of correct, complete, consistent, and verifiable functional specifications. As a result, this approach leads to a significant improvement in quality and distributes the development costs more evenly among the related life-cycle phases.

The purpose of the following subsections is to briefly introduce the practice of the railway control systems development, including the widely applied methodologies, techniques, and principles, and their relation to our proposed methodology.

1.1 Life-Cycle Management

There are many different approaches to life-cycle management in the industry, such as for safety-critical systems [8, 9] or [10]. A possible life-cycle model for railway signaling development is the popular V-model [11]. Our research focuses on the process from the requirements specification phase to the design phase.

Based on practical experiences, the specification and design phases are alternating activities of the development life cycle (see Fig. 1). During the development, the implementation of a system is achieved by reducing the level of abstraction and increasing the formalization step by step (also called a step-by-step refinement). A common experience is that performing specification and design activities simultaneously in engineering practice increases the likelihood of systematic errors. The full implementation of a system usually cannot be immediately completed after a single specification and design step (although it may be possible in the case of trivial systems). This gradual refinement process was taken into account during the construction of the proposed methodology because it has a significant impact on the effectiveness related to the reduction of systematic errors.

1.2 Requirements Engineering and Management

In the railway development life cycle [11], tasks related to requirements are "limited" to two life-cycle phases (namely the "specification of system requirements" and "architecture and apportionment of system requirements") [11], and these will be validated during the activities on the right side (integration and validation) of the V-model. However, recent development in project management and systems engineering emphasizes that the requirement-related activities must not be limited to these phases alone. Requirements engineering (RE) means a set of activities to explore, evaluate and document the objectives, capabilities, constraints, and assumptions of a system to be developed [12]. Requirements management (RM) [13]—as a life-cycle-comprehensive technique—includes such processes as the following up, prioritization, verification, maintenance, etc. of requirements during the whole life cycle. Many widely used methodologies and methods are available nowadays to support RE and RM by tools, toolkits, workbenches, or frameworks (e.g., [14–16]).

In addition to the RE and RM, the examination of the railway domain requirement sources (RS) also influenced the construction of our proposed methodology. For the development of an interlocking system, the requirements can come from many sources. The various description techniques used in RS are at different levels of abstraction and formality. The requirements coming from RS are one of the inputs to the development. The processing of the RS is usually performed during the "specification of system requirements" and "architecture and apportionment of system requirements" phases of the life cycle [11]. As a result, different types of requirement specifications are

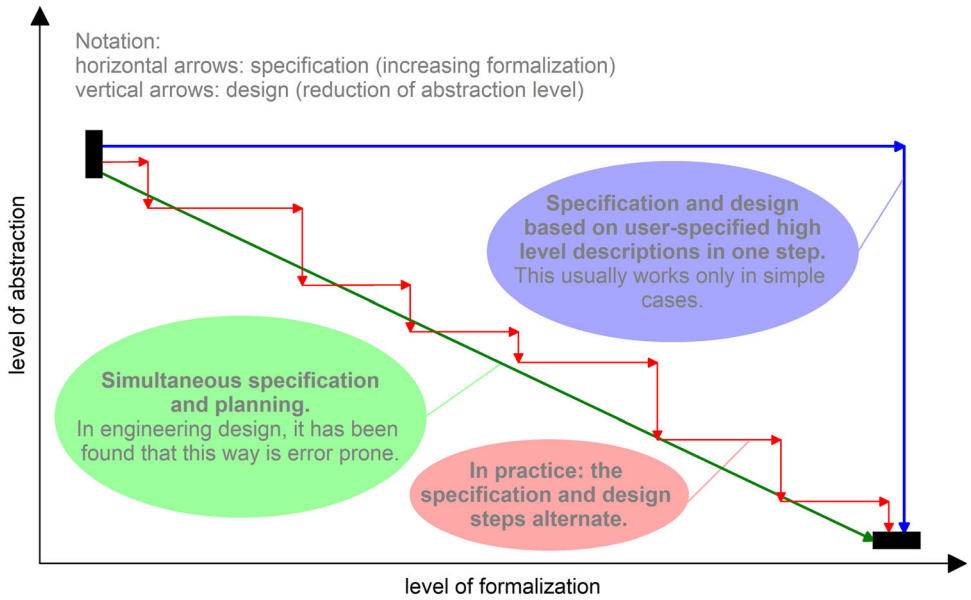

Fig. 1 Relation between abstraction and formalization

created (e.g., safety Requirements specification, SRS; functional requirements specification, FRS).

The main source of requirements are the stakeholder needs, but there are several other sources, too. Additional RS can be divided into four main groups (in Europe): legislation of the European Union (EU), national legislation, standards, and guides by authorities/professional organizations. For example, in the case of an axle counter, RS include a standard [17], a technical specification for interoperability (TSI) [18], and a national regulation [19]. In practice, several RS can be identified as relevant to the development of a system, which may lead to thousands of requirements. Clarifying such a vast requirement set is difficult due to its size, interdependence, and different levels of abstraction and formality. This set of requirements exhibits all sorts of problems: the lack of correctness, completeness, consistency, verifiability, etc. During the first phases of the life cycle [11], developers aim to analyze this set of requirements in order to significantly reduce these problems, and then create systems requirement specifications from it.

1.3 Model-Based Systems Engineering, Model-Driven Development

Model-based systems engineering (MBSE) [20] and model-driven development (MDD) [21] are popular approaches in the current engineering practice. There are several tools (e.g., MATLAB [22] and Simulink [23], etc.) to support MBSE or MDD. Engineers are generally familiar with the tools available for modeling, simulation, or analysis in their respective fields, so when they encounter a problem, they immediately examine it using modeling.

MBSE is a formalized application of modeling to support the life-cycle activities during the development [24], which intends to bring together the model-centric approaches of various engineering disciplines (e.g., electrical, software). MDD is a philosophy of software development that focuses on high-level models [25]. For both approaches, models are the most important artifact, similar to our proposed methodology.

A model is a simplified representation of a system at some particular point in time or space, intended to promote understanding of the real system (Bellinger 2004, [26]). A model can have many appearances, such as textual, mathematical, graphical, or mixed. In railway engineering practice, the graphical form is the most popular; however, sometimes mixed solutions may also be required. Many techniques and tools related to MBSE are applied in the railway field to support modeling, for example, Unified Modeling Language (UML) [27] or Systems Modeling Language (SysML) [28], among others.

Finally, modeling is one of the recommendations of the EN 50128 standard [2] dealing with the development of software for railway control and protection systems. A set of recommended modeling techniques are listed in Table A.17 of the standard [1], including also formal methods.

1.4 Formal Methods

Formal methods are precise modeling and analysis techniques that are based on discrete mathematics and mathematical logic, in particular. These techniques are used primarily in the area of computer science [29]. Formal

methods are useful in system development, including software and hardware development, as well [30]. The semantics and syntax of the formal models are well specified, clear, and complete. They enforce the engineer to think deeply and systematically about the problem, thus reducing significantly the ambiguity and incompleteness of specifications.

Nowadays, the engineering practice of railway interlocking systems is characterized by non-formal and semi-formal (consisting of textual and ad hoc marking systems) specifications written by railway engineers. The cooperation between the participants in the development process often causes misunderstandings, uncertainty, or omissions [31]. Using formal models during the development process can decrease the probability of these inadequacies [32]. Formal methods support the rigorous specification, planning, verification, and modeling of complex systems [33]. They also make the identification of errors possible in the early lifecycle phases. The executable models can be tested early in their creation. When a qualified code generator (e.g., QGen [34]) generates the code, then the proof of correctness of the models will remain valid as a correctness proof for the code as well, in the scope of the verified properties.

The two main approaches to performing formal verification are model checking [35] and theorem proving [36]. Several practical applications are already known in the field of railways for both techniques (e.g. [37–42]). Using theorem proving, the system attributes and behavior, the environment (axioms), the required properties and the domain-specific engineering knowledge is represented as a collection of formulas. The purpose is to investigate whether the given formulas form a consistent set or there is a contradiction. On the other hand, in model checking the system, the environment, the required properties (logic formulas) and the domain knowledge are described by models. During model checking, the (typically symbolic) combined state space of the models is generated, and the truth value of the given formulas is checked in (potentially the whole) state space. Because of the availability of the state space, modeling and model checking can provide a so-called counterexample in the case of property violations, which is beneficial for the detection and elimination of errors. Thus, these techniques are closer to system engineering practice than theorem proving [43]. Therefore, we chose model checking for the proposed methodology. Particular attention has been paid to hide the specifics of formal methods from railway engineers as much as possible.

2 Related Work

In this section, we give an overview of the past research results and case studies related to the purpose described in this paper. These differ from the proposed methodology in that they aim at a different application field, focus on diverse aspects, or employ other tools to handle the specification and verification of critical systems.

Kunnappilly et al. [44] describe a framework based on the UML and the Uppaal tool (a formal modeling and model checking workbench). To introduce the operation of their framework, the authors describe two case studies: a mission-critical 5G-assisted robot surgery e-health application and a less critical video streaming application. The conclusion of the paper is that by combining the benefits of user-friendly UML and Uppaal, they created an effective solution to address the issue by enabling both modeling and formal verification already at the design phase. We use the same tools the authors used (e.g., UML statechart and Uppaal) in our proposed methodology.

In the field of medical systems, the paper by Chunhui et al. [45] presents a framework that facilitates the participation of medical professionals in modeling, formal verification, and root cause identification of safety-critical failures. The proposed environment allows one to perform these activities at two levels of abstraction: at the model and code levels. Clinical validation is also part of the framework. The strategy of the authors to implement this framework was to utilize existing tools designed for validation and verification. They used statechart diagrams to describe the problems of the medical domain. These representations can be transformed automatically to verifiable formal models by the framework, so it is possible to verify safety properties formally. After verification by medical professionals, these models become the basis of executable code generation. The framework transforms safety properties specified at the model level into run-time code monitors to ensure their fulfillment. The authors used a cardiac arrest treatment system as a case study. Their paper is in the medical domain, but it applies the same processes and tools (e.g. Yakindu, Uppaal) that we also use in our proposed methodology.

The framework introduced in the paper [46] presents a formal verification approach to the real-time extension of UML statecharts. The authors specify one subset of the rich UML statechart abstraction extended with real-time constructs (clocks, timed guards, and invariants). The formalism they developed is called hierarchical timed automata (HTA). Our proposed methodology includes a similar suggestion for the subset of the UML state charts. However, we did not extend the defined subset. David et al. [46] overview a possible translation of one HTA to a network of flat timed automata. The set of these timed automata are one of the inputs of the real-time model checking tool Uppaal. Paper [46] gives a report on an XML-based implementation of this translation using the well-known pacemaker case study.

The research goal of the thesis [47] was to analyze the applicability of formal methods in the domain of industrial control systems and propose a specification-verification environment. The main challenges of the chosen domain were performance and usability. The author proposes the application of model checking to support the software development of industrial control systems. The described platform helps domain engineers by hiding the formal details, so they do not need to acquire the special mathematical/logical background needed for formal methods. The case studies of this work come from the practice of the European Organization for Nuclear Research (CERN). In connection with our research, the following results of the thesis are relevant: formal specification for programmable logic controller (PLC) modules and model checking of critical PLC programs. Our research goal is very close to those of this thesis, with differences in the domain and abstraction level.

The Gamma Statechart Composition Framework (GSCF) is an integrated toolset to facilitate the design, verification and validation, and code generation for component-based reactive systems [48]. GSCF provides a modeling language and framework for the hierarchical decomposition of statechart components in an object-oriented way. The framework is integrated with the third-party Yakindu Statechart Tools modeling tool and the Uppaal model checker to provide formal verification of the constructed models. GSCF automatically generates an implementation of an individual component. This environment also allows back-annotation and test generation. The main difference between the GSCF and our proposed framework is that the former targets software engineers as end-users, whereas we target the railway signaling engineers with our specification-verification environment. The objectives, processes, and tools used are similar for both frameworks. However, our methodology does not deal with code generation, because it is not intended to cover that level of abstraction. Both frameworks hide the inherent complexity of using formal methods by offering a high-level user interface. Case studies related to GSCF are presented in [49] from the cyber-physical system domain.

The paper by Jiang et al. [50] describes a methodology related to an extension of the MDD approach. Their MDD process starts from a state-flow (a variant of statecharts) model, followed by simulation, validation, and code generation. Simulink is a popular MDD tool used in the development of industrial software systems. This MDD process has been supplemented with an extended verification environment. The extension consists of a translator and a run-time verifier. The purpose of the translator is to generate an Uppaal model from the Simulink state-flow model. Formal model checking can be performed on the Uppaal model using the property specification (transformed requirements) of the designed system. The verified model can be used for code generation by Simulink, and the run-time verifier supports monitoring the generated software system in operation. The paper also includes a case study related to a train communication control system. The approach of [50] differs from our framework in the targeted end-users, in the initial model (state-flow vs. statechart) and the tool used (Simulink vs. Yakindu Statechart Tools). Both frameworks use Uppaal for formal verification of the designed systems. The last difference is code generation, but as stated earlier, we do not intend to support implementation.

The paper by Yul et al. [51] presents an application of formal model checking-based safety verification of a railway interlocking system. The proposed model checking technique is implemented on a timed automaton of the considered interlocking system. The safety behavioral specification is expressed as a set of computation tree logic formulas. The Uppaal model checker is used to perform the model checking. The distinction between this study and our research is that we perform modeling and model checking at the component design level in this paper, while their study [51] does it at the system level.

Wang et al. [52] also present the simulation results of a performed verification. They describe a general interlocking system function modeling by multistage refinements using the Event-B language. Their methodology can help developers reduce potential design flaws and identify defects in the interlocking data in the early development stage of the life cycle. The validated model can be the foundation for the implementation. In their work [52], the domain (railway interlocking) is the same as the scope of our research. However, the difference is that it proposes a complementary application of the model checking and theorem proving.

The publications described above all have in common that they try to support engineers of a certain domain to incorporate formal methods into their development activities. To enable this, they select appropriate high-level semi-formal descriptions that domain engineers can pick up with little effort; they expect the engineers to prepare models using the selected formalisms, and then automate the rest of the verification process. But they all differ in the way they achieve this. Therein lies the contribution of our paper as well. Although our approach described in Sect. 3 is based on existing tools, and the mathematical basis behind the formalisms and tools used are also well established and not new, the way they are selected and integrated is specially tailored for the engineering of safety-critical railway systems. The novelty lies in the following: (1) it enables the selection and integration of the appropriate high-level semi-formal and low-level formal description forms and tools into a toolchain that fits the railway field; (2) it

illustrates the transformation from the semi-formal to formal models (this transformation can be partly automated, but the transformation rules are not in the scope of in this paper); and (3) the proposed approach was created specifically for the railway engineering domain (also taking systems engineering best practices into account), where these techniques are not yet widely applied.

The similarity of the publications [48–50] means that they use a set of tools and techniques widely used in engineering practice. However, due to their complexity, these tools require an extremely high level of professional knowledge and background. In many cases, in-depth knowledge of a particular field of science is not enough for their proper application, so they require interdisciplinary knowledge (e.g., knowledge of mathematics, information, and engineering). In the methodology proposed by us in Sect. 3, we set the goal of reducing this need for diverse knowledge as much as possible for railway engineers. We plan to give them an easy-to-learn, fast and cost-effective method that can be used in practice, and which gives a result in which the viewpoint of quality meets the expectations of this domain.

3 Methodology

In this section, we describe the principles of our proposed methodology. Hereinafter, it will be referred to as FMBRSE (Formal Model-Based Railway Safety Engineering) methodology.

The purpose of the FMBRSE is to support railway engineers in the application of formal specification and verification in the development of a safety-critical railway system. The result of applying the methodology in a development task is a formally verified, validated functional model of the given railway safety system. The FMBRSE process is shown in Fig. 2.

The inputs of the process are the requirements described by various stakeholders. These requirements are at different levels of abstraction and formality (see Sect. 1.1). The requirement sources that came from stakeholders are presented in Sect. 1.2.

Steps from (1) to (3) in Fig. 2 are well known to railway engineers because they are prescribed by the standard [11]. The FMBRSE methodology provides a specification environment for designing each component of the system in terms of its functionality. This specification environment builds on four pillars: requirements, interfaces, configuration, and behavior. We recommend structured natural language [53] for specifying requirements, a subset of the UML component diagram [27] for interface specifications, a specific tabular method for defining parameters, and a subset of the UML state machines [27] for describing the behavior/functionality of each component. The subsets from the UML component and statechart diagrams have been designed with the following considerations in mind: keeping the syntax of UML, maintaining compatibility with existing tools, and being easy to learn for railway engineers. The methods above are essential parts of the specification environment of the FMBRSE methodology. Of course, if needed/useful, the proposed specification environment can be supplemented with other techniques. However, their integration with the existing FMBRSE methodology must be verified and validated. The reader can find the details of the defined UML subset in [54].

The FMBRSE methodology also provides a formal verification environment using model checking (see step (6) in Fig. 2). Model checking [55] is a method based on discrete mathematics to answer the following question: "Does the model satisfy a set of requirements, or if there is a requirement violation, then what sequence of events can lead to this situation?" Steps (4) and (5) in Fig. 2 aim to create the inputs of model checking—i.e., computation tree logic (CTL) formulas [56] from the requirements, and formal models from the functional specification of the system—the purpose of which is to verify the functionality of the designed system before implementation.

CTL formulas are constructed by applying a rule-oriented technique to transform the natural language descriptions. The behavior (timed automata) of the developed system is generated from the statecharts in an automated way. In certain cases, it is necessary to model the environment of the system and/or the input function (the generator of the input value sequences fed to the system inputs), in order to be able to perform model checking. These are not part of the automated generation of a formal model because they depend too much on the designed system. These should be designed individually with the modeling goals in mind. In summary, a formal model of a component—when the input function does exist—is made up of three elements: the automata representing the functionality, the input (value generator) function, and the necessary elements modeling the environment (e.g., modeling of the function call sequences, modeling of the timing of actions).

The result of the model checking can be of three kinds. If the requirements are fulfilled, then the model is verified correct. When a requirement violation is found in some state(s), then model checking provides a counterexample. This counterexample characterizes an incorrect behavior (the path/steps leading to the requirement violation) of the model. The third possibility, "non termination", means that either the verification fails due to insufficient memory, or the verification is shut down prematurely because it takes too much time. The possible outputs of model checking are summarized in the process shown in Fig. 3.

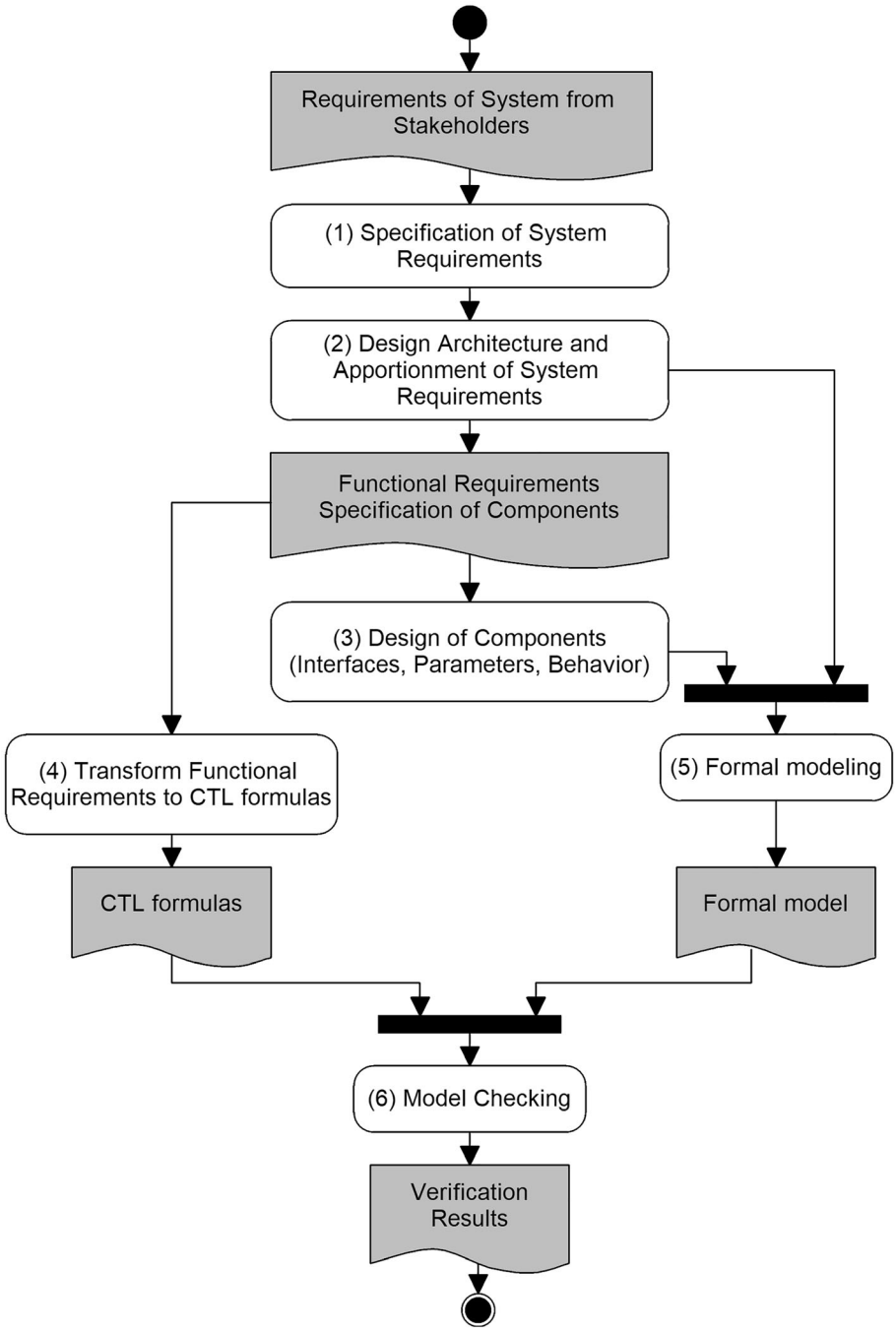

Fig. 2 Process of FMBRSE methodology

The practical implementation of model checking within FMBRSE is based on an existing, widely known framework: UPPAAL [57]. UPPAAL is a tool for modeling and verification of real-time systems developed by Uppsala University and Aalborg University.

UPPAAL models systems as a network of timed automata. A timed automaton is a finite-state machine supplemented with clock variables. The network of timed automata modeled in UPPAAL consists of concurrently operating processes that can interact with each other. These processes are described with graphical process templates, which also have local declarations in textual form. A process template basically consists of two main elements: locations (states) and transitions (state changes). For each process, one location must be appointed as the initial state. It is possible to make a transition dependent on a guard (logical expression formed from variables). A transition can also perform an action during the state change (that e.g. modifies the value of variables); this is called an update. Two additional elements can be defined for a transition:

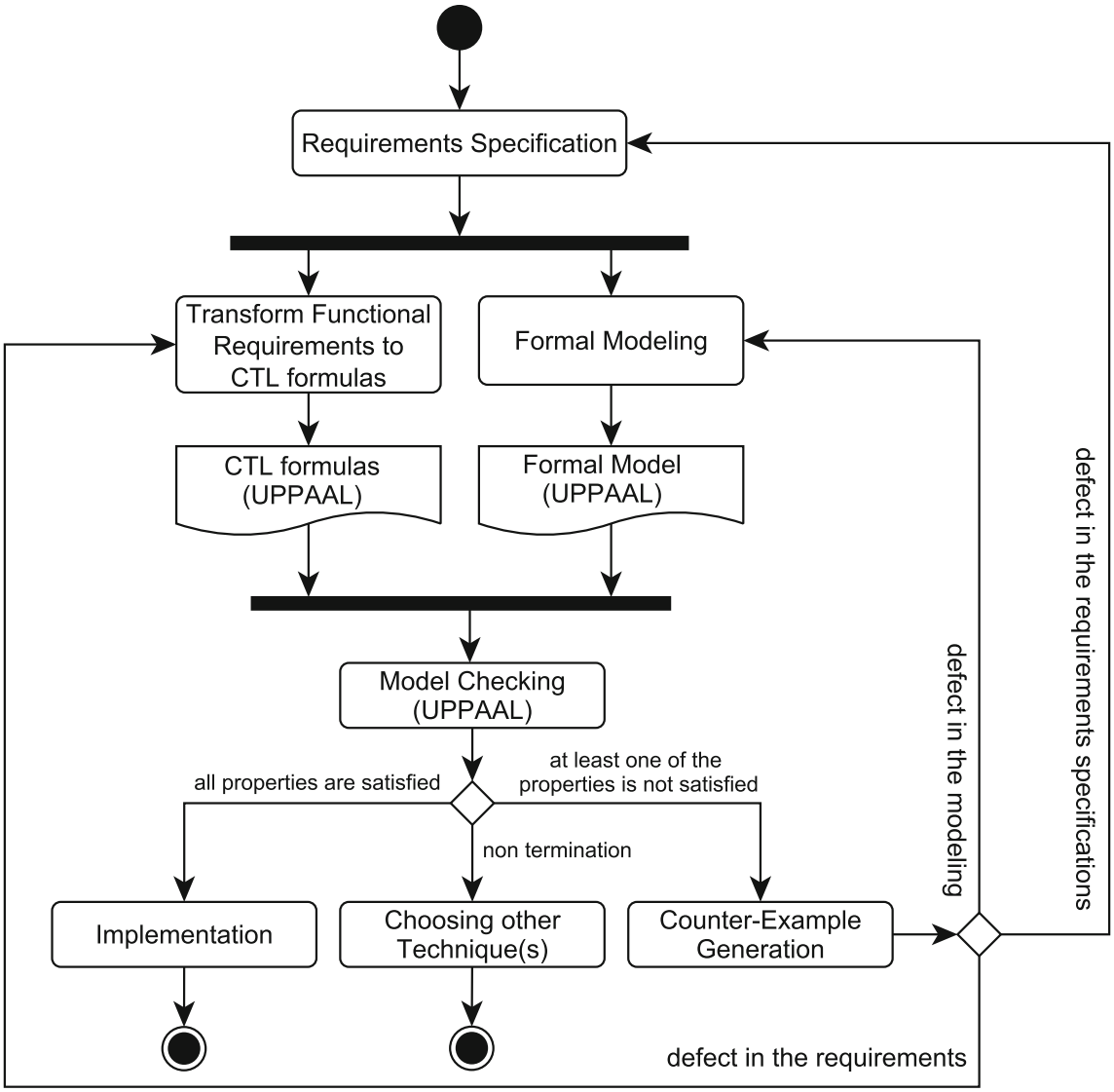

Fig. 3 Model checking (based on [54])

selection and synchronization. The selection non-deterministically binds a given variable to a value in a predefined range. The synchronization implements the interaction between processes in Uppaal. There are two types of synchronization: single-channel synchronization and broadcast synchronization. Both need a declared channel through which the processes can interact with each other. When two processes synchronize on a single channel, both transitions connected by the channel fire together; i.e., the current location of both processes changes simultaneously. Broadcast channels allow one-to many synchronization; i.e., the sender can synchronize with several receivers at the same time.

The timing aspect of timed automata can be modeled by means of so-called clock variables. A clock variable represents an abstract clock with a continuously increasing value. Clocks cannot be stopped, and their value cannot be read, but they can be reset. In summary:

- The value of clock variables increases monotonically unless we explicitly reset them.
- Their value can be used in logical expressions, i.e., the guards of transitions and the invariants of locations.
- Any amount of time (but only finite) or even zero time can be spent in one state. (Invariants and special locations: committed and urgent locations can modify this behavior.)

In addition to modeling (and simulation), Uppaal also includes offers the ability to verify model verifications. The built-in model checking engine evaluates the requirements given as temporal logic expressions over the state space, and reports whether or not they are fulfilled. We can

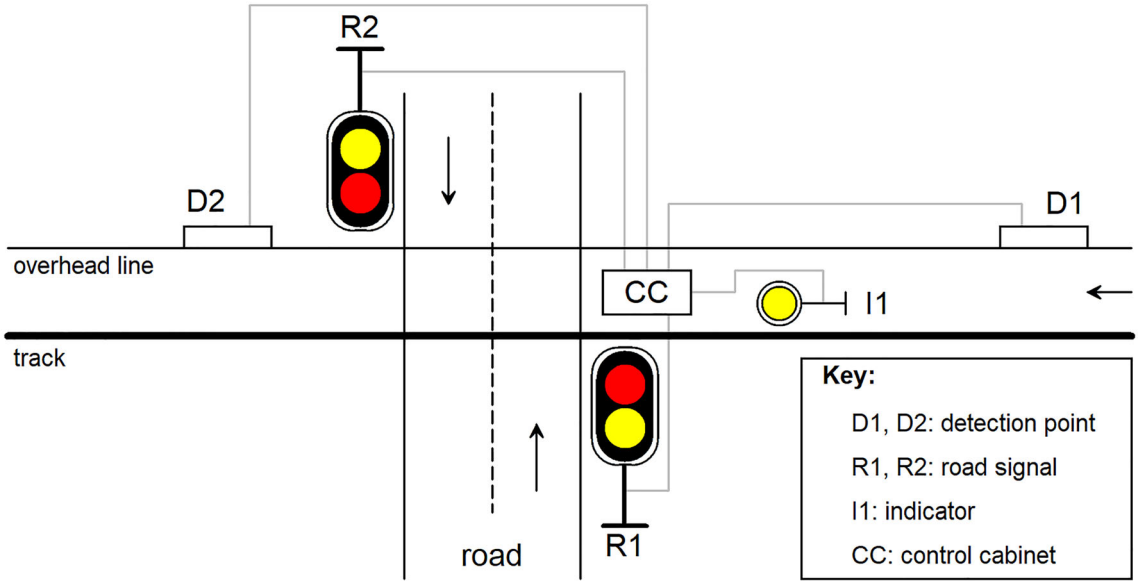

Fig. 4 Level crossing protection system for trams

formalize all our system-related requirements and examine their fulfillment. As a formalism of the requirements, UPPAAL uses a subset of CTL temporal logic language for the requirements. With the help of the CTL, we can examine changes in certain conditions over time. The structure of each CTL expression in UPPAAL is as follows: "temporal operator" and "logical expression". Logical expressions can contain e.g. conditions for variables and clocks variables, combined with comparison and logical operators, e.g. q>0 and expressions related to states, e.g. "p1.working". The temporal CTL operator can be one of the following in UPPAAL (p and q are atomic statements) [57]:

- A$\Box p$: property p is satisfied on every path in every state (invariantly),
- A$\Diamond p$: property p is satisfied on every path in some state of the path (eventually),
- E$\Box p$: there exists a path whose every state satisfies the property p (potentially always),
- E$\Diamond p$: there exists a path whose some state satisfies the property p (possibly),
- $p \rightsquigarrow q$: whenever p holds eventually q will hold as well (leads to). The leads to property $p \rightsquigarrow q$ can also be expressed as A$\Box$ ($p \Rightarrow$ A$\Diamond$ q).

After successfully performing the model checking process (with no requirement violations) described above (see Fig. 2), railway engineers can pass on verified and high-quality detailed requirements and functional specifications to software engineers working at the implementation level. Nonetheless, as mentioned earlier, the implementation phase is beyond the scope of the FMBRSE methodology and is not covered in this paper.

The FMBRSE framework can be used at both the system and component levels. This article discusses the component-level application (see Sects. 4 and 5) using a case study. Another case study illustration of the system-level application is described in the paper [58]. However, the application of FMBRSE at the system level may encounter a well-known difficulty: state space explosion. The reason for this is that the state space of system-level models is much more complex than that of the component-level models. A potential approach for dealing with this is compositional verification [59]. This technique is based on a divide-and-conquer approach to infer the global properties of complex systems from the properties of their components.

An overall characterization of the proposed methodology follows. In terms of features, FMBRSE is a plan-driven/predictive [60] methodology, which means that planning the future to the reachable goal is as detailed as possible. It is a hybrid methodology that combines several existing approaches, the most important of which are the structured and object-oriented design philosophies [61]. FMBRSE applies relevant results from model-driven development and function-based development (FBD) [62]. Its design principle is top-down, with hierarchical decomposition [63] and iterations. The life-cycle ([11] V-model) coverage by FMBRSE is partial. It only covers the design phase, namely, from requirements to the detailed design phase. Overall, FMBRSE integrates widely used existing techniques and tools to facilitate the work related to system development performed by railway engineers.

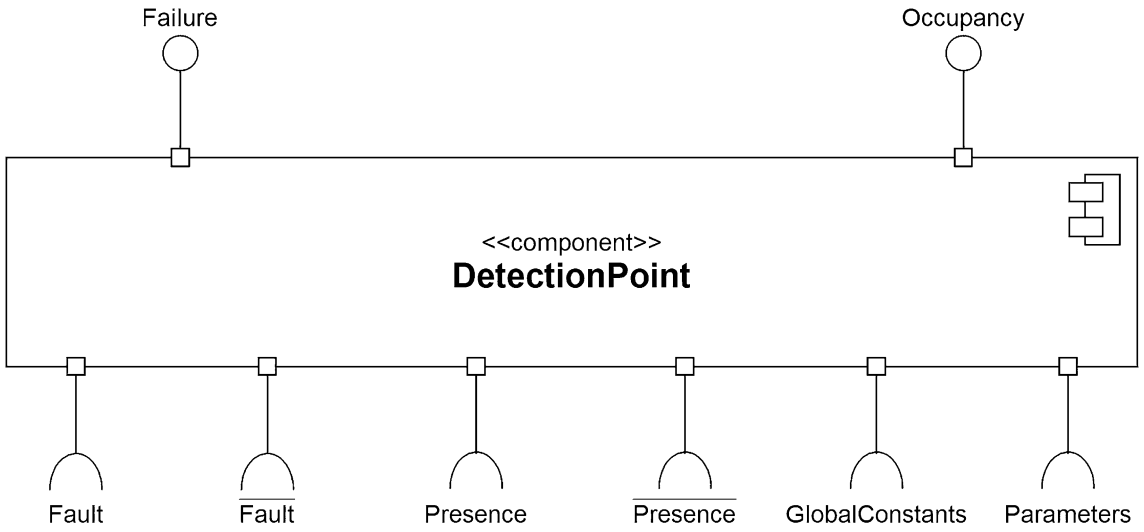

Fig. 5 Interfaces of the detection point

Table 1 Interface specification of the detection point

Identifier	Type	Codomain	Initial value	Brief description
Fault	Input	[non-faulty | faulty]	Faulty	Fault input
/Fault	Input	[non-faulty | faulty]	Faulty	Negated Fault input
Presence	Input	[free | occupied]	Occupied	Presence input
/Presence	Input	[free | occupied]	Occupied	Negated Presence input
Globals	Input	–	–	See Table 2
Parameters	Input	–	–	See Table 3
Failure	Output	[non-faulty | faulty]	Faulty	Failure output
Occupancy	Output	[free | occupied]	Occupied	Occupancy output

Table 2 Specification of global constants

Identifier	Value	Brief description
CInt8Max	255	Maximum value for 8-bit integer type

By constructing FMBRSE, our purpose was to fill the gaps that characterize this special domain of electronic urban railway control system design. Therefore, FMBRSE was developed considering the characteristics and specialties of this domain, and enables fast, simple, cost-efficient modeling and verification process in this field. We achieved this by selecting and matching widely known methods from systems engineering, so their application does not require extra work from railway engineers.

We expect that the application of FMBRSE will result in closer collaboration between the end-user/operator and the manufacturer, and will allow for a more even distribution of development costs from the design phases throughout the life cycle of the system (see Fig. 22). In this methodology, we have integrated techniques that ensure an efficient set of specification tools with minimal effort compared with current solutions, and as a result, the functional requirements specification is significantly improved in terms of quality (completeness, correctness, consistency, controllability).

4 Case Study

The safety of level crossings is of paramount importance to tram operators in Hungary. Accidents are typically prevented with two solutions: traffic control equipment (in the case of complex intersections) and tram-road level crossing (TRLC) protection systems (in the case of simple intersections). The advantage of the latter solution lies in its cost-effectiveness. To ensure the safety of TRLC systems, the Hungarian operator BKV has developed a requirements booklet [64]. This booklet has been the basis for the development of electronic TRLC systems since 2014. Figure 4 shows an example of an application of the TRLC system for a simple single-track site. This system consists of two tram detection points (D1 and D2). The function of these points is to detect tram presence within the scope of the system. Road traffic is controlled by road signals (R1

Table 3 Configuration elements of the detection point

Identifier	Codomain			Brief description
	Min.	Max.	Unit	
PTopn	0	5,000	[ms]	Maximum time allowed for antagonism between Presence inputs
PTomin	0	10,000	[ms]	Minimum time of train presence within the scope of the sensor. It is certainly impossible that a train stays within the scope of the detection point for a shorter time than Tomin.
PTomax	500	20,000	[ms]	Maximum time of train presence within the scope of the detection point. It is certainly impossible that a train stays within the scope of the detector for a longer time than Tomax.
PTomaxE	false	true	–	Presence of upper limit Tomax. When PTomaxE is false, it is allowed that one train could stay above the detection point for an "infinite time".
PTr	100	10,000	[ms]	Release preparation time. The purpose of this time is to await release in a state faulty. In other words, the detection point shall be waiting in a faulty state next to correct inputs, i.e. next to a state non-faulty of Fault inputs and a free state of Presence inputs. The detection point can only be released (i.e. can be entered into a free state) when this time has elapsed.

Table 4 Specification of timers

Identifier	Codomain			Brief description
	Min.	Max.	Unit	
Topn	0	25,500	[ms]	The time of the antagonism between the presence inputs of the detection point
To	0	25,500	[ms]	Occupancy time of detection point
Tr	0	25,500	[ms]	Release time of detection point

and R2), and tram traffic by indicator (I1). The elements of the system are held together by the control cabinet (CC). We chose this system partially because it is relevant to safety, it has a simple architecture and a small number of elements, thus it fits the scope limitations of this article. At the same time, it is suitable for presenting all essential features of our methodology.

In the initial state of the system (no tram in the scope of the system), road signals show flashing yellow, and the indicator is blank. When a tram arrives at detection point D1, the road signals change to continuous yellow. This state lasts for a well-defined, short period (from 4 to 10 seconds) of time. After that, the road signals change to a red aspect, and road traffic must stop. Then the yellow aspect appears on the indicator. To the tram driver, this means that road traffic has been stopped successfully. Thus, the tram can cross the intersection at the maximum speed allowed by national rules. When the tram has left the system at detection point D2, the equipment goes to the initial state.

In terms of functionality, a detailed design (see Sect. 4.1) and formal modeling (see Sect. 5) of an abstract component will be presented in the following sections. We selected the detection point for these purposes because vehicle detection is one of the foundations of system safety.

4.1 Specifications of Detection Point

The purpose of the detection point is to detect a tram vehicle within its scope. Based on [61], the functionality of the detection point is described in simple terms: if the tram is over the detection point, it is "occupied", and when the tram is not over the detection point, it is "free". Railway engineers can significantly supplement this behavior during the functional specification based on the experiences gained so far and domain-specific knowledge. This process should be complemented by the several interactions with end-users (operators) and designers during the development process. In this paper, we do not present the mindset described in Sect. 1.1 in detail. We give only those results that we have reached at the end of the reconciliation process.

The interface specification of the detection point (DP) is given in Fig. 5 and Table 1. The DP has four functional inputs and two outputs. The DP receives these digital inputs from the input-output (IO) management module in the safety-critical embedded software after these physical input signals have passed—among other activities—on the

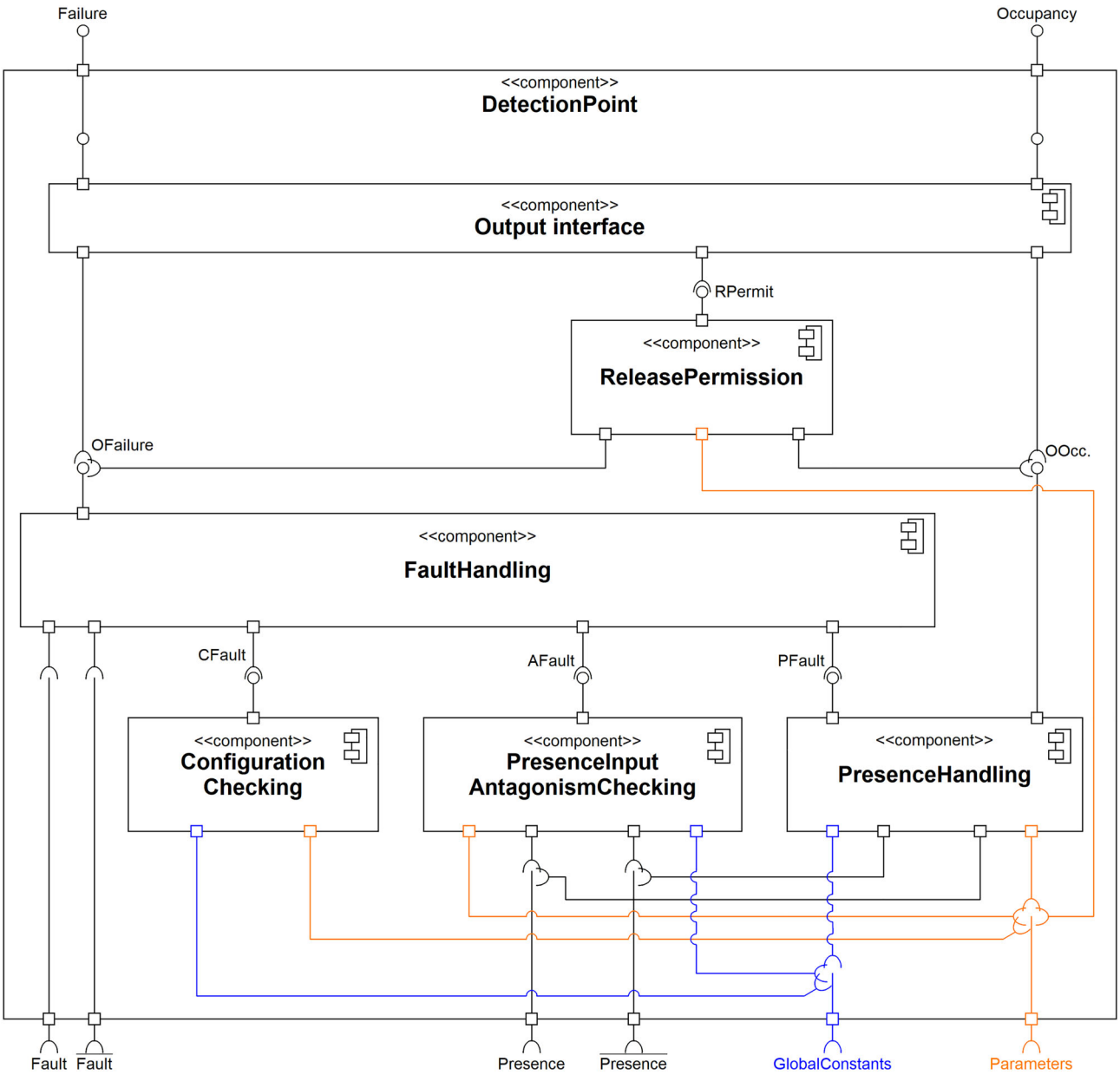

Fig. 6 Internal structure of the detection point

Table 5 Internal interface specification of the detection point

Identifier	Type	Codomain	Initial value	Brief description
CFault	In//Output	[false | true]	True	Configuration fault
AFault	In//Output	[false | true]	True	Antagonism between presence inputs
PFault	In//Output	[false | true]	True	Presence time fault
RPermit	In//Output	[false | true]	False	Release permission
OFailure	In//Output	[non-faulty | faulty]	Faulty	Internal failure state
OOccupancy	In//Output	[free | occupied]	Occupied	Internal occupancy state

input-filtering. The DP component is also connected to further modules through which it accesses global variables (see Table 2, Globals) and parameters (see Table 3, Parameters). These relationships are only given in a simplified way in this specification. The DP object sets the Failure and Occupancy outputs after processing the inputs. The outputs of DP are used by other modules for their operation. We note that we also neglected some outputs of

Table 6 System definitions in Yakindu

```
@CycleBased(100)  // 100 ms long cycle
@ChildFirstExecution

interface:
// External interfaces of detection point - Input variables
var in_fault_p : boolean = true    // Input - Fault
var in_fault_n : boolean = true    // Input - negated Fault
var in_presence_p : boolean = true // Input - Presence
var in_presence_n : boolean = true // Input - negated Presence
// External interfaces of detection point - Output variables
var out_failure : boolean = true   // Output - Failure
var out_occupancy : boolean = true // Output – Occupancy

internal:
// Internal interfaces between modules of the detection point
var CFault : boolean = true     // Configuration fault
var AFault : boolean = true     // Antagonism between presence inputs
var PFault : boolean = true     // Presence time fault
var RPermit : boolean = false   // Release permission
var OFailure : boolean = true   // Failure
var OOccupancy : boolean = true // Occupancy

// Globals
const CInt8Max : integer = 255 // Maximum value of integer (8 bit)

// Configuration elements (parameters)
const PTopn : integer = 10     // Time of antagonism between Presence inputs
const PTomin : integer = 20    // Minimum time of train presence within the scope of the sensor
const PTomax : integer = 50    // Maximum time of train presence within the scope of the detection
const PTomaxE : boolean = true // Existence of upper limit Tomax
const PTr :integer = 10        // Release preparation time, means 1000 ms

// Timers
var Topn : integer = 0 // Presence antagonism time
var To : integer = 0   // Presence time
var Tr : integer = 0   // Release preparation time
```

the DP component. For example, the diagnostics output that helps log the states of each object in this software system.

Table 1 lists the specification of each interface of the DP, with their ID, type of interface, codomain, initial state, and brief description. From the viewpoint of functionality, we assumed based on our practical experiences that in addition to the Presence input, there could exist a negated Presence, Fault, and even a negated Fault input.

In a certain configuration of the detection point, the occupancy time (To, see Table 4) has no upper limit (PTomaxE, To with "infinite time", see Table 3). However, timers in real systems are always finite. In this case, the situation when the To timer reaches its practical maximum, it must be handled properly. This maximum value depends on implementation details, such as the platform that runs the software realizing the functionality. In our case study, we assumed that an 8-bit microcontroller and 8-bit unsigned integer variables reused, so the maximum value is 255 (CInt8Max, see Table 2). This choice was also motivated by the fact that a microcontroller with larger bit-width would significantly increase the state space of the system model. We assume that if the correctness of the functionality is verified under such conditions (see Sect. 6), then by inductive reasoning the verified properties will hold even for a larger microcontroller. Other than this, the functionality is independent of the implementation.

Table 3 contains the configuration elements (parameters) relevant to DP functionality. Closely related to these are the timings defined in Table 4. Both tables are similar in structure, containing the identifier, codomain, and a brief description of the parameters and timers. In terms of

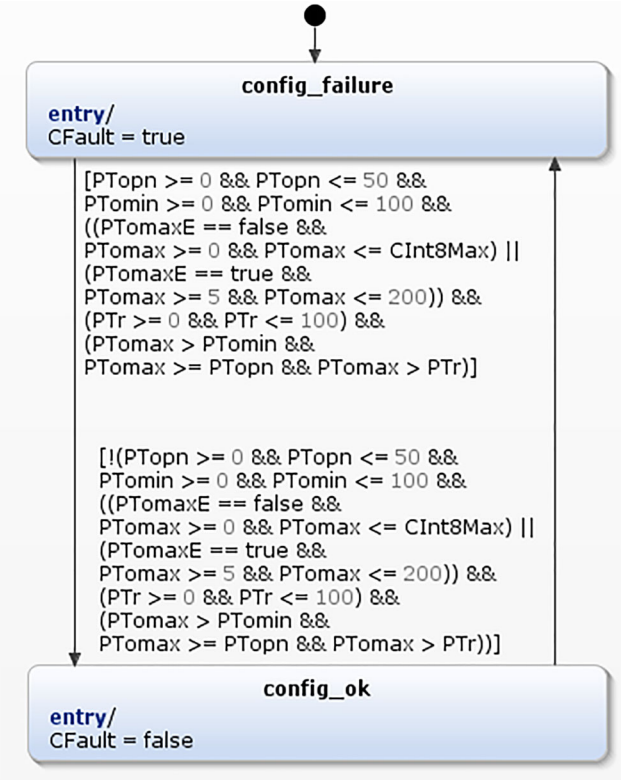

Fig. 7 Checking configuration (1)

timings and parameters, the following relationships exist: Topn with PTopn, To with PTomin, PTomax, PTomaxE, and Tr with PTr. Our goal was to design and verify the functionality of an abstract object, so we did not deal with configuration elements such as the identifier of inputs, the identifier of instances of DP (i.e., with instantiation of DP), etc. Accordingly, these are not shown in Table 3.

The interface specification shown in Fig. 5 can be further detailed in terms of functionality. The following functions can be defined for DP: (1) compliance checking with the configuration rules, (2) antagonism checking between presence and negated presence inputs, (3) handling of presence, (4) handling of faults, (5) handling of release, and (6) setting of outputs of component. Based on (1)–(6), the hierarchical decomposition of the DP component is shown in Fig. 6. The internal interfaces defined for decomposition are described in Table 5.

4.2 Implementation Behavior of Detection Point Using Yakindu

The behavior of the detection point was described as UML statechart in Yakindu (version 3.5.9). We chose a cycle-based execution scheme; i.e., the run-to-completion step is executed periodically at regular time intervals [65]. The definition section implemented in Yakindu is shown in Table 6. This was accomplished based on the specifications described in Sect. 4.1. The clear and simple transformation can be easily traced between Tables 1, 2, 3, 4, 5 and 6. The functions (1) to (6) given in Sect. 4.1 were implemented as state machines (see Figs. 7, 8, 9, 10, 11 and 12).

It is the responsibility of the designer to specify the system parameters for a specific application. Design instructions and application conditions are usually provided in the design manual of the system in question. However, it must be taken into account that the designer may make mistakes in their work. Therefore, the correctness of the designed parameters must be checked according to the rules specified in the design manual. There are several options for this. When we developed this case study, we decided that this function would be part of the task as well. The configuration rules related to DP can be read well from Fig. 7 (see guards). The operation of the "paramcheck" state machine is trivial: when there is a configuration failure in the system, it remains in the config_failure state. If a configuration fault occurs during system operation (e.g., the "parameter store" is damaged), the state machine will transit to the config_failure state from the config_ok state. The "paramcheck" state machine gives failure at its output when it is in the config_failure state (CFault is true) and gives non-failure when it is in the config_ok state (CFault is false).

The presence antagonism checking function ("antagonismcheck", see Fig. 8) checks that the presence (in_presence_p) and negated presence (in_presence_n) inputs are not in contradiction (non_antagonism state). In the non_antagonism state, the Topn timer does not run, and the AFault output of this function is false (i.e., there is no antagonism between presence inputs of DP). If the DP detects a discrepancy between the presence inputs, it enters the state antagonism. In the antagonism state the Topn timer runs. If the antagonism disappears, the state machine returns to the non_antagonism state and resets the Topn timer. If timer Topn is less than or equal to PTopn, there will be no antagonism failure at output AFault (AFault is false). If timer Topn is greater than PTopn and less than CInt8Max, there will be antagonism failure at output AFault (AFault is true). If timer Topn reaches value CInt8Max, it will still be a true value at output AFault. Until Topn reaches CInt8Max, the DP component would be able to provide accurate information to the diagnostics about how long the antagonism fault has occurred.

The "presencehandling" state machine (see Fig. 9) consists of two main states: free and occupied. The system can be entered into the free state if it detects free on both presence inputs. Depending on the type of occupancy from which the DP enters the free state, it can set a failure at its output (PFault is true or false). If the DP detects a presence

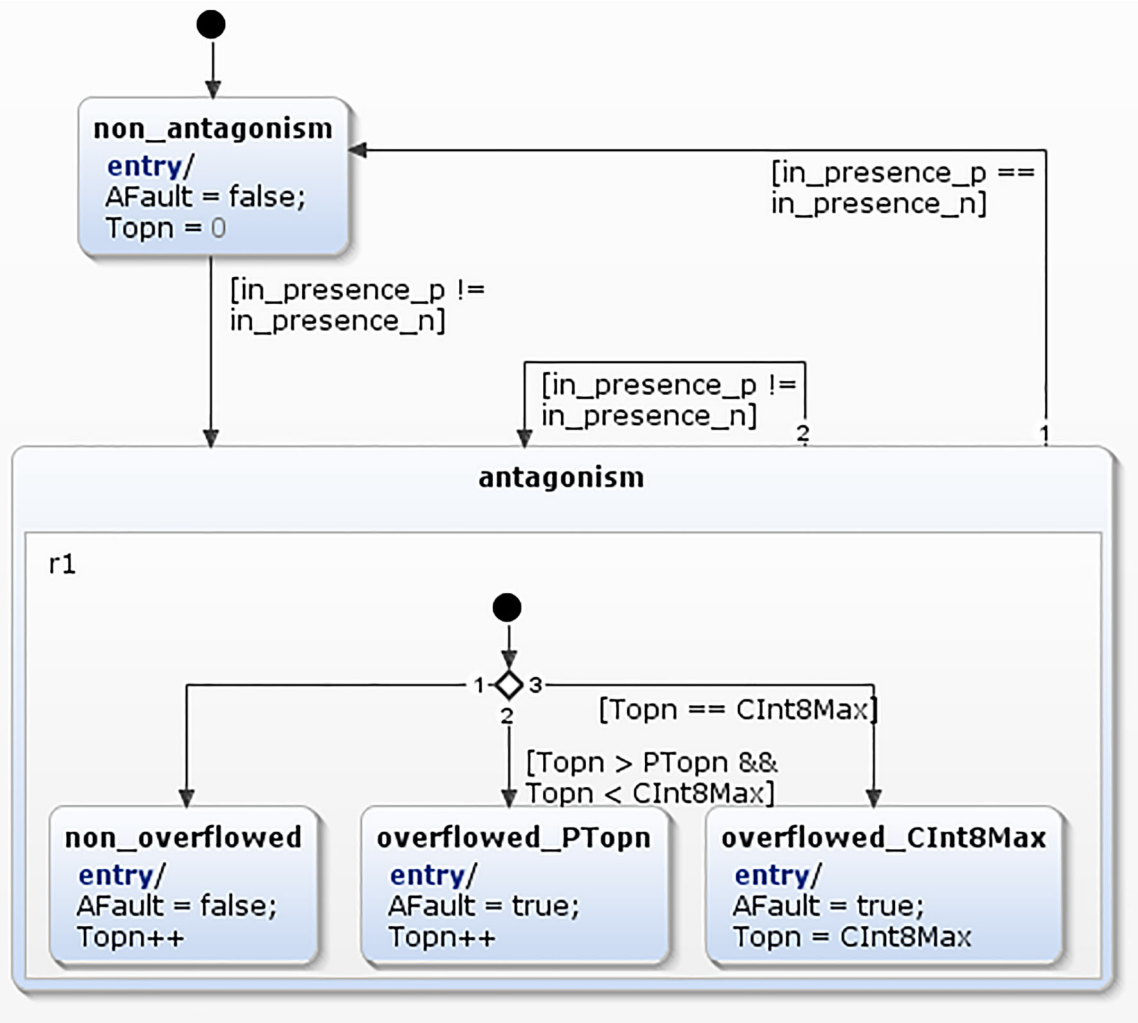

Fig. 8 Checking presence antagonism (2)

on any of its presence inputs, it enters an occupied state. When DP is in an occupied state, the To timer is running. In the occupied state, the DP is in one of the six states shown in Fig. 8, depending on the configuration and the timer To.

The short_occupancy state means that the detection point was occupied for such a short time, that it is physically impossible to be caused by a tram in practice. For the DP, this means that the value of the To timer was less than the PTomin parameter for the duration while both presence inputs were in the free state (with PFault true).

If the presence time To has an upper bound (PTomaxE is true), then the state machine can also enter to state overflowed_PTomax. The PTomax value means that the detection point was occupied for an incredibly long time. Such a long occupancy by a tram cannot occur in practice. For the DP this means that the value of the To timer was more than the PTomax parameter for the duration while both presence inputs were in the free state (with PFault true). Until the moment when To reaches CInt8Max (PTomaxE is true), the DP component would be able to provide accurate information to the diagnostics about how long the presence has occurred (see state awaiting_free_with_fault / overflowed_CInt8Max).

In practice, there may also be a case where it is not necessary to define an upper bound for the time To. In these cases, if DP detects tram presence, it uses the states awaiting_free_without_fault/non_overflowed_CInt8Max and awaiting_free_without_fault/overflowed_CInt8Max.

Finally, if the DP has correctly detected the occupancy according to the configuration (PTomaxE can be also true or false), the tram presence will remain in the state non_overflowed_PTomax. The DP can enter from this state to the free state without failure (i.e., with false PFault).

Function handling of faults (“faulthandling”) collects the outputs of the state machines “paramcheck”, “antagonismcheck”, and “presencehandling” (see Fig. 10). In addition to the former, it also handles fault and negated fault inputs of DP. If there is any error in the system, this

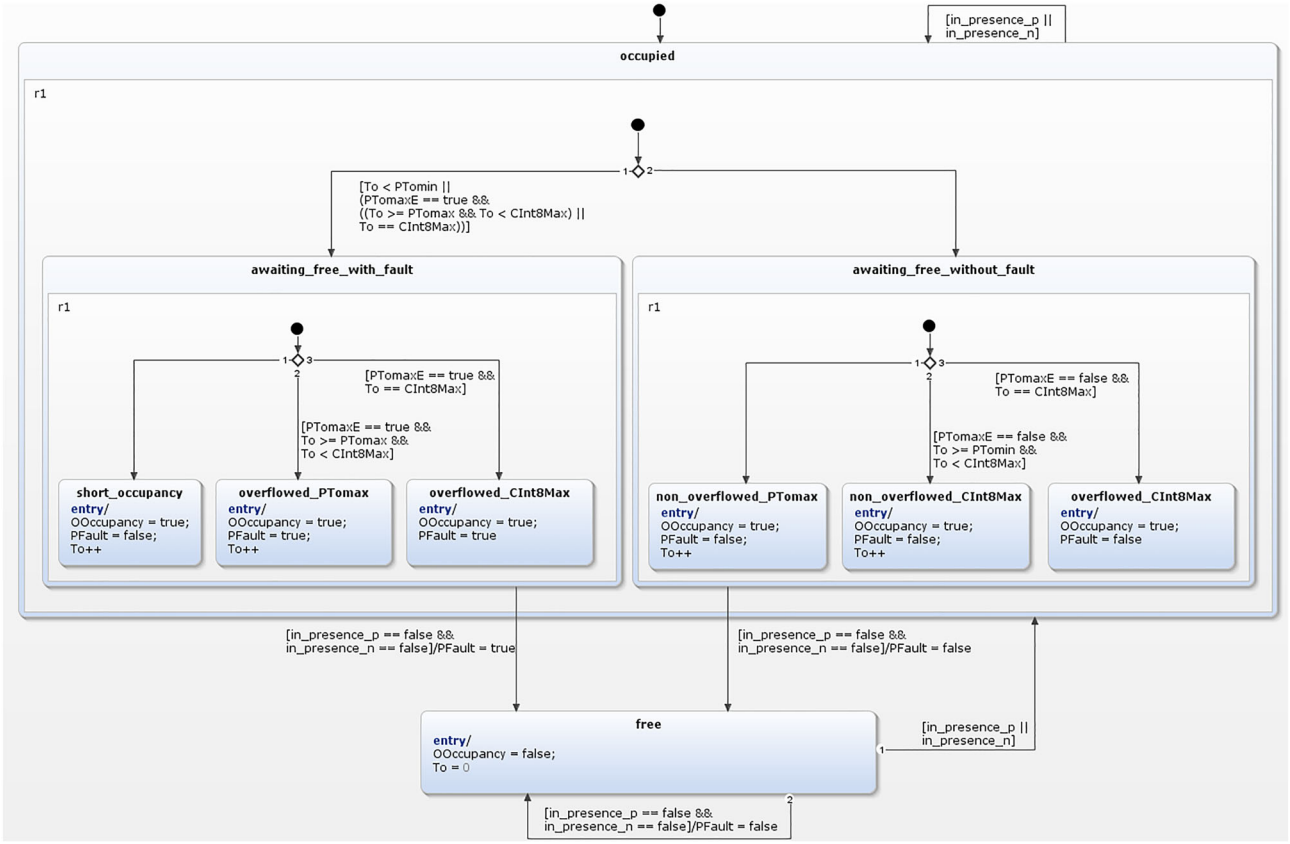

Fig. 9 Handling of presence (3)

state machine enters state faulty and gives a failure at its output (OFailure is true). If there is no fault in the system, "faulthandling" is in the state non_faulty and does not give a failure at its output (OFailure is false).

The state machine "releasepermission" (see Fig. 11) blocks the DP so that it cannot be released immediately after a failure. The condition for release is that the DP must be neither faulty nor in occupied states for a specified period of time (PTr). The release time is measured by the timer Tr. The state machine "releasepermission" uses outputs of the "presencehandling" (OOccupancy) and "faulthandling" (OFailure). Based on these inputs, determine its own output (RPermit) as a function of time Tr.

The outputs (out_failure and out_occupancy) of the DP component are set by the "outputsetting" state machine (see Fig. 12). For this operation, "outputsetting" uses outputs of the "presencehandling" (OOccupancy), "faulthandling" (OFailure), and"releasepermission" (RPermit). The output of DP can be a failure together with occupancy (failure_occupied) and non_failure together with free or occupied.

In this section, we have demonstrated steps 1-3 of the methodology described in Sect. 3. The Yakindu tool did not give any syntax errors to the described model above. The semantics of the model was verified by simulation. However, in this way, we could not be fully convinced of the correctness of the functionality of this model, as only a few use cases were available to us. Therefore, we decided to create this case study also with formal modeling to make model checking.

5 Results

Starting from the statecharts described in Sect. 4.2, we created the formal model of the detection point as UPPAAL process templates (finite automata). In this section, we present this constructed UPPAAL model. The prepared formal model is one of the inputs to the model checking, as discussed in Sect. 3. We used UPPAAL academic version 4.2.24.

5.1 Declarations in UPPAAL

The declarations part of the process templates in UPPAAL (see Table 7) was generated from the system definition part of Yakindu (see Table 6). This transformation is clear and well traceable based on Tables 6 and 7.

For the model checking to be performed, it is not enough to only transform the state machines defined in Yakindu. For example, it must be ensured that the time handling specified in the Yakindu definitions (see Table 6, first row) will also be implemented in some form in UPPAAL. The variables required for time handling in UPPAAL are listed in Table 7, "Channels and variables for simulation". A possible implementation of time handling is discussed in Sect. 5.4.1. In addition, it is necessary to ensure in UPPAAL that each automaton is executed in the correct order. The declarations required for this are given in Table 7, "Run control, permissions within a tick". A possible implementation of run sequences is discussed in Sect. 5.4.2. The above two additions are needed in the declarations section of the UPPAAL model compared with Yakindu.

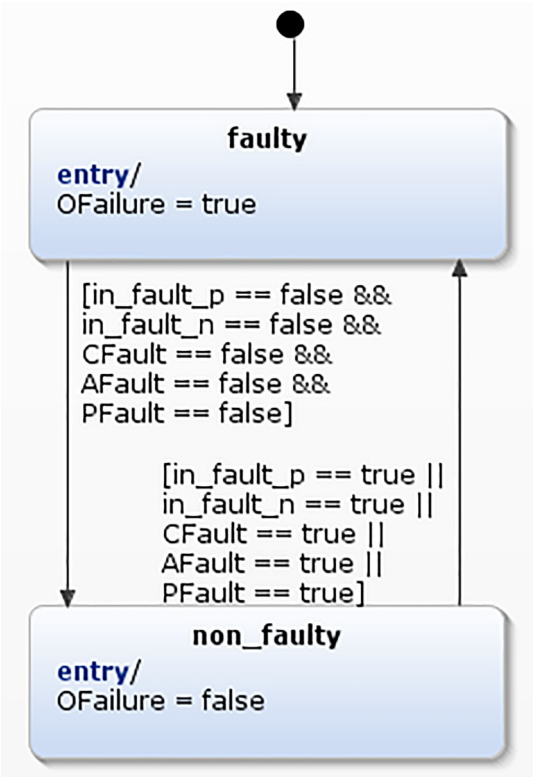

Fig. 10 Handling of faults (4)

5.2 System Declarations in UPPAAL

For the syntactical correctness of the UPPAAL model, the system declaration part must be specified correctly. The transformation of each Yakindu state machine to its equivalent UPPAAL process template can be easily traced based on their names. These automata are "paramcheck", "antagonismcheck", "presencehandling", "faulthandling", "releasepermission", and "outputsetting". Three additional automata ("tick", "runcontrol", "inputgenerator") were needed to make the UPPAAL model suitable for simulation and verification. In the following sections, we first introduce the automata directly transformed from the state machines and then the additional automata.

Based on the parameters and global constants of the Yakindu state machines (see Sect. 4.2), we can also specify these parameters for each automaton in the system

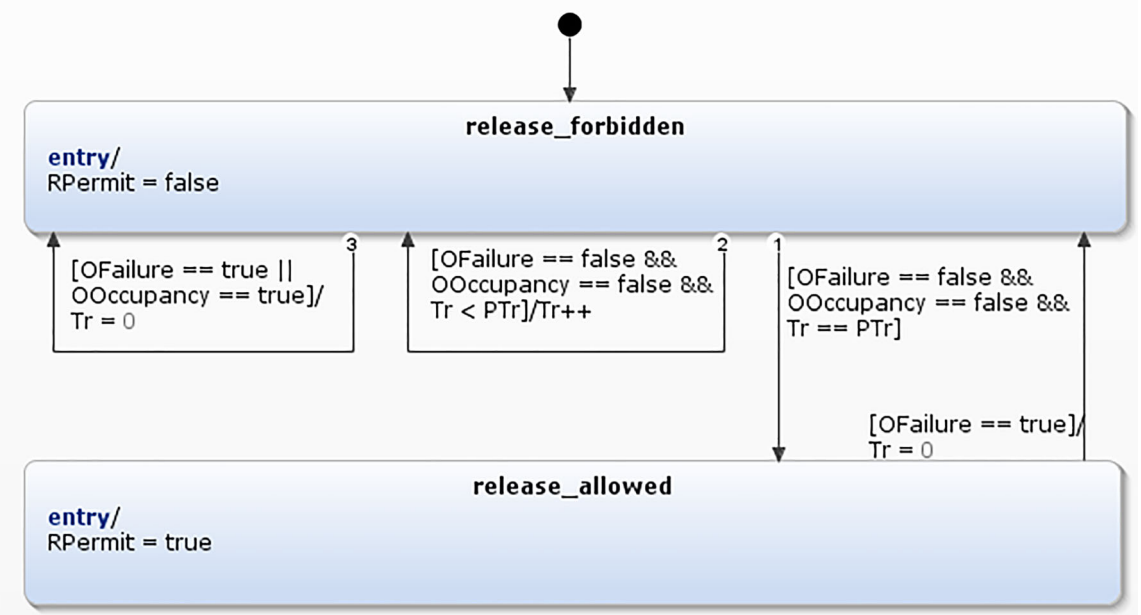

Fig. 11 Release permission (5)

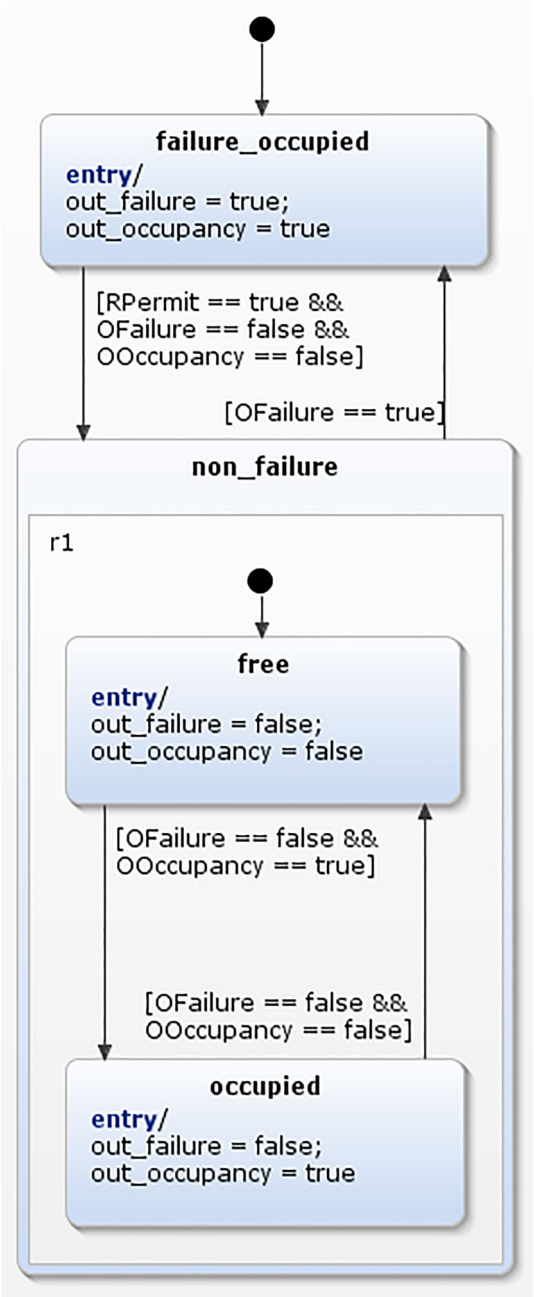

Fig. 12 Handling of outputs (6)

declaration part of the Uppaal model. The parameter list for each Uppaal automaton is shown in Table 8.

5.3 Constructed Automata in Uppaal

The configuration checking function implemented in Uppaal is shown in Fig. 13. This automaton is constructed based on the state machine shown in Fig. 7. Its parameters and declarations sections are described in Table 9. In both implementations, the number and name of the states and the number of transitions are the same. There are only a few differences in notation. The settings of outputs by the state machine became in the "update" section of the state transitions of the automaton.

The only thing that was not directly included in the Yakindu model is the channel (ALLOWEDRUN) defined for all edges of the automata. Through this channel, this automaton is authorized to perform its functions in each cycle. Related to this, the conditions of the guards at the edges also had to be supplemented with the variable PERMISSION. This mechanism is the same for the Uppaal automata described in below, so this is no longer discussed separately.

The presence antagonism checking function implemented in Uppaal is shown in Fig. 14. This automaton is constructed based on the state machine shown in Fig. 8. Its parameters and declarations sections are described in Table 10. The variables, parameters, and constants have identical names in both the Yakindu and Uppaal models, so the transformation can be easily traced.

The composite states help engineers to create a merged, simple, and effective representation of behavior. The "antagonism" state of the "antagonismcheck" state machine is a composite state. There are three additional states in this composite state. The contents of the composite state cannot be transformed directly into Uppaal because, for the conversion, the composite state must be unfolded. This means displaying the edges that were hidden by the merging. Similar cases can be also found in the "presencehandling" and "outputsetting" state machines (see Fig. 9 and 12).

The presence handling function implemented in Uppaal is shown in Fig. 15. This automaton is constructed based on the state machine shown in Fig. 9. Its parameters and declarations sections are described in Table 11. The variables, parameters, and constants have identical names in both Yakindu and Uppaal, so the transformation can be easily traced.

Fig. 15 shows an illustrative example of the process described in Sect. 5.4 for unfolding the composite states in Fig. 9.

The fault handling function implemented in Uppaal is shown in Fig. 16. This automaton is constructed based on the state machine shown in Fig. 10. Its parameters and declarations sections are empty.

The release permission function implemented in Uppaal is shown in Fig. 17. This automaton is constructed based on the state machine shown in Fig. 11. Its parameters and declarations sections are described in Table 12.

The outputs handling function implemented in Uppaal is shown in Fig. 17. This automaton is constructed based on the state machine shown in Fig. 12. Its parameters and declarations sections are empty.

Table 7 Declarations in UPPAAL

```
// External interfaces of detection point - Input variables
bool in_fault_p = true;    // Input - Fault
bool in_fault_n = true;    // Input - negated Fault
bool in_presence_p = true; // Input - Presence
bool in_presence_n = true; // Input - negated Presence
// External interfaces of detection point - Output variables
bool out_failure = true;   // Output - Failure
bool out_occupancy = true; // Output – Occupancy
// Internal interfaces between modules of detection point
bool CFault = true;     // Configuration fault
bool AFault = true;     // Antagonism between presence inputs
bool PFault = true;     // Presence time fault
bool RPermit = false;   // Release permission
bool OFailure = true;   // Failure
bool OOccupancy = true; // Occupancy
// Globals
const int CInt8Max = 255; // Maximum value of integer (8 bit)
// Configuration elements (parameters)
const int PTopn = 10;      // Time of antagonism between Presence inputs
const int PTomin = 20;     // Minimum time of train presence within the scope of the sensor
const int PTomax = 50;     // Maximum time of train presence within the scope of the detection point
const bool PTomaxE = true; // Existence of upper limit Tomax
const int PTr = 10;        // Release preparation time, means 1000 ms
// Channels and variables for simulation
chan CYCLE;         // Tick, start new cycle (1 tick == 100 ms)
bool ISRUN = false; // Tick ran (true: ran, false: did not run)
// Run control, permissions within a tick
broadcast chan ALLOWEDRUN; // Running permissions of functions
int PERMISSION = 0;        // Running permissions of functions, possible values
                           // 0: not used, initial
                           // 1: INPUTGENERATOR is allowed to run
                           // 2: PARAMCHECK is allowed to run
                           // 3: ANTAGONISMCHECK is allowed to run
                           // 4: PRESENCEHANDLING is allowed to run
                           // 5: FAULHANDLING is allowed to run
                           // 6: RELEASEPERMISSION is allowed to run
                           // 7: OUTPUTSETTIGN is allowed to run
```

5.4 Additional Automata for Simulation and Model Checking

Related to the automata given in Sect. 5.3, we already mentioned that the UPPAAL process templates obtained from the corresponding Yakindu state machines are insufficient for simulation and model checking. The specifics of the run-time environment must also be included in the formal model. In this section, we suggest one possible solution for this issue, but it is not an exclusive solution: other (potentially more efficient) solutions may exist as well.

Three additions are needed to make the UPPAAL model of the detection point suitable for simulation and model checking: time handling (A), execution control (B), and an input function (C) for the component (Fig. 18).

5.4.1 Handling of Time

The time handling implemented in UPPAAL is shown in Fig. 19. The parameters and declarations sections of this automaton are described in Table 13.

The “tick” automaton consists of two states. The “start” state is a committed state, so there is no time delay in this state. In the “run” state the component performs the described functionality. When this automaton switches from the “start” to the “run” state, a new cycle begins (CYCLE!). A cycle lasts 1 time unit (which is equivalent to 100 ms in the modeled system). The cycle length is represented by the invariant t<=1 of the “run” state. When the component has finished running, the automaton returns from the “run” to the “start” state. The “tick” and “run-control” (see Sect. 5.4.2) automata are closely related to each other in terms of control. The “tick” automaton gives

Table 8 System declarations in UPPAAL

```
// Automata
tick = TC();                                                    // Cycle generator
runcontrol = RC();                                              // Run control
inputgenerator = INPGEN();                                      // Input generator
paramcheck = PCHK(PTopn, PTomin, PTomax, PTomaxE, PTr, CInt8Max); // Checking configuration elements
antagonismcheck = ACHK(PTopn, CInt8Max);                        // Checking antagonism
presencehandling = PH(PTomin, PTomax, PTomaxE, CInt8Max);       // Presence handling
faulthandling = FH();                                           // Fault handling
releasepermission = RP(PTr);                                    // Release permission
outputsetting = OH();                                           // Output handling

system tick, runcontrol, inputgenerator, paramcheck, antagonismcheck, presencehandling, faulthandling,
releasepermission, outputsetting;
```

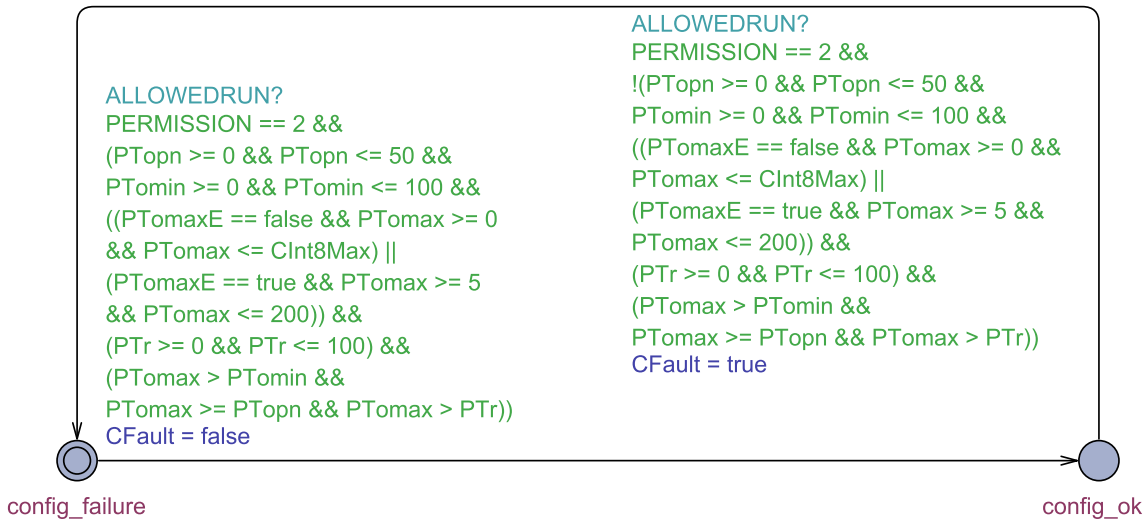

Fig. 13 Automaton for checking parameters

Table 9 Parameters and declarations of automaton "paramcheck"

Parameters
`const int PTopn, const int PTomin, const int PTomax, const bool PTomaxE, const int PTr, const int CInt8Max`
Declarations
-

run permission to the "runcontrol" using channel CYCLE. The "runcontrol" automaton notifies the "tick" automaton when it has finished running using the variable ISRUN. If the cycle ends, the clock variable t will reset, and the variable ISRUN will return to false.

5.4.2 Execution Control

The execution control function is shown in Fig. 20. The parameters and declarations sections of this automaton are empty.

The "runcontrol" automaton is responsible for executing the functions of the component in the correct sequence.

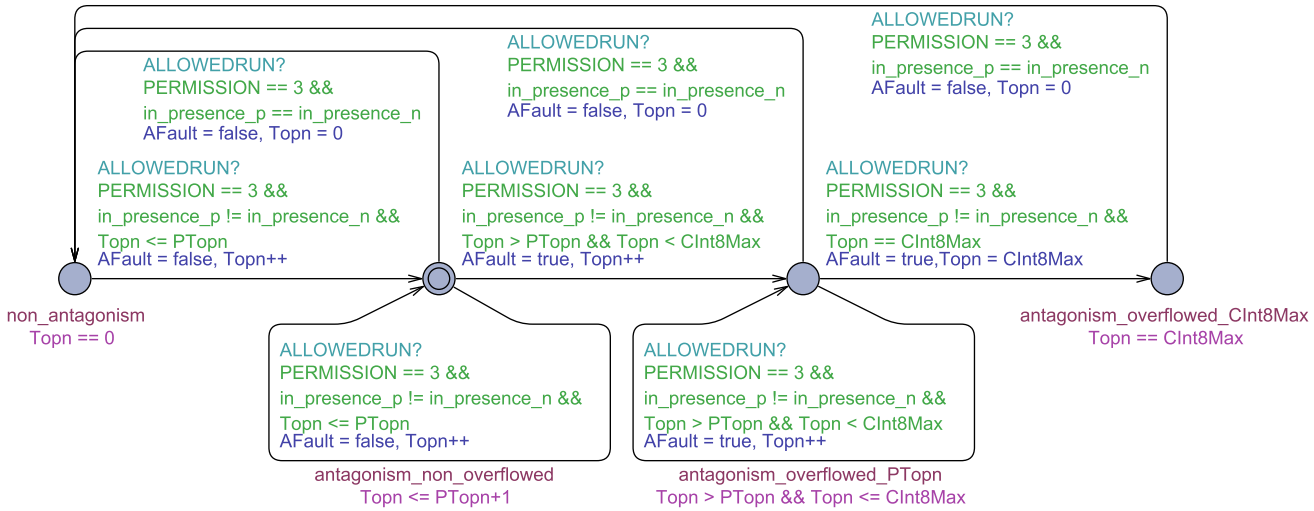

Fig. 14 Automaton for checking presence antagonism

Table 10 Parameters and declarations of automaton "antagonismcheck"

Parameters `const int PTopn, const int CInt8Max`

Declarations
`// Timers`
`int Topn = 0; // Presence antagonism time`

Each of the automata belonging to the detection point functions has a unique identifier (PERMISSION) from 2 to 6. The automaton with identifier 1 is "inputgenerator" (see Sect. 5.4.3). Thus, the order of execution is as follows: "inputgenerator", "paramcheck", "antagonismcheck","presencehandling", "faulthandling", "releasepermission", "outputsetting". Each automaton receives its run permission via the broadcast channel ALLOWEDRUN, and they use the variable PERMISSION to determine whether it is their turn.

Note that UPPAAL ensures that each automaton is executed in the fixed order specified behind the "system" keyword in the system declarations section (see Sect. 5.2, Table 8.). However, we do not exploit this feature, assuming that a component might call a function multiple times within the same cycle.

5.4.3 Input Function

One possible input function of the detection point implemented in UPPAAL is shown in Fig. 21. The parameters and declarations sections of this automaton are described in Table 14.

Depending on our modeling purposes, we may or may not need an input function to examine each component. This function can be specified in several forms. In our case study, we constructed an input function for the presence and fault inputs of the DP component that covers all possible input combinations. The created function is shown in Table 14. Note that the version of UPPAAL that is used is not capable of handling the "switch case".

Each cycle begins with the DP object reading its inputs. The input to be read is always randomly selected (see Fig. 21, inp:int[0,15]) from the set of specified input combinations. After that, each automaton runs based on a given sequence (see Sect. 5.4.2). The automata perform their task depending on the detected inputs, current state, parameters, and timers.

6 Discussion

In Sect. 3, we gave an overview of the principles of the FMBRSE methodology, which supports railway engineers in the application of formal specification and verification during the development of a safety-critical system. The design and specification steps for the proposed methodology were illustrated with the help of a case study in Sects. 4 and 5.

As a result, the FMBRSE approach leads to a significant improvement in quality and distributes the development

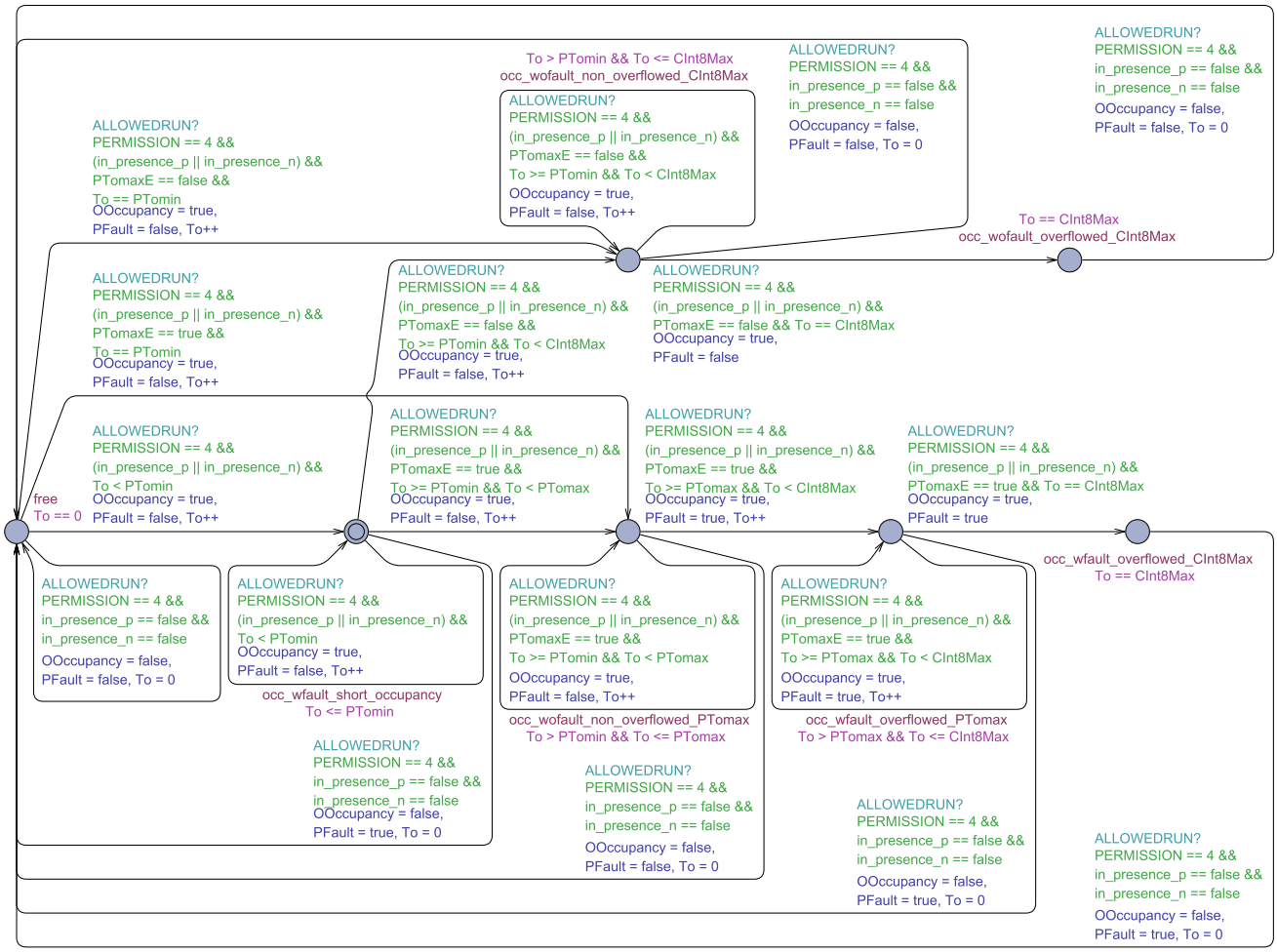

Fig. 15 Automaton for handling of presence

Table 11 Parameters and declarations of automaton "presencehandling"

Parameters
`const int PTomin, const int PTomax, const bool PTomaxE, const int CInt8Max`
Declarations
`// Timers` `int To = 0; // Presence time`

costs more evenly among the related life-cycle phases. An example of the impact of the application of the FMBRSE in practice is shown in Fig. 22. The diagram on the left (see Fig. 22a) shows the conventional development of the functional specification of a component, and the diagram on the right (see Fig. 22b) shows the same process within the FMBRSE framework. In Fig. 22, three levels of abstraction were distinguished:

HIGH: means the level of operator/end-user,
MIDDLE: means the level of railway engineer, and
LOW: means the level of software engineers.

Figure 22a and b both illustrate the number of iterations (vertical axis) required to complete the functional specification of a component.

Based on Fig. 22, the following conclusions can be drawn: In traditional development, users are typically not very involved in the process. They formally communicate their requirements in some textual form (e.g., using requirements booklets). (In this section we use the terms "formal" and "informal" in the "procedural" sense, and

do not refer to the mathematical rigor, as was done previously.) They are unwilling to formally modify these requirements during the development due to its lengthy approval process and costs (in Hungary). This is the reason why Fig. 22a shows only one iteration at the level HIGH. However, the users are often informally helping to clarify the issues related to the set of requirements during the development, even more so with the application of the proposed methodology. Therefore, users became much more involved in the development with FMBRSE than in the traditional case. Because of the increased participation of users, railway engineers were able to create a functional specification of much better quality in fewer iterations. For this reason, the number of detailed design steps for software engineers was also significantly reduced. Altogether, a more ideal process was created in the design phase of the development with the proposed methodology through the equalization of costs and resources, and a better distribution of the work performed during the individual activities, compared with the current practice.

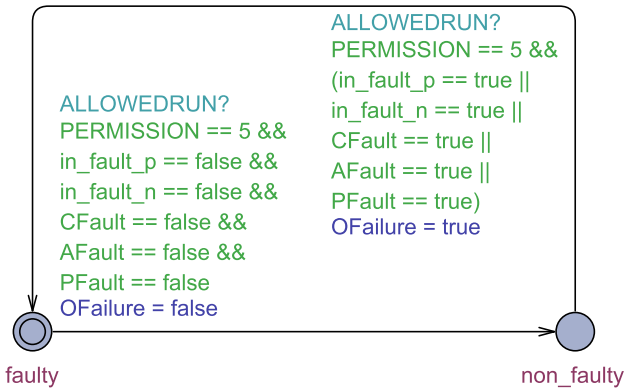

Fig. 16 Automaton for handling of faults

The expected result of the methodology is a formally verified and validated functional model of a component. The verification step related to the case study described in Sects. 4 and 4 is presented in this section using the built-in model checking functionality of the UPPAAL tool.

We used a computer that has an Intel® Core™ i5-7200U CPU and 8 GB memory. The setting of UPPAAL during the

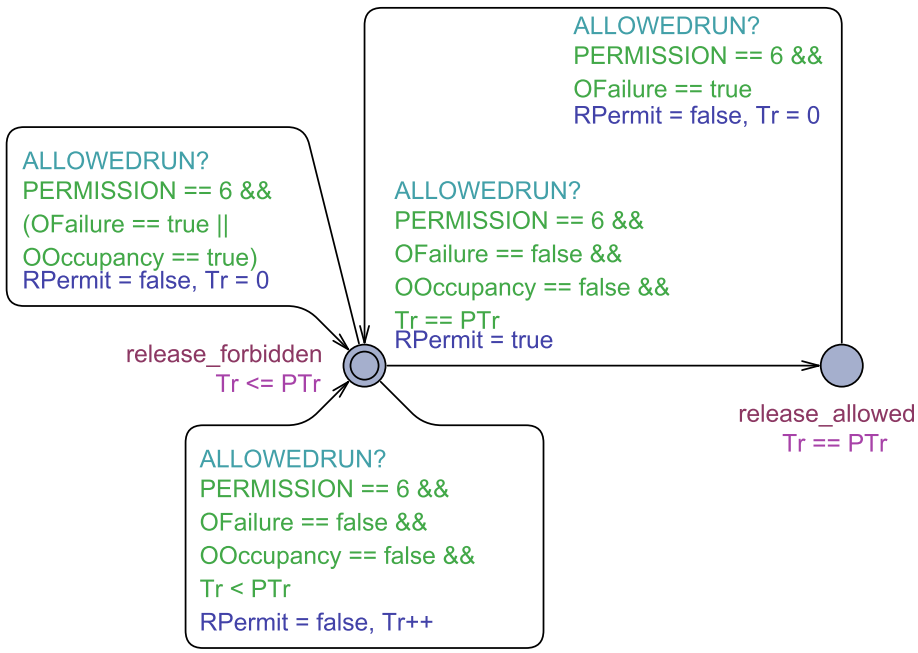

Fig. 17 Automaton for release permission

Table 12 Parameters and declarations of automaton "releasepermission"

Parameters `const int PTr`

Declarations
`// Timers`
`int Tr = 0;  // Release preparation time`

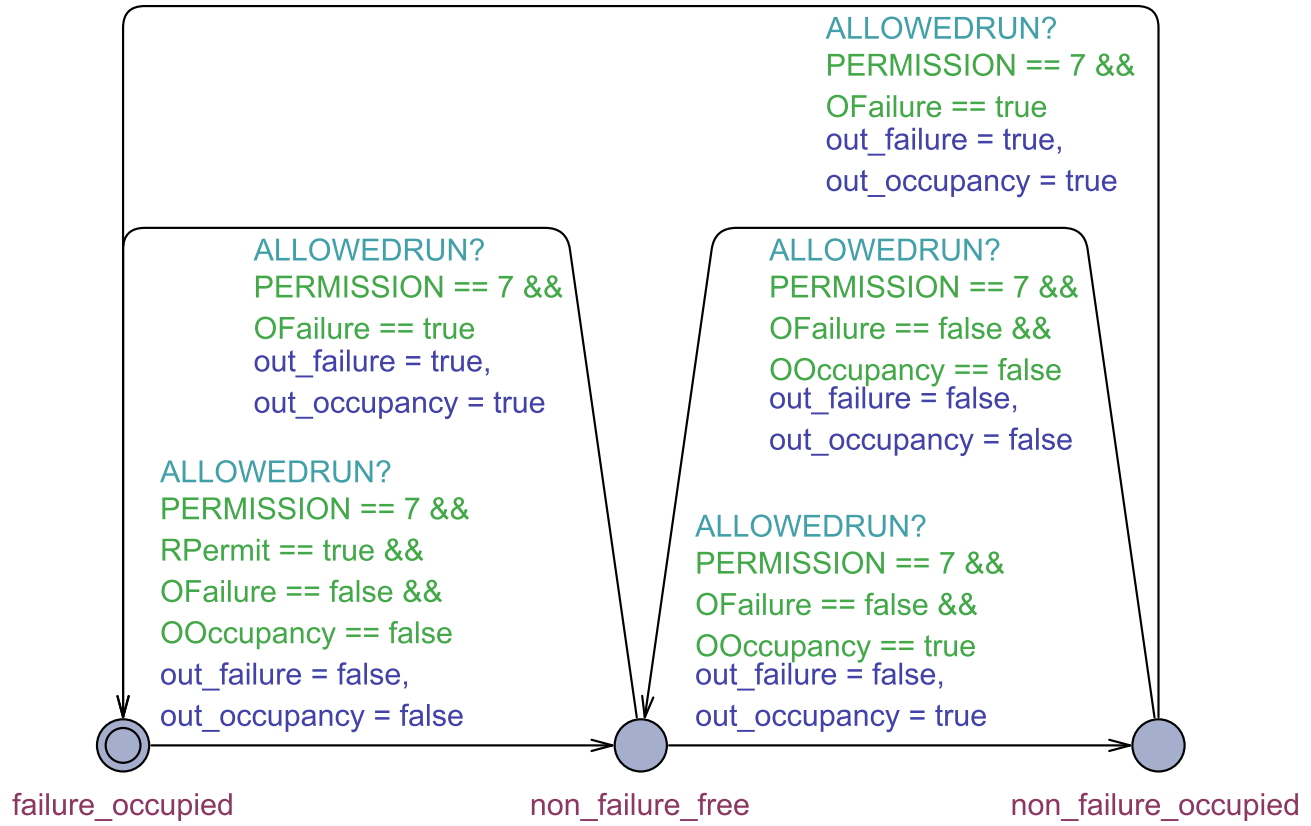

Fig. 18 Automaton for handling of outputs

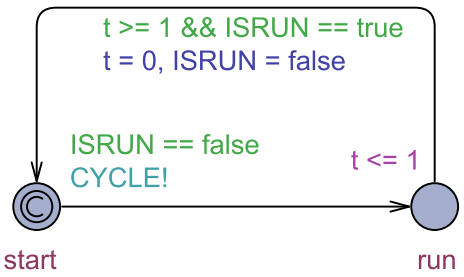

Fig. 19 Automaton for handling of time

model checking was as follows: breadth-first search order, conservative state space reduction, difference bound matrices (DBM) state space representation, no diagnostic trace, automatic extrapolation, 16 MB hash table size.

The subject of the model checking was the formal model described in Sect. 4, and the requirements were given by railway engineers in the form of natural language descriptions. These requirements were converted into branching temporal logic (CTL) formulas. Examples of some of the requirements verified (the natural language form on the left, and the converted CTL formulas on the right) are shown in Table 15. These can basically be divided into three groups: model validation (e.g., row 1), state availability (e.g., rows from 2 to 8), and functional requirements described by railway engineers (e.g., rows 9 and 10).

Note that the requirements given by domain engineers in most cases cannot be directly converted into a CTL formula. This transformation consisted of two steps. As a first step, we rewrote the requirements into an intermediate domain-specific (restricted) language (currently under development). In the second step, the requirements given in the intermediate language were converted into CTL

Table 13 Parameters and declarations of automaton "tick"

Parameters -

Declarations
`// Clock variable`
`clock t; // Length of tick (100 ms)`

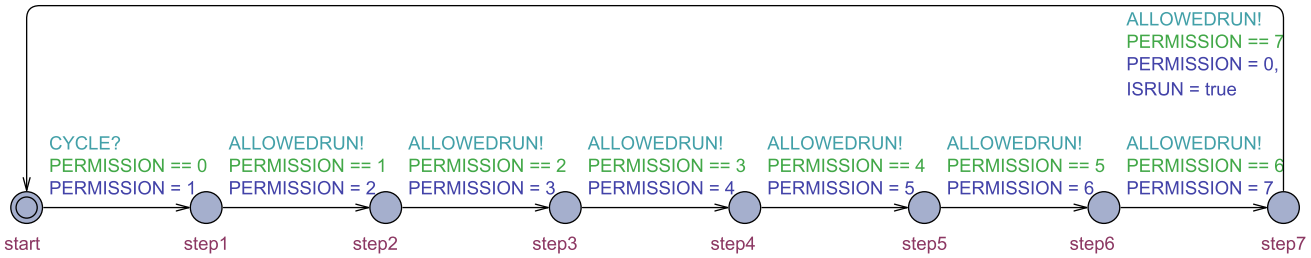

Fig. 20 Automaton for execution control

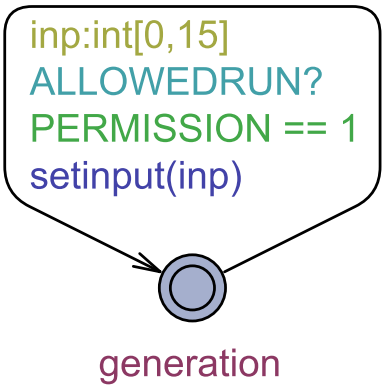

Fig. 21 Automaton for input handling

formulas. We found that expressions written in this intermediate domain-specific language are still easy to understand, but allow simple conversion to CTL formulas.

In our experience, handling requirements in this way required numerous interactions with field engineers. This process is the most resource-intensive part of the proposed FMBRSE methodology, but it is worth the effort, since the safety of the system is also most heavily dependent on this activity.

The results of the model checking related to the case study can be found in Table 17. These results were obtained for the configuration given in Table 16. Note that in addition to the examples given in Table 15, 50 more requirements (not included here) were also subject to model checking. These requirements were verified in 90 different configurations. We found that the requirements related to availability are configuration-dependent, while functional requirements are configuration-independent. Rows 7 and 8 of Table 17 also show two examples of requirements related to state availability not being satisfied.

Table 14 Parameters and declarations of automaton "inputgenerator"

Parameters
-

Declarations

```
// Input generator function
if (inp == 0){in_fault_p = false; in_fault_n = false; in_presence_p = false; in_presence_n = false;}
else if (inp == 1){in_fault_p = false; in_fault_n = false; in_presence_p = false; in_presence_n = true;}
else if (inp == 2){in_fault_p = false; in_fault_n = false; in_presence_p = true; in_presence_n = false;}
else if (inp == 3){in_fault_p = false; in_fault_n = false; in_presence_p = true; in_presence_n = true;}
else if (inp == 4){in_fault_p = false; in_fault_n = true; in_presence_p = false; in_presence_n = false;}
else if (inp == 5){in_fault_p = false; in_fault_n = true; in_presence_p = false; in_presence_n = true;}
else if (inp == 6){in_fault_p = false; in_fault_n = true; in_presence_p = true; in_presence_n = false;}
else if (inp == 7){in_fault_p = false; in_fault_n = true; in_presence_p = true; in_presence_n = true;}
else if (inp == 8){in_fault_p = true; in_fault_n = false; in_presence_p = false; in_presence_n = false;}
else if (inp == 9){in_fault_p = true; in_fault_n = false; in_presence_p = false; in_presence_n = true;}
else if (inp == 10){in_fault_p = true; in_fault_n = false; in_presence_p = true; in_presence_n = false;}
else if (inp == 11){in_fault_p = true; in_fault_n = false; in_presence_p = true; in_presence_n = true;}
else if (inp == 12){in_fault_p = true; in_fault_n = true; in_presence_p = false; in_presence_n = false;}
else if (inp == 13){in_fault_p = true; in_fault_n = true; in_presence_p = false; in_presence_n = true;}
else if (inp == 14){in_fault_p = true; in_fault_n = true; in_presence_p = true; in_presence_n = false;}
else if (inp == 15){in_fault_p = true; in_fault_n = true; in_presence_p = true; in_presence_n = true;}
return 0;
}
```

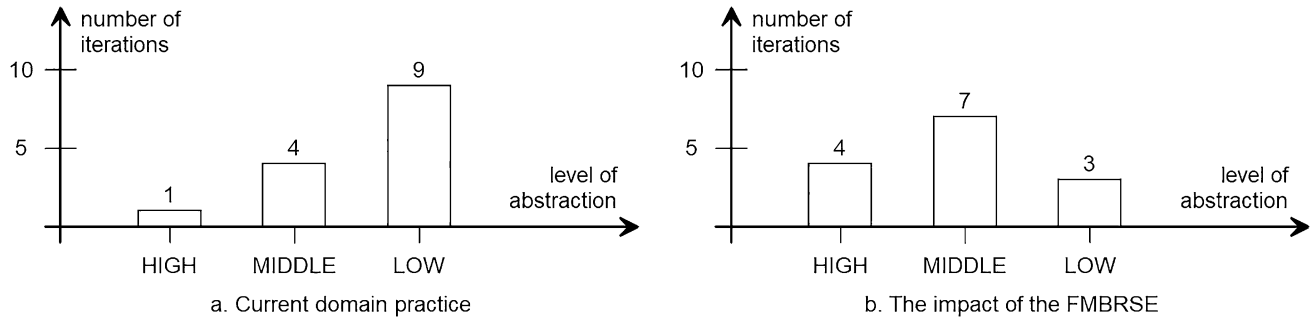

Fig. 22 The impact of the use of FMBRSE in practice in an example of the DP (see Sect. 4)

Table 15 Some examples for verification of requirements of detection point in UPPAAL

ID	Requirement (natural language)	CTL formula
1	Exemption from deadlocks.	A □ not deadlock
2	Availability of the states of automata "presencehandling". "There is a least one route in which a given state is available."	E ◇ presencehandling.free
3		E ◇ presencehandling.occ_wfault_short_occupancy
4		E ◇ presencehandling.occ_wofault_non_overflowed_PTomax
5		E ◇ presencehandling.occ_wfault_overflowed_PTomax
6		E ◇ presencehandling.occ_wfault_overflowed_CInt8Max
7		E ◇ presencehandling.occ_wofault_non_overflowed_CInt8Max
8		E ◇ presencehandling.occ_wofault_overflowed_CInt8Max
9	If the detection point detects a tram presence on one of its presence inputs, it leads to its occupancy output must become occupied.	(in_presence_p == true ‖ in_presence_n == true) ⇝out_occupancy == true
10	If the detection point detects a fault on one of its fault inputs, it leads to its occupancy output must become occupied, and failure output must become faulty.	(in_fault_p ‖ in_fault_n) ⇝out_failure == true && out_occupancy == true

Table 16 Example of one configuration setting for model checking

Configuration element	Specified value
PTopn	10
PTomin	20
PTomax	50
PTomaxE	True
PTr	10

In fact, this result is expected next to the given configuration in Table 16.

With model checking of the DP component, we performed each step of the proposed FMBRSE methodology described in Sect. 3. Note that railway engineers experienced difficulties in evaluating the results of requirement violations. When UPPAAL provided a counterexample, it proved almost impossible for them to decipher where the error causing the requirement violation was. This issue could be solved by developing a backward mapping/annotating method to show the counterexample in the high-level model.

7 Conclusion

In this paper, we presented a formal model-based methodology that facilitates the construction of correct, complete, consistent, and verifiable functional specifications during the development of electronic urban railway control system. The process we propose provides a specification-verification environment for railway engineers. Using this framework they can achieve a higher-quality functional specification compared with traditional development. The most significant advantage of the described approach is that it hides the formal methods-related details from the railway engineers, so they can acquire a formal verification result without learning the necessary mathematical background. We demonstrated the use of the

Table 17 The result of the model checking with the specified configuration in Table 15

ID	Result of model checking	Elapsed time [s]	Resident/Virtual memory usage [KB]
1	Property is satisfied	45.29	417,968/845,588
2	Property is satisfied	0.91	417,968/845,588
3	Property is satisfied	0.001	11,414/33,668
4	Property is satisfied	0.351	15,800/39,968
5	Property is satisfied	1.426	34,876/68,544
6	Property is satisfied	24.415	417,368/847,920
7	Property is not satisfied	14.7	418,636/850,576
8	Property is not satisfied	14.622	418,636/850,576
9	Property is satisfied	27.199	418,884/851,368
10	Property is satisfied	27.562	420,428/855,380

framework through a case study related to a tram-road-level crossing protection system.

We worked together with railway engineers during the development of the case study. We concluded that a small subset of UML enables a fast and efficient design and verification process already in the early stages of the life cycle, thus reducing development costs (with the help of models and verification a significant number of errors are revealed already in the early stages of the life cycle). During the transformation of the Yankidu model to an UPPAAL model, we found that a part of the proposed specification-verification environment can be transformed automatically. We are currently working on the formal definition of these transformation steps.

We found that the main difficulty for railway engineers is preparing the requirement specification. The problem is that they do not (want to) deal with the formalization of the requirements during the preparation of specifications. To solve this issue, we began to develop an intermediate domain-specific restricted textual language for the railway field.

In summary, the proposed methodology proved its suitability for the design of electronic urban railway control systems. Using formal models and model checking, a high-quality functional specification can be achieved, written by railway engineers at the system development level. Finally, the approach described in this paper is theoretically not a new methodology; instead it uses widely applied and well-proven techniques, but an expedient integration of existing methodologies, which thus means that a new contribution is the appropriate integration of existing methodologies, tailored to the domain of electronic urban railway control systems development. Therefore, the main contribution of our work is that we choose from the many existing planning and verification methods a combination that is compatible with the characteristics of this domain. Its novel elements include (1) the selection and integration of the appropriate high-level semi-formal and low-level formal description forms and tools into a toolchain that fits the railway field, (2) the transformation from the semi-formal to formal models as illustrated by the case study, and (3) taking the specifics of the railway engineering domain and the best-practice systems engineering into account during the creation of the proposed methodology.

Acknowledgements The authors would like to express their thanks/gratitude to the itemis AG for providing the academic license of the Yakindu Statechart Tools, furthermore to the Uppsala Universitet and Aalborg University for providing the academic license of the UPPAAL tool.

Authors' contributions All authors contributed to the study's conception and design. Material preparation, data collection and analysis were performed by Gábor Lukács. The first draft of the manuscript was written by Gábor Lukács and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Not applicable.

Data availability The authors confirm that the data supporting the findings of this study are available within the article and/or its supplementary materials.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. CENELEC EN 50129:2018 Railway applications – Communication, signaling and processing systems – Safety related electronic systems for signaling (English version).
2. CENELEC EN 50128:2011 Railway applications – Communication, signaling and processing systems – Software for railway control and protection systems (English version).
3. Vyatkin V, Hanisch H (2001) Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems, In: ETFA 2001. 8th international conference on emerging technologies and factory automation. Proceedings (Cat. No.01TH8597), vol. 2, pp. 113–118. https://doi.org/10.1109/ETFA.2001.997677.
4. Gnesi S; Margaria T (2013) Some trends in formal methods applications to railway signaling, In: Formal methods for industrial critical systems: a survey of applications, IEEE, pp. 61–84, doi: https://doi.org/10.1002/9781118459898.ch4.
5. Alanazi MN (2009) Basic rules to build correct UML diagrams, In: 2009 international conference on new trends in information and service science, pp. 72–76. https://doi.org/10.1109/NISS.2009.252
6. Laibinis L, Troubitsyna E, Leppänen S, Lilius J, Malik Q (2005) Formal model-driven development of communicating systems. In: Lau KK, Banach R (eds) Formal methods and software engineering. ICFEM 2005. Lecture Notes in Computer Science, vol. 3785. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11576280_14.
7. Clarke EM, Wang Q (2015) 25 years of model checking. In: Voronkov A, Virbitskaite I (eds) Perspectives of system informatics. PSI 2014. Lecture Notes in Computer Science, vol. 8974. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-46823-4_2.
8. IEC 61508-1:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements (English version).
9. ISO 26262:2018 Road vehicles – functional safety – from Part 3 to 8 (English version).
10. DO-178B:2012 Software Considerations in Airborne Systems and Equipment Certification (English version)
11. CENELEC EN 50126:2017 Railway applications – The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) – Part 1: Generic RAMS Process (English version).
12. A. van Lamsweerde Requirements Engineering: From System Goals to UML Models to Software Specifications, 2009, ISBN: 978-0-470-01270-3.
13. Colin H (2007) Requirements management, The interface between requirements development and all other systems engineering processes, Springer-Verlag Berlin and Heidelberg GmBH & Co. KG, ISBN: 9783540476894
14. IBM Rational DOORS (2013) Requirements management framework add-on, user manual, release 6.1. https://www.ibm.com/support/pages/rational-doors-requirements-management-framework-add-61
15. RequirementOne ReqMan: Getting started guide v1.1., https://www.em.ag/en/reqman/
16. Siemens Polarion 21 R1 Administrator and User Help, 2021, http://www.teamlive.com.cn/downloads/Polarion_User_and_Administration_Help.pdf
17. CENELEC EN 50617-2:2015 Railway applications. Technical parameters of train detection systems for the interoperability of the trans-European railway system. Part 2: Axle counters (English version).
18. Commission Regulation (EU) 2016/919 of 27 May 2016 on the technical specification for interoperability relating to the 'control command and signaling' subsystems of the rail system in the European Union
19. 18/1998. (VII. 3.) KHVM regulation of the National Railway Regulations II volume (Hungary)
20. Long D, Scott Z (2011) A primer for model-based systems engineering, 2nd edition, ISBN 978-1-105-58810-5
21. Heinrich H (2011) Model-driven development of advanced user interfaces, ISBN13: 9783642145612. https://doi.org/10.1007/978-3-642-14562-9
22. Hunt BR, Lipsman RL, Rosenberg Jm, et al (2001) A guide to MATLAB for beginners and experienced users, Cambridge University, ISBN-13: 978-0-511-07792-0
23. Chaturvedi DK (2010) Modeling and simulation of systems using MATLAB and simulink, ISBN 9781439806722
24. Papke BL et al (2020) Implementing MBSE – an enterprise approach to an enterprise problem. INCOSE Int Symp 30(1):1550–1567. https://doi.org/10.1002/j.2334-5837.2020.00803.x
25. Selic B (2006) Model-driven development: its essence and opportunities, In: Ninth IEEE international symposium on object and component-oriented real-time distributed computing (ISORC'06), p. 7. https://doi.org/10.1109/ISORC.2006.54.
26. SEBoK Editorial Board (2022) The guide to the systems engineering body of knowledge (SEBoK), In: Cloutier RJ (Editor in Chief). v. 2.6, The Trustees of the Stevens Institute of Technology, Hoboken. www.sebokwiki.org
27. Bock C, Cook C (lead), Rivett P, Rutt T, Seidewitz E, Selic B, Tolbert D (2017) OMG Unified Modeling Language (OMG UML) (2017) Version 2.5.1
28. Holt J, Perry S (2013) SysML for model-based systems engineering, Institution of Engineering and Technology, ISBN: 1849196516. https://doi.org/10.1049/PBPC010E
29. Wang J, Tepfenhart W (2019) Formal methods in computer science, ISBN 9781498775328
30. Bartha T, Majzik I (2019) Formal methods for safety planning and proof, ISBN: 978-963-454-2919, doi: https://doi.org/10.1556/9789634542919
31. Menezes J, Gusmão C, Moura H (2019) Risk factors in software development projects: a systematic literature review. Software Qual J 27:1149–1174. https://doi.org/10.1007/s11219-018-9427-5
32. Haxthausen AE (2010) An introduction to formal methods for development of safety-critical applications, DPU Informatics, Technical University of Denmark
33. Xie Y (2019) Formal modeling and verification of train control systems, Thesis, Centrale Lille
34. Adacore, QGen User Guide, Release 24.0w (2018) https://docs.adacore.com/live/wave/qgen/pdf/qgen_ug/qgen_ug.pdf
35. Clarke EM, Henzinger TA, Veith H (2018) Introduction to model checking. In: Clarke E, Henzinger T, Veith H, Bloem R (eds) Handbook of Model Checking. Springer, Cham. https://doi.org/10.1007/978-3-319-10575-8_1
36. Newborn M, Automated theorem proving, theory and practice, Springer New York, NY, ISBN-10: 0387950753. https://doi.org/10.1007/978-1-4613-0089-2
37. Terada N, Toyama T (2013) Application of verification methods to specifications of signalling equipment, quarterly report of RTRI, 2013, Volume 54, Issue 4, Pages 202-207, Released December 05, 2013, Online ISSN 1880-1765, Print ISSN 0033-9008. https://doi.org/10.2219/rtriqr.54.202
38. Zou L et al (2014) Verifying Chinese train control system under a combined scenario by theorem proving. In: Cohen E, Rybalchenko A (eds.) Verified software: theories, tools, experiments. VSTTE 2013. Lecture Notes in Computer Science, vol. 8164.

Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-54108-7_14.
39. Stankaitis P, Dupont G, Singh NK, Ait-Ameur Y, Iliasov A, Romanovsky A (2019) Modelling hybrid train speed controller using proof and refinement, In: 2019 24th international conference on engineering of complex computer systems (ICECCS), pp. 107–113. https://doi.org/10.1109/ICECCS.2019.00019.
40. Eisner C (2002) Using symbolic CTL model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. STTT 4:107–124. https://doi.org/10.1007/s100090100063
41. Eisner C (1999) Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. In: Pierre L, Kropf T (eds.) Correct hardware design and verification methods. CHARME 1999. Lecture Notes in Computer Science, vol 1703. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48153-2_9
42. Haxthausen A, Peleska J (2015) Model checking and model-based testing in the railway domain. In: Drechsler R, Kühne U (eds.) Formal modeling and verification of cyber-physical systems. Springer Vieweg, Wiesbaden. https://doi.org/10.1007/978-3-658-09994-7_4
43. Halpern JY, Vardi MY (1991) Model checking vs. theorem proving: a manifesto, KR. https://doi.org/10.1016/B978-0-12-450010-5.50015-3
44. Kunnappilly A, Backeman P, Seceleanu C (2021) From UML modeling to UPPAAL model checking of 5G dynamic service orchestration, In: ECBS 2021, Conference on the engineering of computer based systems. https://doi.org/10.1145/3459960.3459965.
45. Guo C, Fu Z, Zhang Z, Ren S, Sha L (2020) A framework for supporting the development of verifiably safe medical best practice guideline systems, J Syst Archit, 104: 101693, ISSN 1383-7621.https://doi.org/10.1016/j.sysarc.2019.101693
46. David A., Möller MO, Yi W (2002) Formal verification of UML statecharts with real-time extensions. In: Kutsche RD, Weber H (eds.) Fundamental approaches to software engineering. FASE 2002. Lecture Notes in Computer Science, vol 2306. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45923-5_15
47. Darvas D (2016) Practice-oriented formal methods to support the software development of industrial control systems. https://doi.org/10.5281/zenodo.162950
48. Vince M, Bence G, András V, István M, Dániel V (2018) The gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In: Proceedings of the 40th international conference on software engineering companion. ACM Press, New York, pp. 1–4. ISBN 978-1-4503-5663-3 (In Press). https://doi.org/10.1145/3183440.3183489.
49. Graics B, Molnár V, Vörös A et al (2020) Mixed-semantics composition of statecharts for the component-based design of reactive systems. Softw Syst Model 19:1483–1517. https://doi.org/10.1007/s10270-020-00806-5
50. Jiang Y et al (2016) From stateflow simulation to verified implementation: a verification approach and a real-time train controller design, In: 2016 IEEE real-time and embedded technology and applications symposium (RTAS), pp. 1–11.https://doi.org/10.1109/RTAS.2016.7461337
51. Nazaruddin YY, Tamba TA, Pradityo K, Aristyo B, Widyotriatmo A (2019) Safety verification of a train interlocking timed automaton model, IFAC-PapersOnLine, 52(15): 331-335, ISSN 2405-8963, https://doi.org/10.1016/j.ifacol.2019.11.696
52. Keming W, Zheng W, Chuandong Z (2018) Formal modeling and data validation of general railway interlocking system, In: 16th internal conference on railway engineering design & operation, lisbon, Portugal, 181: 527–538, ISSN 1743-3509. https://doi.org/10.2495/CR180471.
53. Cooper K, Ito MR (2002) Formalizing a structured natural language requirements specification notation, INCOSE International Symposium 12. https://doi.org/10.1002/j.2334-5837.2002.tb02569.x
54. Lukács G, Bartha T (2022) Practical UML subset for railway engineers to support formal modeling. Trans Motauto World 7(2):56–59
55. Baier C, Katoen J-P (2008) Principles of model checking. The MIT Press
56. Chatterjee K, Doyen L (2016) Computation tree logic for synchronization properties, In: 43rd International colloquium on automata, languages, and programing (ICALP 2016). https://doi.org/10.48550/arXiv.1604.06384
57. Behrmann G, David A, Larsen KG (2004) A tutorial on UPPAAL. In: Bernardo M, Corradini F (eds) Formal methods for the design of real-time systems. SFM-RT 2004. Lecture notes in computer science, vol 3185. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30080-9_7
58. Lukács G, Bartha T (2021) Formal modelling of level crossing system for tramsusing UPPAAL framework, Technical Review (EMT) 77: 18–37, 20 p
59. Bensalem S, Bozga M, Sifakis J, Nguyen TH (2008) Compositional verification for component-based systems and application. In: Cha S, Choi JY, Kim M, Lee I, Viswanathan M (eds) Automated technology for verification and analysis. ATVA 2008. Lecture Notes in Computer Science, vol 5311. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-88387-6_7
60. Boehm B, Port D, Winsor Brown A (2002) Balancing plan-driven and agile methods in software engineering project courses. Comput Sci Educ 12(3):187–195. https://doi.org/10.1076/csed.12.3.187.8617
61. Asagba PO, Ogheneovo EE (2007) A comparative analysis of structured and obejct-oriented programming methods, African J (AJOL), 11(4). https://doi.org/10.4314/jasem.v11i4.55190
62. Campean F, Henshall E, Yildirim U, Uddin A, Williams H (2013) A structured approach for function based decomposition of complex multi-disciplinary systems. In: Abramovici M, Stark R (eds.) Smart product engineering. Lecture Notes in Production Engineering. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-30817-8_12
63. Herrigel S, Laumanns M, Nash A, Weidmann U (2013) Hierarchical decomposition methods for periodic railway timetabling problems. Transp Res Rec 2374(1):73–82. https://doi.org/10.3141/2374-09
64. Budapest Transport Privately Held Corporation (BKV) (2011) Requirements booklet for safety elements and equipment for traffic control of trams (BKV-VILL-1.04)
65. itemis, Yakindu statechart tools, User Guide, https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines?hsLang=de