https://doi.org/10.3311/PPtr.23344
Creative Commons Attribution ⓘ

**Periodica Polytechnica Transportation Engineering**, 52(4), pp. 402–411, 2024

# Verification of Railway Control Systems Using Model Checking and CTL, Explained Through a Case Study

Gábor Lukács[1*], Tamás Bartha[1,2]

[1] Department of Control for Transportation and Vehicle Systems, Faculty of Transportation Engineering and Vehicle Engineering, Budapest University of Technology and Economics, Műegyetem rkp. 3., H-1111 Budapest, Hungary
[2] Institute for Computer Science and Control (SZTAKI), Kende street 13-17., H-1111 Budapest, Hungary
[*] Corresponding author, e-mail: lukacs.gabor@edu.bme.hu

Received: 02 September 2023, Accepted: 24 May 2024, Published online: 04 June 2024

**Abstract**
Systematic faults can often occur during the development of a system. The later such faults are discovered, the more expensive it can be to correct them. In systems engineering practice, there are many methods and tools to reduce the likelihood of systematic faults. In this paper, we present the application of a formal model–based verification technique – called model checking – to assist railway engineers in designing and verifying the safety-related functionality of railway control systems. The proposed process is part of a specification-verification environment that facilitates the construction of correct, complete, consistent, and verifiable functional specifications during development. The results and experience in model checking are illustrated by a case study of a vehicle detection point, a common component in this domain. The model checking of the case study has been performed in the widely used UPPAAL modeling and simulation framework, which can also verify formal properties and generate a counterexample in case of a property violation. By analyzing the counterexample, the designer can gain insights into the system's behavior and identify potential design flaws or failures. Model checking can be used to achieve a higher quality functional specification that is typically more complete and/or contains fewer faults compared to the traditional development approach.

**Keywords**
verification, model checking, computation tree logic, railway control system

## 1 Introduction

Society's expectations of safety-critical systems are increasing worldwide in all sectors, including railway engineering. Hazard identification and risk management have become a standardized process in this field (see e.g., CENELEC, 2018). There are many ways to achieve and maintain an appropriate level of safety integrity, supported by well-developed methodologies and techniques, and a wide range of tools to support the system development process (CENELEC, 2011; see CENELEC, 2018; CENELEC, 2017a; CENELEC, 2017b). These can be usefully complemented by formal methods from computer science (Roggenbach et al., 2022), whose semantics and syntax are well defined and complete, thereby forcing the applying engineer to think deeply and systematically about the problem under consideration, thus significantly reducing ambiguities and gaps in the specifications. Research into the application of formal methods in systems engineering practice in the field of railway automation systems engineering has a long history and many results (Nanda and Grant, 2019). Despite the progress made, the translation of these methods into everyday engineering practice is still to be achieved.

Model-Based Systems Engineering (MBSE), (Gnesi and Margaria, 2013) is gaining popularity in the practice. This is also preferred by railway engineers due to the advantages and power of modeling. Using mathematical/logical rules, formal modeling (Vyatkin and Hanisch, 2001) provides a way to precisely specify the functionality of systems.

Based on the motivations described above and on current practice, the aim of our research is to achieve a methodology based on formal methods, suitable for industrial application, at the systems engineering level, to support the development of safety-critical railway automation systems. This methodology includes the selection of appropriate specification and verification technologies and suitable supporting tools and their integration into a unified framework.

In developing the methodology, particular attention has been paid to MBSE techniques that form one basis of current systems engineering practice. The result of the research provides a framework for railway systems engineers that can be used to design and verify the safety-critical functions of the developed systems in a cost-effective and verifiably correct way. The scope of the research has been functional requirements and support for the design and verification of functional safety. Non-functional requirements and related design and verification activities were therefore not included in the research. The principles of the methodology were conceptualized in the paper (Lukács and Bartha, 2022a), hereafter referred to as Formal Model-Based Railway Safety Engineering (FMBRSE).

In this paper, our purpose is to highlight and present in detail the verification part of our framework. To assess the practical applicability of model checking in the railway domain, we developed some case studies in detail (Lukács and Bartha, 2021; Lukács and Bartha 2022a), of which we publish here the results achieved through the example of one case study (Lukács and Bartha, 2022a). For model checking, we used the UPPAAL framework, which proved to be suitable for handling the domain we investigated.

## 2 Related work

The formal verification of various safety-critical control systems has been researched in recent years, and the results are described in several publications. These papers have demonstrated the general applicability of formal verification, in particular model checking, in the context of these systems. In the subsequent paragraphs, we present various examples from some domains.

Cyber-physical systems (CPS) are safety-critical and can be analyzed using experimental testing and model-based verification. However, accurate models have the potential to reach risk-free simulation of system behavior even in extreme scenarios. To address the challenges of CPS modeling and design, the research of (Alshalalfah, et al., 2023) uses the INCOSE/OMG System Modelling Language (SysML), (Friedenthal, et al., 2011) standard to accurately specify CPSs. A limited number of SysML elements are defined to accurately capture the meaning of continuous and discrete time system behaviors. These elements are described by the creation of a novel algebra called Enhanced Activity Calculus (EAC). EAC assists in the creation of comparable timed automata models by developing a new methodical procedure to accurately translate the SysML models into the inputs of the UPPAAL-SMC statistical model checking tool. The latter examines whether the system is accurate and secure. The reliability of the translation mechanism was confirmed, and its efficiency was demonstrated in a real-life scenario, namely the artificial pancreas.

The VerifCar (Arcile et al., 2019) is a framework developed for modeling and model checking communicating autonomous vehicles (CAVs). This approach focuses on the formal modeling of connected autonomous vehicles using timed automata, enabling a formal analysis of vehicle behavior. For model checking, CTL is used.

A study from the railway industry (Laursen et al., 2020) investigates the modeling and model checking of a distributed railway interlocking system algorithm using the UPPAAL framework. The verification of interlocking systems for particular railway networks is achieved by using a generic model instantiated with configuration data describing the network and trains. Various versions of the generic model are used. Verification has been performed on all variants to ensure their correctness and to compare their performance.

Finally, we would also like to present an example from the nuclear industry that demonstrates one possible practical application for model checking. In this research, model checking is presented by (Pakonen, et al., 2017) to prove the correctness of the application logic of instrumentation and control (I&C) systems using some projects from the Finnish nuclear industry. MODCHK is presented as a user-friendly graphical modeling tool. It can be used to verify function block-based application logic. MODCHK creates the necessary input files for the NuSMV 2.6.0 model checker (Cimatti, et al., 2002). It performs the analysis and displays the results. The counterexamples produced by NuSMV are visualized using a 2D animation. The analyst can play the animation back and forth.

Based on these results, we have determined that identical principles and methods have been depending on purposes, tailored to the specialties of the industry. It can be assumed that distinct solutions are occasionally needed in different fields to fulfill the specified goals, regarding specification, model formation, and verification. The FMBRSE approach is based on existing methods and techniques. Its mathematical foundations and tools are well established. The way these are selected and integrated is specifically tailored to the engineering of safety-critical control systems in the railway domain. One of the methods included in FMBRSE is model checking. This paper presents how it is used in FMBRSE and what we have learned from experimenting with it.

## 3 Methodology

In this section, we present an overview of our proposed methodology. For a more detailed explanation of the methodology, please refer to (Lukács and Bartha, 2022a).

The purpose of FMBRSE is to aid railway engineers – designers who comply with the EN 50126-2 standard (CENELEC, 2017b) – in utilizing formal specification and verification when developing a safety-critical railway system. The application of this methodology throughout the development process yields a verified functional model of the railway system under consideration.

FMBRSE is based on a well-defined process that follows the lifecycle model described in the standards (CENELEC, 2017a, CENELEC, 2018). The process receives input in the form of requirements that are specified in detail by different stakeholders. The first stages (specifying the system requirements, designing the architecture, and allocating the system requirements) are familiar to railway engineers because of their inclusion in the standard (CENELEC, 2017a). The FMBRSE methodology provides a framework for specifying how to design system components according to their functionality. This framework has four fundamental pillars: requirements, interfaces, configuration, and behavior. The development of a formal model for the system or component aligns with these pillars.

FMBRSE also offers a verification environment that implements formal verification through the process of model checking (see Fig. 1). Model checking (Baier and Katoen, 2008) is the use of discrete mathematics to answer the question "Does the model satisfy a set of requirements or in the case of deviation, what sequence of events can lead to this situation?".

The goal of model checking is to verify the functionality of a designed system before its implementation. For expressing formal requirements, we use Computation Tree Logic (CTL) formulas (Chatterjee and Doyen, 2016), which in our case are derived from the requirements and formal models based on the system's functional specifications.

The paper (Lukács and Bartha, 2022b) provides a thorough explanation of how CTL formulas, which serve as one of the inputs to model checking, are generated by a rule-based technique that transforms natural language descriptions into a formal specification language. Fig. 2 contains examples that are intended to demonstrate the expressive power of CTL as it is used in FMBRSE and as it compares to other temporal logics.

The system behavior is automatically generated as timed automata models, which are transformed from UML statecharts (Lukács and Bartha, 2022c). We have defined a simplified subset of UML state machines for this mapping (see Lukács and Bartha, 2022c). This subset allows a clear correspondence between UML statecharts and UPPAAL timed automata systems.

The two necessary inputs for model checking – the CTL formulas and the formal model – are both available in the generated UPPAAL timed automata model, as shown in the previous two paragraphs and in Fig. 1. The paper (Lukács and Bartha, 2022a) provides a comprehensive explanation of the theoretical foundation of model checking. In this study, we solely discuss the supplementary requirements for conducting model checking in FMBRSE.

There are two key stages to the FMBRSE model checking process. As a first step (referred to as model validation), the model is checked for deadlocks and state reachability.

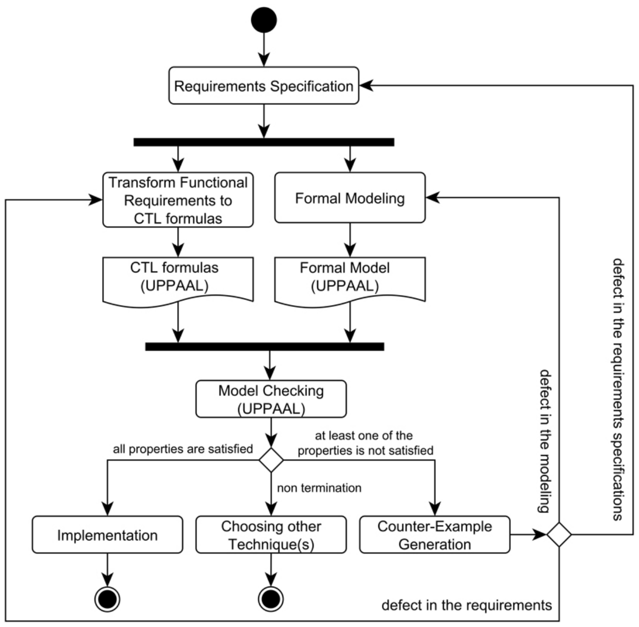
Requirements Specification
Transform Functional Requirements to CTL formulas
Formal Modeling
CTL formulas (UPPAAL)
Formal Model (UPPAAL)
Model Checking (UPPAAL)
all properties are satisfied
at least one of the properties is not satisfied
non termination
Implementation
Choosing other Technique(s)
Counter-Example Generation
defect in the requirements
defect in the modeling
defect in the requirements specifications

**Fig. 1** The process of the model checking

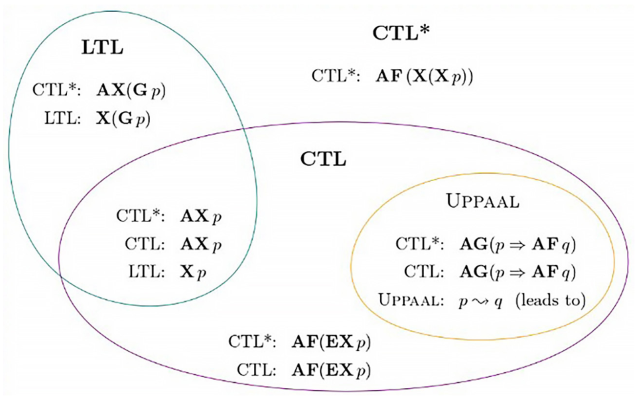

**Fig. 2** Expressive power of temporal logics

The FMBRSE framework currently verifies only these properties, but it could be expanded to other properties deemed significant in practice, such as safety properties. The second step of verification (referred to as functional requirements verification in FMBRSE) involves checking the functional requirements of the model as defined by the stakeholders. Validating the correctness of the model before the second verification step is a critical phase. The reason for this is that a comparison of stakeholder requirements with a model that has been incorrectly implemented can have misleading or inaccurate outcomes.

So far, we have explained an ideal model checking process and some FMBRSE-specific considerations. However, our practical experience with FMBRSE has shown that modeling the system environment is usually also necessary to supplement this process. Modeling the system environment is an additional component required for model checking that can significantly impact the other two inputs, specifically the CTL formulas and the formal model (refer to Fig. 1). Both inputs need to be refactored based on the defined environment. This usually involves augmenting the model and the requirements with environmental conditions.

In FMBRSE, we present a potential model for the environment, with the understanding that this is not the only possible solution. Our goal is to provide a simple template that domain engineers can use as a basis for constructing their environment models within the FMBRSE framework. The proposed environment model consists of three parts: action timing model, function call sequence model, and input function model. These are specific to the designed system; thus, they need to be designed individually with the modeling objectives in mind. In summary, a formal behavioral model of a system is composed of two main elements: the automata representing the functionality and the necessary elements modeling the environment.

Finally, the FMBRSE framework can be applied at both the system and component levels. This paper discusses the application of the framework at the component level through a case study (refer to Section 4). Another illustrative case study of the system-level application is described in (Lukács and Bartha, 2021). However, applying FMBRSE at the system level could result in a well-known difficulty: state space "explosion". This is because the system-level models have a larger and more complex state space than the component-level models. Compositional verification, as proposed by (Bensalem et al., 2008), is one approach to address this issue. It entails deducing global properties of intricate systems from the properties of their constituent parts. For more information on the limitations and constraints of model checking, see, for example, the referenced book (Clarke et al., 2018).

## 4 Case study

The purpose of the detection point (DP) is to detect a train within its range. The functionality of the detection point can be simply described as follows: *If the train is over the detection point, it is 'occupied'; and if the train is not over the detection point, it is 'free'.* Railway engineers can, based on previous experience and domain specific knowledge, significantly extend this behavior during functional specification.

The DP has four functional inputs and two outputs. The inputs are Fault, /Fault, Presence, and /Presence. These inputs and parameters determine how this component behaves and sets its outputs: Failure and Occupancy, which are inputs to other components that are required for their operation. The detailed specification and construction of the UPPAAl model of the DP component is described in the paper (Lukács and Bartha, 2022a). Our goal here is simply to summarize the experience of model checking the DP component. For this purpose, we provide the most relevant parts of the specification and the UPPAAl model of the DP component in the following paragraphs.

From the detailed design of the DP component, the following functions can be defined: checking compliance with the configuration rules, checking antagonism between presence and negated presence inputs, presence handling, fault handling, release handling, setting component outputs. Fig. 3 shows the internal structure of the DP component based on these functionalities. The components within the DP interact via global variables, with the key variables illustrated in Fig. 3.

The structure of the UPPAAL model is built up using the components shown in Fig. 3. Each component is represented in the UPPAAL model as an automaton. As an example, consider the automaton shown in Fig. 4, which describes the behavior of the *PresenceHandling* component. In order to interpret this automaton, the following extra information is required:

- interfaces (see Fig. 3)
- parameters (see Table 1.)
- additional automata to ensure simulation and model checking in UPPAAL (see next paragraph).

To make the simulation and model checking in UPPAAL, the specifics of the runtime environment must also be

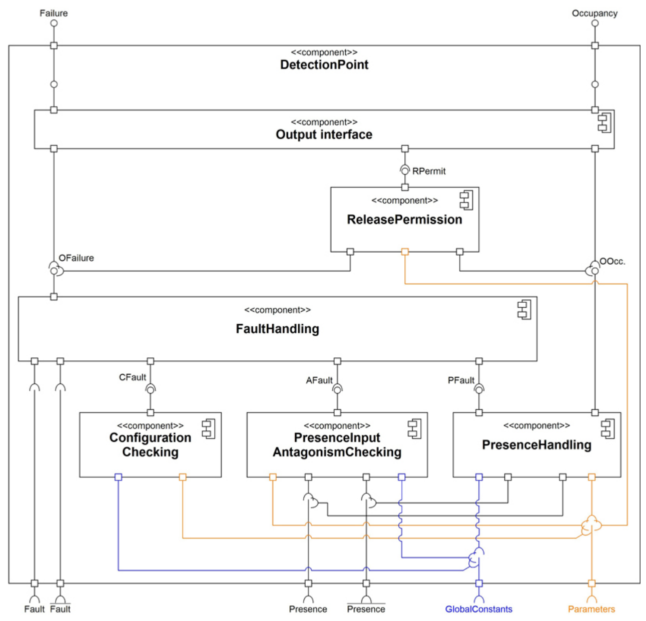

**Fig. 3** The internal structure of the detection point

input function (C). For example, in Fig. 4, the interfaces to these additional automata are a channel (ALLOWEDRUN) and a variable (PERMISSION).

The *PresenceHandling* automaton (Fig. 4) consists of two primary states: free and occupied (which itself consists of six sub-states named as "occ_..."). The system can enter the free state if it identifies a 'false' on both of its presence inputs (in_presence_p, in_presence_n). Depending on the type of occupancy from which the DP enters the free state, it can set a failure at its output (set PFault to 'true'). When a presence ('True') is detected on one of its presence inputs, the DP enters an occupied state. The timer "To" (occupancy time) is activated when the DP is in use. During usage, the DP remains in one of the six "occ_..." states depicted in Fig. 4, which is determined by the configuration (refer to Table 1) and the timer "To".

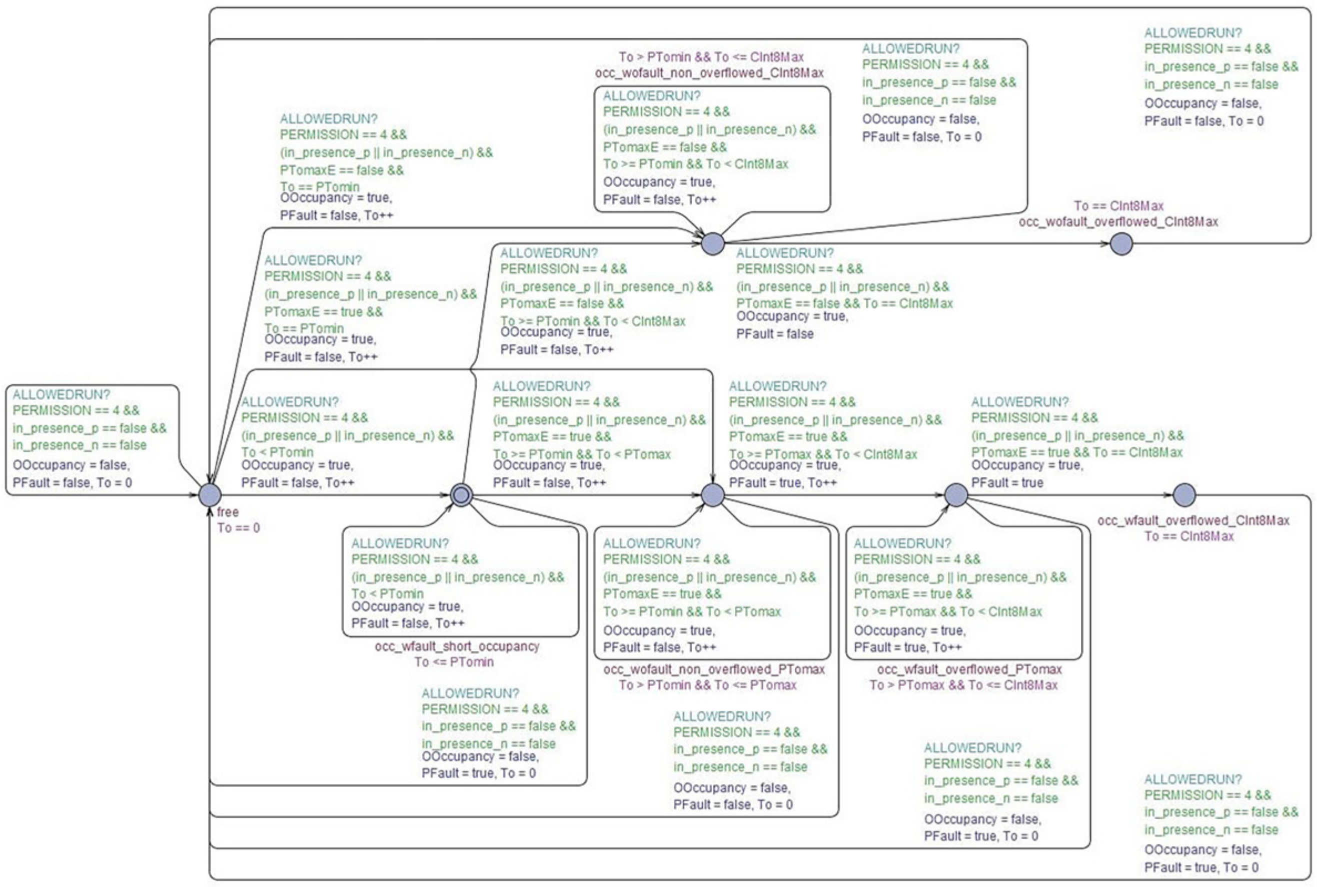
To > PTomin && To <= CInt8Max
occ_wofault_non_overflowed_CInt8Max
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = false, To = 0
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = false, To = 0
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == false && To >= PTomin && To < CInt8Max OOccupancy = true, PFault = false, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == false && To == PTomin OOccupancy = true, PFault = false, To++
To == CInt8Max
occ_wofault_overflowed_CInt8Max
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To == PTomin OOccupancy = true, PFault = false, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == false && To >= PTomin && To < CInt8Max OOccupancy = true, PFault = false, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == false && To == CInt8Max OOccupancy = true, PFault = false
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = false, To = 0
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && To < PTomin OOccupancy = true, PFault = false, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To >= PTomin && To < PTomax OOccupancy = true, PFault = false, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To >= PTomax && To < CInt8Max OOccupancy = true, PFault = true, To++
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To == CInt8Max OOccupancy = true, PFault = true
free
To == 0
occ_wfault_overflowed_CInt8Max
To == CInt8Max
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && To < PTomin OOccupancy = true, PFault = false, To++
occ_wfault_short_occupancy
To <= PTomin
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To >= PTomin && To < PTomax OOccupancy = true, PFault = false, To++
occ_wofault_non_overflowed_PTomax
To > PTomin && To <= PTomax
ALLOWEDRUN? PERMISSION == 4 && (in_presence_p || in_presence_n) && PTomaxE == true && To >= PTomax && To < CInt8Max OOccupancy = true, PFault = true, To++
occ_wfault_overflowed_PTomax
To > PTomax && To <= CInt8Max
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = true, To = 0
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = false, To = 0
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = true, To = 0
ALLOWEDRUN? PERMISSION == 4 && in_presence_p == false && in_presence_n == false OOccupancy = false, PFault = true, To = 0

**Fig. 4** Automaton for handling of presence (PresenceHandling)

included in the formal model. For this purpose, we use the environmental model described in Section 3. We include three additions in the UPPAAL model of the detection point to describe the essential features of the runtime environment: time handling (A), execution control (B), and an

The state occ_wfault_short_occupancy indicates that the detection point was occupied for an exceedingly brief duration. In practice, it is physically impossible for a train to cause such a short occupancy. For DP, this

**Table 1** Parameter settings of DP component for model checking

| Id. | Brief description | Setting |
|---|---|---|
| PTopn | Maximum time allowed for antagonism between Presence inputs. | 10 [s] |
| PTomin | Minimum time of train presence within the range of the detection point. | 20 [s] |
| PTomax | Maximum time of train presence within the range of the detection point. | 50 [s] |
| PTomaxE | Existence of upper limit Tomax. | true |
| PTr | Release preparation time. | 10 [s] |

means that the timer value To was below the minimum parameter (PTomin) while both presence inputs were free (PFault 'true'). If there is an upper limit to the occupancy time (as indicated by PTomaxE being 'true'), the automaton may enter the state occ_wfault_overflowed_PTomax as well. The value of PTomax value indicates that the DP has been occupied for an extraordinary extended period of time. In practice, a train cannot occupy the detection point for such a long time. In the case of the DP, if the value of the To timer exceeds the PTomax parameter and the DP transitions to the free state, a fault will be triggered on its output at the same time (with PFault set to 'true'). Until the point at which To reaches the maximum value for an 8-bit integer (PTomaxE is 'true'), the DP component will provide precise data to the diagnostic regarding the duration of the presence (refer to the status occ_wfault_overflowed_CInt8Max). There may be occasions where it is unnecessary to use an upper limit for the time To. In these instances, if train presence is detected, the DP will utilize the states occ_wfault_non_overflowed_CInt8Max and occ_wfault_overflowed_CInt8Max. Finally, if the DP has correctly detected the occupancy based on the configuration (PTomaxE can be either 'true' or 'false'), the train presence remains in the state occ_wfault_not_overflowed_PTomax. From this state, the DP can enter the free state if there is no fault (i.e. with PFault 'false').

Another important automaton for the case study is *OutputHandling* (see Fig. 5 which corresponds to the Output Interface component in Fig. 3). This automaton sets the outputs of the DP component that can be used by other components connected to the DP. To carry out this operation, the *OutputHandling* automaton interfaces with the *PresenceHandling* automaton (and its OOccupancy output), the *FaultHandling* automaton (and its OFailure output), and the *ReleasePermission* automaton (and its RPermit output). The output of the DP can indicate a failure together with occupancy (the failure_occupied state), and the absence of a failure together with an unoccupied or occupied state.

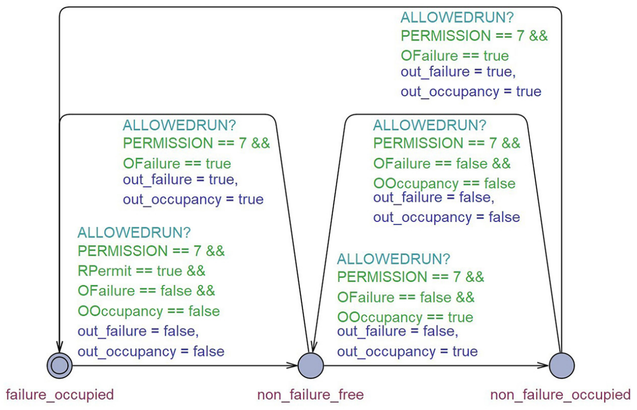

**Fig. 5** Automaton for handling of output (OutputHandling)

The parameters of the DP model have significant impact on model checking. For certain parameter sets, part of the state space may even be inaccessible – so recording these settings is a prerequisite for model checking reproducibility. The model parameters used during model checking are listed in Table 1.

In practice, it is not sufficient to perform validation and verification with only a well-defined set of system parameters. Another known difficulty is that there can be an incredible number of parameter configurations (over $10^8$ variations in our case study). To deal with this problem, we recommend using the approaches suggested by ISTQB (Graham, et. al., 2008): *equivalence partitioning* and *boundary value analysis*. In this paper, we do not present these principles in detail, but only describe the model checking results in addition to the configuration according to Table 1.

Finally, a computer with an Intel® Core™ i5–7200U CPU and 8 GB of memory was used for model checking. The configuration of UPPAAL during the model checking was: breadth-first search order (BFS), conservative state space reduction, Difference Bounds Matrices (DMB) state space representation, no diagnostic trace, automatic extrapolation, 16 MB hash table size, and reuse and parametric comparisons were enabled.

### 4.1 Validation of the model

As mentioned before, we validated the model of the DP object for deadlock freedom and the reachability of the states. In this section, we illustrate the examination of these properties through an example. The special CTL formulas in the following sections are given using the UPPAAL textual notation.

Deadlock freedom is typically a basic expectation against the models of systems in continuous operation, where planned stopping (reaching a deadlock) occurs rarely (or not at all) and is not modeled. Deadlock freedom means that there is no such state that has no subsequent state(s). In the corresponding CTL formula F1, $s_i$ represents the states belonging to the state space S ($S$:{$si$}), and next ($s_i$) represents the set of consecutive states from the state $s_i$.

$$\text{F1}: AG\left(\left|\text{next}\left(s_i\right)\right|\right) > 0$$

In UPPAAL, we can check the deadlock freedom with query where deadlock is a reserved keyword in the tool: `A[] not deadlock`. After executing this query on the model of the DP case study, model checking confirmed that the property is satisfied.

The reachability of particular states of the model can be checked using the CTL formula F2, where state$_i$ is the particular state being checked.

$$\text{F2}: EF\ \text{state}_i$$

The interpretation of formula F2 is that there exists a path in the state space graph where a reachable state has the identifier state$_i$. In other words, formula F2 asserts the existence of an executable path within the system where state$_i$ is included at some point along the path. If no such path can be found, then the proposition is false, indicating that state$_i$ is not reachable. As an example, checking the reachability of the "free" state of the PresenceHandling automaton of the DP case study can be done with formula `E<> presencehandling.free` in UPPAAL. After executing this query on the DP model, the result of the model checking showed that the property is satisfied.

Using formula F2, we checked the reachability of all states of the DP. In summary, this meant checking the reachability of 20 states of the DP case study (based on functional specification) and checking the reachability of the states of the supplementary automata (runtime environment, see Section 4), an additional 11 states. In total, this meant executing 31 queries on the model.

### 4.2 Verification of the functional requirements

The informal functional requirements for the detection point were described in Paragraph 1 of Section 4 in italics. These are high-level system requirements. In other words:

1. When the detection point detects a rail vehicle within its range, it emits an "occupied" signal at its output.
2. If the detection point does not detect a rail vehicle within its range, it emits a "free" signal at its output.

To perform the model checking in UPPAAL, we need to transform the above requirements. There is a detailed description of the transformation steps in the paper (Lukács and Bartha, 2022b). The UPPAAL textual form of the CTL requirements are:

- Q1: `A[] (in _ presence _ p == true || in _ presence _ n == true) imply out _ occupancy == true`
- Q2: `A[] (in _ presence _ p == false && in _ presence _ n == false) imply out _ occupancy == false.`

In the following, we will first present the results of the model checking of property Q1 and then property Q2 and the lessons learned.

The result of checking property Q1 was that the property is not satisfied. The reason is that, due to the structure of the model (multilevel/hierarchical processing), the expected result does not appear on the output immediately - more precisely, in the next step. At the beginning of a given step, the InputGenerator automaton (see Section 4, input function (C)) of the model sets the inputs, but the result can be emitted to the outputs (out_occupancy) only after the OutputHandling automaton (see Fig. 3) has run (at the end of the given cycle). Therefore, the "running time"of the cycle must also be included in the CTL expression to obtain the expected result during model checking. More precisely, the condition ISRUN == true (ISRUN is a global variable) must be included in the property to be checked. Note that the paper (Lukács and Bartha, 2022a) describes the structure of the model in detail.

The lesson from this is that the requirements must be adapted to the model, since the model usually also reflects some kind of implementation. However, even at this stage of the system design, we do not necessarily have all the information about the system architecture and the planned implementation solutions (only the requirements are known). Considering what has been described, property Q1 needs to be completed before model checking. The result of property modification will be according to UPPAAL formula:

Q3: `A[] ((in_presence_p == true || in_presence_n == true) && ISRUN == true) imply out_occupancy == true.`

When model checking is performed on property Q3, the result will be as expected: the property is satisfied.

In the case of model checking based on property Q2, the UPPAAL framework also provided us with a counterexample. We analyzed the counterexample in detail by tracking states, variables, and clock variables during each step (more precisely in each cycle). The counterexample occurred when the 12 cycles were executed. At the beginning of the 12th cycle, the free state was simulated at the inputs of the detection point, but the occupied output was still observable at its outputs, i.e., the input change did not take effect immediately. In the counterexample, the setting of the free state on the inputs was represented by the following lines:

- Pre: (*run, step1, generation, config_ok, non_antagonism, free, non_faulty, release_allowed, non_failure_free*),
- Tr: *ALLOWEDRUN: runcontrol→inputgenerator[15]*,
- Res: (*run, step2, generation, config_ok, non_antagonism, free, non_faulty, release_allowed, non_failure_free*).

In the list above you can see the preceding state (Pre), the state resulting due to the transition (Res) and the state transition (Tr) that led to the counterexample. For the states Pre and Res, each list represents the current state of each automaton in the DP model (see Fig. 3 or in detail in (Lukács and Bartha, 2021)).

We can see the states of the automata implemented in the UPPAAL framework listed in parentheses (see Pre and Res), and that the execution control automaton has given the instruction to the input generator automaton to simulate the occupancy at the input of the detection point (for this the input generator automaton has randomly chosen the input combination [15]). The next line of the counterexample reflects that the input setting has not yet happened (i.e., the effect of changing the input did not immediately take effect in the modeled system, the output of the detection point remained 'free').

The result of the model checking of property Q2 was that the property is not satisfied. Analyzing the counterexample, we found that besides the addition of the property presented in the previous example ('ISRUN == true'), other additions were also needed in this case. The reasons for this are described below.

The detailed design, specification, and modeling were the result of an iterative process of changing the functional requirements of the component (requirements analysis). Because of this, the "fault" inputs also appeared in the specification, and the specification was also supplemented with the release preparation time. As a result, even if the occupancy and error inputs are appropriate, the occupancy output of the evaluation element cannot be free until the release time has expired. In this case, these have influenced the state of the occupancy outputs of the detection point. Therefore, these new conditions must also be included in the original requirement. In addition, changes to the requirements must also be negotiated with the customer. The formula produced by the analysis of the requirements can be given in UPPAAL as follows:

Q4:
```
A[] ((in_presence_p == false &&
in_presence_n == false &&
in_fault_p == false && in_fault_n
== false) && ISRUN == true &&
releasepermission.release_allowed)
imply out_occupancy == false.
```

When model checking is performed on property Q4, the result is as expected: the property is satisfied.

## 5 Discussion

The purpose of this section is to summarize our experience in model checking the case study described in Section 4.

In practice, the design, specification, and development of models usually follow an iterative development process (there may be exceptions for trivially simple systems). To achieve efficient model checking, it is advisable to ensure that, in addition to the artifacts mentioned above, the requirements specification is also properly maintained.

For model checking (and simulation), it is not enough to model the functional requirements, the model must also embody a possible implementation solution. At this level of system design, in most cases only the requirements are known, so it is advisable to choose simple mechanisms for this purpose. Also, the choice of the future implementation solution and its modeling will typically not coincide with the implemented system. It is always necessary to consider that the chosen implementation solution may affect the results of the model checking.

We have identified two basic cases:

3. the modeled implementation solution can automatically (by itself) ensure that certain properties are met,
4. the modeled implementation solution may also contain solutions that need to be known and understood in order to correctly complement the functional requirements with these elements incorporated into the model that reflect the future implementation (the requirements need to be adapted to the model).

In addition to the above conditions, there may also be limitations due to the applied framework (in this case: UPPAAL). For UPPAAL, one such limitation is that the 'next' (next-state/next-time) or 'X' CTL operator has not been implemented. The output of model checking can also be affected by the parameters of the component or system being modeled – e.g., because there may be a set of parameters for which a part of the state space is inaccessible. Fortunately, there are widely known techniques (e.g., equivalence partitioning and boundary value analysis) for dealing with large parameter spaces, which can also be used during model checking.

If the investigated property is not satisfied during model checking, some frameworks (e.g., UPPAAL) can provide counterexamples in various forms. Examining and interpreting counterexamples is not trivial in most cases, and their interpretation requires, among other things, taking into account all of the clauses mentioned above.

Another lesson is that neither the UPPAAL tool (nor any other tool) can determine the expected result of model checking, it has to be defined by the user. Also, the interpretation of any counterexample is left to the user.

In summary, railway engineers should expect the application conditions described in this section when using model checking in the FMBRSE framework.

Finally, we have found that in many cases the informal requirements look good, but they are not precise enough, and this only becomes apparent during modeling/model checking. Therefore, even if we try to hide the formal details from the domain engineer, it is in vain. In our view, it is essential to use formal methods and formal descriptions at least in the requirements analysis phase, because this is where the most difficult and costly mistakes can be made, which will later be deeply embedded in the system design. As long as the requirements are not formally verified, the FMBRSE methodology cannot be used to effectively help the design process.

## 6 Conclusion

In this paper, we have presented the verification part of a formal model–based methodology that facilitates the construction of correct, complete, consistent, and verifiable functional specifications during development. The process we propose provides a specification/verification environment for railway engineers. Using this framework, they can achieve a higher quality functional specification compared to traditional development. The main advantage of the described approach is that it hides the formal method–related details from the railway engineers, so that they can acquire a formal verification result without learning the necessary mathematical background.

Our experience during the development of the case study is that the application of model checking is only possible under several restrictions that the domain engineers have to consider. These limitations may also depend on the method itself, the tool used, and the system being modeled, as well as the possible implementation solutions.

We have found that the main difficulty for railway engineers is the preparation of the requirements 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 problem, we started to develop an intermediate domain-specific restricted textual language for the railway field.

Considering the experiences described in this paper, the proposed methodology proved to be suitable for the design of electronic railway control systems. A high-quality functional specification, written by railway engineers at the system development level, can be achieved using formal models and model checking.

## References

Alshalalfah, A.-L., Mohamed, O. A., Ouchani, S. (2023) "A framework for modeling and analyzing cyber-physical systems using statistical model checking", Internet of Things, 22, 100732.
https://doi.org/10.1016/j.iot.2023.100732

Arcile, J., Devillers, R., Klaudel, H. (2019) "VERIFCAR: a framework for modeling and model checking communicating autonomous vehicles", Autonomous Agents and Multi-Agent Systems, 33, pp. 353–381.
https://doi.org/10.1007/s10458-019-09409-x

Baier, C., Katoen J.-P. (2008) "Principles of model checking", [pdf] The MIT Press Cambridge, Massachusetts London, England, ISBN: 9780262026499.

Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H. (2008) "Compositional verification for component-based systems and application", International Symposium on Automated Technology for Verification and Analysis, Springer, Heidelberg, Berlin.
https://doi.org/10.1007/978-3-540-88387-6_7

CENELEC (2011) "EN 50128 Railway applications – communication, signalling and processing systems – software for railway control protection systems", Management CentreBrussels, Belgium.

CENELEC (2018) "EN 50129 Railway applications – communication, signaling and processing systems – safety related electronic systems for signaling", CEN-CENELEC Management CentreBrussels, Belgium.

CENELEC (2017a) "EN 50126-1 Railway applications – the specification and demonstration of reliability, availability, maintainability and safety (rams) – part 1: generic RAMS process", CEN-CENELEC Management CentreBrussels, Belgium.

CENELEC (2017b) "EN 50126-2 Railway applications – the specification and demonstration of reliability, availability, maintainability and safety (rams) – part 2: systems approach to safety", CEN-CENELEC Management CentreBrussels, Belgium.

Chatterjee K., Doyen L. (2016) "Computation tree logic for synchronization properties", 43rd International Colloquium on Automata, Languages, and Programing (ICALP 2016).
https://doi.org/10.48550/arXiv.1604.06384

Cimatti, A. Clarke, E. Giunchiglia, E. Giunchiglia, F. Pistore, M. Roveri, M. Sebastiani, R. Tacchella, A. (2002) "NuSMV 2: an OpenSource tool for symbolic model checking", International Conference on Computer Aided Verification, pp. 359–364, Berlin, Germany.
https://doi.org/10.1007/3-540-45657-0_29

Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R. (2018) "Handbook of model checking", Springer International Publishing AG, Cham, Switzerland, pp. 1–1197, ISBN 978-3-319-10575-8
https://doi.org/10.1007/978-3-319-10575-8

Friedenthal, S., Moore, A., Steiner, R. (2011) "A practical guide to SysML: the systems modeling language", Morgan Kaufmann Publishers Inc., Elsevier, (2nd ed.). ISBN: 9780123852076
https://doi.org/10.1016/C2013-0-14457-1

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, ISBN: 9781118459898
https://doi.org/10.1002/9781118459898.ch4

Graham D., van Veenendaal E., Evans I., Black R. (2008) "Foundations of software testing: ISTQB certification", Cengage Learning EMEA, edition 4th, ISBN: 1473764793.

Laursen, P. L., Trinh, V. A. T., Haxthausen, A. E. (2020) "Formal modelling and verification of a distributed railway interlocking system using UPPAAL", In: Margaria, T., Steffen, B. (eds) Leveraging Applications of Formal Methods, Verification and Validation: Applications, Springer, Cham, pp. 415–433, ISBN: 978-3-030-61467-6
https://doi.org/10.1007/978-3-030-61467-6_27

Lukács, G., Bartha, T. (2021) "Formal modelling of level crossing system for trams using UPPAAL framework", Műszaki Szemle (Technical Review), Published by EMT as Erdélyi Magyar Műszaki Tudományos Társaság (in English: Hungarian Technical Scientific Society of Transylvania), (EMT) 77, pp. 18–37. (In Hungarian) [online] Avaiable at: https://emt.ro/kiadvanyok/muszaki-szemle [Accessed: 23 May 2024]

Lukács, G., Bartha, T. (2022a) "Formal modeling and verification of the functionality of electronic urban railway control systems through a case study", Urban Rail Transit 8, pp. 217–245.
https://doi.org/10.1007/s40864-022-00177-8

Lukács G. Bartha T. (2022b) "Transformation domain requirements specification into computation tree logic language", 2022 IEEE 1st International Conference on Cognitive Mobility (CogMob), Budapest, Hungary, 2022, pp. 73–78. ISBN: 9781665476317
https://doi.org/10.1109/CogMob55547.2022.10117911

Lukacs G., Bartha T. (2022c) "Practical UML subset for railway engineers to support formal modeling", International Scientific Journals, Trans and Motauto World 7(2), pp. 56–59.

Nanda, S. P., Grant E. S. (2019) "A survey of formal specification application to safety critical systems", IEEE 2nd International Conference on Information and Computer Technologies (ICICT), Kahului, HI, USA, pp. 296–302.
https://doi.org/10.1109/INFOCT.2019.8711369

Pakonen, A., Tahvonen, T., Hartikainen, M., Pihlanko, M. (2017) "Practical applications of model checking in the Finnish nuclear industry", Proceedings of the 10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies, San Francisco, CA, USA, pp. 1342–1352.

Roggenbach, M., Cerone, A., Schlingloff, H., Schneider, G., Shaikh, S. A. (2022) "Formal methods for software engineering: languages, methods, application domains", 1st ed., Springer, ISBN: 978-3-030-38799-0
https://doi.org/10.1007/978-3-030-38800-3

Vyatkin V., Hanisch H.-M. (2001) "Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems", ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation, Proceedings (Cat. No.01TH8597), Antibes-Juan les Pins, France, pp. 113–118.
https://doi.org/10.1109/ETFA.2001.997677