What is the Point: Formal Analysis and Test Generation for a Railway Standard

Mark Bouwman[1], Djurre van der Wal[2], Bas Luttik[1], Mariëlle Stoelinga[2], and Arend Rensink[2]

[1] *Formal System Analysis Group, Eindhoven, University of Technology, The Netherlands.*
E-mail: {m.s.bouwman, s.p.luttik}@tue.nl

[2] *Formal Methods & Tools Group, University of Twente, The Netherlands.*
E-mail: {d.vanderwal-1, arend.rensink, m.i.a.stoelinga}@utwente.nl

EULYNX is an EU-level collaboration between railway infrastructure managers to standardize signaling interfaces. The main goal of EULYNX is to provide, on an EU scale, a modular and flexible railroad architecture where components can easily be exchanged. This also opens the market for specialized manufacturers that do not supply the full range of control assets, but only single components.
Related to EULYNX is FormaSig, an effort to establish the safety of the EULYNX standard with mathematical rigor. In particular, one of the main objectives of FormaSig is to translate the entire EULYNX standard from the semi-formal language SysML to the formal language mCRL2. The resulting mCRL2 models will subsequently be checked for important safety requirements and used for automated testing of actual EULYNX systems.
This paper presents a first case study in this direction, focusing on the EULYNX Point interface, which we have converted to an mCRL2 model. We have also derived nine safety requirements, which have all been automatically compared with the mCRL2 model. Finally, we have used the mCRL2 model to test an industrial simulator of the EULYNX Point interface fully automatically.

Keywords: Formal analysis, mCRL2, Railway systems, SysML, Test automation.

1. Introduction

EULYNX[a] is an initiative involving more than ten European railway infrastructure managers to reduce the cost and installation time of signaling equipment. This is done through standardization of the interfaces between the *interlocking* – the central device that controls most of the signaling infrastructure – and *field elements*, such as signals, points, and level crossings: under the new paradigm, the interlocking communicates over an IP network with an object controller that steers the field element (see Figure 7). The standardization efforts should improve the interoperability between components from different suppliers, and thus lead to a significant reduction of the life cycle costs of signaling systems.

For obvious reasons, infrastructure managers are eager to include EULYNX to their system development cycle, during which EULYNX-based systems will be rigorously validated and verified through a multitude of methods. Such methods can be described as *informal*, *static*, *dynamic*, or *formal* (Debbabi et al., 2010), but despite a significant amount of research into the application of formal methods in the area of safety-critical railway systems (Fantechi, 2013) (Fantechi et al., 2014) (Basile et al., 2018) – focusing on the logic of the interlocking, in particular (James et al., 2014) (Haxthausen and Peleska, 2015) (Bonacchi et al., 2016) – only informal, static, and dynamic methods can be argued to be firmly integrated in contemporary design procedures. The EULYNX project presents an opportunity to resolve obstacles to formal methods being deployed more fully, which has evolved into *FormaSig*[b], a collaboration between academia (Eindhoven University of Technology, University of Twente) and railway infrastructure managers (ProRail, DB Netz AG).

In FormaSig, we use formal techniques to address two major concerns that we identified for infrastructure managers: (i) the possibility of weaknesses in the EULYNX standard – whether they be ambiguities or violated safety requirements – that persist until the production stage; and (ii) the task of testing EULYNX-based systems for conformance to the EULYNX standard, which is laborious and error-prone when performed manually. The approach of FormaSig is inspired by Bouwman et al. (2019), in which the model of an interlocking in the formal language mCRL2 (Bunte et al., 2019) was used for both automated requirement checking with the mCRL2 toolkit and

[a] https://www.eulynx.eu

[b] Formal Methods in Railway Signaling Infrastructure Standardization Processes

Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference
Edited by Piero Baraldi, Francesco Di Maio and Enrico Zio
Published by Research Publishing, Singapore.
ISBN: 978-981-14-8593-0; doi:10.3850/978-981-14-8593-0_4410-cd

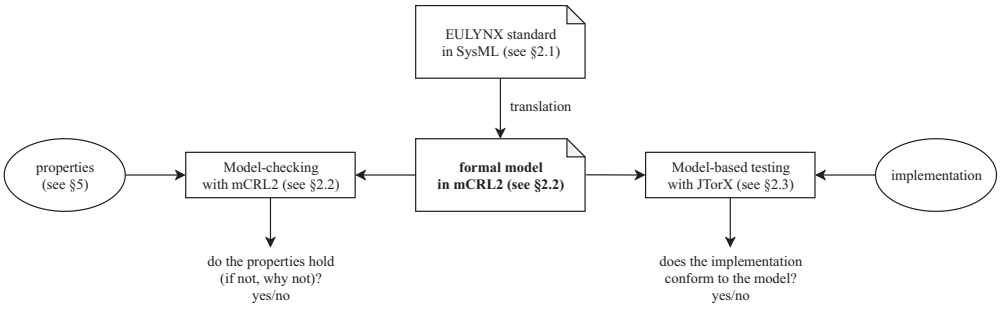

Fig. 1. FormaSig setup. The formal model is used for property analysis and for implementation testing.

automated testing with JTorX (Belinfante, 2014). In order to make the approach accessible to formal non-experts, we extend it with a translation from the semi-formal EULYNX specifications to mCRL2, which is done manually at the current stage of FormaSig but should ultimately become an automated process. An overview of the approach can be found in Figure 1.

This paper presents an early case study, in which we applied the FormaSig approach to the EULYNX interface for the Point subsystem in order to discover the more nuanced challenges of the FormaSig project. Among other things, this resulted in the exposure of several weaknesses in EULYNX standard (such as when events that are triggered by one system component are observed by another system component), of which we notified the responsible parties. In the big picture, this paper contributes to the body of research into applying formal methods in the railway industry. Notable of our work is that (i) the derived formal models are used for both verification *and* automated testing, (ii) a railway *standard* is analyzed rather than a railway *system*, and (iii) formal models are derived directly from specifications that are composed in a semi-formal language.

Our paper is organized as follows: the basics of the EULYNX standard, the mCRL2 toolkit, and model-based testing are explained in Section 2; we give an overview of the EULYNX Point interface in Section 3; Sections 4 to 7 describe the different components of the case study – namely formalization, requirement elicitation, formal verification, and model-based testing – and the discussion, conclusions, and future work can be found in Sections 8 and 9.

2. Preliminaries

This section explains the basic concepts behind the EULYNX standard, the mCRL2 toolkit, and model-based testing.

2.1. *The EULYNX standard*

EULYNX defines the interfaces of ten crucial signaling systems, ranging from level crossings to other nearby interlockings. The interfaces are defined primarily with a custom (still maturing) variant of the graphical semi-formal language *SysML*[c]. SysML – which stands for 'Systems Modeling Language' – is a popular systems engineering modeling language that is closely related to the ubiquitous Unified Modeling Language (UML). SysML defines nine different diagram types, several of which are extended versions of UML diagram types, and two of which are relevant to our case study.

Internal block diagrams (IBDs) are used to show which data can be communicated to other components of a system. This is done by drawing components as *blocks* and by drawing data that is consumed/produced by those blocks as inward/outward *ports* on the borders. A connection from an outward port to an inward port means that the data that is produced by the former is consumed by the latter. Figure 2 shows an example.

The behavior of a block is defined in EULYNX by one or more *state machine diagrams*, which is the second diagram type in EULYNX. State machine diagrams – or simply 'state machines' – make use of *states* that are connected by arrows that define behavior (*transitions*). Such behavior is only enabled when its transition starts at the currently active state, but there is also exit/entry-behavior, which is executed when a state becomes active/inactive. States can also contain other states, and multiple contained states can be active simultaneously if they belong to different *parallel regions* of the containing state.

Component behavior itself is expressed with ASAL, the Atego Structured Action Language[d].

[c] https://www.omg.org/spec/SysML/

[d] https://support.ptc.com/help/modeler/r9.0/en/index.html#page/Integrity_Modeler%2Fsysim%2FSySim_Atego_structured_action_language.html%23

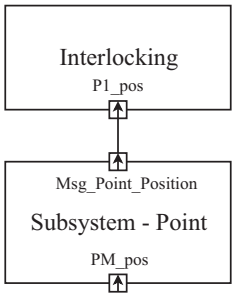

Fig. 2. Example internal block diagram. 'Subsystem - Point' exposes its 'Msg_Point_Position' data to the 'Interlocking' component; the 'Interlocking' component references the incoming data as 'P1_pos'. There is also an unconnected port, 'PM_pos'; this port is either connected to a block in another internal block diagram, or it defines a connection with the system environment.

The choice for ASAL is EULYNX-specific: SysML does not give an explicit behavioral language. ASAL is a mostly straightforward imperative programming language, including conditional statements, while loops, and basic operations on Booleans, integers, and strings. See Figure 3 for an example.

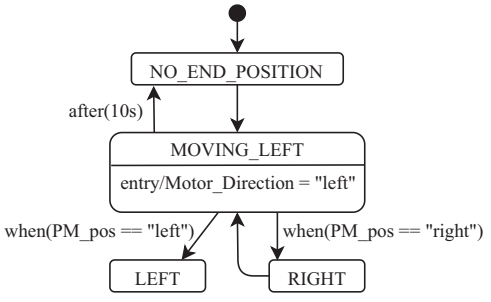

Fig. 3. Example state machine. Initially, the component is in the 'NO_END_POSITION' state, moving to the 'MOVING_LEFT' state at an arbitrary moment. Upon entering the new state, 'Motor_Direction' (an outward port of the block) is set to a new value. Once there, the component waits until 'PM_pos' changes to "left" or "right", moving to the 'LEFT' or 'RIGHT' state, accordingly. It will not wait longer than 10 seconds, however, because then the 'after(10s)' event forces the component back to the 'NO_END_POSITION' state.

2.2. *mCRL2*

mCRL2 is a formal modeling language with an associated toolset. It has an exact mathematical interpretation – making it suitable for describing a system with the precision that is required for requirement verification – and it can be used from both the command-line interface and from a user-friendly GUI.

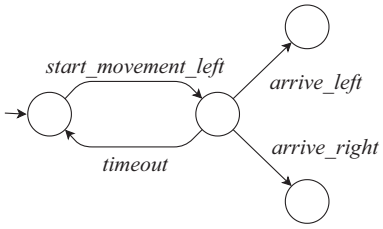

Fig. 4. Example of an LTS. The arrow without a label or source state points to the initial state. From this LTS it can be inferred that after starting a movement, either a timeout can occur (after which a new movement may be initiated) or we arrive at some position (the one we were moving towards or not).

The semantics of mCRL2 is based on *labeled transition systems* (LTSs), which consist of states and labeled transitions between those states. An example of an LTS is shown in Figure 4. Since an LTS is an explicit representation of all possible behaviors of a system, in practice it often has a huge – or even infinite – number of states and transitions. mCRL2 provides tools that assist with this problem.

Before requirements of a system can be verified with mCRL2, they must first be expressed in the so-called *modal μ-calculus* (Kozen, 1983), which just like mCRL2 has a precise LTS-based interpretation. The mCRL2 toolkit can subsequently be used to check whether a μ-calculus formula holds for a given mCRL2 model (it provides a counter-example when this is not the case; see Figure 1).

2.3. *Model-based testing*

Model-based testing is a technique for automatically generating, executing, and evaluating tests (Tretmans, 1996) (Haxthausen and Peleska, 2015). Its main prerequisite is the availability of a system model in the form of an input/output-LTS, or *IOLTS*: an extension of an LTS in which a distinction is made between inputs (by convention given names that end with a '?') and outputs (with names that end with a '!'). Figure 5, for example, shows an input-output transition system similar to the LTS from Figure 4.

Tests that are generated from an IOLTS are essentially *decision trees* that track which stimuli ('?') are sent to a system and which responses ('!') are expected. Branches always end with a **pass** or **fail** verdict: when a test reaches a **pass** verdict, it terminates (and another test can begin); when a test reaches a **fail** verdict, the tested system does not conform to the model, and testing is typically stopped altogether.

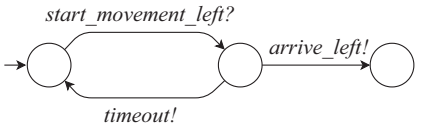

Fig. 5. Example of an IOLTS. It has one input, namely *start_movement_left?*, and two outputs, namely *arrive_left!*, and *timeout!*.

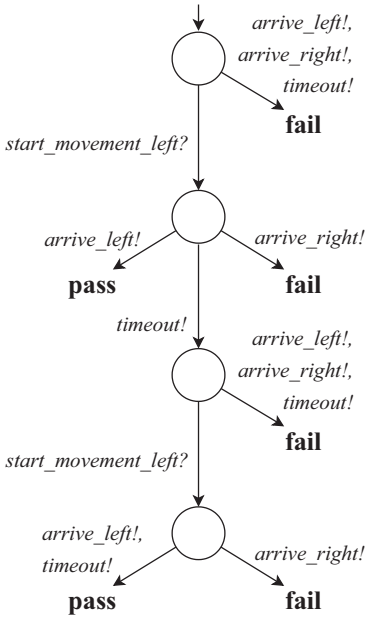

Fig. 6. Example of a test that could have been generated from the IOLTS from Figure 5, depicted as a decision tree.

Figure 6 depicts the decision tree of a test for the IOLTS in Figure 5. At the start of this tree, one input is accepted and all outputs are rejected; then, *arrive_left!* and *timeout!* are accepted (but only *arrive_left!* ends the test) whereas *arrive_right!* is rejected; and after *timeout!*, the behavior can be tested a second time. Note that if we were to use the decision tree to test a system that is described by the LTS in Figure 4 (ignoring missing '?' and '!'), the system would receive a **fail** verdict if it would move along the *arrive_right* transition (but this does not necessarily happen).

In this paper, we use the model-testing tool JTorX (Belinfante, 2014), which supports test generation for mCRL2 models.

3. Case study: Point

A point is a common railway element that makes it possible to split one railway track into two railway tracks – or, conversely, to merge two railway tracks into one railway track. Typically, a point is implemented with two rails that are moved laterally between two *end positions* (left and right) by the motors of one or more point machines. Depending on the end position of the point – which is detected by sensors – one of the two branching railway tracks is the current destination/origin track of an approaching train.

Under EULYNX, a field element such as a point is controlled by an *object controller*: a component that manages a field element locally, communicating with the interlocking (the central computer that controls the signaling equipment in an area) with a specific communication interface that is independent of the underlying implementation (see Figure 7). In other words, EULYNX specifies the interaction between the interlocking and the object controller, and not how the object controller should interact with its field element.

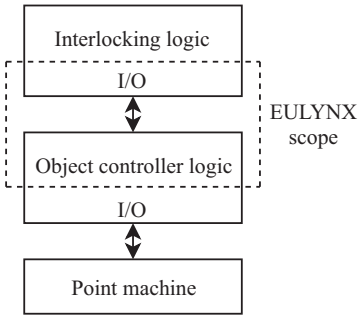

Fig. 7. Scope of the EULYNX point interface specification.

The EULYNX point interface specification is divided into a generic part and a specialized part: the former is the same for all field element types and includes behavior for situations such as connection start, incoming instructions, power loss, and timeouts; the latter describes the actual steering of the point and the reporting of its position. The two parts are mutually dependent: the specialized part can access the current connection status, for example, and the generic part is notified when the specialized part has transmitted its status to the interlocking.

The EULYNX point interface specification consists of 12 blocks and 131 communication channels that are distributed over five internal block diagrams. The behavior of these blocks is defined by nine state machine diagrams with a total of 64 states (between three and 13 states per state machine). Note that not all ports in the internal block diagrams are connected: the core interlocking logic, for example, is part of the environment, meaning that the position of the point is sent to the environment without specifying what the interlocking logic does with that information.

Unconnected input ports are assumed to always be able to receive a message, such as a request from the interlocking logic to move the point.

4. Formalization of the Point interface

In order to unlock the potential of verification and model-based testing, the SysML model of the Point interface needs to be converted to a formal model, or *formalized*, as shown in Figure 1. Formalization of a system that is specified in SysML is not a new challenge, by itself, including the development of tools for editing and analyzing SysML models (Linhares et al., 2007) (Debbabi et al., 2010) (Pétin et al., 2010) (Pedroza et al., 2011) (Pétin et al., 2010) (Chouali and Hammad, 2011) (Morkevicius and Jankevicius, 2015). Based on our estimation of existing work, however, there is no research into the dialect of SysML that is used in EULYNX, and we could not find a formalization in a format that is suitable for both model-checking and model-based testing (such as mCRL2). Developing a new formalization has therefore become part of our enterprise.

The decision of the EULYNX initiative to use the semi-formal language SysML for its specifications has advantages, such as its inviting imagery and expressiveness, but for us it also means that we are occasionally forced to make subjective choices in our interpretation. UML/SysML intentionally leaves the behavior of event buffers of state machines unspecified, for example; and the action language of EULYNX, ASAL, lacks a formal semantics completely, forcing us to consider whether or not sequences of ASAL instructions can be interrupted by incoming events from other state machines (which we permit in order to maximize the potential for unintended interactions).

We justify our interpretation as much as possible through discussions with experts in the railway signaling domain and by consulting the literature on other SysML formalizations (interestingly, these do not always align). In addition, we referenced literature on available semantics (Lilius and Paltor, 1999) (Varró, 2002) (Ober et al., 2011), algorithms (Gnesi et al., 2004), and overviews of ambiguities, contradictions, and underspecifications (Fecher et al., 2005).

To obtain a formal model we manually translated the state machine diagrams and the internal block diagrams to an mCRL2 model such that each state machine is represented by a parallel component in the mCRL2 language and each flow from the internal block diagrams is defined as a communication channel. The mCRL2 model therefore combines the behaviors of all state machines and all of their interactions.

5. Requirement elicitation for the Point interface

To assess the quality of the EULYNX standard we intend to verify requirements on all the execution paths of the system. Because of the safety-critical nature of railway signaling systems, such requirements should go beyond checks for reachability and deadlock-freedom. Eliciting these more advanced requirements is an explicit concern of the FormaSig project.

A challenge in formulating more advanced requirements is that the prerequisite knowledge is mostly split between the academic partners of FormaSig and the infrastructure managers: the former only possess the skills to formulate formal requirements, and the latter only possess the signaling domain knowledge. To overcome this challenge, we adopted an iterative process of requirement elicitation. Each iteration started with some initial requirements in natural language, which we refined based on feedback from interviews with signaling experts. We then attempted to verify the refined requirements with mCRL2, resulting in a better understanding of their context and meaning, and possibly resulting in even more refined requirements. We translated these requirements back to natural language, and interviewed signaling experts again.

For example, the following requirement was derived from an interview with a signaling expert: *"The object controller may only instruct the point machines to move when this is commanded by the interlocking."* In order to formalize this requirement in μ-calculus, it had to be expressed in terms of events, and so we added that *"a movement command from the interlocking allows the object controller to initiate a movement to a certain position until either a timeout occurs or when the end position is reached"*. When checking the mCRL2 model for this requirement, however, we found a counterexample in which the interlocking sends two movement commands for opposing end positions in a row; we resolved this issue by allowing the movement until *"either a timeout occurs, the end position is reached, or a movement command for the opposing position is received"*, instead. With this addition the requirement holds.

In total, we formulated nine requirements, seven relating to generic behavior and two specific for point behavior.

6. Formal verification of the Point interface

Once we have a formal model, we can use formal verification techniques to check the quality of the model and, by extension, the specification. To this end, we used the mCRL2 toolkit to verify the elicited requirements. The number of states in the LTS of the model containing both the generic and

point specific behavior, turned out to be huge (in excess of 5 billion states). We got around this by splitting the mCRL2 model into two models: one containing the generic behavior and one containing only the point interface behavior. We were able to check all requirements using these two models.

The statespace induced by the model containing only the point specific interface behavior consists of 1.518.070.128 states and 12.819.240.064 transitions. The two relevant requirements both hold, which took a week to verify. The statespace induced by the model containing only the generic interface behavior consists of 32.971.396 states and 332.333.594 transitions. Checking the seven relevant properties took 162 minutes, one property does not hold, all others do.

The mCRL2 toolset provided a counterexample for the property that does not hold: a part of the statespace that violates the requirement. This counterexample showed that a specific sequence of messages could lead to a deadlocked state: a state of the system after which no further behavior is possible. Further analysis revealed the cause of the deadlock. SysML does not prescribe how event buffers should be implemented and neither does EULYNX. Event buffers, which contain messages sent by other state machines, are assumed to be of fixed size in our model and can hence be full. The counterexample showed a situation where two state machines with a full buffer want to send something to each other. Since the buffers are full, these send actions cannot take place and the state machines will infinitely wait on each other to empty their buffer.

7. Model-based testing of the Point interface

The mCRL2 model combining generic field element behavior and Point-specific behavior was used to perform tests with JTorX on a third-party simulator that was derived from the same SysML diagrams. The mCRL2 model was customized to hide communications that are internal to the simulator and can hence not be observed during testing.

JTorX generates tests on-the-fly (online testing), but we chose to use offline test generation instead. This made it possible for us to re-use pregenerated test suites during the development of the test execution code, avoiding the time that JTorX requires for test generation (between 5 and 15 minutes for generating a sequence of $\sim$100 test steps). Because JTorX does not support offline test generation directly, we used AutoHotkey[e], a framework for automated interaction with the MS Windows GUI. Our AutoHotkey script applies stimuli by simulating keyboard and mouse inputs to JTorX's GUI, and it extracts responses by requesting the contents of control elements from the operating system and by determining the color of pixels at specific positions. Figure 8 gives an overview of the script's architecture.

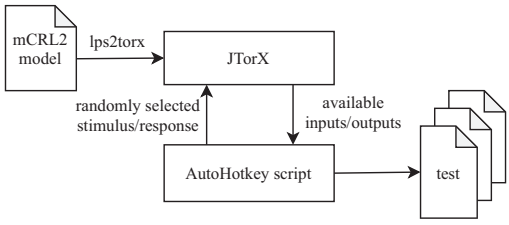

Fig. 8. Test generation setup.

Just like the test generation itself, we execute the generated tests with an AutoHotkey script; see Figure 9 for its setup. As a consequence of the offline test generation, the script may encounter an output in the contents of the simulator GUI that is accepted, but for which there is no subsequent behavior defined; in such cases, the AutoHotkey script terminates – prematurely, as it were – producing a **pass** verdict. It is also possible that outputs appear simultaneously in the simulator GUI but exist as separate outputs in the model; this has been resolved by capturing simulator outputs in a buffer and consuming them on demand.

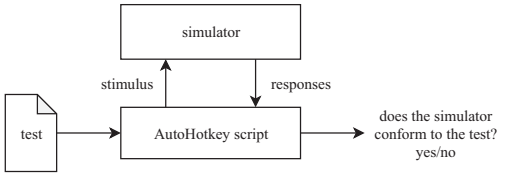

Fig. 9. Test execution setup.

For this paper, we generated 82 tests of $\sim$100 steps from the PDI/Point mCRL2 model and executed them. Test generation took $\sim$10 minutes on average; test execution took $\sim$ 24 minutes on average. The third-party simulator eventually passed all of the tests, although several iterations of test execution were required before the model and the simulator were properly integrated (the model assumed that the point would start in 'no end position', for example, whereas the simulator could also be in the 'left' or 'right' end position).

A third of all tests (24 out of 82) terminated prematurely with a **pass** verdict, completing about half of their test steps on average (note that the impact on the usefulness of our manner of test generation is limited because the mCRL2 model of the EULYNX point is fairly predictable: it does

[e] https://www.autohotkey.com/

not contain many diverging paths that produce different outputs for the same input).

8. Discussion

The case study in this paper is a step towards using unambiguous formal models to improve EULYNX standards and test delivered components, as shown in Figure 1. In this section we will reflect on the lessons learned, potential shortcomings and remaining challenges.

Let us begin by discussing the formalization. Firstly, as mentioned in Section 4, SysML contains a number of ambiguities. By choosing a specific interpretation of SysML and translating to a formal language, these ambiguities are no longer present. If the translation is also automatic, a unique unambiguous interpretation can be associated with each SysML model. However, whether our choices are consistent with the interpretations of signaling experts is uncertain, and requires further research.

Secondly, EULYNX defines all communication between state machines as asynchronous and buffered. Asynchronous communication is realistic for communication between, for example, the interlocking and the field element. However, in the case of communication between state machines within the same system other forms of communication – e.g. shared variables – might also be realistic; this would reduce the total number of states, making verification easier.

Finally, our models do not contain an aspect of time. A number of the requirements that we elicited are related to time: under certain conditions some output must be produced quickly. We verified these requirements by checking whether the desired output is produced within a finite number of steps in the model. It would be desirable to check the stronger requirement of whether the desired output is always produced within a specific time frame. At the moment, the EULYNX SysML models lack necessary information on the duration of events, such as the time it takes for a message to reach its destination.

Our approach of eliciting and formalizing requirements requires back and forth translation between natural language and the modal μ-calculus. Moreover, the final formal requirements cannot be read by signaling experts. The process of eliciting requirements would benefit from an intermediate requirements format: a (visual) format close to SysML from which a modal μ-calculus formula can be derived automatically. This would make formal requirements accessible to signaling experts.

With regard to the model-based testing in our case study, we mainly consider the performance of the test generation and test execution. Test generation took fairly long, although from the literature we surmise that this underperformance is not representative, and there are several changes that we could make to our test environment in order to resolve this (we may currently be running a non-optimally compiled version of the `lps2torx` tool, for example). Test execution is also time-consuming. Naturally, this could be alleviated in the case of a simulator by integrating the test environment directly at source code level, but we should anticipate that tests with live systems may be similarly sluggish, meaning that randomly generated tests may have to be abandoned in favor of more targeted testing methods.

Finally, we have not yet quantified the effectiveness of our proof-of-concept test setup. Our thoughts on how to proceed on that front can be found in the next section.

9. Conclusions and future work

We have shown that it is feasible to formalize EULYNX SysML models to mCRL2 and to subsequently perform verification and model-based testing. This case study has helped us to gain insights into the semantics of SysML and the ASAL action language and how they can be translated to mCRL2.

Future work of FormaSig will be directed at streamlining the application of formal methods in the context of EULYNX and making it more accessible for signaling engineers. The goal of FormaSig is that, eventually, formal methods will be applied to all interfaces standardized in EULYNX.

This will partly be achieved by automating the translation from SysML to mCRL2, which will eliminate the time needed to construct and maintain mCRL2 models by hand as well as prevent human translation errors. A likely challenge with an automated translation is that it might be more difficult to abstract from details to reduce the size of the state space. We will also put effort into translating counterexamples to SysML sequence diagrams, and finding a notation for requirements that can be understood by signaling experts and which is unambiguous enough to be directly translated to a modal μ-calculus formula. To do so, we are considering visual requirements languages with a formal semantics in the literature, such as Live Sequence Charts (Brill et al., 2004).

We also plan to investigate ways to improve our test setup. One possibility is to deploy *coverage-based testing* (Briones et al., 2006) (van den Bos and Tretmans, 2019), which is a model-based testing method in which tests are generated with the aim to reach all transitions in a model. We should also provide an indication of the thoroughness of our test methods, for example by using *mutant testing* (Jia and Harman, 2010) (counting the number of tests/steps that is required on average in order to determine that an incorrect implementation does not conform to a model).

Acknowledgements and disclaimer

FormaSig and this work (by extension) are fully funded by DB Netz AG and ProRail; at the same time, the vision illustrated in this article is not part of the strategy of DB Netz AG or ProRail, but reflects the personal views of the authors.

References

Basile, D., M. H. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, A. Piattino, D. Trentini, and A. Ferrari (2018). On the industrial uptake of formal methods in the railway domain. In *International Conference on Integrated Formal Methods*, pp. 20–29. Springer.

Belinfante, A. (2014). *JTorX: Exploring Model-Based Testing*. Ph. D. thesis, University of Twente, Enschede, Netherlands.

Bonacchi, A., A. Fantechi, S. Bacherini, and M. Tempestini (2016). Validation process for railway interlocking systems. *Science of Computer Programming 128*, 2–21.

Bouwman, M., B. Janssen, and B. Luttik (2019). Formal modelling and verification of an interlocking using mCRL2. In *Proceedings of FMICS 2019*, pp. 22–39. Springer.

Brill, M., W. Damm, J. Klose, B. Westphal, and H. Wittke (2004). Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification. In *Integration of Software Specification Techniques for Applications in Engineering*, pp. 374–399.

Briones, L. B., E. Brinksma, and M. Stoelinga (2006). A semantic framework for test coverage. In *Proceedings of ATVA*, pp. 399–414. Springer.

Bunte, O., J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse (2019). The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *Proceedings of TACAS 2019, Part II*, pp. 21–39.

Chouali, S. and A. Hammad (2011). Formal verification of components assembly based on SysML and interface automata. *Innovations in Systems and Software Engineering 7*(4), 265–274.

Debbabi, M., F. Hassaine, Y. Jarraya, A. Soeanu, and L. Alawneh (2010). *Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models*. Springer Science & Business Media.

Fantechi, A. (2013). Twenty-five years of formal methods and railways: What next? In *SEFM 2013 Collocated Workshops, Revised Selected Papers*, pp. 167–183.

Fantechi, A., F. Flammini, and S. Gnesi (2014). Formal methods for railway control systems. *Int. J. Softw. Tools Technol. Transf. 16*(6), 643–646.

Fecher, H., J. Schönborn, M. Kyas, and W. P. de Roever (2005). 29 new unclarities in the semantics of UML 2.0 state machines. In *Proceedings of ICFEM 2005*, pp. 52–65.

Gnesi, S., D. Latella, and M. Massink (2004). Formal test-case generation for UML statecharts. In *Proceedings of ICECCS*, pp. 75–84. IEEE.

Haxthausen, A. E. and J. Peleska (2015). Model checking and model-based testing in the railway domain. In *Formal Modeling and Verification of Cyber-Physical Systems*, pp. 82–121. Springer.

James, P., F. Moller, N. H. Nga, M. Roggenbach, S. A. Schneider, and H. Treharne (2014). Techniques for modelling and verifying railway interlockings. *STTT 16*(6), 685–711.

Jia, Y. and M. Harman (2010). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering 37*(5), 649–678.

Kozen, D. (1983). Results on the propositional μ-calculus. *Theoretical computer science 27*(3), 333–354.

Lilius, J. and I. P. Paltor (1999). Formalising UML state machines for model checking. In *International Conference on the Unified Modeling Language*, pp. 430–444. Springer.

Linhares, M. V., R. S. de Oliveira, J.-M. Farines, and F. Vernadat (2007). Introducing the modeling and verification process in SysML. In *IEEE Conference on Emerging Technologies and Factory Automation*, pp. 344–351. IEEE.

Morkevicius, A. and N. Jankevicius (2015). An approach: Sysml-based automated requirements verification. In *International Symposium on Systems Engineering (ISSE)*, pp. 92–97. IEEE.

Ober, I., I. Ober, I. Dragomir, and E. A. Aboussoror (2011). UML/SysML semantic tunings. *Innovations in Systems and Software Engineering 7*(4), 257–264.

Pedroza, G., L. Apvrille, and D. Knorreck (2011). AVATAR: A SysML environment for the formal verification of safety and security properties. In *International Conference on New Technologies of Distributed Systems*, pp. 1–10. IEEE.

Pétin, J.-F., D. Évrot, G. Morel, and P. Lamy (2010). Combining sysml and formal methods for safety requirements verification.

Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence. *Software-concepts and tools 17*(3), 103–120.

van den Bos, P. and J. Tretmans (2019). Coverage-based testing with symbolic transition systems. In D. Beyer and C. Keller (Eds.), *Tests and Proofs*, Cham, pp. 64–82. Springer International Publishing.

Varró, D. (2002). A formal semantics of UML statecharts by model transition systems. In *International Conference on Graph Transformation*, pp. 378–392. Springer.