A Case in Point: Verification and Testing of a EULYNX Interface

MARK BOUWMAN, Eindhoven University of Technology, The Netherlands
DJURRE VAN DER WAL, University of Twente, The Netherlands
BAS LUTTIK, Eindhoven University of Technology, The Netherlands
MARIËLLE STOELINGA and AREND RENSINK, University of Twente, The Netherlands

We present a case study on the application of formal methods in the railway domain. The case study is part of the FormaSig project, which aims to support the development of EULYNX — a European standard defining generic interfaces for railway equipment — using formal methods. We translate the semi-formal SysML models created within EULYNX to formal mCRL2 models. By adopting a model-centric approach in which a formal model is used both for analyzing the quality of the EULYNX specification and for automated compliance testing, a high degree of traceability is achieved.

The target of our case study is the EULYNX Point subsystem interface. We present a detailed catalog of the safety requirements, and provide counterexamples that show that some of them do not hold without specific fairness assumptions. We also use the mCRL2 model to generate both random and guided tests, which we apply to a third-party software simulator. We share metrics on the coverage and execution time of the tests, which show that guided testing outperforms random testing. The test results indicate several discrepancies between the model and the simulator. One of these discrepancies is caused by a fault in the simulator, the others are caused by false positives, i.e. an over-approximation of fail verdicts by our test setup.

CCS Concepts: • **Theory of computation** → **Verification by model checking**; • **Networks** → **Protocol testing and verification**; **Formal specifications;**

Additional Key Words and Phrases: Formal analysis, mCRL2, railway systems, SysML, test automation

ACM Reference format:
Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. 2023. A Case in Point: Verification and Testing of a EULYNX Interface. *Form. Asp. Comput.* 35, 1, Article 2 (March 2023), 38 pages.
https://doi.org/10.1145/3528207

FormaSig is 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.
Authors' addresses: M. Bouwman and B. Luttik, Eindhoven University of Technology, Groene Loper 3, 5612 AE, Eindhoven, PO Box 513, 5600MB, Eindhoven, Noord-Brabant, The Netherlands; emails: m.s.bouwman@tue.nl, s.p.luttik@tue.nl; D. van der Wal, M. Stoelinga, and A. Rensink, University of Twente, Drienerlolaan 5, 7522 NB, Enschede, PO Box 217, 7500 AE, Enschede, Overijssel, The Netherlands; emails: d.vanderwal-1@utwente.nl, m.i.a.stoelinga@utwente.nl, arend.rensink@utwente.nl.

0934-5043/2023/03-ART2
https://doi.org/10.1145/3528207

1 INTRODUCTION

ProRail and DB Netz AG, two railway infrastructure managers, are interested in applying formal methods to EULYNX.[1] EULYNX is an initiative involving 11 other European railway infrastructure managers to reduce the cost and installation time of signaling equipment. One of the primary activities of EULYNX is the standardization of the interfaces between the *interlocking* – the central device that controls most of the signaling infrastructure in an area – and *field elements*, such as signals, points, and level crossings. The standardization efforts should improve the interoperability between components from different suppliers, and thus lead to a significant reduction of life cycle costs.

The correctness of the EULYNX standard and the conformance of the corresponding implementations to that standard are essential for the safety of future railway systems. The correctness and conformance should be clearly established, which can be achieved using *formal methods* such as model checking and automated testing. For this reason, a collaboration called *FormaSig*[2] was started in 2018 between ProRail, DB Netz AG, Eindhoven University of Technology, and the University of Twente. The aim of FormaSig is to apply a state-of-the-art *model-centric approach*, i.e. to use the same formal model for verification and for automated testing of EULYNX implementations. The model-centric approach has gained traction because (i) only one formal model has to be created instead of two and (ii) it means that the two activities become mutually reinforcing [25].

The EULYNX project has adopted the semi-formal *SysML* language [33] to specify the standardized interfaces. In the case of a semi-formal specification model – as opposed to a natural language specification – there is an opportunity to trace back weaknesses (ambiguities, inconsistencies, and missing assumptions) that are exposed by a formal analysis more clearly and easily to elements in the original design. This opportunity is explored in FormaSig via a translation from SysML to *mCRL2*,[3] a formal modeling language with accompanying analysis software; see Figure 1. The translation also serves as the *formalization* of EULYNX, giving EULYNX specifications an unambiguous interpretation. This unambiguous interpretation is aligned with the intuition of signaling engineers based on interviews with them.

The SysML tool used for EULYNX specifications, PTC Windchill,[4] includes several non-standard features such as a custom communication mechanism (pulse ports) and a specific action language. A general translation from SysML to mCRL2 [8] was customized to support these features.

Case study. In this paper, we present a case study on the application of the FormaSig approach to the EULYNX interface for point subsystems (also called 'turnouts' or 'switches'). The FormaSig approach will eventually be applied to other EULYNX interfaces, such as train detectors, light signals, and so on. Future work will therefore be enhanced by the experiences and lessons learned from the Point case study.

We extend – and improve upon – an earlier publication, in which we describe an early version of the Point case study [10].

A *point* is a type of mechanical installation that guides trains from one set of rails to another. This relatively simple function is described in EULYNX as the composition of a number of subcomponents, and built upon a layer of composed subcomponents for generic functionality (connection management, timeouts, power loss, etcetera). Consequently, the complexity of the EULYNX Point interface is greater than one might expect, and analyzing it (either with formal methods or through plain human understanding) becomes non-trivial.

[1]European Initiative Linking Interlocking Subsystems; see https://www.eulynx.eu.
[2]Formal Methods in Railway Signaling Infrastructure Standardization Processes.
[3]mini Common Representation Language.
[4]https://www.ptc.com/en/products/windchill/integrity/.

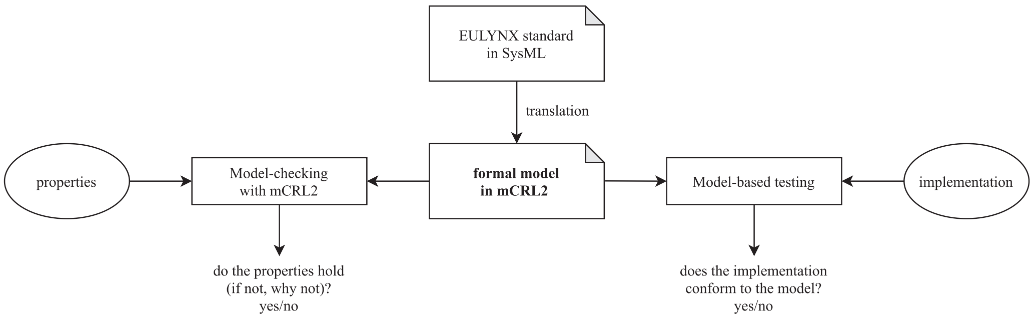

EULYNX standard in SysML
translation
properties
Model-checking with mCRL2
formal model in mCRL2
Model-based testing
implementation
do the properties hold (if not, why not)? yes/no
does the implementation conform to the model? yes/no

Fig. 1. FormaSig setup. The formal model is obtained with an automated translation and then used for model-checking and conformance testing.

We used the SysML-to-mCRL2 translation to obtain an mCRL2 model from the specification of the EULYNX Point interface. The state space of this mCRL2 model is extremely large, and we therefore performed *symbolic model checking* with the mCRL2 toolkit. For model-based testing, a similar solution is not available in the mCRL2 toolkit, so we generated tests from only a part of the Point state space.

For the verification, we elicited nine requirements from signaling experts for the Point interface, and used mCRL2 to show that six of the requirements are met and two need a fairness assumption (which led to a recommendation to clarify the EULYNX specification). The mCRL2 toolset was not able to check the last requirement in a reasonable time frame.

To perform model-based testing, we preprocessed the partial Point state space by removing all events except those that model interactions with the environment (the interlocking on one side and motors and sensors of the point on the other). We manually mapped these interactions to the inputs and outputs of a third-party software simulator of the EULYNX Point interface, developed by the SIGNON Group.[5] We implemented an integrated adapter and test generator. Using this setup, we have executed 1,000 random tests and 1,000 guided tests. Measurements show that the guided test suite was faster to execute and achieved better coverage than the random test suite.

The simulator failed several of the generated tests. In total, we discovered three (types of) discrepancies between model and simulator. We confidently identified one of the discrepancies as an error; that is, behavior of the simulator that should not be possible according to the Point specification. Another discrepancy is caused by the fact that the model and the simulator do not have the same granularity of inputs and outputs, a difference that we have made smaller but not yet eliminated. The final discrepancy consists of false positives (incorrect **fail** verdicts) that are caused by the fact that the Point state space is partial, and an unexplored state was reached.

Contributions. Our work on the Point case study has resulted in:

- A formal model in mCRL2 for the EULYNX Point interface specification;
- Safety requirements and their formalization in terms of modal μ-calculus formulas;
- Suggestions for quality improvements of the EULYNX standards, supported by a formal verification of the safety requirements using the mCRL2 model checker; and
- A model-based testing setup to automatically test an implementation for compliance to the EULYNX standard.

[5]See https://signon-group.com/.

This paper extends and improves on our earlier work [10], by presenting:

- A systematic method to automatically formalize EULYNX SysML models in mCRL2;
- A more detailed description of the requirements, including the corresponding μ-calculus formulas;
- A different model-based testing approach, using on-the-fly test case generation;
- A test setup that executes tests between 16 and 32 times faster;
- A more detailed description of the testing setup; and
- Novel verification and testing results.

Related work. A significant amount of research has been put into the application of formal methods in the area of safety-critical railway systems [2, 17, 18, 26]. These include studies in which a formal model is used for both verification and model-based testing [6, 7, 22–24]. However, the aforementioned work is focused on interlockings and fixed railway layout configurations, whereas FormaSig is concerned with the interaction between an interlocking and its field elements. Moreover, the interaction must conform to a *standard*, which can be more permissive than a (product) specification.

More generally, researchers have been exploring the use of formal models for both verification and testing since the start of the century [11, 25]. Over time, several challenges with this approach have also been identified. In the case of cyber-physical systems, for example, it has been reported that problems may not be discovered with formal methods when the abstraction level of a formal modeling language is too high [46]. Part of FormaSig is to determine to which extent EULYNX is affected by such weaknesses, and provide solutions, if necessary.

The formal specification language mCRL2 comes with a software toolkit [14] that supports model-checking and which has been used successfully in the past for **model-based testing (MBT)** in combination with TorX [38] and JTorX [3, 4]. In particular, an mCRL2-and-JTorX setup has been used to automatically test an interlocking [7]. TorX and JTorX generate, execute, and evaluate tests based on **input-output conformance (ioco)** theory [37, 39, 40]. Several other MBT approaches exist [41], differentiated by aspects such as model scope, coverage criteria and test selection.

In this paper, we combine the automatic derivation of a formal model from a SysML model and the usage of a single model for model checking and testing. These aspects together create a high degree of traceability between semi-formal model, formal model, verification results and testing results. Furthermore, we apply said techniques in an industrial case study.

Data sources. All models and requirements used in this case study are publicly available online via a Zenodo repository, see https://doi.org/10.5281/zenodo.5075647. The repository also contains the logs of all performed tests and the source code of the simulator.

Paper organization. The paper is organized as follows. Sections 2, 3 and 4 introduce the Point subsystem, how it is specified in SysML and how we have derived a formal mCRL2 model from the semi-formal SysML model, respectively. Verification of the Point interface is presented in Section 5. Section 6 describes how we tested the software simulator of the Point interface, and the results. Sections 7 and 8 conclude this work with discussion, conclusions, and future work.

2 POINT ARCHITECTURE IN EULYNX

A railway point (also known as a 'turnout' or 'switch') is a mechanical safety-critical installation enabling trains to be guided from one set of rails to another (see Figure 2). Although implementations of points are country-specific, they all consist of one or more movable elements, which are controlled by point machines and monitored by sensors. Point machines are essentially engines moving the movable parts of a point. The positions of the movable elements determine the

Fig. 2. Point at Broomhill station, Scotland [45].

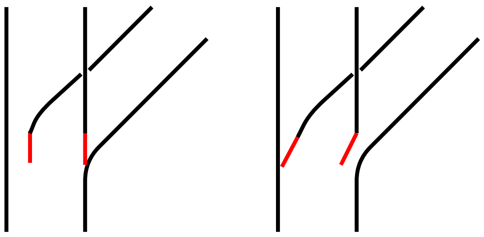


Fig. 3. Schematic view of point positions, with movable elements in red. The left picture shows a point in the 'left' position, the right picture shows a point in the 'right' position.

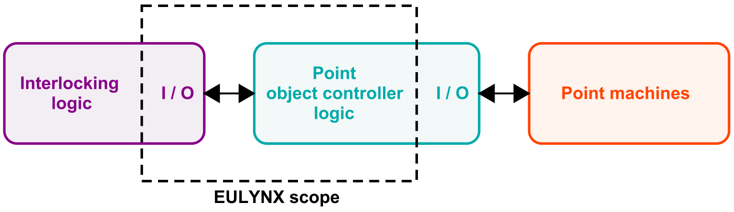


Fig. 4. Scope of the EULYNX Point interface. The connection between the interlocking and the object controller runs over an IP network. The connection between the object controller and the point machines is not specified by EULYNX but is typically electrical.

position of the point itself, namely 'left' or 'right' (see Figure 3), or 'neither' when changing from one to the other.

A point is controlled by an interlocking. The interlocking controls all the points, signals, level crossings, etcetera, in an area and ensures non-conflicting routes of trains.

EULYNX philosophy. In traditional points, the motors and sensors of the point machines are directly connected to the interlocking. This makes the implementation of a point highly dependent on the implementation of the interlocking, reducing interoperability.

A key innovation by EULYNX is to change this architecture by decoupling the interlocking and the point machines and controlling the point over an IP network. The point machines are controlled locally by an *object controller*, which in turn communicates with the interlocking. By decoupling the life cycles of the interlocking and trackside equipment and standardizing the interface, EULYNX creates a larger European market for trackside equipment. Moreover, this decoupling offers a higher resilience to cable failures, since IP packets can now be routed dynamically.

The messages exchanged over the network, the logic of the object controller and the interfacing logic of the interlocking constitute the *EULYNX Point interface* (see Figure 4). Note that the electrical connections between the object controller and the point machines are left unspecified.

High level architecture. The communication between the interlocking and the Point object controller is distributed over three layers (see Figure 5): The first layer manages a channel that follows the *RaSTA protocol* [44]. RaSTA is a safety-focused rail network protocol, described in the DIN VDE

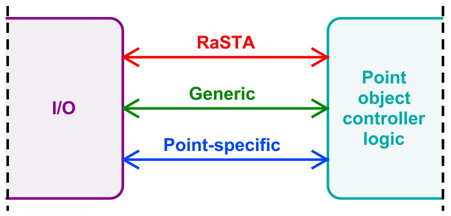


Fig. 5. Layers of communication between the interlocking and the object controller.

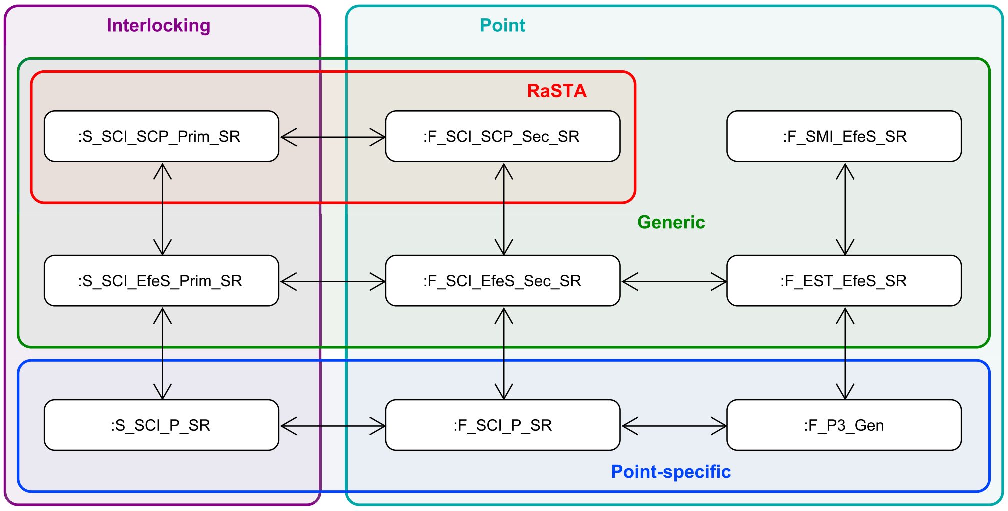


Fig. 6. Structure of the subcomponents of the EULYNX Point interface. Arrows indicate communication channels between subcomponents.

V 0831-200 standard. The second layer extends the first layer with functionality for identifying the (version of the) device on the other side. It also reports general failure modes, such as power loss. The final layer adds point-specific commands and messages.

The first and second layer are shared by all interlocking and object controller interfaces; the third is specific to the subsystem type at hand (point, light signal, etcetera).

Point-specific functionality. The object controller controls the point via one or more point machines. The two functions of the point-specific layer are steering the point to a requested position and reporting the current position of the point to the interlocking. The output from the object controller to the point machines is either ‘left’, ‘right’ or ‘stop’. The point machines, in turn, send back the current position of the point. The position can have three possible values, ‘end position right’, ‘end position left’ or ‘no end position’. Since each point machine detects and reports its own position, there may not be consensus between point machines on the point position, in which case the object controller reports a ‘no end position’ message to the interlocking.

Division in subcomponents. The EULYNX Point interface is modeled using nine *subcomponents*, see Figure 6. The interface captures both the physical systems and their communication layers.

- The subcomponents F_SCI_SCP_Sec_SR and S_SCI_SCP_Prim_SR manage the RaSTA connection: one on the side of the interlocking (S_SCI_SCP_Prim_SR) and one on the side of the object controller (F_SCI_SCP_Sec_SR).

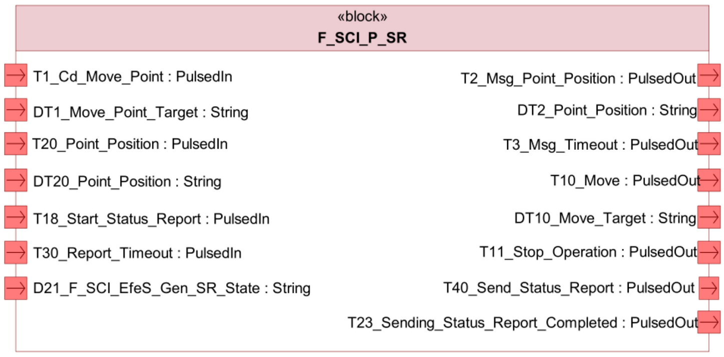

«block»
F_SCI_P_SR
T1_Cd_Move_Point : PulsedIn
DT1_Move_Point_Target : String
T20_Point_Position : PulsedIn
DT20_Point_Position : String
T18_Start_Status_Report : PulsedIn
T30_Report_Timeout : PulsedIn
D21_F_SCI_EfeS_Gen_SR_State : String
T2_Msg_Point_Position : PulsedOut
DT2_Point_Position : String
T3_Msg_Timeout : PulsedOut
T10_Move : PulsedOut
DT10_Move_Target : String
T11_Stop_Operation : PulsedOut
T40_Send_Status_Report : PulsedOut
T23_Sending_Status_Report_Completed : PulsedOut

Fig. 7. IBD that defines the ports of the ‘F_SCI_P_SR’ subcomponent of the EULYNX Point interface. Ports have a name, a data type, and a direction (‘in’ or ‘out’).

- The subcomponents S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR are responsible for the generic communication behavior of the second layer.
- The last pair of interlocking/object controller subcomponents (S_SCI_P_SR and F_SCI_P_SR) implement the third layer, exchanging point-specific commands and messages. Subcomponent F_SCI_P_SR relays point-specific messages to F_P3_Gen and ensures that the position is reported in the initialization phase.
- The three remaining subcomponents are F_SMI_EfeS_SR, used for maintenance access, F_EST_EfeS_SR, defining interaction with generic electronics, and finally F_P3_Gen, which interacts with the point machines (not included in the figure). The subcomponent F_P3_Gen performs two tasks in parallel: it monitors the current positions of the point machines and reports the information to F_SCI_P_SR, and it steers the point to the position that was most recently received from F_SCI_P_SR. When the generic layer reports a loss of connection or another failure, F_P3_Gen stops reporting the position or moving the point.

3 SYSML MODELING OF POINT

EULYNX interface specifications are given in a dialect of SysML [33]. SysML is a popular systems engineering modeling language, closely related to the **Unified Modeling Language (UML** [32]). SysML defines nine different diagram types, several of which are extended versions of UML diagram types. Five are used in EULYNX specifications: **Block Definition Diagrams (BDDs)**, **Internal Block Diagrams (IBDs)**, **Use Case Diagrams (UCDs)**, **Sequence Diagrams (SDs)**, and **State Machine Diagrams (SMDs)**.

To determine the complete behavior of the Point interface and to generate the complete mCRL2 model, IBDs and SMDs are sufficient. Below we show how they are used in EULYNX.

3.1 Internal Block Diagrams

IBDs are used to describe the *ports* of EULYNX subcomponents. Ports are essentially variables, with a name and type. Ports are defined as either an *input port*, denoted by an arrow-in-a-box pointing *into* the block, or as an *output port*, denoted by an arrow that points outwards. A subcomponent cannot change the value of its input ports.

Figure 7 presents the IBD of the F_SCI_P_SR subcomponent, which intermediates between the interlocking and Point hardware. Its header provides the name of the subcomponent, and the area below the header lists input ports (‘T1_Cd_Move_Point’, for example) and output ports (such as ‘T2_Msg_Point_Position’).

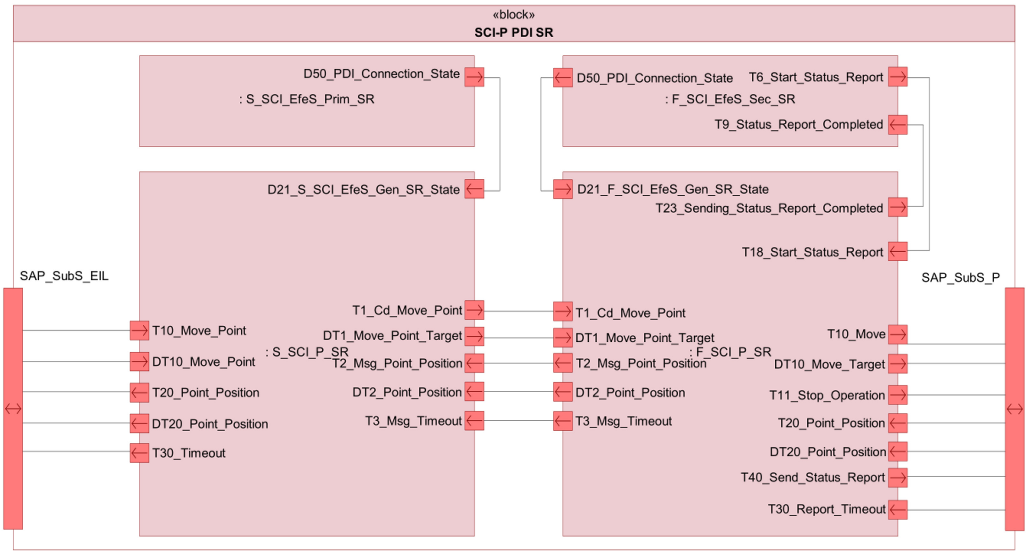


Fig. 8. IBD that defines how certain Point subcomponents are interconnected; a so-called “context”.

The IBD in Figure 7 also declares ten *pulse ports*: four input pulse ports, recognizable by their ‘PulsedIn’ type, and six output pulse ports, recognizable by their ‘PulsedOut’ type. Pulse ports are an addition to SysML by EULYNX. They can have the value ‘TRUE’ or ‘FALSE’, just like Boolean ports, but they automatically reset themselves from ‘TRUE’ to ‘FALSE’. They are frequently accompanied by data ports; for example, when the T1_Cd_Move_Point pulse port of F_SCI_P_SR becomes ‘TRUE’, it means that the value of the DT1_Move_Point_Target port is (temporarily) valid and can be used to choose new behavior.

IBDs can also be used to define how EULYNX subcomponents are interconnected. In such an IBD, the main block of the IBD does not define an individual subcomponent, but rather a context in which the ports of multiple subcomponents are connected by *(data) flows*. Two ports can only be connected to each other by a flow if they have the same type and opposite directions. An output port may be connected to multiple input ports, if they have the same type; input ports, on the other hand, may be connected to at most one output port. Input ports take on the data value of the output ports to which they are connected.

We illustrate this way of using IBDs in Figure 8. As stated in the header, the diagram defines the context called ‘SCI-P PDI SR’. The area below the header contains the subcomponents that are included in this context; among them is the subcomponent from Figure 7, F_SCI_P_SR, which is connected to the subcomponent S_SCI_P_SR with five flows. F_SCI_P_SR and S_SCI_P_SR have flows that lead to two other subcomponents, as well as to two objects on the edge of the diagram, labeled ‘SAP_SubS_EIL’ and ‘SAP_SubS_P’; these are *interface flow ports*, and can be used to combine contexts.

If a port of a subcomponent is not connected to another port (anywhere in the specification), it is connected to the *environment*. For an input port, this implies that its value can change non-deterministically at any moment. Supposing that the IBD in Figure 8 is the only IBD of a specification, the T20_Point_Position port of the F_SCI_P_SR subcomponent is manipulated by the environment. This does not hold for the two T1_Cd_Move_Point ports, because they are connected to each other.

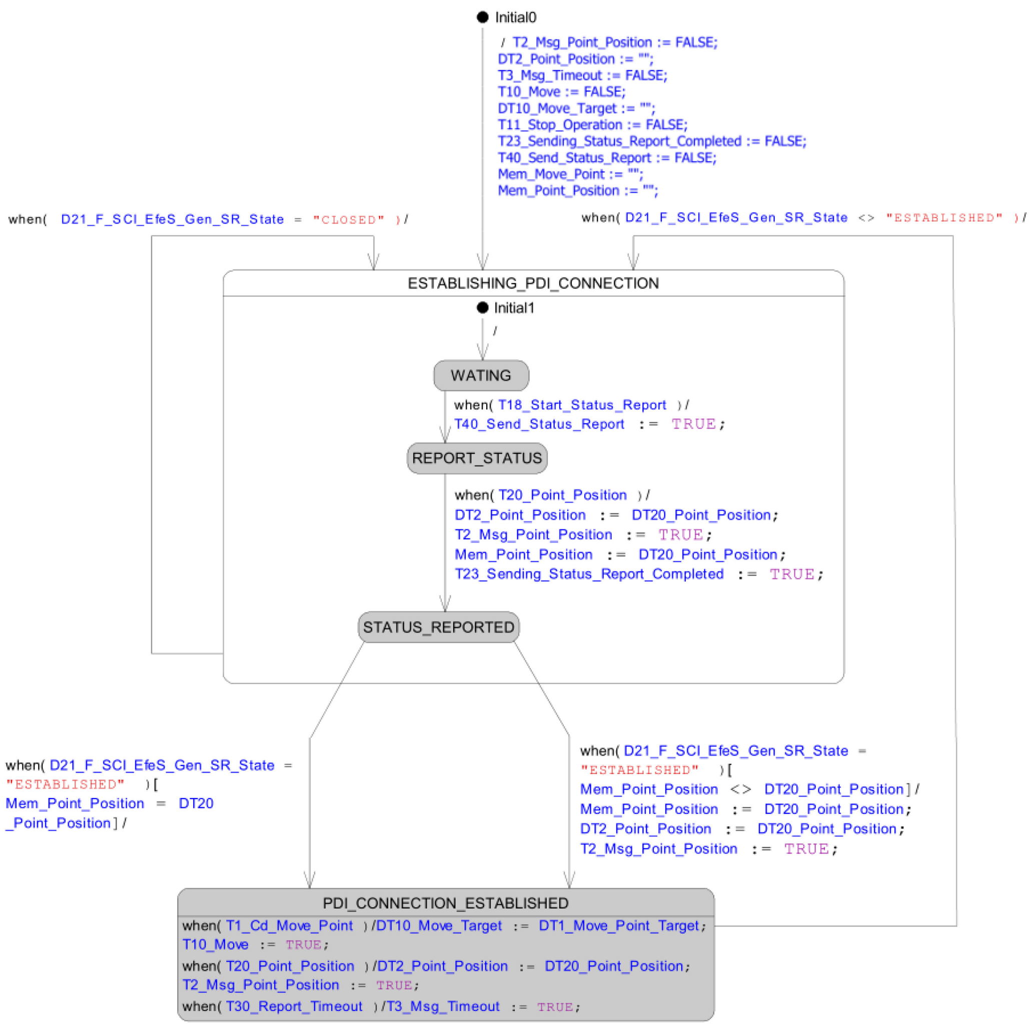

Initial0
/ T2_Msg_Point_Position := FALSE;
DT2_Point_Position := "";
T3_Msg_Timeout := FALSE;
T10_Move := FALSE;
DT10_Move_Target := "";
T11_Stop_Operation := FALSE;
T23_Sending_Status_Report_Completed := FALSE;
T40_Send_Status_Report := FALSE;
Mem_Move_Point := "";
Mem_Point_Position := "";
when(D21_F_SCI_EfeS_Gen_SR_State = "CLOSED")/
when(D21_F_SCI_EfeS_Gen_SR_State <> "ESTABLISHED")/
ESTABLISHING_PDI_CONNECTION
Initial1
/
WATING
when(T18_Start_Status_Report)/
T40_Send_Status_Report := TRUE;
REPORT_STATUS
when(T20_Point_Position)/
DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
Mem_Point_Position := DT20_Point_Position;
T23_Sending_Status_Report_Completed := TRUE;
STATUS_REPORTED
when(D21_F_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[Mem_Point_Position = DT20 _Point_Position]/
when(D21_F_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[Mem_Point_Position <> DT20_Point_Position]/
Mem_Point_Position := DT20_Point_Position;
DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
PDI_CONNECTION_ESTABLISHED
when(T1_Cd_Move_Point)/DT10_Move_Target := DT1_Move_Point_Target;
T10_Move := TRUE;
when(T20_Point_Position)/DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
when(T30_Report_Timeout)/T3_Msg_Timeout := TRUE;

Fig. 9. SMD that defines the behavior of the F_SCI_P_SR subcomponent of the EULYNX Point interface.

3.2 State Machine Diagrams

The behavior of EULYNX subcomponents is defined by **state machine diagrams (SMDs)**. Each subcomponent must have exactly one corresponding SMD. We use the SMD of the F_SCI_P_SR subcomponent as an example (see Figure 9).

The principal elements of an SMD are *states*. These are displayed as rounded rectangles, with a header at the top with their name. States are connected by *transitions*, displayed as arrows with a label. Transition labels may define a *trigger*, a *guard*, an *effect*, or some combination of the three:

- The trigger of a transition is a change event 'when(c)', which occurs when a Boolean expression 'c' changes from 'FALSE' to 'TRUE', or a timeout event 'after(d)', which occurs 'd' time units after a state was entered. As an example of a change event, the outgoing transition of the state 'WATING' (sic) can only happen when the value of the input port 'T18_Start_Status_Report' has just changed from 'FALSE' to 'TRUE'. Figure 9 does not contain an example of a timeout event.

- The guard of a transition is of the form '[g]', where 'g' is a Boolean expression that must hold in order for the transition to be enabled. There are two guards in Figure 9, namely in the labels of the outgoing transitions of the state 'STATUS_REPORTED'.
- The effect of a transition consists of a number of *action language* statements. For these statements, EULYNX uses the **Atego Structured Action Language**[6] **(ASAL)** which is tied to the PTC Windchill tool. For an example, the outgoing transition of the state 'WATING' sets the output port 'T40_Send_Status_Report' to 'TRUE' when the transition is executed.
  ASAL is a fairly straightforward programming language, featuring typical constructs such as variable assignments, if-statements, and while loops; we will not describe it in more detail in this paper.

A transition may "fire" when its source state is active, its trigger (if any) is in effect, and its guard (if any) holds. When a transition fires, its effect is executed; afterwards, the target state of the transition becomes active and the source state of the transition becomes inactive (unless the source state is equal to the target state).

Pseudo-states, internal transitions and composite states. In addition to states, there are also various types of *pseudo-states*, such as 'initial', 'final', 'choice', 'junction', 'fork', and 'join' pseudo-states. Pseudo-states are connected to each other and to regular states by transitions. Transitions that start in a pseudo-state must not have a trigger. Furthermore, transitions that start in an active pseudo-state must fire *before* transitions that start in a regular state. Figure 9 contains only two pseudo-states, which are both initial. These are the small solid-black circles labeled 'Initial0' and 'Initial1'. Initial (pseudo-)states may never be the target of a transition, and they must be the source of exactly one transition, which must not have a guard. Initial (pseudo-)states indicate the "first" behavior that should be executed (by a state machine in its entirety or by a region inside a state machine).

States can also define *internal* transitions. These do not change the state of the state machine, but only affect the values of the variables. Internal transitions are not visualized as an arrow; instead, their label is listed inside the rounded rectangle of their state. Figure 9 shows three examples of local transitions, all owned by the state 'PDI_CONNECTION_ESTABLISHED'.

States can nest (i.e. contain) other states and pseudo-states. For example, 'ESTABLISHING_PDI_CONNECTION' nests the states 'WATING', 'REPORT_STATUS', and 'STATUS_REPORTED', and the initial state 'Initial1'. States that nest other (pseudo-)states are called *composite states.*

4 FORMALIZING THE POINT INTERFACE IN MCRL2

To unlock the potential of verification and model-based testing, we need to be able to convert EULYNX SysML models to a formal language (see Figure 1). This section explains how we formalize EULYNX SysML models in mCRL2, the first contribution of this paper. Section 5.1 discusses how the mCRL2 toolkit is used for model checking.

Extracting an mCRL2 model from a SysML model is non-trivial. Due to the interplay of pseudo-states, nested states and change events, it can be hard to determine which transitions fire and in what order. Formalization of a system specified in SysML is not a new challenge. Indeed, various formalizations of SysML models have been proposed [15, 16, 29, 31, 34, 35]. However, the many variants of semantic interpretation of diagrams and the many ways in which SysML diagrams can be combined to describe the behavior of a composed system, make it difficult to apply existing work in other contexts. This is also the case for EULYNX, which uses its own action language and a port-based communication mechanism. We therefore developed a new formalization of the EULYNX-specific dialect.

[6] https://support.ptc.com/help/modeler/r9.0/en/index.html#page/Integrity_Modeler/sysim/SySim_Atego_structured_action_language.html.

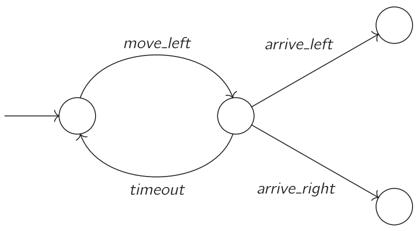


Fig. 10. Example of an LTS. The arrow without a label or source state points to the initial state. From this LTS, it can be inferred that after starting a movement, either a timeout can occur (after which a new movement may be initiated) or we arrive at some position (the one we were moving towards or not).

mCRL2. The mCRL2 toolkit is designed to model and analyze concurrent and distributed systems. The mCRL2 language is an ACP-style [5] process algebra and contains an expressive and flexible data language containing several built-in primitive data types, such as integers, natural numbers and booleans, together with common operations on them. Users can also define their own abstract data types and operations.

The mCRL2 toolkit can transform complex models consisting of parallel interacting processes into a linearized model in a normal form: a **linear process specification (LPS)** [21]. This step removes all parallelism and applies various operators such as hiding and communication. The semantic interpretation of an mCRL2 model is a **labeled transition system (LTS)**, which can be derived from a linearized specification. An example of an LTS can be found in Figure 10. By translating from SysML to mCRL2 we indirectly associate an LTS with the EULYNX SysML model.

The mCRL2 toolkit contains tools for the verification of parameterized modal μ-calculus formulas, bisimulation reduction, counterexample generation, simulation and visualization. For more information on mCRL2 we refer to mcrl2.org and [20].

Translation tool. In [8] we formalized the semantics of UML state machines with port-to-port communication directly in the mCRL2 language, without assuming any particular action language. This formalization in mCRL2 is a partial model generically specifying the semantics of state machines, which needs to be complemented by encoding a concrete set of state machines (and their port connections) in the mCRL2 data language. To make this previous work suitable for EULYNX SysML models, some extensions are necessary. The mCRL2 model containing the formalization [8] needs to be extended with (i) the semantics of the action language and (ii) a set of concrete state machines with port connections encoded in the mCRL2 data language. We could encode the state machine diagrams and port connections of the EULYNX point interface manually in mCRL2, but this would be inefficient and error-prone. Instead, we have built a tool that automatically converts EULYNX models consisting of state machine diagrams and internal block diagrams to the correct mCRL2 encoding. The internal block diagrams are preprocessed by recognizing which blocks occur in multiple diagrams and combining the diagrams into one big overview of all the port connections. We have also formalized ASAL in mCRL2.

SysML diagrams of the EULYNX point-specification are available in a proprietary format that is defined by the PTC Windchill tool. We found that files in the PTC format were not straightforward to parse, and also that the digital instances of the diagrams were not entirely consistent with how these diagrams appeared in the EULYNX specifications in PDF format. A third party is developing software to convert files in the PTC Windchill format to a format that is easier to parse. In the mean time, we have avoided the issues mentioned above by introducing an intermediate format (jEULYNX) for inputting SysML diagrams. jEULYNX is a DSL in Java which we intend to submit a paper on in the near future. For the Point case study, we transcribed SysML diagrams from PDFs to jEULYNX by hand.

EULYNX adaptations to the mCRL2 formalization. To accommodate verification of EULYNX models we have enriched the mCRL2 model that encodes the semantics of SysML state machines in two ways. We added selfloops with the label `inState(`c`,`s`)`, where c is the name of a component and s the name of a state machine state. These selfloops are necessary to verify properties that refer to state machine states.

Secondly, we needed to adjust the semantics to accommodate pulse ports, which are boolean-valued ports with a semantics deviating from other data types. Suppose that we have some pulse output port X connected to input port Y. When a state machine sets port X to the value true, then port X will *automatically* revert to false after a brief time, and so will Y. The exact semantics of this mechanism is imprecise in EULYNX specifications. Our models do not model time explicitly, but we can capture the semantics of pulse ports in an alternative way. Our interpretation is that X and Y are true only in one atomic moment, during the communication from X to Y. The sender and receiver may use pulse ports to trigger *change events*: if the receiver has a transition with trigger 'when(Y)', a change event is placed in the event queue.

Transition labels. The LTS with the mCRL2 model contains two types of labels that are of interest for model checking: `inState`, which was previously mentioned, and `send|receive`. A transition with the label `send(`$c1$`,`$p1$`,`v`)| receive(`$c2$`,`$p2$`,`v`)` indicates that component $c1$ sends value v along port $p1$ to port $p2$ of component $c2$. The LTS contains other labels related to internal steps of state machines, which can be considered unobservable. In process algebras, it is common to abstract from such unobservable transitions by renaming the label to the special label τ.

Resolution of ambiguities. Variables, including ports, should have an initial value as otherwise it might not be possible to evaluate ASAL expressions referencing those variables. In EULYNX, all output ports and local variables are initialized in the transition from the initial state in the root of the state machine. Input ports can then be initialized by looking up the initial value of the connected output port. However, as discussed in Section 3.1, ports need not be connected, because they should interact with the environment (systems outside the scope). Input ports open to the environment are therefore not initialized in EULYNX specifications. Our translation tool allows manual specification of the initial value of these ports. For example, we choose the value 'no end position' for the initial value of the detected position. Also notable is that we set the country code to 'The Netherlands' (which disables some behavior that is specific to other EULYNX participants).

For input ports open to the environment, it is also not clear what values can be received while the system is running. Our default interpretation is that any value in line with the data type of the port is allowed. Our translation tool also allows us to disable or restrict selected input ports. For the Point interface we disabled some input ports representing configuration parameters, such as timeout durations and country settings. For ports with data type 'String' we restricted what values can be sent. For the input port measuring the position of the point we allowed three values: 'left', 'right' and 'no end position'. The input port allowing the interlocking to request a position was restricted to 'left' and 'right'.

Model versions. We created three formal models for the point interface, a full one for model based testing and two partial ones for verifying. Splitting the model for verification significantly reduces the state space. The following models were created (see Figure 6 to cross-reference component names):

(1) **Generic.** A model containing only the components modeling the generic interface: S_SCI_EfeS_Prim_SR, F_SCI_EfeS_Sec_SR, F_EST_EfeS_SR and F_SMI_EfeS_SR.

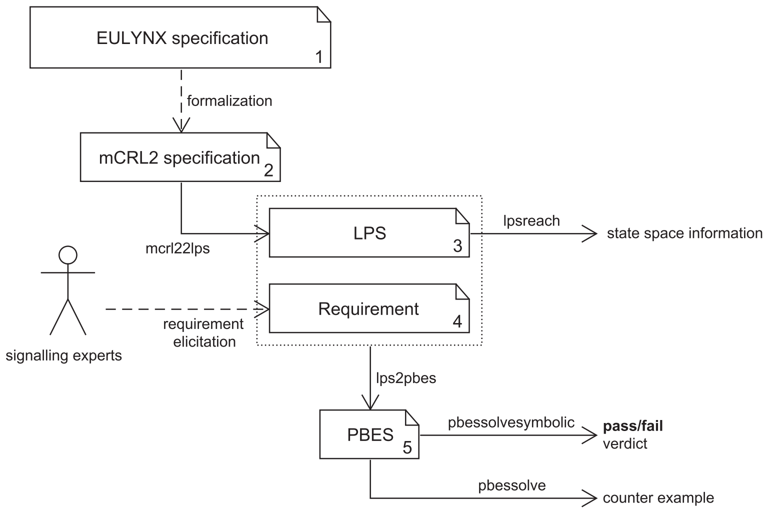


Fig. 11. Overview the mCRL2 tool chain to verify EULYNX models.

(2) **Point-specific.** A model containing only the components modeling the point-specific interface: S_SCI_P_SR, F_SCI_P_SR and F_P3_Gen.
(3) **Full.** A model containing the components S_SCI_EfeS_Prim_SR, F_SCI_EfeS_Sec_SR, F_EST_EfeS_SR, F_SMI_EfeS_SR, S_SCI_P_SR, F_SCI_P_SR and F_P3_Gen. This model is used for MBT; see Section 6.3.

Note that we consider the RaSTA protocol to be part of the environment in all three variants of the model. We do this for two reasons: EULYNX does not specify the behavior of the RaSTA blocks with state machines (so we would need to model these ourselves) and to reduce the state space.

Note that all three models contain a generic StateMachine process and generic data operations which formalize the EULYNX-specific adaptations of pulse ports, the ASAL action language and `inState` selfloops.

5 VERIFICATION

Formal verification techniques provided by mCRL2 make it possible to determine if the EULYNX Point interface satisfies conditions ranging from deadlock freedom to obedience of the field element to the commands of the interlocking. The conditions are proven by exhaustively checking all reachable states of the model, which means that verification is among the most thorough methods for assessing whether a model conforms to certain requirements.

In Section 5.1 we explore how requirements are verified using the mCRL2 toolkit. Sections 5.2 and 5.3 explain the method of eliciting requirements and the list of requirements, respectively. This section culminates in Section 5.4, where we present and discuss the results of model checking the EULYNX Point interface.

5.1 Model Checking with mCRL2

Figure 11 shows the verification procedure that we applied to the EULYNX Point interface. The procedure starts with our formalization of the Point interface (from 1 to 2). Our formalization outputs an mCRL2 model containing a number of communicating processes. The mCRL2 model is converted to an LPS (3) (see Section 4), using the tool `mcrl22lps`, making it a suitable target for mCRL2's analysis tools. Computing the LPS only takes a few seconds for our models. We can compute the number of states that the LPS represents – an indication of the complexity of behavior – and, more importantly, the conformance of its behavior to a set of requirements. The latter is accomplished by the analysis of a **parameterized boolean equation system or PBES** (5), which is computed with `lps2pbes` from the LPS and a single requirement (4) expressed in modal μ-calculus. Note that we do not need to compute the LTS associated with the mCRL2 model in order to verify requirements. Solving the PBES will indirectly explore (part of) the state space.

mCRL2 has algorithms for computing both state space size and conformance to requirements. These algorithms must store large sets of states, which can be done in two ways. The "classic" way is to store each state *explicitly*. The second way is to store states *symbolically*, drastically reducing the representation of the state space. Note, however, that the development of mCRL2 tools that use the symbolic approach has started recently, and that they are not yet documented. The tools are based on existing (documented) theory [28, 30]. In [10], we only used the explicit state tools.

The size of our Point model has persuaded us to use the symbolic tools. More specifically, we use `lpsreach` to compute the state space size and `pbessolvesymbolic` to solve PBESs. Unfortunately, `pbessolvesymbolic` does not (yet) support counterexample generation. To obtain counterexamples we still rely on the "classic" PBES solver `pbessolve`. This means that we cannot obtain a counterexample when the state space is too large for `pbessolve`.

5.2 Requirements Elicitation

The EULYNX standard specifies, through sequence diagrams, several scenarios for the Point interface. The model of the Point interface should at least admit the correct execution of these scenarios, thus they can be interpreted as requirements for the Point interface. However, to assess the quality of the EULYNX standard, we need to verify stronger requirements: rather than one scenario, we want to verify that properties hold along all execution paths of the system. Eliciting such requirements is an explicit concern of the FormaSig project, as good requirements are essential to assess the quality of EULYNX specifications.

A challenge in formulating pertinent requirements is that the prerequisite knowledge is, currently, split between the academic partners of FormaSig and the infrastructure managers: only the former possess the skills to formulate formal requirements, and only the latter possess the signaling domain knowledge. To overcome this challenge, we adopted an iterative process of requirement elicitation. We started out by gathering background information and identifying hazards by interviewing signaling experts. These general hazards were translated to initial requirements in natural language. We then attempted to verify the refined requirements using the mCRL2 model checker. For requirements that do not hold for the model we assessed whether the model contains an error or the requirement is too strong. In the latter case we refined the requirement.

For example, we derived the following requirement: *"When component F_EST_EfeS_SR signals to component F_SCI_EfeS_Sec_SR that the object controller is not ready for a connection by sending a message on port 'T18_Not_Ready_For_PDI_Connection', then component F_SCI_EfeS_Sec_SR is not allowed to be in the state 'PDI_CONNECTION_ESTABLISHED' until a message on port 'T21_Ready_For_PDI_Connection' is received."* By formalizing the requirement to a μ-calculus formula and checking the mCRL2 model for this requirement, we found a counterexample in which the communication over port 'T18_Not_Ready_For_PDI_Connection' happens while F_SCI_EfeS_Sec_SR is in the state 'PDI_CONNECTION_ESTABLISHED'. Clearly, the component needs some time to process the message and move out of the state 'PDI_CONNECTION_ESTABLISHED'.

We weakened the requirement to *"When component F_EST_EfeS_SR signals to component F_SCI-_EfeS_Sec_SR that the object controller is not ready for a connection by sending a message on port 'T18_Not_Ready_For_PDI_Connection', then component F_SCI_EfeS_Sec_SR will always eventually move out of the state 'PDI_CONNECTION_ESTABLISHED' and not establish a connection again until a message on port 'T21_Ready_For_PDI_Connection' is received."*

The corresponding μ-calculus formula is:

```
%For any trace ending with a communication that the object controller is not ready for a connection
[true*.send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),Value_Bool(true))|receive(CompPortPair(BEQ_seec
      ,T18_Not_Ready_For_PDI_Connection),Value_Bool(true))](
%when in PDI_connection established, move to not ready for connection
((<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true)
  %least fixed point is used to express that all paths eventually lead to a state where we have exited
        PDI_CONNECTION_ESTABLISHED. inState transitions are excluded as they allow infinite selfloops, making the
        formula trivially false
  => (mu X. (([!(exists c:CompName,s:StateName. inState(c,s))]X)
    || [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false)))
%greatest fixed point is used to express that on all paths a connection is not established until it is allowed
      again by F_EST_EfeS_SR
&& (nu X. [!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true))|receive(CompPortPair(
      BEQ_seec,T1_Ready_For_PDI_Connection),Value_Bool(true))]X
  && (<inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true =>
    [(!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true))|receive(CompPortPair(BEQ_seec,
          T1_Ready_For_PDI_Connection),Value_Bool(true)))*.inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false)))
```

Section 5.3 presents the derivation of nine requirements for the Point interface. Appendix A, lists them again including the μ-calculus formulas. The appendix also includes a short introduction to the μ-calculus.

5.3 Requirements

The two main hazards related to points are derailments and train-train collisions. The hazard of derailment has many aspects. Physical failures of the track might lead to derailment. A high speed in a tight curve could lead to derailment. Another possible cause is when a train goes over a point that is not in a proper (left or right) end position. Since the object controller of the point controls the movement of the point and informs the interlocking on the current position, correct behavior of the point object controller is essential to prevent derailments.

Since points determine the route of trains, correct behavior of the object controller is also essential in preventing train-train collisions. The considerations for the object controller are again the correct control of the movement and reporting of the position. The principle for movement of the point is that only the interlocking knows when it is safe to move a point, so only the interlocking can initiate a movement. Two principles apply to reporting the position of the point: the interlocking should always be kept up to date concerning the position of a point; and when the position of a point is unknown, it is always assumed not to have an end position (meaning that it is unsafe to drive over the point).

We can derive the following hazards for the point specific interface:

(1) The object controller reports an end position that is not accurate.
(2) The object controller initiates a movement that was not expected by the interlocking.

These two hazards are countered by the following requirements:

ID	REQ_P_001
Summary	The object controller must report changes in position
Detailed description	The position of the point is determined by combining the input from the point machines. When all point machines report right the position is right. When all point machines report left the position is left. In any other case the point does not have an end position. When the position of the point is different from the last reported position, the new position must be reported to the interlocking. This obligation is lifted if communication with the interlocking is not possible due to a connection problem, power failure, etc. If the object controller cannot report the changed position, the connection with the interlocking must be closed, so that the interlocking knows that the position is unknown.

ID	REQ_P_002
Summary	The object controller must not change position unless commanded by the interlocking
Detailed description	The object controller may only instruct the point machines to move when this is commanded by the interlocking. A movement command from the interlocking gives an authorization to initiate movement in a certain position that ends when either a timeout occurs, the end position is reached or a movement command for the opposing position is sent.

The main role of the generic interface is connection management. Initializing the connection consists of a number of steps. Firstly, a RaSTA connection is established. Secondly, the interlocking and object controller message each other to request a connection and exchange the version of the EULYNX protocol that they are using. Finally, the current position of the point is communicated. After these initialization steps, the connection is fully established and the interlocking and object controller can freely send each other movement commands and position updates.

During initialization or when the connection is established, the connection can be aborted by either side. When they encounter an error (which can occur non-deterministically in the model), such as a power failure or timeout, they move to an error state and notify the other side by closing the RaSTA connection. The connection can then be re-established using the normal procedure.

The main hazard related to the connection management is that a connection is wrongly established or maintained. The reason that this would pose a risk is that the interlocking would continue to believe it has a connection with the field element while it is no longer updated on the status of the field element. A secondary hazard is that a connection is prevented from being established at all. This might not be a direct safety hazard but could disrupt train services.

We can derive the following hazards for the generic interface:

(1) The object controller or interlocking has some reason to not be able to have a connection but a connection is still established/maintained.
(2) A deadlock or livelock prevents the system from establishing a connection.

These two hazards are countered by multiple requirements, which are listed below. The second hazard is countered by requirement REQ_PDI_004. The other requirements expand the first hazard by specifying different scenarios in which the object controller or interlocking is not allowed to establish a connection.

ID	REQ_PDI_002
Summary	Disconnect at version unequal
Detailed description	Whilst establishing a connection, in the case that the object controller sees that the interlocking uses a different version of the protocol, it sends a message to the interlocking notifying the failed version check and moves to the state not ready for a connection.

ID	REQ_PDI_003
Summary	Disconnect at checksum unequal
Detailed description	Whilst establishing a connection, in the case that the interlocking receives an incorrect checksum (indicating a malformed message) from the object controller in the Msg_PDI_Version_Check message, the interlocking terminates the connection.

ID	REQ_PDI_004
Summary	A connection remains possible
Detailed description	As long as there is no telegram error (the reception of a malformed message, which may occur non-deterministically in the model it always remains possible to reach PDI_Connection_Established in the future.

ID	REQ_PDI_005
Summary	Close connection on error
Detailed description	At the moment a protocol error or telegram error occurs at either the side of the object controller or the interlocking, both the interlocking and the object controller will eventually move to PDI_Connection_Closed before reattempting a connection.

ID	REQ_PDI_006
Summary	Close connection when not ready
Detailed description	When F_EST_EfeS_SR signals it is not ready for a connection, the object controller will move to not ready for connection and only reattempts a connection after receiving a message ready for connection.

ID	REQ_PDI_007
Summary	Close connection after timeout
Detailed description	When S_SCI_EfeS_Prim_SR does not change in state PDI_Connection_Established after entering state Establishing_PDI_Connection within the time D2_Con_tmax_PDI_Connection, both S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR reach PDI_Connection_Closed before reattempting a connection.

ID	REQ_PDI_008
Summary	One closes, both close
Detailed description	When S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR are both in state PDI_Connection_Established, and one leaves that state, the other will eventually leave the state as well.

5.4 Results & Performance

We used the toolchain described in Section 5.1 to measure the size of the state spaces and verify the requirements. In this section we present the verification results.

All tests are run on a machine equipped with 4 12-core Intel Xeon Gold 6136 CPUs and 3TB of RAM. To determine the size of the state space we use the symbolic tool `lpsreach` with 12 threads. To solve PBESs we use the symbolic tool `pbessolvesymbolic` with 6 threads. The requirements, point-specific mCRL2 model and generic mCRL2 model are all available in the Zenodo repository. Table 1 shows the size of the state spaces and Table 2 lists the verification results.

State space exploration and verification are significantly slower for the point-specific model. Further investigation revealed that one SysML component, F_P3_Gen, proves to be a bottleneck. The number of states discovered per second during explicit state space exploration is also remarkably low for the F_P3_Gen component. Since F_P3_Gen is the only component featuring parallel regions it is possible that some of the auxiliary data operations in the generic mCRL2 model defining the semantics of SysML (e.g. transition selection) are computationally more intensive in the presence of parallel regions. Whether this can be improved needs further investigation.

Not all requirements hold for the model (see Table 2). We will go over these unsatisfied requirements and see why they do not hold.

Requirement REQ_P_001, which states that changes to the position of the point will always eventually be reported to the interlocking, does not hold. Since the state space of the point-specific model is too large for the explicit state tools we cannot generate a counterexample. However, we can gain insights by verifying slightly altered formulas. The formula REQ_P_001_1 (see Appendix A) is true. REQ_P_001_1 only considers paths in which the environment is silent, removing loops of behavior. When F_P3_Gen wants to send the update position to F_SCI_P_SR, F_SCI_P_SR may be kept busy by continually receiving messages from F_SCI_EfeS_Sec_SR or S_SCI_P_SR.

Table 1. Statistics on the Size of the State Spaces, Computed with `lpsreach`

Model	Size state space	Time (s)
Generic PDI	$1.14262 \cdot 10^8$	20
Point-specific	$5.14198 \cdot 10^{10}$	1829

Table 2. Results of our Requirement Verification with `pbessolvesymbolic`

Requirement	result	Time (s)
REQ_P_001	false	5,507
REQ_P_001_1	true	3,188
REQ_P_002	?	?
REQ_PDI_002	true	178
REQ_PDI_003	true	147
REQ_PDI_004	false	180
REQ_PDI_005	true	228
REQ_PDI_006	false	226
REQ_PDI_006_1	true	142
REQ_PDI_007	true	157
REQ_PDI_008	true	333

A fairness assumption would be needed to also make the communication between F_P3_Gen and F_SCI_P_SR possible. EULYNX should add a fair scheduling requirement in its specification to avoid undesired behavior.

For now, we are not able to verify REQ_P_002 due to the previously mentioned performance issue with component F_P3_Gen.

Requirement REQ_PDI_004, stating that a connection always remains possible, does not hold. By using the explicit state tools, we were able to obtain a counterexample, which showed that two components can deadlock when trying to send a message to each other. Sending a message usually results in adding an event to the event queue (see [8]). The event queue mostly acts as a communication buffer, though state machines may also add events to their own queue by triggering change events. The event queue of state machines is finite in our models; when the event queue of a state machine is full, communication is no longer possible. When two components with a full event queue want to send a message to each other, they get into a deadlock. In EULYNX it is assumed that event queues are unbounded. However, unbounded event queues would cause an infinite state space due to communications from the environment and therefore make model checking unfeasible. Moreover, an event queue of arbitrary length can always be filled due to communications from the environment. Hence, increasing the size of the event queue will not remove the deadlock. In a model that includes timing, the probability of a deadlock should be inversely proportional to the size of the event queue. However, it would be even better if the system is designed to be more robust against bursts of communication. We have made the recommendation to EULYNX to explicitly specify what happens when a buffers becomes full.

Requirement REQ_PDI_006 – stating that when the object controller is not ready for a connection, the connection is closed and not established again until it is ready – also does not hold. The counterexample produced by the explicit state tools shows us that a component may not always eventually move to PDI_CONNECTION_CLOSED due to a loop of behavior of another component. The other component loops by receiving a value from the environment over and over again. The requirement might hold under a mild component-based progress assumption, such as *justness* [9, 43], which excludes unrealistic computations in which a component never gets the chance to make progress. Adding justness assumptions to formulas in the context of EULYNX mCRL2 models remains future work. Requirement REQ_PDI_006_1 (see Appendix A), which just checks whether a new connection is not established, does hold.

6 MODEL-BASED TESTING

Model-based testing (MBT) is a technique for automatically generating, executing and evaluating tests [37, 39, 40], which has also been applied in the railroad domain [23]. The main prerequisite of MBT is the availability of a model of the system-under-test 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 12 shows an IOLTS similar to the LTS from Figure 10, and Figure 13 and 14 show two other IOLTSs.

Test cases (or simply *tests*) generated from an IOLTS are essentially *decision trees* that track which stimuli ('?') are sent to a system-under-test 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.

When generating a decision tree from the IOLTS in Figure 13, one of the possible results is depicted in Figure 15. 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 (arbitrarily or according to some strategy, other behavior could be tested here, or the test could end with a **pass** verdict). Note that if we were to use the decision tree to test a system that is described by the LTS in Figure 12, the system would receive a **fail** verdict if it would move along the *arrive_right!* transition (due to non-determinism, this does not necessarily happen).

Depending on how the decision tree of a test is generated, its branches may not be deep enough to detect all discrepancies between a system-under-test and its model. For example, the decision tree in Figure 15 fails in this regard because it does not give the IOLTS in Figure 14 a **fail** verdict even though it behaves differently than the IOLTS in Figure 13. Full test coverage typically requires multiple decision trees such that for each transition in the model IOLTS there exists a decision tree that can reach that transition *and* confirms that the subsequent behavior can only correspond with the target state of the transition in the model (the number of states of the system-under-test must not exceed the number of states of the model, or such a conclusion cannot be drawn; this approach also discounts non-determinism).

6.1 Automated Testing with mCRL2 models

We generate a decision tree for testing "on-demand"; that is, we add branches to it *while* we are testing. We start with a new decision tree when we have reached a manually determined depth that is suitable for the system-under-test in question. On-demand testing has the advantage that we can keep testing as long as there is time available, or, similarly, that we test more behavior the longer we test. By choosing according to a particular strategy which new branches to add and when to start a new decision tree, we can focus more on parts of the model that contain more behavior and/or that are more critical.

Naturally, we may only add branches that are consistent with the mCRL2 model, and we must therefore obtain its explicit states and transitions through *exploration*. Exploration starts in S, the set of states in which the system-under-test could currently be, and computes $T(S)$, the set of all

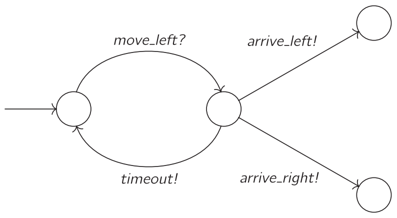


Fig. 12. Example IOLTS that is the same as the LTS from Figure 10, but with input labels that end with '?' and output labels that end with '!'.

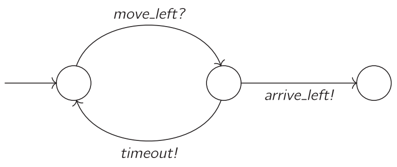


Fig. 13. Example IOLTS with input *move_left?*, and two outputs *arrive_left!*, and *timeout!*.

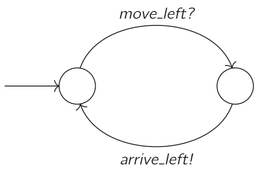


Fig. 14. Example IOLTS that behaves differently than the IOLTS in Figure 13, but does not receive a **fail** verdict from the decision tree in Figure 15.

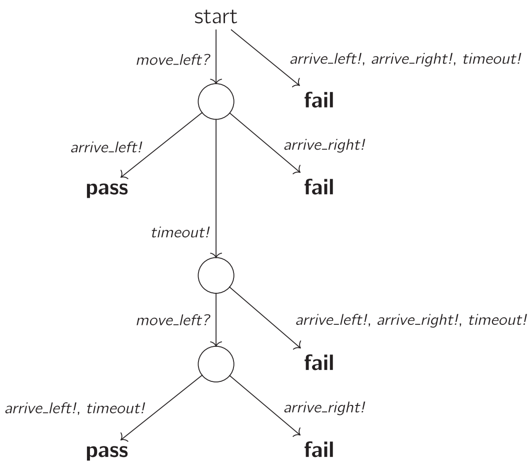


Fig. 15. Test case, expressed as a decision tree, that can be generated from the IOLTS in Figure 13.

transitions with an input/output label that can be reached from one of the states in S, possibly via internal transitions (τ transitions in an mCRL2 model: transitions without an input/output label, representing actions that the system-under-test can do independently of the environment and that cannot be observed). Given a used input or observed output a, the new value of S becomes the set of target states of all transitions in $T(S)$ that have the label a.

Like the generation of decision trees for testing, exploration can be performed on-demand: it is not necessary to explore the entire LTS, but only those parts that are reached by the tests that we generate. We do not use on-demand model exploration because the performance of mCRL2 is better on Unix platforms than on the Windows platform, on which our current system-under-test runs. Instead, we explore the mCRL2 model in advance on a Unix platform, using breadth-first search and limited by a time constraint, resulting in an incomplete LTS.

We then apply *weak trace equivalence reduction* [20]. Weak trace equivalence – also described as *stutter trace equivalence* [1] – holds when one LTS produces the same traces as another LTS, where a *trace* is a possible sequence of non-τ actions that can be encountered in an LTS. Weak trace equivalence reduction is conveniently available in mCRL2, and prevents us from having to consider internal actions during testing.

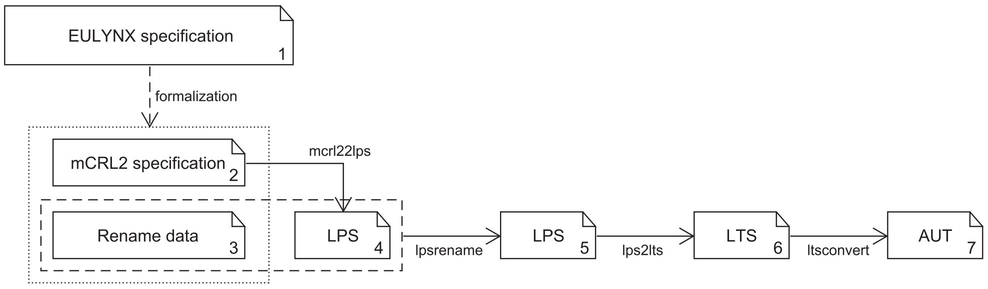


Fig. 16. Overview of how we obtain a partial mCRL2 model for the purpose of model-based testing.

Assuming the absence of cycles of internal actions – which is reasonable for the system in question – *quiescence* [36, 40] is preserved by weak trace equivalence reduction. Not preserved are *unexplored states*: states without any outgoing transition, which are a part of incomplete LTSs by definition. Clearly, tests will yield an incorrect **fail** verdict (a false positive) when such a state is reached and a model-compliant system-under-test produces an output. We therefore cause tests that receive a **fail** verdict while in a state without any outgoing transition to receive an **unexplored** verdict instead. However, the weak trace equivalence reduction may have merged some unexplored states with explored states, adding transitions to the unexplored states and making our method for detecting false positives an *under-approximation.* In other words, not all false positives are automatically removed, and we have to determine for each failed test that it is indicative of a fault in the test and/or simulator. (It is possible to mark unexplored states *before* the weak trace equivalence reduction – with transitions with a special label for example – but we have not done this at this stage of our research.)

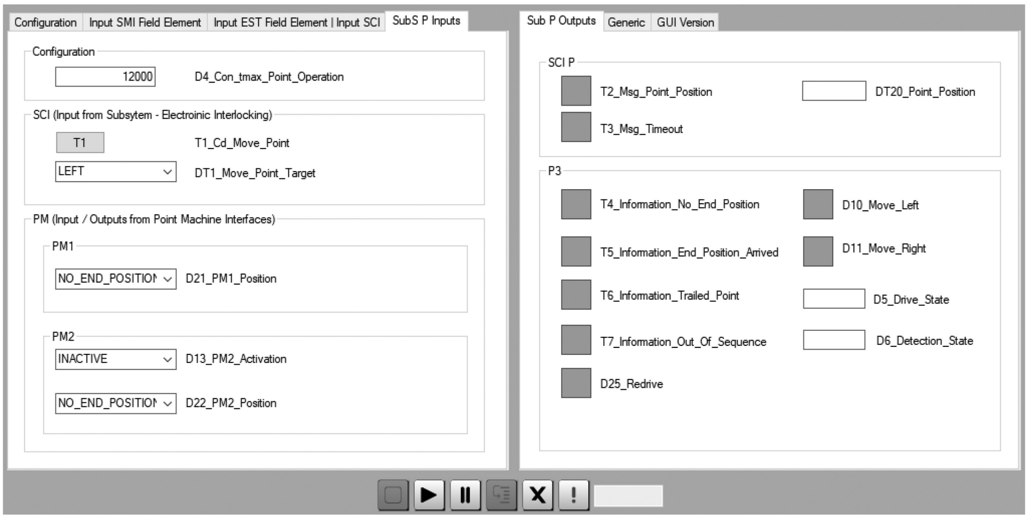
Figure 16 gives an overview of the tool chain that we use to obtain an incomplete LTS for automated testing. As for verification, we first parse the text-based mCRL2 specification (2) that has resulted from our translation with `mcrl22lps` to an LPS (4), a process that we mention in Section 4. Using information that is generated by our automated translation (3), we then use `lpsrename` to rename all internal transitions of the system-under-test to the special action τ. We explore the new LPS (5) for a certain amount of time with `lps2lts`, yielding the explicit (as opposed to symbolic) states and transitions of an LTS that describes a part of the model behavior (6). We use `ltsconvert` to reduce the LTS modulo weak trace equivalence. `ltsconvert` saves the new LTS (7) in the textual Aldebaran format (developed for use in the formal analysis toolkit CADP [19]).

6.2 System-Under-Test

The target of our model-based testing activities is a *software simulator* of the EULYNX Point interface, developed by the SIGNON Group. Normally, the simulator is intended for human interaction, and therefore presents a **graphical user interface (GUI)**; see Figure 17. Inputs are available as buttons and drop-down boxes on the left, outputs are displayed as colored boxes and text fields on the right, and controls to start and step through the simulation can be found at the bottom. The software simulator is normally used by signaling engineers to explore the behavior of EULYNX models.

The SIGNON Group simulator was the best option for a system-under-test for our experiments, since real implementations of EULYNX interfaces are currently not available. Just like our mCRL2 model, the simulator was generated from SysML semi-automatically, but there are sufficient differences between the semantic interpretations of the SysML diagrams that comparing the simulator to the mCRL2 model is non-trivial.

The SIGNON Group provided us access to the source code so that we can programmatically access the GUI elements and their data values. The source code of the Point simulator, in the


Configuration | Input SMI Field Element | Input EST Field Element | Input SCI | SubS P Inputs

Configuration
12000 D4_Con_tmax_Point_Operation

SCI (Input from Subsytem - Electroinic Interlocking)
T1 T1_Cd_Move_Point
LEFT DT1_Move_Point_Target

PM (Input / Outputs from Point Machine Interfaces)
PM1
NO_END_POSITION D21_PM1_Position
PM2
INACTIVE D13_PM2_Activation
NO_END_POSITION D22_PM2_Position

Sub P Outputs | Generic | GUI Version

SCI P
T2_Msg_Point_Position
DT20_Point_Position
T3_Msg_Timeout

P3
T4_Information_No_End_Position
D10_Move_Left
T5_Information_End_Position_Arrived
D11_Move_Right
T6_Information_Trailed_Point
D5_Drive_State
T7_Information_Out_Of_Sequence
D6_Detection_State
D25_Redrive

Fig. 17. A view of the GUI of the Point simulator. Inputs are located on the left, outputs on the right, and simulation controls at the bottom.

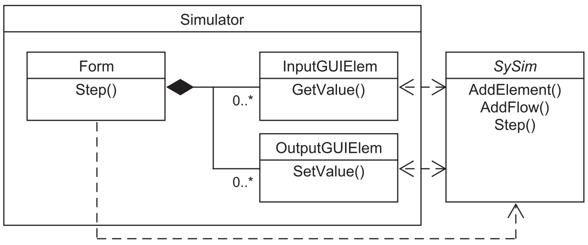


Fig. 18. Simplified class diagram of the Point simulator.

object-oriented programming language Visual Basic, is generated by the PTC Windchill software, and depends on the PTC Windchill library ‘SySim’, which determines the “flow” of a simulation, i.e. the times when data is read from or written to GUI elements and when the effects of SMD transitions are applied. GUI elements are created and managed by ‘Form’, the main class of the implementation, and they interact with the SySim library at the start of the program to define which ports and blocks exist and how flows connect them. See Figure 18 for a simplified class diagram of the simulator.

Modifications. We have modified the source code of the simulator to automatically execute tests based on an mCRL2 sub-model. We have added three classes to the simulator, namely ‘TestGenerator’, ‘Model’, and ‘Adapter’. The class diagram in Figure 19 gives a simplified overview of the relationships between the classes. We still make use of the ‘Form’ class, but only of its functional behavior, not of its GUI. Because we affect the simulation only via (the functional parts of) its GUI elements, it behaves as if manual testing were performed on the unmodified simulator.

‘TestGenerator’ contains the new entry point of the simulator, which creates an instance of Form and which then makes calls to ‘DoTest()’ repeatedly. This method performs test steps until (i) an unexpected output is detected, (ii) the simulator does not generate outputs and there are no available inputs according to the incomplete LTS, or (iii) a predetermined limit on the number of test

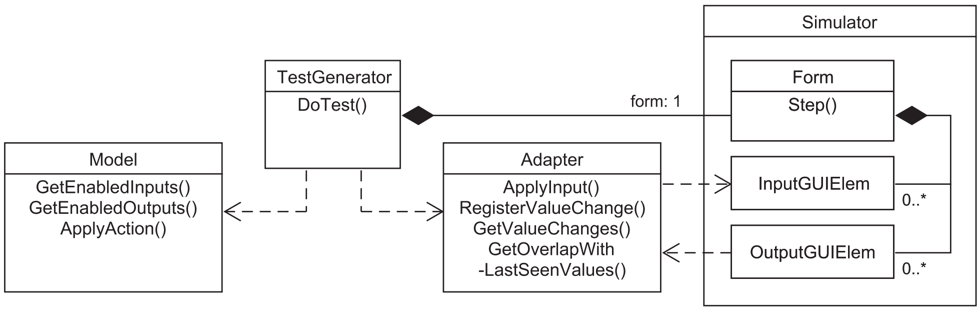


Fig. 19. Class diagram of the modified source code of the Point simulator.

steps has been reached. Gathering information on the incomplete LTS is delegated to the 'Model' class; the conversion of mCRL2 actions to simulator inputs/outputs and vice versa is handled by the 'Adapter' class. Another responsibility of 'Adapter' is the registration of simulator outputs, of which it is notified by the OutputGUIElems themselves.

Language compatibility. Actions in the mCRL2 model are port-value pairs, because they set a specific port to a certain value. We denote one such pair with (x, v), where x is the port and v is its new value. The simulator, on the other hand, consumes/publishes the values of all ports *simultaneously* each time that it takes an action. We must therefore express its actions as *sets of* port-value pairs.

For example, the assignments in the transition in Figure 20 would result in a sequence of multiple actions in the LTS of the mCRL2 model; see Figure 21. We make actions so fine-grained[7] because we want to model as much different interference behavior between state machines as possible. The core of the simulator (SySim) also executes the assignments sequentially, but for GUI elements the new values of the variables become visible at the same time (see Figure 22).

We make the language of the mCRL2 model and the language of the simulator compatible by extracting one by one the port-value pairs from a simulator action. We base the order in which we extract port-value pairs on a manual ordering of all ports; this works because EULYNX specifications are consistent in the order in which values are assigned to ports.

A related problem is that the simulator sends an update (x, v) to the GUI element of a port x at every time step of the simulation, regardless of whether an assignment to x has occurred (in which case v is its new value) or not (in which case v is its current value). It seems, therefore, as if all simulator ports are updated at every step, which is not the case in the mCRL2 model. We can greatly reduce this dissimilarity by ignoring GUI port updates that do not change the value of their ports. This leaves situations in which the mCRL2 model can do an action (x, v) when the value of port x already is v in both the mCRL2 model and the simulator. When such a situation occurs, we greedily take the (x, v) action in the mCRL2 model, knowing that, for our specific mCRL2 models, actions that are enabled *before* (x, v) are also enabled *after* (x, v). Alternatively, we could prevent the mCRL2 model from producing (x, v) in the first place, but then we could not use the same mCRL2 model for testing systems in which (x, v) *is* visible.

Test loop. We visualize the entire test loop in Figure 23. At the start of the loop (2), the actions that are currently enabled are retrieved from 'Model'. We determine each action (x, v) that is enabled

[7]Partially in response to our fine-grained interpretation of EULYNX actions, the EULYNX modeling standard has been modified to state explicitly that all effects of a transition take place simultaneously; in other words, the semantics of the simulator in this context have been officially adopted.

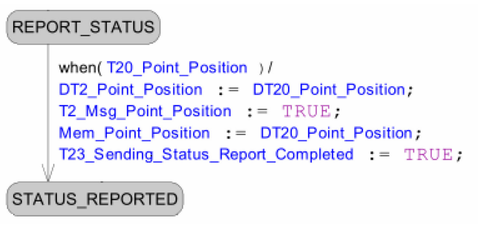

REPORT_STATUS
when(T20_Point_Position)/
DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
Mem_Point_Position := DT20_Point_Position;
T23_Sending_Status_Report_Completed := TRUE;
STATUS_REPORTED

Fig. 20. Transition from the STM in Figure 9. In the effect of the transition, four assignments occur. Mem_Point_Position happens to be an internal variable, and assignments to it are not visible to the environment.

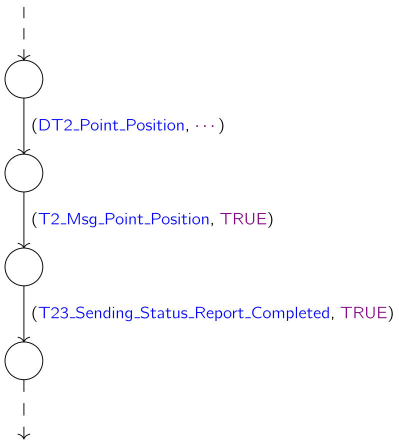

(DT2_Point_Position, ···)
(T2_Msg_Point_Position, TRUE)
(T23_Sending_Status_Report_Completed, TRUE)

Fig. 21. A possible occurrence of the assignments from Figure 20 as actions in the LTS that represents the mCRL2 model. The actions are expressed as a pair: the first element is the assigned port, the second element is the assigned value. Actions by parallel components may interleave the assignment actions, but the ordering of the assignment actions must remain the same.

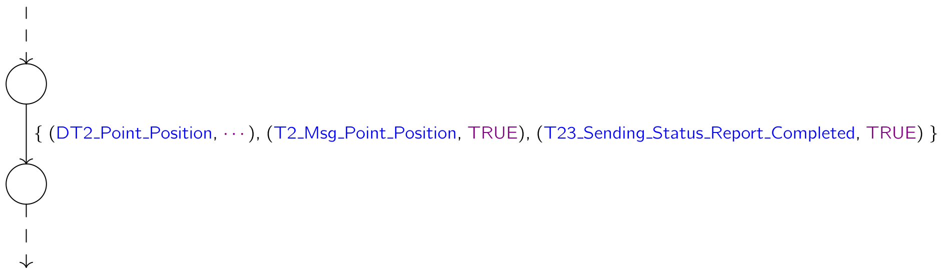

{ (DT2_Point_Position, ···), (T2_Msg_Point_Position, TRUE), (T23_Sending_Status_Report_Completed, TRUE) }

Fig. 22. From the outside, the effect of the transition from Figure 20 as produced by the simulator could be accurately represented in an LTS with a single action expressed as a set of port-value pairs.

in the mCRL2 model and of which the GUI element that corresponds with the port x currently has the value v (3); such actions are processed first (4). Afterwards, we check if the test has become stuck (5), which happens when the simulator will not provide an output on its own – 'last_output' is equal to 'NO_OUTPUT' in that case – and when there are no enabled input actions.

If the test can continue, we choose (6) between stimulating the simulator with an input action (if one is enabled) and observing outputs. When stimulating (7), an available input action is applied to both 'Adapter' and 'Model', and 'last_output' is cleared. When observing, actions that were detected because of value changes are retrieved from 'Adapter' (8). If there are none, we do a time step and try again up to n times, where n is at least the number of time steps in the longest timeout event of the simulator (we must ensure that timeout behavior is included in our tests). If n attempts still do not produce any actions, we conclude that we must send a new input before the simulator produces a new output (meaning that the simulator is *quiescent*). Naturally, this approach is insufficient if the timeouts of the system-under-test do not adhere to their prescribed durations.

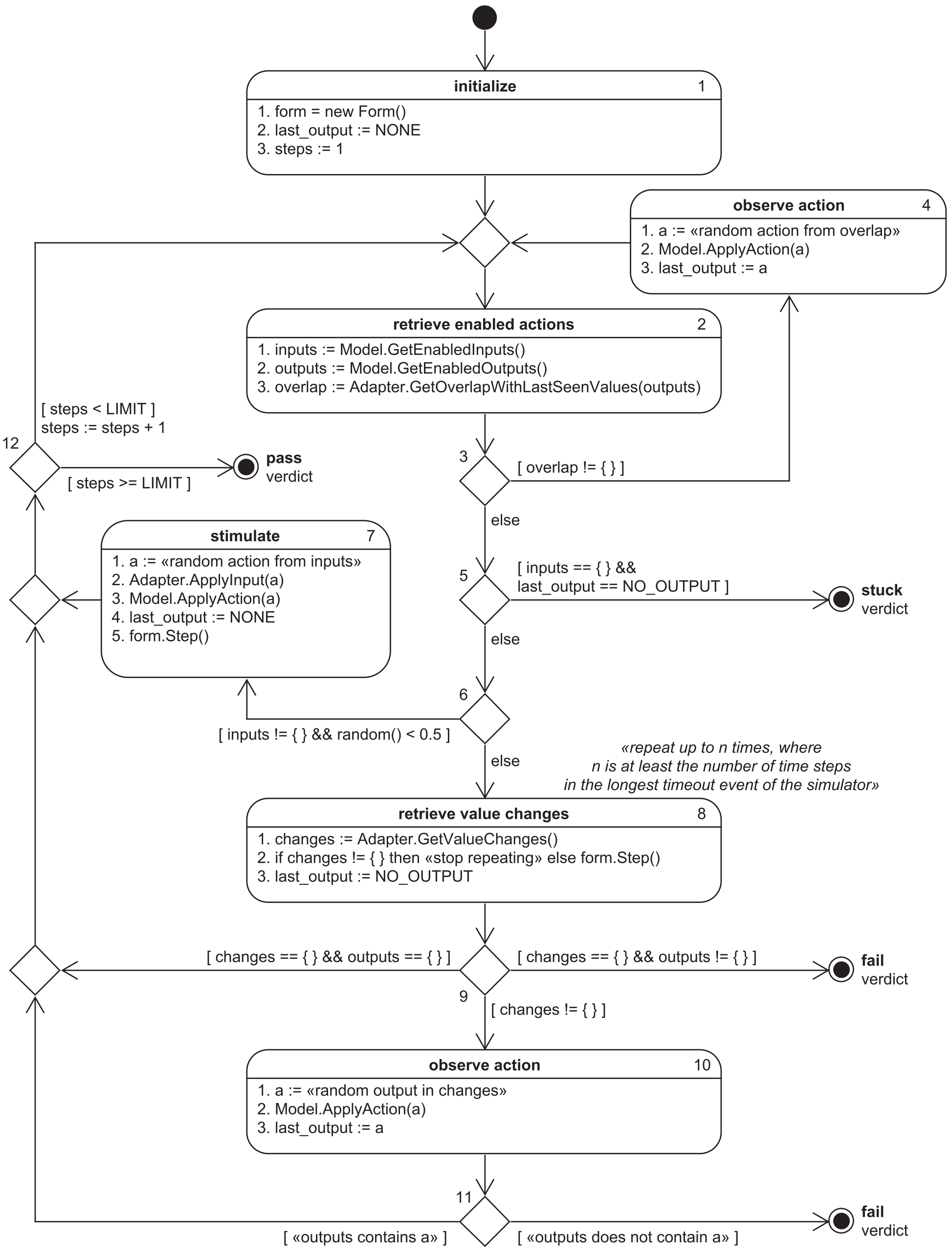

initialize 1
1. form = new Form()
2. last_output := NONE
3. steps := 1
observe action 4
1. a := «random action from overlap»
2. Model.ApplyAction(a)
3. last_output := a
retrieve enabled actions 2
1. inputs := Model.GetEnabledInputs()
2. outputs := Model.GetEnabledOutputs()
3. overlap := Adapter.GetOverlapWithLastSeenValues(outputs)
[steps < LIMIT]
steps := steps + 1
12
[steps >= LIMIT]
pass verdict
3
[overlap != { }]
else
stimulate 7
1. a := «random action from inputs»
2. Adapter.ApplyInput(a)
3. Model.ApplyAction(a)
4. last_output := NONE
5. form.Step()
5
[inputs == { } && last_output == NO_OUTPUT]
stuck verdict
else
6
[inputs != { } && random() < 0.5]
else
«repeat up to n times, where n is at least the number of time steps in the longest timeout event of the simulator»
retrieve value changes 8
1. changes := Adapter.GetValueChanges()
2. if changes != { } then «stop repeating» else form.Step()
3. last_output := NO_OUTPUT
[changes == { } && outputs == { }]
[changes == { } && outputs != { }]
fail verdict
9
[changes != { }]
observe action 10
1. a := «random output in changes»
2. Model.ApplyAction(a)
3. last_output := a
11
[«outputs contains a»]
[«outputs does not contain a»]
fail verdict

Fig. 23. Behavior of the ‘DoTest()’ method of the ‘TestGenerator’ class.

We finish the test step or yield a **fail** verdict depending on whether the retrieved actions are consistent with the incomplete LTS or not (9 to 11). Inconsistent behavior results in an **unexplored** verdict when the incomplete LTS specifies no inputs and no outputs. (Remember from Section 6.1 that this is an under-approximation of false positives, meaning that there may be tests that receive a **fail** verdict that should have received an **unexplored** verdict.)

Table 3. Discrepancies Between the mCRL2 Model and the Point Simulator, and Their Causes

Discrepancy	**Cause**
F_P3_Gen seems to produce the following outputs in the wrong order: (T4_Information_No_End_Position, TRUE) and (D6_Detection_State, "NO_END_POSITION") (T5_Information_End_Position_Reached, TRUE) and (D6_Detection_State, "END_POSITION")	The outputs are extracted from the simulator in an order that is different than their order of appearance in the mCRL2 model.
When communication is interrupted or times out while simulated S_SCI_EfeS_Prim_SR subcomponent is in the state 'ESTABLISHED_PDI_CONNECTION', the subcomponent produces the output (T12_Terminate_SCP_Connection, TRUE), which is not possible in the mCRL2 model.	An unexplored state was reached in the mCRL2 model; in the full model, the output is available.
Two minutes after the input (T1_Power_On_Detected, TRUE), the simulated F_P3_Gen subcomponent produces the output (T4_Information_No_End_Position, TRUE), which is not possible (yet) according to the mCRL2 model.	The output is triggered by F_EST_EfeS_SR moving to the state 'INITIALISING' 12.000 time steps (= 2 minutes) after entering the state 'BOOTING'. This is *incorrect* (unspecified) simulator behavior.

At (12), we check if the test limit has been reached. We choose a value of LIMIT that is considerably higher than the depth of the incomplete LTS; consequently, it is essentially impossible for a test to receive a **pass** verdict.

Random vs. guided testing. The choice between stimulation and observation (6) and the choice of an input (7) are both *random* in Figure 23. We also explore another strategy, which we call 'guided': this strategy only makes choices that, according to the model, most quickly lead to the occurrence of a predetermined transition (falling back on random choices when this transition is no longer reachable). Note that the outputs of the system-under-test can prevent the guided strategy from reaching its target transition.

6.3 Results & Performance

From the full mCRL2 model – which combines the generic layer of the communication with the point-specific layer; see Section 4, Model versions – we generated according to the process described in Section 6.1 an incomplete LTS over the course of ~48 hours. The LTS consists of 151 states and 628 transitions (~200.000 states and ~1 million transitions before weak trace equivalence reduction). We used the LTS to automatically perform 1,000 tests with the modified Point simulator (machine: AMD Ryzen 5 3600, 3.59 GHz). We have done this both with random testing and with guided testing.

After performing tests, we evaluated the results, and we repeated the tests if the cause of a failed test was in the mCRL2 sub-model or in the test setup, and if it was feasible for us to fix it. We also did this when the cause of a failed test was in the simulator and it was feasible for us to "bypass" it (potentially discovering new causes of failure in the next iteration). Naturally, we recorded all problems; we give an overview of these problems in Table 3.

First, we discovered that the order in which we extract outputs from the GUI elements of the simulator was incorrect for the F_P3_Gen subcomponent. We fixed the ordering, and no more problems of this nature were encountered.

Second, we found two situations in which our method to avoid false positives (by determining whether one of the current potential states in the model has no outgoing transitions) was inadequate, so that simulator outputs were (incorrectly) being rejected by our testing algorithm. We

Table 4. Metrics of Two Test Suites, One Based on Random Testing and One on Guided Testing

Method	#Fails	State coverage	Transition coverage	Avg. #steps	Time (s)
Random	37	89 (59%)	313 (50%)	18.0	4305
Guided	51	90 (60%)	381 (61%)	7.6	903

have identified these situations as false positives by replaying the test by hand in the complete mCRL2 model.

Our third and final finding is that the implementation of F_EST_EfeS_SR has an unspecified timeout that causes it to autonomously enter a state 'INITIALISING', which, according to the state machine diagram, can only be entered in response to events from the outside. We are confident that this is incorrect behavior of the simulator, and suspect that the behavior is a remnant from an older version of the simulator. We were able to identify the port that determines the length of the illegal timeout, which can be set to MAX_INT to disable the event.

In the last series of tests that we executed, only tests that failed due to false positives remained. We share several interesting metrics of these tests (see Table 4):

- The number of tests that were labeled with a **fail** verdict (all false positives).
- The number of states/transitions in the mCRL2 model that were encountered during testing, and the percentage that this number is relative to the total number of states/transitions in the mCRL2 sub-model. Note that we know in advance that some states and transitions are unreachable because we prioritize certain outputs (actions that are enabled in the mCRL2 model but which do not cause changes in GUI elements of the simulator) over others. Still, the shown values are a convenient metric for comparing the two test suites.
- The average number of steps (inputs/outputs) per test. This number gives us an impression of the extent to which unexplored parts of the sub-model were reached.
- The time required for each test suite, providing a measure of efficiency.

We make the following general observations. First, as expected, neither test suite covered all states/transitions of the model. Second, both test suites encountered differences between the model and the simulator. The guided test suite encountered more differences than the random test suite, which is consistent with the moderately greater coverage that the guided test suite achieves. Third and finally, the guided test suite is over four times faster than the random test suite. This is explained partially by the fact that guided tests are more likely to stimulate the simulator with an input instead of observing its outputs, which (since quiescence is a possible output) is a more expensive activity, on average. In addition, guided tests are probably much faster in reaching the unexplored part of the mCRL2 model and stop; this suspicion is supported by the lower average length of guided tests.

7 DISCUSSION

The case study in this paper is a step towards using formal verification to improve the EULYNX standard and model-based testing to confirm compliance of delivered components. In this section we reflect on improvements compared to [10] as well as on known and potential shortcomings in our work.

In the Point case study, we have applied a procedure that follows the model-centered principle of FormaSig (see Figure 1) to the EULYNX Point interface. Because we have automated several steps in the procedure that had to be performed manually before, the procedure has become much more efficient. For example, we now automatically generate the mCRL2 model from a number of text files that define SysML diagrams, which is much easier than inputting the Point interface in mCRL2 directly. For now, producing these text files themselves is still a manual (and therefore error-prone) process.

For performance reasons, we removed subcomponents from the mCRL2 model of Point when verifying requirements that did not (directly) involve those subcomponents; for example, REQ_PDI_002 was verified on a model from which point-specific behavior (S_SCI_P_SR, F_SCI_P_SR, F_P3_Gen) was removed. Such optimizations are not always sufficient; for example, REQ_P_002 could not be verified after removing generic subcomponents from the model (the root of the bottleneck seems to be located in the Point-specific F_P3_Gen subcomponent, arguably because of its state machine's parallel regions; see Section 5.4).

The verification results only inform us about the correctness of behavior modeled in EULYNX. In particular, we cannot analyze whether EULYNX components interact correctly with their environment (core interlocking, RaSTA, point machines) because we do not model the environment. To illustrate that this may cause this method to miss errors, in a (still ongoing) case study of the EULYNX Level Crossing interface we also modeled the behavior of RaSTA. Preliminary results indicate a mismatch between the interface with RaSTA as modeled in EULYNX and as specified in the RaSTA specification. This mismatch results in a deadlock, which is also present in the Point interface. The deadlock was not discovered in the Point case study since, like EULYNX, we treated RaSTA as part of the environment (as discussed in Section 4).

With regard to testing, we were able to improve performance considerably relative to our earlier work [10]. The main reason for the improvement is that we are now hooking directly into the source code of the system-under-test, instead of interfacing with its GUI. The number of test steps has increased from approximately one every four seconds (on average) to four and eight test steps per second for random and guided testing, respectively (remember that guided testing is faster than random testing because it is more likely to stimulate the simulator instead of observing it; see Section 6.3). In other words, we improved the performance of testing by a factor of 16 to 32.

Finally, the current test results clearly indicate some behavior of the simulator that is not found in the EULYNX specification of Point; the simulator can therefore be said to contain a fault. Because the behavior occurs after waiting for 2 minutes, it is unlikely that the fault would have been detected through manual testing, which clearly demonstrates the value of automated model-based testing. On the other hand, the tests are generated from a *partial* LTS; consequently, there is a significant amount of behavior of the EULYNX Point interface that is not covered. Moreover, the partial LTS comes with unexplored states, which cause false positives that make the current evaluation of test results non-trivial.

Threats to Validity

An important question arises when applying formal methods: Can we trust the results? For the FormaSig project we identify the following possible points of failure:

(1) The translation from SysML to mCRL2;
(2) The encoding of SysML models in our framework;
(3) The formalization of requirements in the modal μ-calculus;
(4) The mCRL2 toolset;
(5) The test generation approach;
(6) The adapter between the mCRL2 model and the simulator.

For all hazards it holds that incorrectly failing tests and formulas that do not hold for the model are not very severe (although they can be cumbersome to identify). For example, if a μ-calculus formula evaluates to false, we simply check the counterexample provided by the tools and manually replay the scenario on the original SysML model.

1. Translation from SysML to mCRL2. Establishing that the translation form SysML to mCRL2 is fully correct is difficult due to complexity of the resulting mCRL2 model. At the moment we verify the mCRL2 model produced by our translation framework by stepping through a part of the model using `lpsxsim`, checking whether the behavior is as intended.

In the future we would like to align the semantics of our models to the **Precise Semantics of State Machines (PSSM)**. This is a standard from the OMG group, which also manages the UML and SysML standards. PSSM very precisely (but not formally) specifies the semantics of state machines and includes a test suite. As PSSM only specifies the behavior of a single state machine we would need to add test cases that cover interaction between state machines over ports.

2. Encoding the SysML models. The measures taken to check the translation framework will also catch errors in the encoding of the SysML models. Moreover, the translation framework itself performs a number of sanity checks, such as whether state machine diagrams contain states that are not connected to the initial state via transitions, or states without outgoing transitions.

3. Formalization of requirements. To mitigate the risk of specifying incorrect formulas we check whether the formula is trivially true or false (which can be deduced from output of the `pbessolve` tool). Moreover, the μ-calculus formulas are checked by multiple people from the FormaSig project. In the future we would like to research formal (visual) requirements languages that can be understood by signaling engineers, allowing them to validate the formalized requirements.

4. mCRL2 toolset. The correctness of our findings relies on the correctness of the mCRL2 toolset. Verifying the correctness of the mCRL2 toolset is outside the scope of FormaSig.

5. Test generation. Our test generation algorithm is a custom implementation of well-known MBT techniques. The custom implementation could contain errors, but this would more than likely reveal itself during testing (especially guided testing). Essentially, we use the correct behavior of the simulator to validate our test environment.

The test generation algorithm is configured with a maximum test depth LIMIT and with the NO_OUTPUT limit (the number of times that the simulator must not produce an output before we conclude that no output is coming). We never reach the maximum test depth, so LIMIT is sufficiently high. We found a fault after a delay that is longer than the longest timeout event of the simulator, which validates the reasoning behind the value chosen for the NO_OUTPUT limit.

6. Adapter. Our adapter between the model and the system-under-test is more complicated than is typical because of the language differences (see Section 6.2, Language compatibility). The additional complexity may lead to an incorrect implementation, and – although the adapter is validated to some extent by the successful test activities – it may be worthwhile to modify the mCRL2 model so that the adapter can be simplified.

8 CONCLUSION AND FUTURE WORK

We have shown with our extended and improved case study of the EULYNX Point interface that our automated translation produces mCRL2 models that are amenable to formal verification and model-based testing. Among other things, we discovered that some requirements of the Point interface only hold under fairness assumptions, and that there exist some discrepancies between the Point model and the Point simulator, including one fault. We also gained insight into the scalability of verification and automated testing.

Since the ultimate goal of FormaSig is to formally analyze all 10 EULYNX interfaces, we plan to continue streamlining our approach for applying formal methods to EULYNX interfaces. In particular, we want to be able to extract SysML diagrams directly from PTC Windchill files instead of transcribing them by hand to a text file. We also want to expedite requirement elicitation by using diagrams to visualize modal μ-calculus formulas and mCRL2 counterexamples for signaling experts. We have yet to select a suitable type of diagram; a possible candidate is the *live sequence chart* [12].

More fundamentally, we will invest time in our formalization of SysML diagrams in mCRL2. First, we intend to address a possible performance bottleneck caused by the way in which parallel regions of state machines are modeled in mCRL2. Second, we want to add an option to our automated translation to produce an mCRL2 model in which actions that do not change a port value (see Section 6.1, Language compatibility) do not occur, so that test execution becomes more reliable and elegant. We may also change our mCRL2 models so that they produce *simultaneous* outputs, so that they become directly language compatible with systems-under-test such as the Point simulator (because the semantics of simultaneous outputs have been officially adopted by EULYNX in the time between performing the experiments of the case study and finalizing the article). Third, we want to explore making (a number of the) communication channels synchronous (as opposed to asynchronous), in particular communication channels between subcomponents that in practice will be executed on the same physical device. This would reduce the number of states, making verification easier. Fourth and finally, we want to investigate if we can add explicit time to our mCRL2 models, since we have come across several requirements with time constraints in the course of the Point case study. This would require EULYNX specifications to provide more information than is currently the case, such as the time it takes for a message to travel to its destination.

We also plan to improve our test setup. Naturally, testing based on an incomplete LTS that is computed in advance is only a stepping stone towards a more effective approach, which would involve either the computation of a complete LTS or an LTS that is explored on-demand. To this end, we already started experimenting with the **Windows Subsystem for Linux**[8] **(WSL)**, which makes it possible to achieve mCRL2 performance on the Windows platform that is comparable to that of Unix platforms. If this performance is insufficient, we will work on combining symbolic model exploration techniques with MBT.

After achieving an effective MBT method, our attention will be directed at improving the *quality* of generated tests, where quality could be defined in various ways: it could be based on decision/transition coverage (i.e. the fraction of the transitions of state machines or LTSs that tests touch [13, 42]), or on the likelihood of tests to detect *mutants* [27], implementations to which random mistakes are deliberately added.

APPENDICES

A REQUIREMENTS

This appendix lists all the requirements of Section 5.3 with all the μ-calculus formulas. Unlike in other places in the paper, where we reference subcomponents by their type name, we give subcomponents unique instance names in these formulas (this way, we can support multiple instances of the same subcomponent type). Use Table 5 to convert the instance name of a subcomponent to its type name, and vice versa.

[8]https://docs.microsoft.com/en-us/windows/wsl/about.

Table 5. Type Names and Instance Names that Correspond with each Other

Instance name	**Type name**
BEQ_pe51	F_P3_Gen
BEQ_fp	F_SCI_P_SR
BEQ_sp	S_SCI_P_SR
BEQ_prim	S_SCI_EfeS_Prim_SR
BEQ_seec	F_SCI_EfeS_Sec_SR
BEQ_eest	F_EST_EfeS_SR
BEQ_smi	F_SMI_EfeS_SR

Brief Introduction to the μ-calculus

mCRL2 uses a first-order modal μ-calculus extended with data and regular formulas; see Table 6 for the most important operators. We refer to the book on mCRL2 [20] for a more in-depth treatment of the logic.

Table 6. Main μ-calculus Operators Besides Standard Logical Connectives

Operator	**Meaning**
`<a>`ϕ	Diamond operator; there exists an a-labeled transition to a state where ϕ holds
`[a]`ϕ	Box operator; all a-labeled transitions lead to a state where ϕ holds
`mu X.` ϕ	Least fixed point operator
`nu X.` ϕ	Greatest fixed point operator

The diamond and box operators support regular formulas. The formula `[true*]`ϕ, from REQ_P_001, expresses "ϕ holds after any trace from the initial state"; `true` represents any action label and the `*` indicates a sequence of arbitrary length. Quantifiers can also be used in regular formulas: the formula `[!(exists st:StateName, c:CompName. inState(c,st))]`ϕ, also from REQ_P_001, specifies that ϕ must hold after any transition that is not labeled `inState`.

ID	REQ_P_001
Summary	The object controller must report changes in position
Detailed description	The position of the point is determined by combining the input from the point machines. When all point machines report right the position is right. When all point machines report left the position is left. In any other case the point does not have an end position. When the position of the point is different from the last reported position, the new position must be reported to the interlocking. This obligation is lifted if communication with the interlocking is not possible due to connection problem, power failure, etc. If the object controller cannot report the changed position, the connection with the interlocking must be closed, so that the interlocking knows that the position is unknown.

μ-calculus formula

```
% Once upon an arbitrary state vector:
[true*](%in which the point is in an end position
        (<inState(BEQ_pe51, ALL_LEFT)>true || <inState(BEQ_pe51, ALL_RIGHT)>true) =>
  % When PM1 or PM2 reports end position 'none':
  ([receive(CompPortPair(BEQ_pe51, D21_PM1_Position), Value_String(STR_NO_END_POSITION))|
        send(CompPortPair(BEQ_pe51_Environment, D21_PM1_Position), Value_String(
        STR_NO_END_POSITION))]
        (mu X. [!(exists st:StateName, c:CompName. inState(c,st))]X
        %and require that always eventually the no_end_position is reported to the
              interlocking
    || <receive(CompPortPair(BEQ_sp, T20_Point_Position), Value_String(
          STR_NO_END_POSITION))|send(CompPortPair(BEQ_sp_Environment,
          T20_Point_Position), Value_String(STR_NO_END_POSITION))>true
        %or the connection is no longer up
        || <inState(BEQ_pe51,WAITING_FOR_INITIALISING)>true
        || [inState(BEQ_fp,PDI_CONNECTION_ESTABLISHED)]false
        || [inState(BEQ_sp,PDI_CONNECTION_ESTABLISHED)]false)))
```

Alternative formula REQ_P_001_1 (see Section 5.4 for the rationale of this requirement):

```
% Once upon an arbitrary state vector:
[true*](%in which the point is in an end position
        (<inState(BEQ_pe51, ALL_LEFT)>true || <inState(BEQ_pe51, ALL_RIGHT)>true) =>
  % When PM1 or PM2 reports end position 'none':
  ([receive(CompPortPair(BEQ_pe51, D21_PM1_Position), Value_String(STR_NO_END_POSITION))|
        send(CompPortPair(BEQ_pe51_Environment, D21_PM1_Position), Value_String(
        STR_NO_END_POSITION))]
        % We examine all paths for which the environment is silent
        (mu X. [!(exists st:StateName, c, env: CompName, p1, p2: VarName, v: Value.
                (val(env in [BEQ_sp_Environment, BEQ_seec_Environment,
                      BEQ_eest_Environment, BEQ_fp_Environment, BEQ_pe51_Environment,
                      BEQ_prim_Environment]))
                || inState(c,st))]X
    %and require that always eventually the no_end_position is reported to the
          interlocking
    || <receive(CompPortPair(BEQ_sp, T20_Point_Position), Value_String(
          STR_NO_END_POSITION))|send(CompPortPair(BEQ_sp_Environment,
          T20_Point_Position), Value_String(STR_NO_END_POSITION))>true
        %or the connection is no longer up
        || <inState(BEQ_pe51,WAITING_FOR_INITIALISING)>true
        || [inState(BEQ_fp,PDI_CONNECTION_ESTABLISHED)]false
        || [inState(BEQ_sp,PDI_CONNECTION_ESTABLISHED)]false)))
```

ID	REQ_P_002
Summary	The object controller must not change position unless commanded by the interlocking
Detailed description	The object controller may only instruct the point machines to move when this is commanded by the interlocking. A movement command from the interlocking gives a pass to initiate movement in a certain position that ends when either a timeout occurs, the end position is reached or a movement command for the opposing position is sent.
***μ*-calculus formula**	

```
%Fixedpoint parameters track whether the object controller is allowed to move the
      point and in which direction
nu X(allowed: Bool = false, direction: Value = Value_String(STR_LEFT)).
%When a movement command for the position left is received, we update parameters
      allowed and direction to true and left, respectively
[send(CompPortPair(BEQ_sp,DT1_Move_Point_Target),Value_String(STR_LEFT))
        |receive(CompPortPair(BEQ_fp,DT1_Move_Point_Target),Value_String(STR_LEFT))]
              X(true,Value_String(STR_LEFT))
%When a movement command for the position right is received, we update parameters
      allowed and direction to true and right, respectively
&& [send(CompPortPair(BEQ_sp,DT1_Move_Point_Target),Value_String(STR_RIGHT))
        |receive(CompPortPair(BEQ_fp,DT1_Move_Point_Target),Value_String(STR_RIGHT))
              ]X(true,Value_String(STR_RIGHT))
%When movement of the point times out, it may no longer be moved. Parameter allowed
      is set to false.
&& [send(CompPortPair(BEQ_fp,T3_Msg_Timeout),Value_Bool(true))
        |receive(CompPortPair(BEQ_sp,T3_Msg_Timeout),Value_Bool(true))]X(false,
              direction)
%When the end position is reached, it may no longer be moved. Parameter allowed is
      set to false.
&& [send(CompPortPair(BEQ_fp,DT2_Point_Position),direction)
        |receive(CompPortPair(BEQ_sp,DT2_Point_Position),direction)]X(false,
              direction)
%When allowed is false or the direction is not right then the object controller may
      not start a movement to the right
&& ((val(!allowed || (direction != Value_String(STR_RIGHT)))) => [send(CompPortPair(
      BEQ_pe51,D11_Move_Right),Value_Bool(true))
        |receive(CompPortPair(BEQ_pe51_Environment,D11_Move_Right),Value_Bool(true))
              ]false)
%When allowed is false or the direction is not left then the object controller may
      not start a movement to the left
&& ((val(!allowed || (direction != Value_String(STR_LEFT)))) => [send(CompPortPair(
      BEQ_pe51,D10_Move_Left),Value_Bool(true))
        |receive(CompPortPair(BEQ_pe51_Environment,D10_Move_Left),Value_Bool(true))]
              false)
%For any other transition than the transitions above the parameters allowed and
      direction stay the same
&& [!(send(CompPortPair(BEQ_sp,DT1_Move_Point_Target),Value_String(STR_LEFT))|
      receive(CompPortPair(BEQ_fp,DT1_Move_Point_Target),Value_String(STR_LEFT)))
        && !(send(CompPortPair(BEQ_sp,DT1_Move_Point_Target),Value_String(STR_RIGHT)
              )|receive(CompPortPair(BEQ_fp,DT1_Move_Point_Target),Value_String(
              STR_RIGHT)))
        && !(send(CompPortPair(BEQ_fp,T3_Msg_Timeout),Value_Bool(true))|receive(
              CompPortPair(BEQ_sp,T3_Msg_Timeout),Value_Bool(true)))
        && !(send(CompPortPair(BEQ_fp,DT2_Point_Position),direction)|receive(
              CompPortPair(BEQ_sp,DT2_Point_Position),direction)
        )]X(allowed,direction)
```

ID	REQ_PDI_002
Summary	Disconnect at version unequal
Detailed description	Whilst establishing a connection, in the case that the object controller sees that the interlocking uses a different version of the protocol, it sends a message to the interlocking notifying the failed version check and moves to the state not ready for a connection.
μ-calculus formula	

```
[true*.send(CompPortPair(BEQ_seec,DT13_Result),Value_String(STR_note32match))|
      receive(CompPortPair(BEQ_prim,DT13_Result),Value_String(STR_note32match))]
%The object controller moves to the state Not_Ready_For_Connection before it is
      allowed to succesfully establsish a connection
((nu X. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false
    && [true]X)
|| <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true)
%The interlocking moves to the state Connection_closed before it is allowed to
      succesfully establsish a connection
&& (nu X. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]false
    && [true]X)
|| <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>true))
```

ID	REQ_PDI_003
Summary	Disconnect at checksum unequal
Detailed description	Whilst establishing a connection, in the case that the interlocking receives an incorrect checksum (indicating a malformed message) from the object controller in the Msg_PDI_Version_Check message, the interlocking terminates the connection.
μ-calculus formula	

```
[true*.inState(BEQ_prim,PDI_CHECKSUM_UNEQUAL)]
%The object controller moves to the state Not_Ready_For_Connection before it is
      allowed to succesfully establsish a connection
((nu X. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false && [true]X)
        || <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true || <inState(BEQ_seec,
              READY_FOR_CONNECTION)>true)
%The interlocking moves to the state Connection_closed before it is allowed to
      succesfully establsish a connection
&& (nu X. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]false && [true]X)
        || <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>true))
```

ID	REQ_PDI_004
Summary	A connection remains possible
Detailed description	As long as there is no telegram error, it always remains possible to reach PDI_Connection_Established in the future.
μ-calculus formula	

```
[true*]%after any trace
%we either have that PDI_Connection_Impermissible is inevitable
((<inState(BEQ_prim,PDI_CONNECTION_IMPERMISSIBLE)>true
  || <inState(BEQ_seec,PDI_CONNECTION_IMPERMISSIBLE)>true
  || <inState(BEQ_prim,PDI_TELEGRAM_ERROR)>true
  || <inState(BEQ_seec,PDI_TELEGRAM_ERROR)>true)
%or we can eventually establish a connection
|| (<true*>(<inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>true
  && <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true)))
```

ID	REQ_PDI_005
Summary	Close connection on error
Detailed description	At the moment a protocol error or telegram error occurs at either the side of the object controller or the interlocking, both the interlocking and the object controller will eventually move to PDI_Connection_Closed before reattempting a connection.
μ-calculus formula	

```
[true*]((
  (<inState(BEQ_seec,PDI_TELEGRAM_ERROR)>true || <inState(BEQ_seec,PDI_PROTobject
        controllerOL_ERROR)>true) =>
  (nu X. ([inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false && [true]X)
    || <inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true))
&& (
  (<inState(BEQ_prim,PDI_TELEGRAM_ERROR)>true || <inState(BEQ_prim,PDI_PROTobject
        controllerOL_ERROR)>true) =>
  (nu X. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]false && [true]X)
    || <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>true)))
```

ID	REQ_PDI_006
Summary	Close connection when not ready
Detailed description	When F_EST_EfeS_SR signals it is not ready for a connection, the object controller will move to not ready for connection and only reattempts a connection after receiving a message ready for connection.
μ-calculus formula	

```
[true*.send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),Value_Bool(true))
      |receive(CompPortPair(BEQ_seec,T18_Not_Ready_For_PDI_Connection),Value_Bool(
      true))](
%when in PDI_connection established, move to not ready for connection
((<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true)
  => (mu X. (([!(exists c:CompName,s:StateName. inState(c,s))]X)
    || [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false)))
%do not establish a connection until it is allowed again by F_EST_EfeS_SR
&& (nu X. [!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true)
      )|receive(CompPortPair(BEQ_seec,T1_Ready_For_PDI_Connection),Value_Bool(true))]
      X
  && (<inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true => (
    [(!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true))|
          receive(CompPortPair(BEQ_seec,T1_Ready_For_PDI_Connection),Value_Bool(true)
          ))*]              [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false))))
```

Subformula	REQ_PDI_006_1 (see Section 5.4 for the rationale of this requirement):

```
[true*.send(CompPortPair(BEQ_eest,T18_Not_Ready_For_PDI_Connection),Value_Bool(true))
      |receive(CompPortPair(BEQ_seec,T18_Not_Ready_For_PDI_Connection),Value_Bool(
      true))]
(nu X. [!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true))|
      receive(CompPortPair(BEQ_seec,T1_Ready_For_PDI_Connection),Value_Bool(true))]X
  && (<inState(BEQ_seec,NOT_READY_FOR_CONNECTION)>true => (
    [(!send(CompPortPair(BEQ_eest,T21_Ready_For_PDI_Connection),Value_Bool(true))|
          receive(CompPortPair(BEQ_seec,T1_Ready_For_PDI_Connection),Value_Bool(true)
          ))*]
    [inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)]false)))
```

ID	REQ_PDI_007
Summary	Close connection after timeout
Detailed description	When S_SCI_EfeS_Prim_SR does not change in state PDI_Connection_Established after entering state Establishing_PDI_Connection within the time D2_Con_tmax_PDI_Connection, both S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR reach PDI_Connection_Closed before reattempting a connection.
μ-calculus formula	

```
[true*](<inState(BEQ_prim,PDI_INIT_TIMEOUT)>true =>
%The interlocking moves to the state Connection_closed before it is allowed to
      succesfully establsish a connection
(nu X. ([inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)]false && [true]X)
  || <inState(BEQ_prim,PDI_CONNECTION_CLOSED)>true))
```

ID	REQ_PDI_008
Summary	One closes, both close
Detailed description	When S_SCI_EfeS_Prim_SR and F_SCI_EfeS_Sec_SR are both in state PDI_Connection_Established, and one leaves that state, the other will eventually leave the state as well.
μ-calculus formula	

```
[true*]((
  <inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>true
  && <inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true) =>[true](
    ((!<inState(BEQ_prim,PDI_CONNECTION_ESTABLISHED)>true)
      => mu X. [!(exists c:CompName,s:StateName. inState(c,s))]X || (<inState(
            BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true))
    && ((!<inState(BEQ_seec,PDI_CONNECTION_ESTABLISHED)>true)
      => mu X. [!(exists c:CompName,s:StateName. inState(c,s))]X || (<inState(
            BEQ_prim,PDI_CONNECTION_ESTABLISHED)>true))))
```

ACKNOWLEDGMENTS

We would like to thank Maurice Laveaux for his help in using the symbolic tools of mCRL2, and the Signon Group for providing access to the source code of the EULYNX Point simulator.

REFERENCES

[1] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[2] Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. 2018. On the industrial uptake of formal methods in the railway domain - A survey with stakeholders. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5–7, 2018, Proceedings (Lecture Notes in Computer Science)*, Carlo A. Furia and Kirsten Winter (Eds.), Vol. 11023. Springer, 20–29. https://doi.org/10.1007/978-3-319-98938-9_2

[3] Axel Belinfante. 2010. JTorX: A tool for on-line model-driven test derivation and execution. In *TACAS'10 Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 266–270.

[4] Axel Belinfante. 2014. *JTorX: Exploring Model-Based Testing*. Ph.D. Dissertation. University of Twente, Enschede, Netherlands. http://purl.utwente.nl/publications/91781.

[5] Jan A. Bergstra and Jan Willem Klop. 1985. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.* 37 (1985), 77–121. https://doi.org/10.1016/0304-3975(85)90088-X

[6] Andrea Bonacchi, Alessandro Fantechi, Stefano Bacherini, and Matteo Tempestini. 2016. Validation process for railway interlocking systems. *Sci. Comput. Program.* 128 (2016), 2–21. https://doi.org/10.1016/j.scico.2016.04.004

[7] Mark Bouwman, Bob Janssen, and Bas Luttik. 2019. Formal modelling and verification of an interlocking using mCRL2. In *24th International Conference on Formal Methods for Industrial Critical Systems, FMICS 2019*. 22–39.

[8] Mark Bouwman, Bas Luttik, and Djurre van der Wal. 2021. A formalisation of SysML state machines in mCRL2. In *Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, held as part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings (Lecture Notes in Computer Science)*, Kirstin Peters and Tim A. C. Willemse (Eds.), Vol. 12719. Springer, 42–59. https://doi.org/10.1007/978-3-030-78089-0_3

[9] Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. 2020. Off-the-shelf automated analysis of liveness properties for just paths. *Acta Informatica* 57, 3–5 (2020), 551–590. https://doi.org/10.1007/s00236-020-00371-w

[10] Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. 2020. What is the point: Formal analysis and test generation for a railway standard. In *Proceedings of ESREL2020-PSAM15*, Piero Baraldi, Francesco Di Maio, and Enrico Zio (Eds.). Research Publishing, Singapore, 921–928. https://doi.org/10.3850/978-981-14-8593-0_4410-cd

[11] J. P. Bowen, K. Bogdanov, J. A. Clark, M. Harman, R. M. Hierons, and P. Krause. 2002. FORTEST: Formal methods and testing. In *Proceedings 26th Annual International Computer Software and Applications*. 91–104.

[12] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut 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, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report (Lecture Notes in Computer Science)*, Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper (Eds.), Vol. 3147. Springer, 374–399. https://doi.org/10.1007/978-3-540-27863-4_21

[13] Laura Brandán Briones, Ed Brinksma, and Mariëlle Stoelinga. 2006. A semantic framework for test coverage. In *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October*

23–26, 2006 (Lecture Notes in Computer Science), Susanne Graf and Wenhui Zhang (Eds.), Vol. 4218. Springer, 399–414. https://doi.org/10.1007/11901914_30

[14] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. 2019. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Tomás Vojnar and Lijun Zhang (Eds.), Vol. 11428. Springer, 21–39. https://doi.org/10.1007/978-3-030-17465-1_2

[15] Samir Chouali and Ahmed Hammad. 2011. Formal verification of components assembly based on SysML and interface automata. *Innov. Syst. Softw. Eng.* 7, 4 (2011), 265–274. https://doi.org/10.1007/s11334-011-0170-3

[16] Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. 2010. *Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models*. Springer. https://doi.org/10.1007/978-3-642-15228-3

[17] Alessandro Fantechi. 2013. Twenty-five years of formal methods and railways: What next?. In *SEFM 2013 Collocated Workshops, Revised Selected Papers*. 167–183. https://doi.org/10.1007/978-3-319-05032-4

[18] Alessandro Fantechi, Francesco Flammini, and Stefania Gnesi. 2014. Formal methods for railway control systems. *Int. J. Softw. Tools Technol. Transf.* 16, 6 (2014), 643–646. https://doi.org/10.1007/s10009-014-0342-1

[19] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer* 15, 2 (2013), 89–107.

[20] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and Analysis of Communicating Systems*. MIT Press. https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems.

[21] Jan Friso Groote, Alban Ponse, and Yaroslav S. Usenko. 2001. Linearization in parallel pCRL. *J. Log. Algebraic Methods Program.* 48, 1–2 (2001), 39–70. https://doi.org/10.1016/S1567-8326(01)00005-4

[22] Anne E. Haxthausen, Marie Le Bliguet, and Andreas A. Kjær. 2008. Modelling and verification of relay interlocking systems. In *Monterey'08 Proceedings of the 15th Monterey Conference on Foundations of Computer Software: Future Trends and Techniques for Development*. 141–153.

[23] Anne E. Haxthausen and Jan Peleska. 2015. Model checking and model-based testing in the railway domain. In *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, Rolf Drechsler and Ulrich Kühne (Eds.). Springer, 82–121. https://doi.org/10.1007/978-3-658-09994-7_4

[24] Anne E. Haxthausen, Jan Peleska, and Sebastian Kinder. 2011. A formal approach for the construction and verification of railway control systems. *Formal Aspects of Computing* 23, 2 (2011), 191–219.

[25] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. *Comput. Surveys* 41, 2 (2009), 9.

[26] Phillip James, Faron Moller, Nguyen Hoang Nga, Markus Roggenbach, Steve A. Schneider, and Helen Treharne. 2014. Techniques for modelling and verifying railway interlockings. *Int. J. Softw. Tools Technol. Transf.* 16, 6 (2014), 685–711. https://doi.org/10.1007/s10009-014-0304-7

[27] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[28] Gijs Kant and Jaco van de Pol. 2014. Generating and solving symbolic parity games. In *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2014, Grenoble, France, 5th April 2014 (EPTCS)*, Dragan Bosnacki, Stefan Edelkamp, Alberto Lluch-Lafuente, and Anton Wijs (Eds.), Vol. 159. 2–14. https://doi.org/10.4204/EPTCS.159.2

[29] Marcos Vinicius Linhares, Rômulo Silva de Oliveira, Jean-Marie Farines, and François Vernadat. 2007. Introducing the modeling and verification process in SysML. In *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2007, September 25–28, 2007, Patras, Greece*. IEEE, 344–351. https://doi.org/10.1109/EFTA.2007.4416788

[30] Jeroen Johan Gerardus Meijer. 2019. *Efficient Learning and Analysis of System Behavior*. Ph.D. Dissertation. University of Twente.

[31] Aurelijus Morkevicius and Nerijus Jankevicius. 2015. An approach: SysML-based automated requirements verification. In *International Symposium on Systems Engineering (ISSE)*. IEEE, 92–97.

[32] Object Management Group. 2017. OMG Unified Modeling Language, version 2.5.1. (2017). https://www.omg.org/spec/UML/.

[33] Object Management Group. 2019. OMG Systems Modeling Language, version 1.6. (2019). https://www.omg.org/spec/SysML/.

[34] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. 2011. AVATAR: A SysML environment for the formal verification of safety and security properties. In *11th Annual International Conference on New Technologies of Distributed Systems, NOTERE 2011, Paris, France, 9–13 May 2011*, Isabelle M. Demeure, Thomas Robert, and Ahmed Serhrouchni (Eds.). IEEE, 1–10. https://doi.org/10.1109/NOTERE.2011.5957992

[35] Jean-François Pétin, Dominique Evrot, Gérard Morel, and Pascal Lamy. 2010. Combining SysML and formal methods for safety requirements verification. In *22nd International Conference on Software & Systems Engineering and their Applications*.

[36] W. G. J. Stokkink, M. Timmer, and M. I. A. Stoelinga. 2013. Divergent quiescent transition systems. In *Proceedings of the 7th International Conference on Tests And Proofs (TAP) (Lecture Notes in Computer Science)*, Margus Veanes and Luca Viganò (Eds.), Vol. 7942. Springer, 214–231. https://doi.org/10.1007/978-3-642-38916-0_13

[37] M. Timmer, H. Brinksma, and M. I. A. Stoelinga. 2011. Model-based testing. In *Software and Systems Safety - Specification and Verification*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 30. IOS Press, 1–32. https://doi.org/10.3233/978-1-60750-711-6-1

[38] G. J. Tretmans and Hendrik Brinksma. 2003. TorX: Automated model-based testing. *First European Conference on Model-Driven Software Engineering* (2003), 31–43.

[39] Jan Tretmans. 1996. Test generation with inputs, outputs and repetitive quiescence. *Softw. Concepts Tools* 17, 3 (1996), 103–120.

[40] Jan Tretmans. 2008. Model based testing with labelled transition systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.). Lecture Notes in Computer Science, Vol. 4949. Springer, 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

[41] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification & Reliability* 22, 5 (2012), 297–312.

[42] Petra van den Bos and Jan Tretmans. 2019. Coverage-based testing with symbolic transition systems. In *Tests and Proofs - 13th International Conference, TAP@FM 2019, Porto, Portugal, October 9–11, 2019, Proceedings (Lecture Notes in Computer Science)*, Dirk Beyer and Chantal Keller (Eds.), Vol. 11823. Springer, 64–82. https://doi.org/10.1007/978-3-030-31157-5_5

[43] Rob van Glabbeek and Peter Höfner. 2019. Progress, justness, and fairness. *ACM Comput. Surv.* 52, 4 (2019), 69:1–69:38. https://doi.org/10.1145/3329125

[44] VDE. 2015. Electric signalling systems for railways - Part200: Safe transmission protocol according to DIN EN50159 (DIN VDE V 0831-159). DIN VDE V 0831-200. (6 2015).

[45] Wikimedia Commons. 2021. File:Facing points Broomhill.jpg. (2021). https://commons.wikimedia.org/wiki/File:Facing_points_Broomhill.jpg. [Online; accessed July 9, 2021].

[46] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. 2017. Perceptions on the state of the art in verification and validation in cyber-physical systems. *IEEE Systems Journal* 11, 4 (2017), 2614–2627.

Received 12 July 2021; accepted 17 March 2022