Experimenting with Formal Verification and Model-based Development in Railways: the case of UMC and Sparx Enterprise Architect

Davide Basile(✉), Franco Mazzanti, and Alessio Ferrari

Formal Methods and Tools Lab
ISTI–CNR, Pisa, Italy
{davide.basile,franco.mazzanti,alessio.ferrari}@isti.cnr.it

Abstract. The use of formal methods can reduce the time and costs associated with railway signalling systems development and maintenance, and improve correct behaviour and safety. The integration of formal methods into industrial model-based development tools has been the subject of recent research, indicating the potential transfer of academic techniques to enhance industrial tools. This paper explores the integration of an academic formal verification tool, UML Model Checker (UMC), with an industrial model-based development tool, Sparx Enterprise Architect (Sparx EA). The case study being analyzed is a railway standard interface. The paper demonstrates how formal verification techniques from academic tools can be integrated into industrial development practices using industrial tools, and how simulation in Sparx EA can be derived from traces generated by the UMC formal verification activity. From this experience, we derive a set of lessons learned and research challenges.

Keywords: umc · sparx enterprise architect · formal verification · uml

1 Introduction

The adoption of formal methods and railway standard interfaces has been identified as crucial in reducing the time for developing and delivering railway signalling systems, as well as decreasing the high costs associated with procurement, development, and maintenance [15, 12, 47]. Formal methods tools are essential to ensure correct behaviour, interoperability of railway interfaces, and safety. Formal methods are mainly used and developed by academia, and their uptake in the railway industry has been the subject of recent studies [31, 33, 55, 36, 38, 41, 17, 10].

Model-based development is an industrially adopted software engineering technique that supports the creation of models to represent a system's behaviour and structure. These models are used to generate code, documentation, test cases, system simulations, and perform other tasks. Examples of commercial tools are PTC Windchill Modeler SySim [5], Sparx Systems Enterprise Architect [6], Dassault Cameo Systems Modeller [2]. These tools are often based on the

Unified Modeling Language (UML) OMG standard [49, 50], which is considered a semi-formal method. Semi-formal methods owe their name to their lack of a formal semantics. The semantics of semi-formal methods is informally described in natural language documents (e.g., [51]). This informal semantics suffers either from semantic aspects intentionally left open by the standards or unintentional ambiguities [9, 27, 21, 26]. As a result, the same semi-formal model executed on different simulators may behave differently.

There is a growing body of literature on the integration of formal methods techniques into model-based development tools (e.g., [40, 52, 54, 13, 56, 20, 59, 23, 19, 34]), and the formalization of UML state diagrams has been recently surveyed in [9]. This integration shows how techniques developed in academia—typically formal—can be transferred to enhance current industrial tools—generally semi-formal.

This paper explores the combination of an academic formal verification tool with an industrial model-based development tool to develop a railway interface. The formal verification tool used is the UML Model Checker (UMC), while the selected model-based development tool is Sparx Enterprise Architect (Sparx EA). The models developed using the two tools are related, with the Sparx EA model used for model-based development activities and the UMC model used for formal verification, in particular model checking.

This paper builds upon previous activities carried out during the Shift2Rail project 4SECURail [7]. The case study being analyzed is a fragment of the UNISIG Subset 039 [61] and Subset 098 [60] standard interface called the Communication Supervision Layer (CSL). It is borrowed from the first release of the "Formal development Demonstrator prototype" of the 4SECURail project [43, 53] and is specifically dedicated to the control of the communication status between two neighboring Radio Block Centre (RBC). The paper aims to demonstrate how formal verification techniques from academic tools can be integrated into industrial development practices using industrial tools. In particular, [9] reports that "counterexamples are rarely mapped back to the original models" and more specifically that "UMC could be used to verify UML models". We use UMC to formally verify the UML state diagrams of Sparx EA and the traces generated by the UMC verification phase are reproduced as simulations in Sparx EA.

The contributions of this paper are: i) a set of UML notation constraints oriented towards maintaining the correspondence between Sparx EA and UMC models; ii) a set of actionable rules to map the two notations; iii) the mapping of traces generated by the UMC formal verification into simulations in Sparx EA; iv) a set of lessons learned and challenges derived by applying the proposed methodology to a case study from the railway industry.

Structure of the paper We start with the related work in Section 2. Background on the used tools is in Section 3. Section 4 discusses the methodology for connecting the semi-formal and formal models. A concrete example showing how the methodology is applied to a case study is in Section 5. The lessons learned, limitations and challenges are discussed in Section 6. Section 7 concludes the paper and discusses future work.

2 Related work

Several works have been carried out in the railway domain concerning the usage of formal and semi-formal notations to represent a wide diversity of systems [31], according to the formal model-based development paradigm [28].

Among them, Chiappini et al. [25] consider a portion of the ERTMS/ETCS system as case study and propose an approach to manually translate natural language requirements into an enhanced UML language. The UML representation is then translated and verified by means of the NuSMV model checker.

Ferrari et al. [29, 32] start from requirements expressed by means of UML component diagrams, and use Simulink/Stateflow, with the aid of Simulink Design Verifier, to verify the model behaviour of an automatic train control system. Similar to us, the authors also define a set of modelling guidelines and notation restrictions to remove ambiguities from the models, with the goal of achieving clearer models and generated code.

In [48] Miller et al. report their infrastructure to translate Simulink models into different formal languages, including SPIN and NuSMV, to perform formal verification. Model translation is also the target of Mazzanti et al. [46], who also report a method to increase confidence in the correctness of the transformation. The method starts from UML state machines, which are translated into multiple formal notations.

Still on the translation from UML-like models to other formal notations, recent works have focused on transforming these models into mCRL2 [59, 20]. Many studies also focus on the translation from UML into the B/Event-B notation [57, 58], with formal verification performed by means of Atelier B and ProB [22].

A recent set of works by Mazzanti and Belli [18, 47] focuses on the incremental modelling of natural language requirements as UMC state machines, and associated formal validation. Initially, a UML state machine modelling the set of requirements under analysis is created. This initial model is not targeting any specific tool and it contains pseudo-code. Once consolidated, the state machine model will eventually be written using the UMC syntax. In [18, 47] it is showed how, under certain notation restrictions, it is possible to automatically translate the state machines from UMC to other verification tools such as ProB and CADP [35], where the models are formally verified to be equivalent. In [18] it is discussed how the formal verification of UMC state machines can be used independently to transform natural language requirements into formally verified structured natural language requirements.

Similarly to [18, 47], in this paper we use a preliminary version of the same case study and a relaxed version of the restrictions imposed on the UML state machines. Moreover, our work complements [18, 47] by showing how the developed UMC models can be imported into Sparx EA to enable various tasks other than formal verification, including generating diagrams, documentation, code, and interactive simulations. More details on the rationale for the choice of the case study of 4SECURail, the tools, and the prototype architecture are in [44, 43, 53, 45].

3 MBSD, Sparx Enterprise Architect and UMC

Model-Based Software/System Development (MBSD) is a methodology for creating software and hardware artifacts using models expressed as graphical diagrams. Models are used throughout the development cycle. The development process is guided by a model of the software architecture, which represents a semi-formalization of the system's abstract level without implementation details. Semi-formal models can be complemented by their formal specifications, enabling formal techniques like model checking or theorem proving. Early detection of errors is possible by verifying the model against requirements using techniques such as model checking.

UML, an OMG standardised notation [49, 50], is the standard for many MBSD environments. Models support modular design by representing different views of the system at distinct levels, such as requirements definition, implementation, and deployment. Code and test generation ensures that the implementation is derived directly from the models with traceability. In this paper, we will focus on a specific subset of UML, detailed in Section 4.

In UML, a model consists of multiple classes, each with its own set of attributes. Objects are created by instantiating these classes and assigning values to the attributes using the object-oriented paradigm. A classifier behaviour can be assigned to a class in the form of a UML state machine. A state machine can be triggered by events, e.g., signals. The state machine includes various states and transitions connecting them. Transitions have labels of the form `trigger[conditions]/effects`, where the conditions are on the variables of the class and the trigger arguments, and the effects can modify these variables and generate outgoing signals. Two examples of state machines, accepted by the tools UMC and Sparx EA, are in Figure 1 and Figure 2.

3.1 Sparx Enterprise Architect

Sparx Enterprise Architect is an MBSD tool based on OMG UML [49]. It was selected after an initial task was conducted during the 4SECURail project to test and gather information about factors such as licensing costs, customer support, and training [43]. The most desirable feature was the modelling and simulation of state machine diagrams.

MBSD tools differ in the way in which UML state machines can be composed. Different MBSD tools provide different solutions. Sparx EA [6] is an MBSD tool that offers an Executable State Machine (ESM) artifact specifically designed for simulating the *composition* of different state machines. These machines can interact through a straightforward instruction for sending an event, and do not require excessive notation. An example of ESM for composing the state machine in Figure 2 with other state machines is depicted in Figure 3. ESM provide all the necessary elements for easy translation to/from a formal specification amenable to verification and graphical display of an informal specification, as well as simulation.

In addition to simulating a composition of state machines, the standard simulation engines of Sparx EA can be used to interact with each machine individually. Source code is automatically generated from such ESM models, which is then executed/debugged. It is possible to generate source code in JavaScript, Java, C, C++ and C#. The source code also contains the implementation of the behavioural engine of state diagrams, for example the pool of events for each state machine, the dispatching method and so on. Once designed, a system composed of several interacting state machines can be simulated interactively, by sending triggers, to observe its behaviour. The ESM is used for generating code, and the simulation gives an interactive graphical animation of the system being debugged. In this paper, we used Sparx EA unified edition version 15.2 build 1559.

3.2 UML Model Checker

The UML Model Checker (UMC) [39, 8, 16] is an open-access tool explicitly oriented to the fast prototyping of systems constituted by interacting state machines. UMC allows the user to design a UML state diagram using a simple textual notation, visualise the corresponding graphical representation, interactively animate the system evolutions, formally verify (using on-the-fly model checking) UCTL [16] properties of the system behaviour. Detailed explanations are given when a property is found not to hold, also in terms of simple UML sequence diagrams. With UMC it is possible to check if/how a given transition is eventually fired, if/when a certain signal is sent, if/when a certain variable is modified, or a certain state reached.

The formal semantics of UMC models is provided by an incremental construction of a doubly labelled transition system [16]. In addition, for a restricted set of UMC notation this can be given through the automatic translation into the LOTOS NT language [24, 37, 42] described in [18, 47, 45]. The strong bisimilarity of source and target models can be proved with, e.g., mCRL2 `ltscompare` [3] or CADP `bcg_cmp` [1]. Note that UMC is an academic tool that is primarily utilized for research and teaching. We used UMC version 4.8f (2022).

4 Methodology

In this section, we present the methodology used to connect the UMC and Sparx EA models. It is worth noting that the proposed methodology works in both directions. It is possible to use a forward engineering method by analyzing the model in UMC first and then translating it into Sparx EA for development. Alternatively, it is possible to use a reverse engineering approach, and if the Sparx EA models meet the specified condition described below, they can be translated into UMC for formal verification. The rationale is that the two tools are complementary and their features shall be used jointly. While Sparx EA supports typical MBSD activities, such as interactive simulation and code generation, UMC supports formal verification.

UMC supports a subset of UML State Machine Diagrams, polished from some syntactic sugar notations, and each construct can be mapped one-to-one to a construct in Sparx EA, if the Sparx EA model follows the same restrictions. Relating the UMC model and the Sparx EA model is almost straightforward when notation constraints are enforced. The following adopted restrictions on the model are exploited to keep the notation light and as much independent as possible from UML technicalities. Indeed, many of the constructs that are discarded are syntactic sugar that can be expressed using a lighter notation.

Syntactic Restrictions on UML State Machines

- no `entry`, `exit`, or `do` behaviour is present in the states of the model (these behaviors can be equivalently expressed in state transitions),
- interaction happens using only *signals*, and no operation calls are used,
- only one-to-one interactions are used, i.e., no signals broadcast,
- conflicts in enabled transitions are only allowed in the environment,
- no timing behaviour is present (time elapsing is explicated using a TICK event), no internal and local transitions are used, no hierarchical states are used, no history, fork, join and choice nodes are used.

Environment In Sparx EA interactive simulations the human user acts as the environment. Consider, for example, a system composed of two components C1 and C2. When only one of the two components (e.g., C1) is fully modelled, then events from C2 can be considered part of the environment of C1 and manually triggered. In model checking tools like UMC a (possibly non-deterministic) environment needs to be explicitly modelled to obtain a fully closed system on which the verification is automatic.

Semantics of Sparx ESM and UML models In the context of UML State Machine models, certain aspects of the model's behaviour are left unspecified by the ISO standard [49]. For example, the order in which events occur is left open to interpretation. As a result, it is difficult to formally verify the accuracy of a translation from a formal to a semi-formal model and vice-versa, from the semi-formal to the formal one. Indeed, the Sparx EA models do not have a formal semantics and Sparx EA does not have the ability to exhaustively generate the state space of the model. In our case, the correctness of the translation (i.e., the correspondence between the formal and semi-formal model) has been validated informally, and by translating traces derived from UMC proofs into simulations in Sparx EA (cf. Section 5).

The UML state diagrams in both UMC and Sparx EA are avoiding the presence of aspects with ambiguous semantics. Sparx EA provides a way to inspect and review the code generated by an ESM to disambiguate the semantics choices left open by the UML standard. Thus, a code review has been performed to check that the semantics of Sparx EA and UMC state machines are aligned. Regarding how events are ordered in each pool of events, both UMC and Sparx EA use a first-in-first-out policy. In Sparx EA, the scheduling of state machines and the dispatching of messages are fixed, with each state machine completing its run-to-completion cycle before another one starts. Conflicts in enabled transitions

are not present in the Sparx model (i.e., the model is deterministic), so there is no need to specify a choice strategy. However, UMC allows for all orders of scheduling, it interleaves all run-to-completions steps of different state machines, and permits all possible behaviours obtained by fixing a specific strategy for selecting one among many enabled transitions (UMC models only allow conflicts in the environment, which is not translated in Sparx EA). Thus, the semantics of UMC includes the semantics of Sparx ESM, as well as all semantics obtained by changing the scheduling order. If a safety property holds in the UMC model, it will also hold in the Sparx model.

The effects of each transition contain Java code, limited to performing arithmetic operations on variables, sending signals, and reading values. These restrictions on Sparx models are necessary to disambiguate the semi-formal semantics and proceed in external formal verification using model checking.

Rules for relating the UMC model with the Sparx EA model We now describe the rules to relate a UMC model with a Sparx EA model. An example of application of the rules is in the next section, where Figure 1 and Figure 2 show how the state machines of Sparx EA and UMC are related.

1) Each class in UMC corresponds to a class in Sparx EA.
2) Attributes of a class in UMC correspond to attributes of the corresponding class in Sparx EA.
3) Each Object in UMC, with its variables' instantiation, is mapped into a Property of an ESM (i.e., an instantiation of class), to where the values of the attributes can be instantiated.
4) Both UMC and Sparx EA classes have a relation "has-a" with other classes, in such a way that every object has a reference to other objects to whom it is interacting with.
5) Each class in UMC is specified as a state machine. Similarly, in Sparx EA a classifier behaviour will be assigned to each class in the form of a state machine.
6) States and transitions of a machine in UMC are in one-to-one correspondence with those of a machine in Sparx EA.
7) Signals that are attributes of each class in UMC are in correspondence with global trigger events in the Sparx model, accessible by each state machine. These events are of type `Signal` and have the same parameters as in UMC.
8) In UMC the sending of a signal with, for example, two parameters, is performed using the instruction `Object.Signal(value1, value2)` where `Object` is the receiver object argument, `Signal` is the signal invoked in that object, and `value1` and `value2` are the values to be passed as arguments. In an ESM, the objects are connected by connectors typed with the relation "has-a" coming from the class diagram. Each end of a connector identifies the partner of a communication. The above send operation is performed in Sparx EA with the macro `%SEND_EVENT("TRIGGER.sig(value1,value2)",CONTEXT_REF(RECIPIENT))%;` where `sig` denotes the specification of the signal assigned to the trigger (i.e., names and types of the parameters), `TRIGGER` corresponds to `Signal` in UMC, and `RECIPIENT` corresponds to `Object` in UMC. `RECIPIENT` is the identifier provided in the corresponding connector end of the ESM. In case values of signals

must be accessed inside the guard or effect of a transition, in UMC this can be done by simply accessing the parameter with its declared name. In Sparx EA values of signals are accessed as follows. In the effect of a transition, the instruction `signal.parameterValues.get("arg")` is used, where `arg` is the name of the parameter of the signal. If the value is accessed in a condition, the above command becomes `event.signal.parameterValues.get("arg")`. We also note that this syntax is specific to Java code generation.

5 Case Study

The chosen case study is a subset of the RBC/RBC handover protocol borrowed from the 4SECURail project [43, 53]. This protocol is a crucial aspect of the ERTMS/ETCS train control system, in which a Radio Block Centre (RBC) manages trains under its area of supervision. An RBC is a wireless component of the wayside train control system that manages the trains that can be reached from its assigned geographical area (i.e., the area of supervision). When a train approaches the end of an RBC's area of supervision, a handover procedure with the neighbouring RBC must take place to manage the transfer of control responsibilities. Since neighbouring RBCs may be manufactured by different providers, the RBC/RBC interface must ensure interoperability between RBCs provided by different suppliers. The selected subset of the RBC/RBC protocol is the Communication Supervision Layer (CSL), responsible for opening/closing a communication line between RBCs and maintaining connection through life signs. The CSL's functional requirements are specified in UNISIG SUBSET-039 [61] and UNISIG SUBSET-098 [60]. In particular, two sides are identified: the Initiator CSL (ICSL), and the Called CSL (CCSL). The initiator is the RBC responsible for opening the connection. The layers above and below the CSL are called, respectively, RBC User Layer and Safe intermediate Application sub-Layer (SAI). In this paper, these other layers are treated as external environment and thus are not specified. All models and other artifacts are available in [11].

Formal and Semi-formal Models We now describe some aspects of the CSL model, focusing on the ICSL. The state machine modeling the ICSL is provided both as a formal model in UMC (see Figure 1) and as a semi-formal model in Sparx EA (see Figure 2). Each transition is labeled with a name (e.g., R1) to keep track of the correspondence between the two models.

The ICSL state machine is composed of two states `NOCOMMS` (the two RBC are disconnected) and `COMMS` (the two RBC are connected). The initial state is `NOCOMMS`. From state `NOCOMMS`, a counter `connect_timer` is incremented at each reception of a `TICK` signal from the clock (R6). If the threshold `max_connect_timer` is reached, a request for connection `SAI_CONNECT_REQUEST` is signaled to the SAI (which will be forwarded to the CCSL), and the counter is reset (R5).

The signal of connection `SAI_CONNECT_CONFIRM` (signaling the connection of the CCSL) coming from the `SAI` triggers the transition to state `COMMS` (R7). In state `COMMS` two counters are used. A counter `receive_timer` is used to keep track of the last message received. A counter `send_timer` is used to keep track

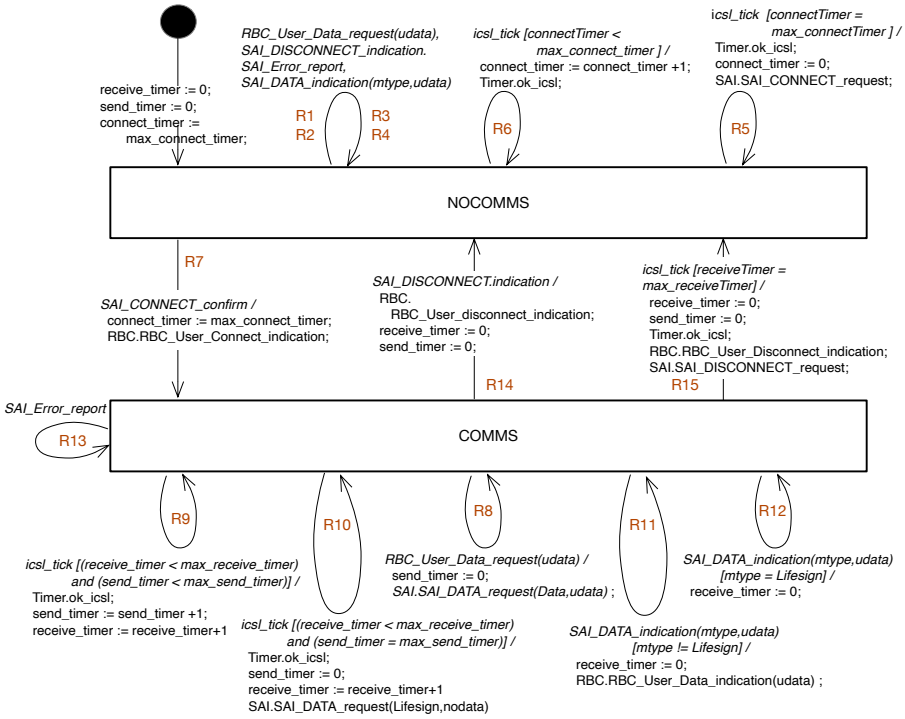

Fig. 1. The Initiator Communication Supervision Layer State Machine of UMC

of the last time a message was sent. These counters are incremented at the reception of a signal from the clock (R9). Each time a message is received from the SAI, the `receive_timer` is reset (R11,R12). Moreover, if the message is not of type `LifeSign`, it is forwarded to the user (R11). Similarly, if a message is received from the user, it is forwarded to the SAI (R8), and the `send_timer` is reset. Whenever the threshold `max_send_timer` is reached (R10), a `LifeSign` message is sent to the `SAI` (which be forwarded to the CCSL) and `send_timer` is reset. This message is used to check if the connection is still up. Whenever the threshold `max_receive_timer` is reached, the connection is closed because no message has been received within the maximum allowed time. In this case, a signal of disconnection is sent to both the user and the SAI (R15). If the message of disconnection is received from the SAI (R14), then it is only forwarded to the user and the connection is closed.

Mapping We now discuss how the rules from Section 4 have been applied to provide a correspondence between the two models in Figure 1 and Figure 2. The class diagram (not displayed here) contains the classes `I_CSL`, `C_CSL`, `SAI` and `RBC_User`. An additional class `TIMER` is used to model the elapse of time, and it is part of the environment. These classes and their attributes are the same for both Sparx EA and UMC, according to rule 1 and rule 2. Following rule 4, the classes `I_CSL`, `C_CSL` both have relations "has-a" with `TIMER`, `RBC_USER` and `SAI`.

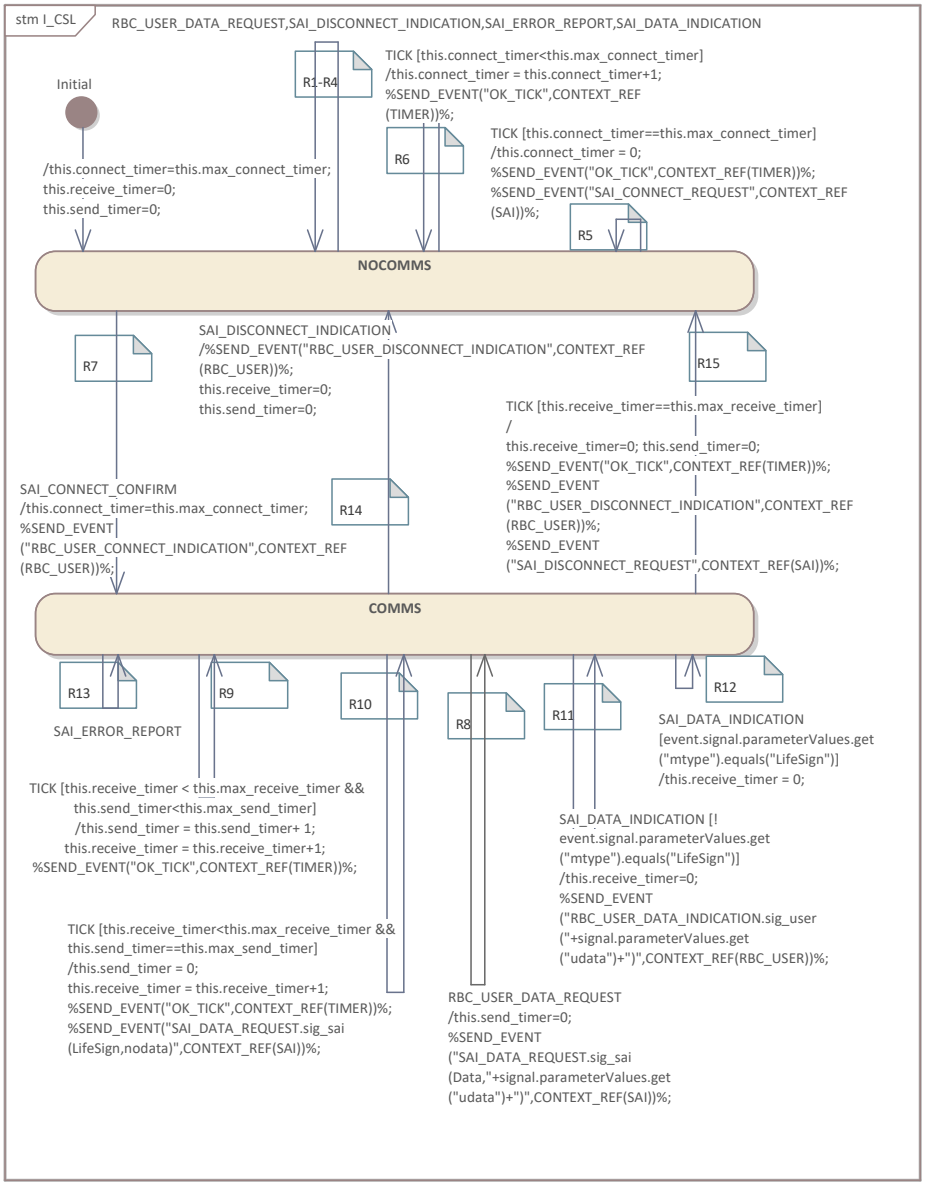

Fig. 2. The Initiator Communication Supervision Layer State Machine of Sparx EA

We recall that in Sparx EA the environment classes `TIMER`, `SAI` and `RBC_USER` are just stubs with a dummy behaviour assigned to them.

The ESM artifact is displayed in Figure 3. The ESM illustrates the composition of different class instances. There are two instantiations of the environment classes `RBC_USER` and `SAI`, one for each CSL. There is one instantiation of the environment class `TIMER` for both CSL. The objects `C_CSL` and `I_CSL` are instantiating their respective classes, and they are initialising their attributes. These objects are the same in Sparx EA and UMC according to rule 3.

Moreover, following rule 4, each CSL will refer to its RBC_USER, SAI and TIMER using the context references `RBC_USER`, `SAI` and `TIMER`, respectively, as depicted in the ends of the corresponding connectors in Figure 3.

According to rule 5, the behaviour of the classes `I_CSL` and `C_CSL` is specified by state machines in both UMC and Sparx EA. The states and transitions of these state machines are in correspondence according to rule 6. Following rule 7, the signals with their parameters are also in correspondence.

Finally, we use the transition R11 as an example for showing how rule 8 is applied. In UMC, `SAI_DATA_indication(mtype,udata)` is the trigger of R11. In Sparx EA, the trigger `SAI_DATA_INDICATION` does not report its parameters, which are declared separately in the type of the signal associated with the trigger. The parameters have the same name in both Sparx EA and UMC. In UMC, the condition of R11 is `mtype != LifeSign`. Indeed, it checks whether the type of the received message is not a life sign. In Sparx EA, the condition of R11 is `!event.signal.parameterValues.get("mtype").equals("LifeSign")`. We remark that Sparx EA uses Java as code for the conditions and the effects (other languages are also supported, e.g., C++). The code present in the effects and conditions (with the exception of the macros) will be injected into the generated source code as is. Finally, `RBC.RBC_User_Data_indication(udata)` and `receive_timer:=0` are the instructions of the effect of R11 in UMC. Basically, the message received by the SAI is forwarded to the user and the timer is reset. In Sparx EA, the effect of R11 also contains two instructions. The timer is reset with the instruction `this.receive_timer=0;`. The signal is forwarded with:

```
%SEND_EVENT("RBC_USER_DATA_INDICATION.sig_user("+signal.
parameterValues.get("udata")+")",CONTEXT_REF(RBC_USER))%;
```

we note that the macro is mixed with Java code. Indeed, it uses the code `signal.parameterValues.get("udata")` to read the field `udata` of the signal received by the SAI. The signal `sig_user` only contains one parameter (`udata`) and is assigned to the trigger `RBC_USER_DATA_INDICATION`.

5.1 Model Checking Sparx EA Models

We now show one of the benefits of our approach, i.e., the formal verification of semi-formal models. We remark that, generally, MBSD tools such as Sparx EA are not equipped with facilities for performing formal analyses. Through the mapping described in Section 4, it becomes possible to perform model checking of Sparx EA models by exploiting the connection with UMC. The model checking of a formal property produces a trace showing that the property holds or is

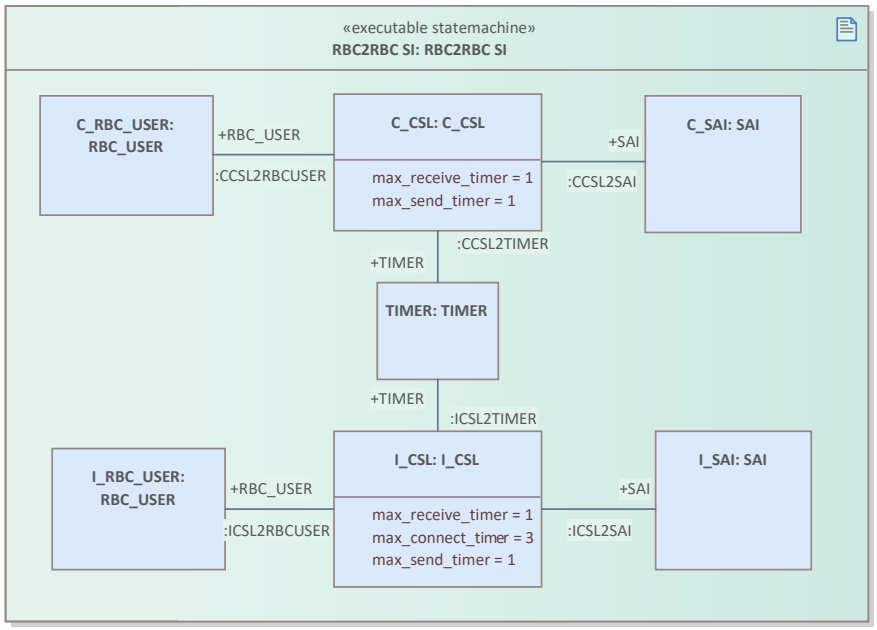

Fig. 3. The Executable State Machine of Sparx EA, showing the composition of the various instantiations of the state machines

violated. We show below how the produced trace is simulated in the Sparx EA semi-formal model.

Reproducing UMC traces in Sparx EA has two main benefits. Firstly, it allows the reproduction of the detected issues in Sparx EA. This is generally desirable for interacting with stakeholders that are only knowledgeable of the used industrial tool but are not aware of the underlying formal verification that has been carried out. The second goal is to validate the correspondence between the semi-formal and formal models presented in this paper. Indeed, if the trace is not reproducible in Sparx EA then we have detected a misalignment between the semantics of the formal and semi-formal models. Even if the correspondence is sound, this is still possible since UMC overapproximates all possible behaviours of Sparx EA. Therefore, an issue signalled by UMC might not be detectable by only relying on the interactive simulation capabilities provided by Sparx EA. Indeed, the simulation engine shows only a subset of all behaviours that the real system may have because, e.g., it fixes the order in which state machines are executed.

We now provide an example of a temporal property formally verified with UMC, using the models in Figure 1 and Figure 2. We need to mutate the models to cause the violation of an invariant, in such a way that a trace showing the violation is generated by UMC. This is typical of model-based mutation testing [14], where mutations are applied to the model to measure the effectiveness of the validation. We apply a mutation changing the condition of

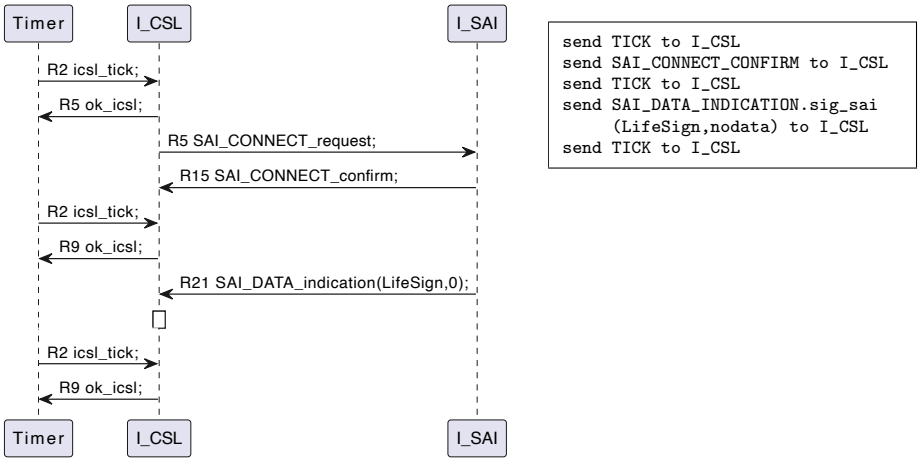

Fig. 4. On the left, a fragment of the sequence diagram generated by UMC showing that the property `EF sendTimer_Error` holds. On the right, the instructions needed to reproduce the trace in Sparx EA interactive simulation

transition R9 from a conjunction to a disjunction. The mutated condition becomes (in UMC) `(receive_timer < max_receive_timer) or (send_timer < max_send_timer)`. The introduced mutation enables a scenario where `send_timer` exceeds its maximum threshold `max_send_timer`. This violation can be detected, for example, by verifying the property `EF sendTimer_Error` where `EF` is a temporal operator stating that something will eventually happen in the future, and `sendTimer_Error` is defined by the instruction `Abstractions {State: I_CSL.send_timer > I_CSL.max_send_timer -> sendTimer_Error}` as a state property holding when the `send_timer` has a value greater than `max_send_timer`.

Figure 4 (left) shows a fragment of the sequence diagram automatically generated by UMC after model-checking the property (the full sequence diagram is in [11]). The sequence diagram graphically depicts a trace proving that the property holds in the model with the current set-up of variables (displayed in Figure 3). This means that the counter exceeds its maximum allowed value. Figure 4 (left) only highlights the necessary environment interactions that are needed in Sparx EA to reproduce the trace. The first `TICK` event received causes `I_CSL` to request a connection (R5), which is confirmed by `I_SAI` (R7), causing the switch to state `COMMS`. At the reception of the second `TICK`, the transition R9 is executed, which increments both `send_timer` and `receive_timer` to their maximum allowed value (i.e., `1`). A life sign is subsequently received, which causes the reset of `receive_timer` (R12). After that, another tick is received, triggering again the mutated transition R9, causing `send_timer` to exceed its maximum allowed value.

To reproduce the trace in Sparx EA, the interactive user assumes the role of the environment. In particular, all signals sent from the environment to one

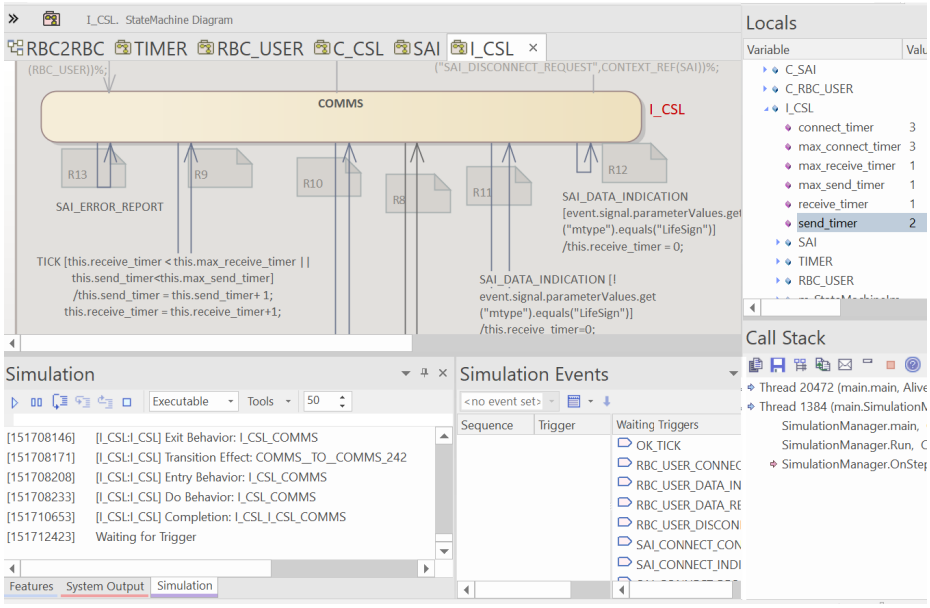

Fig. 5. A portion of the Sparx EA simulation where `sendTimer_Error` is true

of the machines will be replicated by the interactive user. Indeed, Sparx EA simulations permits sending these signals from the simulation console. As showed in Figure 4 (left), all signals sent to the `I_CSL` are coming from the environment components, and thus will be replicated by the interactive user. In particular, the instructions inserted at console during the simulation are displayed in Figure 4 (right). Each instruction causes all machines in the model to execute their run-to-completion cycle. After all instructions are executed, the simulation reaches a state where `I_CSL.send_timer = 2` (see Figure 5), proving the error. A short video reproducing this experiment and the logs of the simulation are in [11].

6 Lessons Learned and Limitations

The experience of connecting the UMC formal model and the Sparx EA semi-formal model led to a set of lessons learned, which are reported below.

Modelling Restrictions. The introduction of modelling restrictions was fundamental to avoid ambiguities that are present in the informal semantics of UML state machines. The adoption of the proposed restrictions enabled the connection of Sparx EA semi-formal models with UMC formal models. Moreover, thanks to a limited and simple set of notation constraints, not only the translation process was straightforward, but the models appeared cleaner and easier to inspect. Our notation constraints are stricter than those used in [29, 32], therefore, we conjecture that our restricted UMC state machines could also be easily modelled as Simulink/Stateflow state charts.

Complementary Tools. UMC and Sparx EA played complementary roles in this experience, and have been developed in academia and industry, respectively. Their combined usage allows the cross-fertilization of academic techniques and industrial practices. This allowed us to check fine-grained temporal properties that were hard to verify by only using the simulation capabilities offered by Sparx EA. On the other hand, the Sparx EA model represents the starting point of the model-based development activity (including visual simulation) that will eventually lead to the final product.

Bidirectional approach. Whilst generally the literature proposes unidirectional approaches (cf. Section 2), our methodology supports both forward and reverse engineering. This is also helpful in maintaining aligned the formal and semi-formal artifacts during the evolution of the system to newer versions.

Tool Competence. The researchers involved in this experience have complementary expertise in the two tools considered. This was fundamental to achieve a sufficient degree of confidence in the correctness of the developed models, as full control of the used notations is needed to prevent misrepresentations. Together with other works [30, 33], we argue that similar case studies shall involve a diversity of experts to successfully carry out the process described in this paper.

Integrated environment. We express as particularly desirable the possibility to rely on a single MBSD framework for typical MBSD activities (e.g., design, code generation, documentation) and formal verification. This is currently out of reach, especially if a semi-formal language such as UML is kept as a reference underlying notation. This remains an important direction to be further explored.

Limitations and challenges We now discuss some limitations of the proposed approach and the challenges ahead.

Manual Translation. In regards to manual translation, this was addressed by having two researchers (first and second author) work collaboratively to translate and verify the consistency of the models through model inspection. Additionally, the simulation of the Sparx EA model and formal verification of the corresponding UMC model increased the confidence in the accuracy of the correspondence. This was achieved by demonstrating how traces from the formal verification process can be replicated by simulating the semi-formal model. Moving forward, we intend to fully automate both the translation process and the verification of model conformance to the restrictions outlined in Section 4.

Correspondence of Models. Concerning the lack of formal verification of correspondence between formal and semi-formal models, it is worth noting that this is an inherent limitation of semi-formal approaches. These methods, by definition, lack formal semantics, and as a result, formal verification of behavioral correspondence is not feasible. In fact, while inspecting the semantics of Sparx EA models, in particular the code generated from the ESM, we ran into corner cases that needed interactions with the support at Sparx Systems. The next released version (15.2) fixed the issues detected in our experiments [4].

Generalisability. Concerning representativeness and generalisability of the results, it should be noticed that the restrictions on the notation and the translation process have been evaluated in reference to our specific case study from

the railway domain. In other domains, and for other systems, different needs may emerge that require additional restrictions, or the relaxation of existing ones. In particular, in the current case study the two modelled state machines are not directly interacting. Each machine (ICSL and CCSL) is only interacting with the surrounding environment components. Further case studies are needed to extend the scope of validity of the proposed constraints and implemented process.

Partial Representation and Scalability. The verified Sparx EA model only provides a partial representation of the final product, as the code generated from the ESM is utilized for simulation purposes. Thus, further development is required to refine the generated code and produce the final implementation. This raises the challenge of ensuring that the verified properties are maintained during the refinement process. A possible solution is to minimize the difference between the verified Sparx EA model and the final implementation. Formal verification may become challenging if the size of the models increases significantly. It should also be noted that the complete implementation of an industrial system requires significant resources, which may not be readily available for a research activity like the one described in this paper. Therefore, substantial involvement of practitioners from both academia and industry is required. In this case, the issue of non-disclosure and confidentiality must also be considered, particularly when the intention is to make the models publicly available, as in this paper [11].

7 Conclusion

This paper has presented an investigation into the combination of an academic formal verification tool, UMC, with an industrial semi-formal model-based development tool, Sparx EA. The integration has been achieved through the definition of a set of notation restriction rules and rules for relating semi-formal and formal models. We have demonstrated how the output of the UMC formal verification can be connected to Sparx EA interactive simulations. The presented approach has been experimented on a case study from the railway domain. From this experience, we have derived a set of lessons learned and limitations driving future research challenges.

In the future, we plan to investigate how much the UML notation constraints presented in this paper can be relaxed to allow more freedom in the design of the models whilst preserving formality. We would also like to fully implement an application that is formally verified using the proposed methodology.

Acknowledgements This work has been partially funded by the 4SECURail project (Shift2Rail GA 881775). Part of this study was carried out within the MOST – Sustainable Mobility National Research Center and received funding from the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.4 – D.D. 1033 17/06/2022, CN00000023). The content of this paper reflects only the author's view and the Shift2Rail Joint Undertaking is not responsible for any use that may be made of the included information.

References

1. CADP: bcgcmp man page, https://cadp.inria.fr/man/bcg_cmp.html
2. Dassault Cameo Systems Modeler, https://www.3ds.com/products-services/catia/products/no-magic/cameo-systems-modeler/, accessed April 2023
3. mCRL2: ltscompare man page, https://www.mcrl2.org/web/user_manual/tools/release/ltscompare.html
4. Multiple improvements to executable state machine code generation, https://sparxsystems.com/products/ea/15.2/history.html, accessed May 2023
5. PTC Windchill Modeler SySim, https://www.ptc.com/en/products/windchill/modeler/sysim, accessed April 2023
6. Sparx Systems Enterprise Architect, https://sparxsystems.com/products/ea/index.html, accessed May 2023
7. The Shift2Rail 4SECURail project site, https://projects.shift2rail.org/s2r_ip2_n.aspx?p=s2r_4securail, accessed May 2023
8. UMC project website, http://fmt.isti.cnr.it/umc
9. André, É., Liu, S., Liu, Y., Choppy, C., Sun, J., Dong, J.S.: Formalizing UML State Machines for Automated Verification–A Survey. ACM Comput. Surv. (2023). https://doi.org/10.1145/3579821
10. Basile, D., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., Trentini, D., Ferrari, A.: On the Industrial Uptake of Formal Methods in the Railway Domain. In: Furia, C.A., Winter, K. (eds.) iFM. LNCS, vol. 11023, pp. 20–29. Springer (2018). https://doi.org/10.1007/978-3-319-98938-9_2
11. Basile, D., Mazzanti, F., Ferrari, A.: Experimenting with Formal Verification and Model-based Development in Railways: the case of UMC and Sparx Enterprise Architect - Complementary Data (2023). https://doi.org/10.5281/zenodo.7920448
12. Basile, D., ter Beek, M.H., Fantechi, A., Ferrari, A., Gnesi, S., Masullo, L., Mazzanti, F., Piattino, A., Trentini, D.: Designing a demonstrator of formal methods for railways infrastructure managers. In: Margaria, T., Steffen, B. (eds.) ISoLA. LNCS, vol. 12478, pp. 467–485. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_30
13. Basile, D., ter Beek, M.H., Ferrari, A., Legay, A.: Modelling and Analysing ERTMS L3 Moving Block Railway Signalling with Simulink and Uppaal SMC. In: Larsen, K.G., Willemse, T.A.C. (eds.) FMICS. LNCS, vol. 11687, pp. 1–21. Springer (2019). https://doi.org/10.1007/978-3-030-27008-7_1
14. Basile, D., ter Beek, M.H., Lazreg, S., Cordy, M., Legay, A.: Static detection of equivalent mutants in real-time model-based mutation testing. Empir. Softw. Eng. **27**(7), 160 (2022). https://doi.org/10.1007/s10664-022-10149-y
15. Basile, D., Fantechi, A., Rosadi, I.: Formal analysis of the UNISIG safety application intermediate sub-layer: Applying formal methods to railway standard interfaces. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS. LNCS, vol. 12863, pp. 174–190. Springer (2021). https://doi.org/10.1007/978-3-030-85248-1_11
16. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. **76**(2), 119–135 (2011). https://doi.org/10.1016/j.scico.2010.07.002
17. ter Beek, M.H., Borälv, A., Fantechi, A., Ferrari, A., Gnesi, S., Löfving, C., Mazzanti, F.: Adopting Formal Methods in an Industrial Setting: The Railways Case. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM. LNCS, vol. 11800, pp. 762–772. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_46

18. Belli, D., Mazzanti, F.: A case study in formal analysis of system requirements. In: Masci, P., Bernardeschi, C., Graziani, P., Koddenbrock, M., Palmieri, M. (eds.) SEFM Workshops. LNCS, vol. 13765, pp. 164–173. Springer (2022). https://doi.org/10.1007/978-3-031-26236-4_14
19. Bougacha, R., Laleau, R., Dutilleul, S.C., Ayed, R.B.: Extending SysML with refinement and decomposition mechanisms to generate Event-B specifications. In: Ameur, Y.A., Craciun, F. (eds.) TASE. LNCS, vol. 13299, pp. 256–273. Springer (2022). https://doi.org/10.1007/978-3-031-10363-6_18
20. Bouwman, M., Luttik, B., van der Wal, D.: A formalisation of SysML state machines in mCRL2. In: Peters, K., Willemse, T.A.C. (eds.) FORTE. LNCS, vol. 12719, pp. 42–59. Springer (2021). https://doi.org/10.1007/978-3-030-78089-0_3
21. Broy, M., Cengarle, M.V.: UML formal semantics: lessons learned. Softw. Syst. Model. **10**(4), 441–446 (2011). https://doi.org/10.1007/s10270-011-0207-y
22. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F., Voisin, L.: The First Twenty-Five Years of Industrial Use of the B-Method. In: ter Beek, M., Ničković, D. (eds.) FMICS. LNCS, vol. 12327, pp. 189–209. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_8
23. Cavada, R., Cimatti, A., Griggio, A., Susi, A.: A formal IDE for railways: Research challenges. In: Masci, P., Bernardeschi, C., Graziani, P., Koddenbrock, M., Palmieri, M. (eds.) SEFM Workshops. LNCS, vol. 13765, pp. 107–115. Springer (2022). https://doi.org/10.1007/978-3-031-26236-4_9
24. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Lang, F., McKinty, C., Powazny, V., Serwe, W., Smeding, G.: Reference manual of the LOTOS NT to LOTOS translator (2023), https://cadp.inria.fr/ftp/publications/cadp/Champelovier-Clerc-Garavel-et-al-10.pdf, accessed May 2023
25. Chiappini, A., Cimatti, A., Macchi, L., Rebollo, O., Roveri, M., Susi, A., Tonetta, S., Vittorini, B.: Formalization and validation of a subset of the European Train Control System. In: Proceedings of the 32nd International Conference on Software Engineering (ICSE). pp. 109–118. ACM (2010). https://doi.org/10.1145/1810295.1810312
26. Cook, S.: Looking back at UML. Softw. Syst. Model. **11**(4), 471–480 (2012). https://doi.org/10.1007/s10270-012-0256-x
27. Derezińska, A., Szczykulski, M.: Interpretation Problems in Code Generation from UML State Machines: A Comparative Study. In: Kwater, T., Zuberek, W.M., Ciarkowski, A., Kruk, M., Pekala, R., Twaróg, B. (eds.) Proceedings of the 2nd Scientific Conference on Computing in Science and Technology (STI). pp. 36–50. Monographs in Applied Informatics, Warsaw University of Life Sciences (2012)
28. Ferrari, A., Fantechi, A., Gnesi, S., Magnani, G.: Model-Based Development and Formal Methods in the Railway Industry. IEEE Softw. **30**(3), 28–34 (2013). https://doi.org/10.1109/MS.2013.44
29. Ferrari, A., Fantechi, A., Magnani, G., Grasso, D., Tempestini, M.: The Metrô Rio case study. Sci. Comput. Program. **78**(7), 828–842 (2013). https://doi.org/10.1016/j.scico.2012.04.003
30. Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H., Fantechi, A.: Comparing formal tools for system design: a judgment study. In: Proceedings of the 42nd International Conference on Software Engineering (ICSE). pp. 62–74. ACM (2020). https://doi.org/10.1145/3377811.3380373
31. Ferrari, A., ter Beek, M.H.: Formal methods in railways: A systematic mapping study. ACM Comput. Surv. **55**(4), 69:1–69:37 (2023). https://doi.org/10.1145/3520480

32. Ferrari, A., Fantechi, A., Bacherini, S., Zingoni, N.: Modeling guidelines for code generation in the railway signaling context. In: Denney, E., Giannakopoulou, D., Pasareanu, C.S. (eds.) Proceedings of the 1st NASA Formal Methods Symposium (NFM). NASA Conference Proceedings, vol. CP-2009-215407, pp. 166–170 (2009), https://ntrs.nasa.gov/citations/20100024476
33. Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H.: Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. IEEE Trans. Software Eng. **48**(11), 4675–4691 (2022). https://doi.org/10.1109/TSE.2021.3124677
34. Filipovikj, P., Mahmud, N., Marinescu, R., Seceleanu, C., Ljungkrantz, O., Lönn, H.: Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems. In: FM. LNCS, vol. 9995, pp. 748–756. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_46
35. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z
36. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 Expert Survey on Formal Methods. In: ter Beek, M., Ničković, D. (eds.) FMICS. LNCS, vol. 12327, pp. 3–69. Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
37. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd, LNCS, vol. 10500, pp. 3–26. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_1
38. Gleirscher, M., Marmsoler, D.: Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. Empir. Softw. Eng. **25**(6), 4473–4546 (2020). https://doi.org/10.1007/s10664-020-09836-5
39. Gnesi, S., Mazzanti, F.: An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In: Wirsing, M., Hölzl, M.M. (eds.) Rigorous Software Engineering for Service-Oriented Systems, LNCS, vol. 6582, pp. 390–407. Springer (2011). https://doi.org/10.1007/978-3-642-20401-2_18
40. Horváth, B., Molnár, V., Graics, B., Hajdu, A., Ráth, I., Horváth, A., Karban, R., Trancho, G., Micskei, Z.: Pragmatic verification and validation of industrial executable sysml models. Systems Engineering (2023). https://doi.org/10.1002/sys.21679
41. Huisman, M., Gurov, D., Malkis, A.: Formal Methods: From Academia to Industrial Practice. A Travel Guide (2020), https://arxiv.org/abs/2002.07279
42. Leduc, G.: Information technology-enhancements to LOTOS (E-LOTOS). ISO/IEC International Standard (2001), https://www.iso.org/obp/ui/#iso:std:iso-iec:15437:ed-1:v1:en
43. Mazzanti, F., Basile, D.: 4SECURail Deliverable D2.2 "Formal development Demonstrator prototype, 1st Release" (2020), https://www.4securail.eu/pdf/4SR-WP2-D2.2-Formal-development-demonstrator-prototype-1st%20release-CNR-3.0.pdf, accessed May 2023
44. Mazzanti, F., Basile, D., Fantechi, A., Gnesi, S., Ferrari, A., Piattino, A., Masullo, L., Trentini, D.: 4SECURail Deliverable D2.1 "Specification of formal development demonstrator" (2020), https://www.4securail.eu/pdf/4SR-WP2-D2.1-Specification%20of%20formal%20development%20demonstrator-CNR-1.0.pdf, accessed May 2023
45. Mazzanti, F., Belli, D.: 4SECURail Deliverable D2.5 "Formal development demonstrator prototype, final release" (July 2021), https://www.4securail.eu/pdf/4SR-WP2-D2.5-Formal-development-demonstrator-prototype.final-release-CNR-1.0.pdf, accessed May 2023

46. Mazzanti, F., Ferrari, A., Spagnolo, G.O.: Towards formal methods diversity in railways: an experience report with seven frameworks. Int. J. Softw. Tools Technol. Transf. **20**(3), 263–288 (2018). https://doi.org/10.1007/s10009-018-0488-3
47. Mazzanti, F., Belli, D.: The 4SECURail formal methods demonstrator. In: Dutilleul, S.C., Haxthausen, A.E., Lecomte, T. (eds.) RSSRail. LNCS, vol. 13294, pp. 149–165. Springer (2022). https://doi.org/10.1007/978-3-031-05814-1_11
48. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010). https://doi.org/10.1145/1646353.1646372
49. Object Management Group: Unified Modelling Language (December 2017), https://www.omg.org/spec/UML/About-UML/
50. Object Management Group: OMG Systems Modeling Language (OMG SysML) (November 2019), http://www.omg.org/spec/SysML/1.6/
51. Object Management Group: Precise Semantics of UML State Machines (PSSM) (May 2019), https://www.omg.org/spec/PSSM
52. Peres, F., Ghazel, M.: A proven translation from a UML state machine subset to timed automata. ACM Trans. Embed. Comput. Syst. (2023). https://doi.org/10.1145/3581771
53. Piattino, A.: 4SECURail Deliverable D2.3 "Case study requirements and specification" (2020), https://www.4securail.eu/pdf/4SR-WP2-D2.3-Case-study-requirements-and-specification-SIRTI-1.0.pdf, accessed May 2023
54. Salunkhe, S., Berglehner, R., Rasheeq, A.: Automatic transformation of SysML model to Event-B model for railway CCS application. In: Raschke, A., Méry, D. (eds.) ABZ. LNCS, vol. 12709, pp. 143–149. Springer (2021). https://doi.org/10.1007/978-3-030-77543-8_14
55. Seisenberger, M., ter Beek, M.H., Fan, X., Ferrari, A., Haxthausen, A.E., James, P., Lawrence, A., Luttik, B., van de Pol, J., Wimmer, S.: Safe and Secure Future AI-Driven Railway Technologies: Challenges for Formal Methods in Railway. In: Margaria, T., Steffen, B. (eds.) ISoLA. LNCS, vol. 13704, pp. 246–268. Springer (2022). https://doi.org/10.1007/978-3-031-19762-8_20
56. Sheng, H., Bentkamp, A., Zhan, B.: HHLPy: Practical verification of hybrid systems using Hoare logic. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) FM. LNCS, vol. 14000, pp. 160–178. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_11
57. Snook, C.F., Butler, M.J.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006). https://doi.org/10.1145/1125808.1125811
58. Snook, C.F., Butler, M.J., Hoang, T.S., Fathabadi, A.S., Dghaym, D.: Developing the UML-B modelling tools. In: Masci, P., Bernardeschi, C., Graziani, P., Koddenbrock, M., Palmieri, M. (eds.) SEFM Workshops. LNCS, vol. 13765, pp. 181–188. Springer (2022). https://doi.org/10.1007/978-3-031-26236-4_16
59. Stramaglia, A., Keiren, J.J.A.: Formal verification of an industrial UML-like model using mCRL2. In: Groote, J.F., Huisman, M. (eds.) FMICS. LNCS, vol. 13487, pp. 86–102. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_7
60. UNISIG: RBC-RBC Safe Communication Interface – SUBSET-098 (February 2012), https://www.era.europa.eu/system/files/2023-01/sos3_index063_-_subset-098_v300.pdf, accessed May 2023
61. UNISIG: FIS for the RBC/RBC Handover – SUBSET-039 (December 2015), https://www.era.europa.eu/system/files/2023-01/sos3_index012_-_subset-039_v320.pdf, accessed May 2023