Comparing Model Checking and Model-based Simulation

Davide Basile(✉) and Franco Mazzanti

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

Abstract. We use a railway-related case study to illustrate the differences that can be encountered while modeling and verifying a system using an academic formal verification framework and an industrial model-based framework. The different roles and structures of the two approaches are illustrated. We analyze instances where the exclusive use of interactive simulation cannot replicate the formal verification activity, and we derive some future research directions.

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

1 Introduction

Model-based development is an industrially adopted software engineering technique, supported by commercial tools such as PTC Windchill Modeler SySim [2], Sparx Systems Enterprise Architect [3], Dassault Cameo Systems Modeller [1]. This technique supports the creation of models to represent the behavior and structure of a system and is often based on the Unified Modeling Language (UML) OMG standard [24, 25]. These models are used to generate code, documentation, test cases, system simulations, and perform other tasks.

Formal methods are used and developed mainly by academia with the goal of achieving rigorous and exhaustive analysis of a system. There is a growing body of literature on the integration of formal methods into model-based development tools, and this integration is often based on alternative approaches to the formalization of UML state machines (see, e.g. [6, 21, 26, 8, 14, 29, 28]).

The combination of an academic formal verification tool (UMC [5]) with an industrial model-based development tool (Sparx EA [3]) has been investigated in [10]. The models developed using the two tools were related through a mapping of their models that preserves their semantics. A set of actionable rules to map the UMC and Sparx notations have been defined, and it was shown in detail how simulations in Sparx EA can be derived from traces generated by the UMC formal verification activity. This allowed to perform model checking of models developed through an industrial model-based development tool, to enhance the validation generally performed using interactive simulations. In particular, UMC has been used to guide the transformation of the initial, ambiguous, and

incomplete natural language requirements into more rigorous UML designs and to perform a formal analysis of them. Sparx EA has been used to introduce the initial UML designs into an industrial development framework to exploit its advanced features of simulation, documentation, and code generation.

In this paper we bring forward the activity started in [10]. We take here a wider point of view, illustrating the main differences in the formal verification of the system as done by UMC with respect to the simulations that can be performed using Sparx EA. This comparison is performed in the framework of a railway case study, and is made possible through their connection established in [10]. We show a series of examples of formal verification performed in UMC, which goes beyond the simulation capabilities offered by Sparx EA. A comparison of formal verification in UMC is conducted between modelled scenarios and a generic model of the environment, emphasizing the advantages and disadvantages of these two approaches.

The selected case study is a component of a standard railway interface called the Communication Supervision Layer (CSL) [31, 30]. This component is at the base of the handover protocol that allows a train to migrate from one RBC controlled zone to another.

Finally, we share a set of future challenges, highlighting the differences between models designed for formal verification and those designed for code generation, the handling of parallelism and others.

Related Work This work continues the line of research published in [10]. Previously, a recent set of works [22, 23] focuses on the incremental modelling of natural language requirements as UMC state machines, and associated formal validation. Initially, a UML state machine used to verify a 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. 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 [13] and CADP [18, 16, 19], where the models are formally verified to be equivalent.

In [12] 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.

Still on the translation from UML-like models to other formal notations, recent works have focused on transforming these models into mCRL2 [29, 14]. Many studies also focus on the translation from UML into the B/Event-B notation [27, 28], with formal verification performed by means of Atelier B and ProB [15]. The selected case study has also been modelled and analysed in Uppaal [9].

Structure of the paper In Section 2, we introduce the two tools used for performing simulations and model checking, respectively, and the case study. In Section 3, we describe the different roles that these tools have played in our investigation and the different structure of their models. Section 4 is where model

checking and simulations in these environments are compared. In Sections 5 and Section 6, we discuss a set of limitations and future challenges highlighted by this activity and draw some conclusions.

2 Background

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 standardized notation [24, 25], is the standard for many MBSD environments. 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 behavior 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 also have triggers, conditions and effects (denoted as `trigger[conditions]/effects`). The trigger and conditions specify when the transition is enabled, and the effects can modify the class variables and generate outgoing signals. A transition may have no trigger, in which case it is called a completion transition, and completion transitions typically take precedence over triggered transitions. A run-to-completion step represents the sequence of actions that need to be performed when a specific transition is executed in a UML state machine. Two examples of state machines, accepted by the tools UMC and Sparx EA, are in Figure 2 and Figure 3.

2.1 Sparx Enterprise Architect

Sparx Enterprise Architect [3] is an MBSD tool based on OMG UML [24]. Sparx EA offers an Executable State Machine (ESM) artifact specifically designed for simulating the *composition* of different state machines.

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 Ada, JavaScript, Java, C, C++ and C#. The source code contains the implementation of the behavioral 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 behavior. 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.

2.2 UML Model Checker

The UML Model Checker (UMC) [20, 11] is an open-access tool explicitly oriented to the fast prototyping of systems constituted by interacting state machines, developed at the Formal Methods and Tools Lab of ISTI-CNR. This tool can be easily accessed and used through its web interface at [5]. UMC allows the user to design a UML state machine using a simple textual notation, visualize the corresponding graphical representation, interactively animate the system evolutions, and formally verify (using on-the-fly model checking) UCTL [11] properties of the system behavior. Detailed explanations are given when a property is checked, when possible 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 if/when a certain state is reached.

In UMC a system is specified as a set of interacting objects. Objects are instances of Classes and are possibly customized at instantiation time with specific parameters. Classes describe the behavior of their instances in the form of UML state machines. There are a few restrictions on the UML notation supported by UMC. The most important are the absence of entry / exit /do fields inside states, the absence of history / deep history / junction and choice states, and the absence of numerical types other than integers. The formal semantics of UMC models is provided by an incremental construction of a doubly labeled transition system (L2TS) [11]. In this L2TS, nodes correspond to the composition of the internal states of the objects in the system, and each edge corresponds to an object run-to-completion step. States of the L2TS can be enriched with labels that make visible the values of the local variables of the objects. Edges of the L2TS can be enriched with information on the actions executed as part of the run-to-completion step of the evolving state machine or with the label associated with the executing transition. The supported logic is called UCTL [11]: it is an ACTL-like, state- and event-based, branching time logic that supports the typical operators (e.g., eventually, globally, next, until and fix points). In [4] a more detailed interactive presentation of the supported grammar for the models and the logical properties is available.

UMC is an on-the-fly model checker. It generates the state space (L2TS) only for the fragments needed to evaluate the formula in a top-down way, starting from the initial state and the outermost level of the formula. The validity or invalidity of a formula can be analyzed interactively by traversing the L2TS graph that explains the result. Linear sequences of verification steps can be displayed also in the graphical form of a message sequence chart.

UMC is an academic tool that is primarily used for research and teaching. A comparison of UMC with other tools can be found in [17]. In these experiments, we made use of UMC version 4.9 (2024).

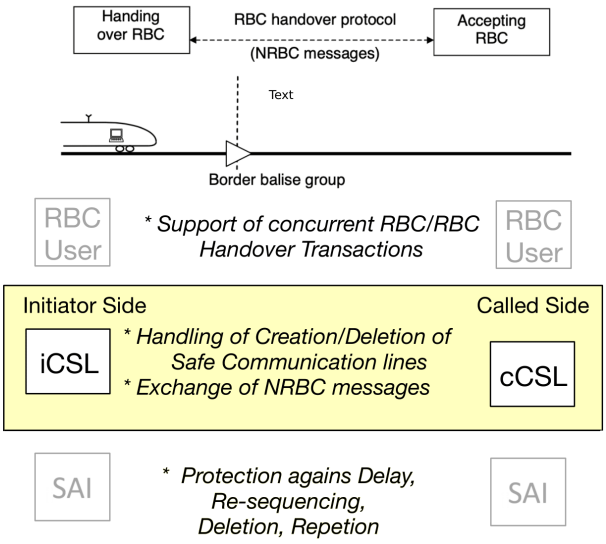

Fig. 1. The overall structure of the handover protocol

2.3 Case study

The chosen case study is a subset of the RBC/RBC handover protocol [31, 30]. The RBC/RBC handover 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 trains inside its assigned geographic 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 neighboring RBC must take place to manage the transfer of control responsibilities. This exchange of information is supported by the communication layer specified within the documents: UNISIG SUBSET-039 [31], UNISIG SUBSET-098 [30], and the whole stack is implemented by both sides of the communication channel.

Figure 1 summarizes the overall relation between the components of the handover protocol. The RBC/RBC communication system consists of two sides that are respectively configured as "initiator" and "called". Each side is composed of levels. For example, the RBC User level is responsible for handling concurrent handover transactions involving multiple trains. The Communication Supervision Layer (CSL) of the SUBSET-039 and the Safe Application Intermediate SubLayer (SAI) of the SUBSET-098 support, respectively, the creation/deletion of safe communications and the protection of the transmission of messages exchanged. In particular, the CSL is responsible for requesting the activation – and in the event of failure, the reestablishment – of the communication, for controlling its liveliness, and for forwarding the handover transaction messages. The RBC User communicates with the CSL, and the CSL with the SAI. The SAI

of the two sides communicate with each other through lower levels not reported here for brevity.

In this paper, we focus on the CSL model and show all the details of its initiator side (ICSL). In other words, the SAI, the communication network, and the RBC User are considered as the environment of the modeled CSL (both for initiator and called sides). The called side (CCSL) is also modelled, and can be inspected at [7]. It is not shown here for reasons of space.

The CSL specification is highly parametric. In particular, its behavior strictly depends on the delay values used for transmitting live signals, for triggering the reset of the communications in the absence of signals from the other side, and for triggering the restart of the connection process. Together with the modeling of the environment, the instantiation of these parameters becomes part of the scenarios under analysis.

The state machine modeling the ICSL is provided both as a formal model in UMC (see Figure 2) and as a semi-formal model in Sparx EA (see Figure 3). These two models are related through a mapping described in [10], and the difference relies on the underlying semantics of the two tools (formal and semi-formal). Each transition is labeled with a name (e.g., R1) to keep track of the correspondence between the two models. In both Figure 2 and Figure 3 transitions from R1 to R4 are grouped to enhance readability. The send and receive operations are represented, respectively, by the trigger and effect of a transition label (see Section 2). Communications are asynchronous and each state machine has a FIFO buffer.

The ICSL state machine is made up of two states `NOCOMMS` (the two RBC are disconnected) and `COMMS` (the two RBC are connected). The initial state is `NOCOMMS`. From the state `NOCOMMS`, a counter `connect_timer` is incremented at each reception of a `TICK` signal from the clock (R6) (note that the model is not real time and the length of the interval of time between ticks is not modelled).

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 called CSL (CCSL)), and the counter is reset (R5).

The signal of connection `SAI_CONNECT_confirm` (signaling the connection of the CSL) 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 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 will 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 disconnection

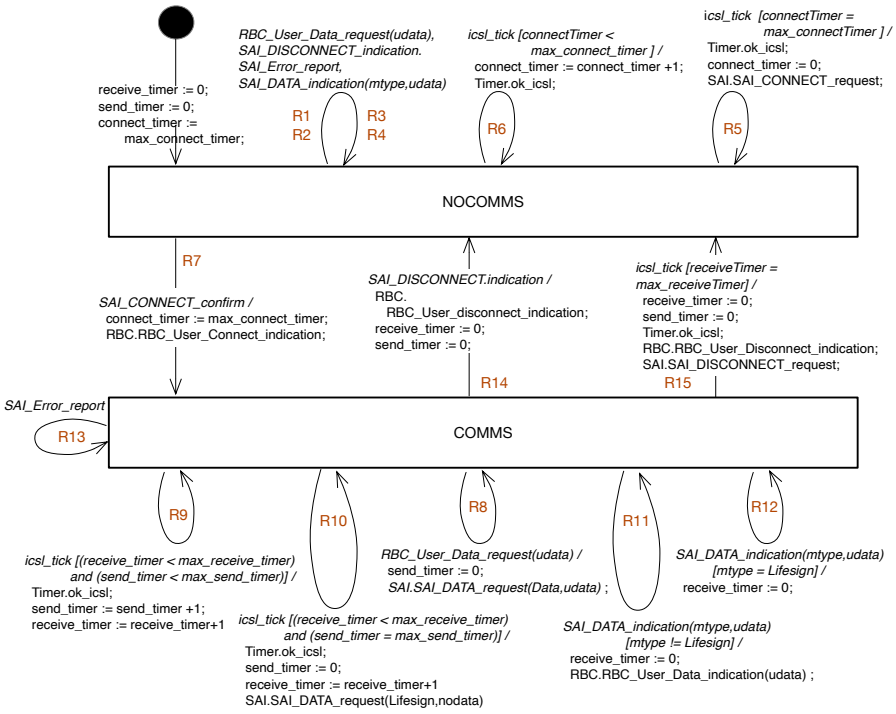

Fig. 2. The Initiator Communication Supervision Layer State Machine of UMC

signal is sent to both the user and the SAI (R15). If the disconnection message is received from the SAI (R14), then it is only forwarded to the user and the connection is closed.

3 The different roles and structure of UMC and Sparx EA environments

Starting from the initial requirements in natural language, UMC has been used to develop a clear, correct, and executable UML design. In fact, several ambiguities and uncertainties were identified and corrected during the UMC design process.

This executable UMC description is already at a lower level than the higher-level natural language requirements. For example, if we consider transition R15 in Figure 2, the original natural language requirements stated that:

> "when ICSL is in the `COMMS` state, if the receive timer expires, a disconnect indication is sent to the user, and a disconnect request is sent to the SAI."

Nothing is said about the order in which these two messages should be sent. Nothing is specifically stated about how the timer expiration should be observed (here, we have used an integer `receive_timer` variable periodically incremented

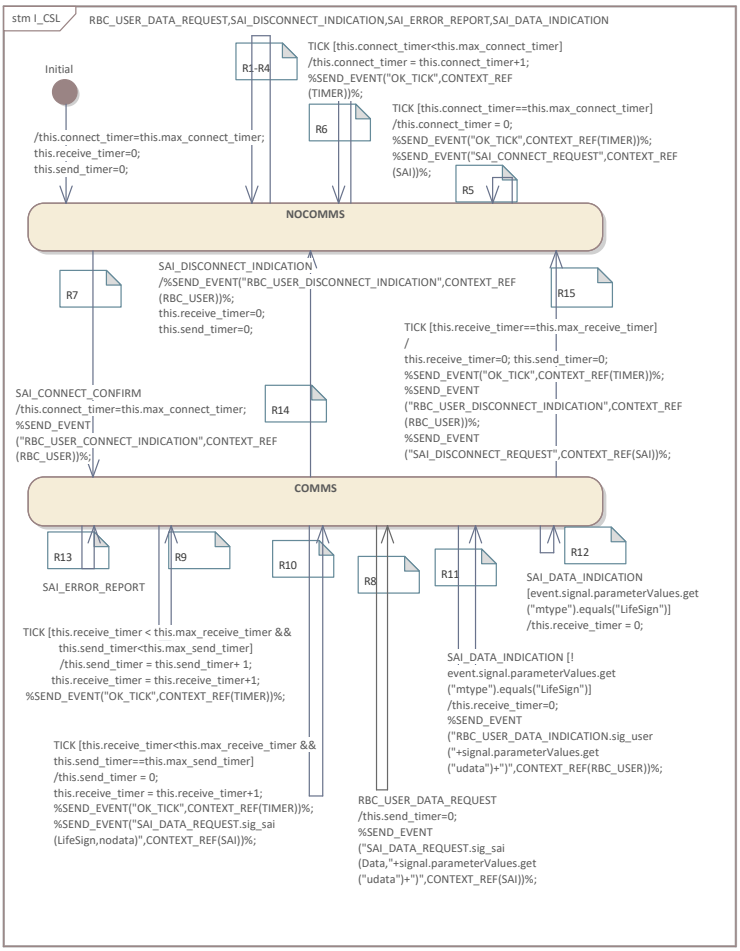

Fig. 3. The Initiator Communication Supervision Layer State Machine of Sparx EA

by tick events generated by a Timer component), and nothing is said about resetting the `receive_timer` variable. In particular, the choice to reset this variable immediately—rather than, for example, when reentering the `COMMS` state and restarting the timer—was made to avoid unnecessary growth of the state space during verification. We decided to adopt the same implementation choices in Sparx EA to preserve, as much as possible, a strict coherence between the two platforms. In Section 5, we will discuss some limitations and challenges ahead for this kind of choice.

In UMC, all nondeterministic aspects of UML are preserved and considered. All components are modeled as concurrent, and all possible interleavings among them are explored. This is the correct approach if we aim to verify the correctness of UML designs in an abstract way. However, Sparx EA uses a deterministic execution engine to execute the system components, making proprietary deci-

sions to resolve all nondeterministic aspects of UML. The Sparx choice might be reasonable from the standpoint of constructing a sequential system that should be as deterministic as possible, to simplify monitoring and testing. However, it is important that the design is also proved correct at the more generic UMC level. This ensures that future changes in internal code generation or system simulation details will not affect the correctness of the system. From a documentation standpoint, it is also important that the correctness of the state machine diagrams describing the system does not rely on hidden assumptions about the underlying execution engine. Of course—as will be described in more detail in the next section—not all UMC executions will be repeatable in the Sparx EA setting.

UMC requires the presence of a complete, closed system to analyze all possible system evolutions. This necessitates the generation of usually nondeterministic environment stubs in addition to the specific components under design. In our case, the environment stubs that must be composed with the two initiator and called CSL components are those modeling the RBC Users, the lower levels of communication (SAI and network levels), and a Timer.

Sparx EA, instead, allows us to stimulate a system component through direct user interactions via the GUI, triggering a specific execution trace. In this case, only the code of the component under development needs to be generated, and it is the user's task to provide all the missing interactions from the RBC User, the SAI components, the Timer, and the network. Some "sink" stubs may still be needed to receive outgoing signals from the CSL components. Note that nothing prevents us from encoding in Sparx the same environment stubs as the ones used in the UMC context, but their execution within the Sparx simulation environment would be subject to all the internal deterministic choices of the execution engine, making them scarcely useful for modeling purposes. When more than one component is involved, several execution steps may be performed before control is returned to the user.

4 Comparing Model Checking and Simulations

In this section, we will analyze scenarios where it is not possible to replicate the outcome of the model checking phase (i.e., the trace) as simulations. Indeed, even if the correspondence between models is sound, this is still possible since UMC overapproximates all possible behaviors of Sparx EA.

All models are available at [7].

Example of traces not reproducible in Sparx EA We consider a scenario in which the RBC User on the initiator side (as shown in Figure 4) may nondeterministically decide to send a data request to the initiating CSL, but only if it is in a connected status (i.e., after having received from CSL a connect indication without any subsequent disconnect indication).

In this scenario, when the CSL is in state `COMMS` and the RBC User `connected` variable is true, two different signals could arrive at the CSL: a data request

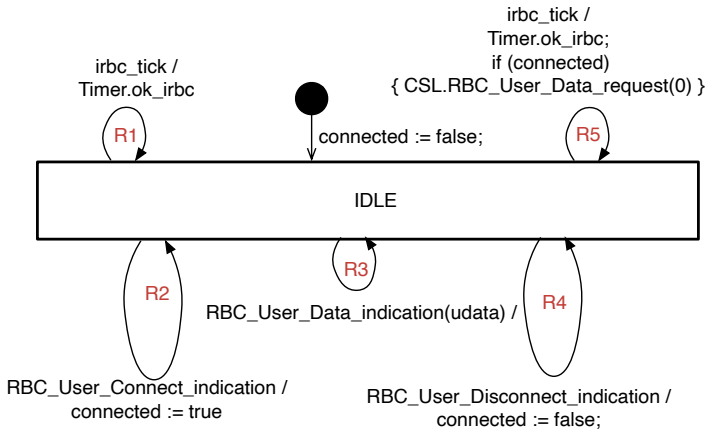

Fig. 4. The UMC state machine of the RBC User of the initiator side for the second example

from the initiator RBC User, and a disconnect indication from the SAI. Both events are stored in the buffer of the initiator CSL before it can process any of them. In this case, a possible system evolution might be that the disconnect indication is treated as first, changing the CSL state to `NOCOMMS` and forwarding a disconnect indication to the RBC User. At this point, the RBC user data request is dispatched from the event queue. In this case, being the CSL in state `NOCOMMS`, the signal is discarded.

This evolution is clearly represented by the sequence diagram in Figure 5,[1] which can be produced, for example, when UMC evalutes the formula:

```
EF {R1_ICSL_discard_userdata}
```

This formula checks whether there exists an execution in which the transition R1 of the initiator CSL is eventually fired (for brevity, in Figure 2 the transitions are labelled with abbreviations of the original labels, e.g., R1 instead of R1_ICSL_discard_userdata).

To reproduce the trace in Sparx EA, the Sparx user assumes the role of the environment (in this case, the RBC User and the SAI). However, in the Sparx EA simulator, it is not possible for the interactive user to send to (and enqueue in) the CSL two consecutive signals, because as soon as the first event is triggered, it immediately activates its corresponding transition. In particular, when the Sparx user sends the first signal `SAI_DISCONNECT_indication`, this will trigger a run-to-completion-cycle of the CSL that will execute the transition R14 of the initiator CSL (see Figure 3). This will cause the immediate emission of the signal `RBC_User_Disconnect_indication` from the initiator CSL to the RBC User. Therefore, the last three interactions shown at the end of the

[1] The event names appearing in the formulas and in the arrows of Figure 5 denote the event of sending a signal (and its storage in the receiver's buffer), and not the event of picking up the message from the buffer as done by the receiver.

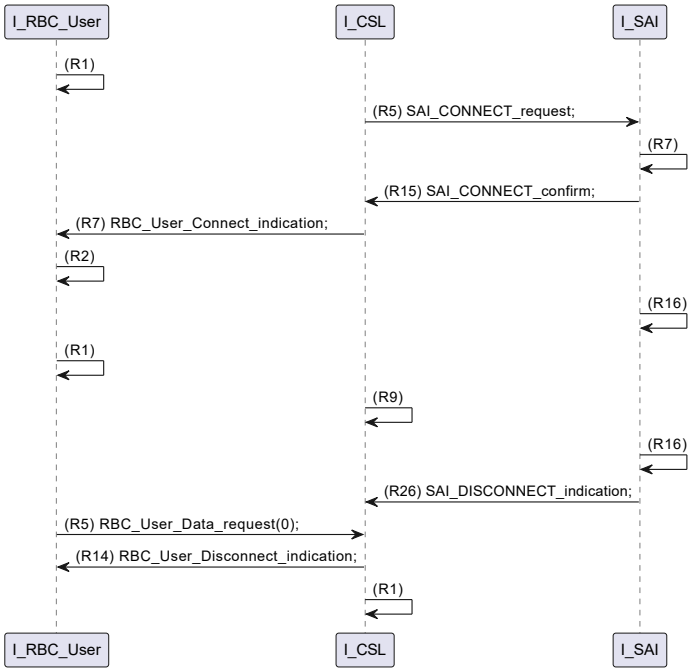

Fig. 5. The trace produced by UMC explaining the verification of the first example, not reproducible in Sparx EA

trace in Figure 5 (i.e., `SAI_DISCONNECT_indication`, `RBC_User_Data_request`, `RBC_User_Disconnect_indication`) appears not to be reproducible in Sparx EA.

To precisely reproduce this scenario, it would be necessary for the Sparx user to send two signals to the CSL (one coming from the SAI and one coming from the RBC User) before the system executes the run-to-completion.

Example of a formal verification that does not produce any trace The model checking of certain formal properties (e.g. reachability) produces a trace indicating whether the property holds. However, there are instances where a formal property is confirmed (or not) to hold in the model, yet no single trace can be given as explanation. This occurs when the successful proof of a universal formula (or the counterexample of an existential formula) can be described only by reporting a complex fragment of state space of the model, usually in the form of a tree or subgraph, rather than just a single trace. Consider, for example, the following property:

```
AG(
  [IRBC_User_Connect_indication] (
    A[{not IRBC_User_Connect_indication}
    W {IRBC_User_Disconnect_indication}]))
```

The formula makes use of three temporal operators: `AG phi1` stating that subformula `phi1` should hold for all states reachable from the current one; where

`phi1=[cond2] phi2` stating that subformula `phi2` should hold for all states reachable after a transition satisfying `cond2` (if any); and `phi2=A[ {cond3} W phi3 ]` stating that for all paths starting from the current state the transitions should satisfy the condition `cond3` until a state is reached that satisfies the subformula `phi3` (if ever found).

This property asserts that whenever the RBC User receives a connect indication (from the CSL), no other connect indication will be received until a disconnected indication is received. The weak until operator `W` specifies that it could also be the case that no disconnect indication will ever be received by the RBC User. We verify this formula using the previous scenario. The property is formally verified by UMC to hold in the model. However, no single trace is sufficient to prove that this formula holds. In fact, UMC necessitates visiting the whole state space, which consists of 51303 states, to prove the formula.

In summary, whenever a universally quantified property is verified not to hold or an existentially quantified property is verified to hold, a trace (generated by UMC) exists to witness the verification. In such cases, it is possible to test whether the same trace also exists in the Sparx EA simulation. However, for other formulas, one can only rely on UMC for formal property verification. In this case, formal verification is useful in providing guarantees that would otherwise be hard to obtain solely by relying on the semi-formal Sparx EA model.

Example of an infinite trace Finally, it could also be the case that the formal verification produces, as witness, an infinite trace. For example, consider the following property:

```
EF { IRBC_User_Connect_indication }
        ( EG {not IRBC_User_Disconnect_indication} )
```

The formula makes use of two temporal operators: `EF {cond1} phi1` stating that there exists a state reachable from the current one with a transition satisfying `cond1` that lead to to a state satisfying `phi1`; and `phi1=EG {cond2}` stating that there exists a path starting from the current state whose transitions continuously satisfy `cond2`.

This property asserts that there exists a trace where the RBC User receives a connect indication (from the CSL), and subsequently, it may never receive a disconnect indication from the CSL. In other words, in the model, it is possible that the connection remains open forever.

The formal verification produces two sequence diagrams that explain the two nestings in the formula. The first sequence diagram illustrates the connection phase (not shown here). The second sequence diagram is shown in Figure 6. Figure 6 contains an infinite loop, within which the initiator CSL continuously exchanges life signals with the other side. Therefore, it is not possible to entirely reproduce this trace in Sparx EA, but only a finite fragment of it. Moreover Sparx EA does not provide the user with any help for detecting that during trace simulation we have encountered a state already seen in the same trace,

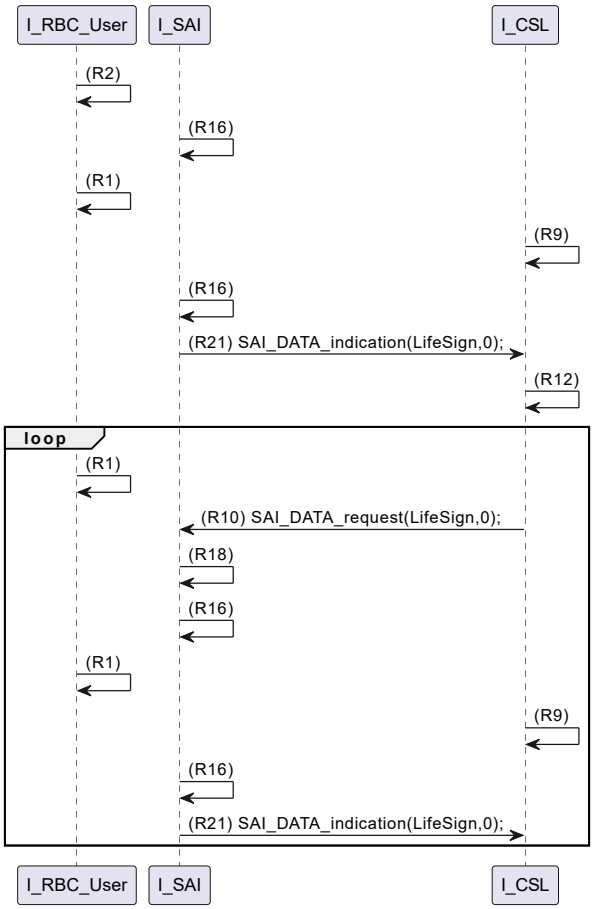

Fig. 6. An infinite trace generated by UMC

therefore it becomes very difficult to understand when just a trace fragment would be sufficient.

In summary, formal verification encompasses infinite behavior that is not reproducible solely by relying on the interactive simulation capabilities of Sparx EA.

The Role of the Environment The modeling of the environment is peculiar to the formal verification performed in UMC, while it may not be necessary for interactive simulations. In fact, during the interactive simulation, the Sparx user acts as the environment and sends signals to the state machines.

In the previous examples, the environment was used to model specific *scenarios* where the system is being analyzed. Another possible approach is to model a generic environment that does not implement any specific scenario. In this case, the environment receives and discards all messages from the system and may nondeterministically send the available signals to the system.

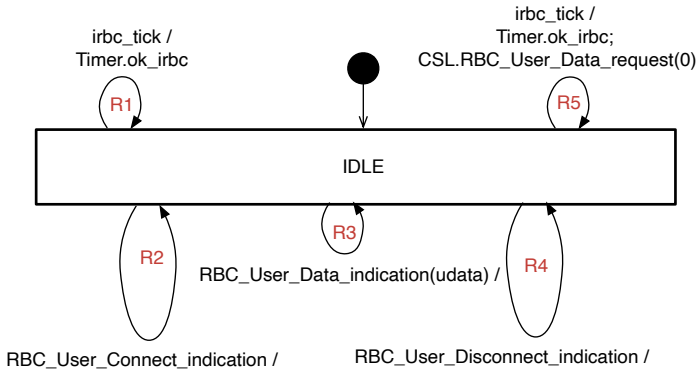

Fig. 7. The UMC state machine of a generic RBC User

The advantage of having a generic environment is that it encompasses all the behavior that can be manifested by any specific scenario. Therefore, a safety property (i.e., ensuring that nothing bad ever happens) verified to hold using the generic environment will also hold for any specifically modeled scenario. The disadvantage is that many false positives may be present, i.e., behavior manifested by the environment that is not present in the real system. The over-approximation of a generic environment may also increase the state space during formal analysis.

Consider again the first example discussed in this section (i.e., an example of traces not reproducible in Sparx EA). We replace a portion of the environment, specifically the RBC User depicted in Figure 4, with a generic component shown in Figure 7. The RBC User in Figure 7 is generic: it receives and discards all signals, and nondeterministically sends signals to the system. By comparing these two RBC User components, we observe that the generic RBC User may send a signal to the CSL at any time, even when disconnected. This is an example of over-approximation: in the real system, data request signals will only be emitted when the user is connected.

We demonstrate how the verification using this generic environment impacts our analysis. Firstly, in the original example (without replacing the RBC User), the overall state space of the entire model consists of 9495 states. By switching to the generic RBC User, we observe a deterioration in the size of the state space, which now consists of 10336 states. Note that we only generalized a single component of the environment. Moreover, the formal verification of the property

$$\texttt{EF}\ \{\texttt{R1\_ICSL\_discard\_userdata}\}$$

produces a trace shown in Figure 8. In this case, the generated trace is not significant as it represents a false positive. Indeed, the trace simply shows an execution where the disconnected RBC User sends a data request to the disconnected CSL, which discards it. This trace is not significant because in the real system it will never occur.

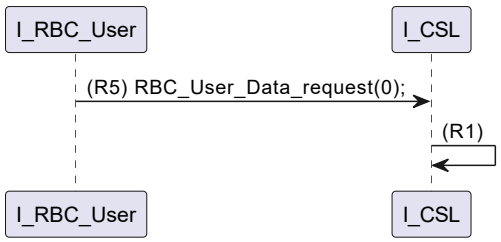

Fig. 8. The trace produced by UMC using a generic RBC User component

In summary, adopting an environment that models specific scenarios has proven to be more effective. Indeed, when switching to a generic environment, the results of formal verification may not be significant (i.e., false positives), and the performance (i.e., the state space) may deteriorate. In addition, the presence of false positives makes the discovery of true positives (if any) more difficult. The risk of not covering unexpected errors is a downside.

In the Sparx EA context, since it is the user who directly triggers the environment events, both approaches (generic user and specific user) can be easily simulated.

The use of a "generic" environment could actually be useful in evaluating the robustness of a component, even in the presence of misbehavior by the environment or other components. Indeed, transitions R1 to R4 (see Figure 2) have been introduced to model that messages arriving at the wrong time are simply discarded and have no effect on the system.

The diagram in Figure 2 does not explicitly describe what should happen if a `SAI_CONNECT_confirm` message arrives when the ICSL is already in the `COMMS` state. This is a situation that should be impossible in the real system but can still be triggered by user interactions in Sparx EA or by a generic ISAI stub in UMC. In any case, the UML semantics requires that the message must be discarded, but this fact is left implicit in the design. It is a design choice whether or not to make this kind of behavior explicit in the diagram. During the UMC analysis, such situations (i.e., messages implicitly discarded because they do not immediately trigger a transition when dispatched) can be identified and flagged as possible design errors.

5 Limitations and Future Research Directions

We now discuss the limitations and future research directions.

5.1 Parallelism and Atomicity

We recall that a run-to-completion of a UML state machine consists of the execution of different actions. Internally to each state machine, the run-to-completion steps appear to be executed atomically (even in the presence of parallel regions of states). Externally, when different state machines are executed in parallel, the

external effects (e.g., signals) produced by different run-to-completions of different state machines may overlap and interfere. For example, this may occur when a transition of a UML state machine sends several signals (e.g., the transition R15 of the initiator CSL in Figure 2). At the system level, the sending of these signals may interleave with signals sent by other state machines.

Currently, in the latest version of UMC, the external effects of different state machines' run-to-completions are never interleaved. Therefore, some behaviors that could be observable in the real system are not reproduced by UMC.

This is a current limit of UMC that should be overcome in future releases.

A temporary solution to this problem is to require that the effects of transitions of state machines not contain more than one communication action. This may force the splitting of a transition that sends several messages (e.g., R15 in Figure 2) into a sequence of transitions, each sending only one message.

5.2 Differences Between Modelling Activities

The modelling activities concerning formal verification and software development, although being related, also present differences. Indeed, formal verification may require the generation of the state space, as in model checking. In this case, optimizations may be performed during modeling to reduce the state space. On the other hand, in software development, rules are followed to maintain clarity and readability of the models. These rules may be conflicting with the optimizations performed for formal verification. As already discussed in Section 3, consider for example the case of resetting a variable. In a model used for code generation, the variable should be reset only when strictly necessary to avoid unnecessary instructions. Conversely, in a model used for formal verification, a variable is reset as soon as possible to keep the state space from growing. We note that these changes are made in a way that does not affect the overall semantics of the model.

For example, consider Figure 2. The variables `send_timer` and `reset_timer` are reset after moving from state `COMMS` to state `NOCOMMS`. This guarantees that, whenever the system is in state `NOCOMMS`, both variables have value zero. However, the reset operation is only needed when entering state `COMMS`. Thus, for readability, the reset should only be performed when entering the state `COMMS`, instead of being anticipated earlier. Therefore, the related model used for the development and code generation (ie Sparx EA) may need to be slightly adjusted to adhere to readability and other model-based development guidelines (e.g., in Figure 3 the reset of the two variables should be moved to `R7`).

5.3 Automatic Translations and Sparx EA Profile for Model Checking

It might be interesting to exploit model checking on the same specific Sparx EA executable model, and not only on the abstract UML designs. To achieve this goal, two problems should be dealt with:

1. The model should be automatically regenerated from the Sparx EA model (probably by exploiting the XMI export feature of Sparx).
2. A Sparx EA profile should be introduced in UMC that constrains all the nondeterministic aspects in precisely the same way as Sparx EA does at runtime. This would require a deep analysis of Sparx EA's behavior but might actually be of great help for analyzing the actual executable system.

5.4 Sparx EA Custom Runtime

Another possibility that could be investigated is the manipulation of the default Sparx EA engine managing the deterministic choices that affect the allowed system execution traces, by introducing pseudo-random or user-guided choices instead of fixed ones. In this case, the generated code would not be suitable for the real product but could serve as a tool for more extensive testing and more controlled simulations of the system.

6 Conclusion

We have analyzed some of the benefits provided by the formal verification activity compared to interactive simulations. The tools adopted were an academic formal verification tool, UML Model Checker (UMC), and an industrial model-based development tool, Sparx Enterprise Architect (Sparx EA). This analysis is part of growing interest in the integration of formal techniques within industrial tools for the development of safety-critical systems. The selected case study belongs to the railway domain.

We have shown that formal verification enhances validation based solely on interactive simulations. We have analyzed instances where the exclusive use of interactive simulation cannot replicate the formal verification activity. From this experience, we have derived some future research directions.

Extensions to UMC are needed to manage parallelism and atomicity in the execution of run-to-completion steps of different state machines. Concerning other comparison points, a systematic evaluation (as previously done in [17]) is left for future work. Another direction is researching modeling approaches that balance modeling activities oriented towards formal verification and software development, and developing methodologies for the transformation of models between these two purposes. Finally, a custom Sparx EA profile and custom runtime could be exploited to align the execution traces of the two tools.

Acknowledgements 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), and the MUR PRIN 2022 PNRR P2022A492B project ADVENTURE (ADVancEd iNtegraTed evalUation of Railway systEms) funded by the European Union - NextGenerationEU.

References

1. Dassault Cameo Systems Modeler, https://www.3ds.com/products-services/catia/products/no-magic/cameo-systems-modeler/, accessed April 2025
2. PTC Windchill Modeler SySim, https://www.ptc.com/en/products/windchill, accessed April 2025
3. Sparx Systems Enterprise Architect, https://sparxsystems.com/products/ea/index.html, accessed April 2025
4. UMC interactive syntax help, http://fmt.isti.cnr.it/umc/V4.9/sdhelp.html]
5. UMC project website, http://fmt.isti.cnr.it/umc
6. 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
7. Basile, D., Mazzanti, F.: Comparing UMC Model Checking and Sparx EA Simulation - Complementary Material (September 2025). https://doi.org/10.5281/zenodo.17045133
8. 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
9. 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
10. Basile, D., Mazzanti, F., Ferrari, A.: Experimenting with formal verification and model-based development in railways: The case of UMC and sparx enterprise architect. In: Cimatti, A., Titolo, L. (eds.) Formal Methods for Industrial Critical Systems - 28th International Conference, FMICS 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14290, pp. 1–21. Springer (2023). https://doi.org/10.1007/978-3-031-43681-9_1
11. 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
12. 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
13. Bendisposto, J., Clark, J., Dobrikov, I., Körner, P., Krings, S., Ladenberger, L., Leuschel, M., Plagge, D.: ProB 2.0 Tutorial. In: Butler, M., Hallerstede, S., Waldén, M. (eds.) Proceedings of the 4th Rodin User and Developer Workshop. TUCS Lecture Notes, Turku Centre for Computer Science (2013)
14. 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
15. 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
16. 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 January 2024
17. 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
18. 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
19. 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
20. 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
21. 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
22. Mazzanti, F., Belli, D.: Formal Modeling and Initial Analysis of the 4SECURail Case Study. In: Proceedings of Fifth Workshop on Models for Formal Analysis of Real Systems (MARS), Munich, Germany, 2nd April 2022. pp. 118–144. Springer (2022). https://doi.org/10.4204/EPTCS.355.6
23. 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
24. Object Management Group: Unified Modelling Language (December 2017), https://www.omg.org/spec/UML/About-UML/
25. Object Management Group: OMG Systems Modeling Language (OMG SysML) (November 2019), http://www.omg.org/spec/SysML/1.6/
26. 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
27. 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
28. 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
29. 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
30. 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 April 2025
31. 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 April 2025