Automated Verification of Executable UML Models

Helle Hvid Hansen[1], Jeroen Ketema[2], Bas Luttik[1], MohammadReza Mousavi[1], Jaco van de Pol[2], and Osmar Marchi dos Santos[3]

[1] Eindhoven University of Technology, Eindhoven, the Netherlands
[2] University of Twente, Enschede, the Netherlands
[3] University of York, York, England

Abstract. We present a fully automated approach to verifying safety properties of Executable UML models (xUML). Our tool chain consists of a model transformation program which translates xUML models to the process algebra mCRL2, followed by symbolic model checking using LTSmin. If a safety violation is found, an error trace is visualised as a UML sequence diagram. As a novel feature, our approach allows safety properties to be specified as UML state machines.

1 Introduction

UML has become the popular modeling approach, driving the development of industrial applications in many different fields. One such field is the railway industry, where *Executable UML (xUML)* [35] is gaining popularity for specifying critical applications such as railway interlockings. The main goal of interlockings is to ensure that trains neither collide nor derail. This is achieved by establishing routes, which comprise tracks, points and other railway components, along which trains can pass safely. Correctness of interlockings is certainly imperative, and hence rigorous methods should be employed to verify their safety properties.

In this paper, we report on an automated approach to verifying safety properties of xUML models. These xUML models may specify the functional requirements of interlocking systems, but in principle they are not restricted to the interlocking domain. One of the target groups for our tool chain are modeling engineers with no training in formal methods. We accommodate this target group by allowing the functional requirements (i.e. the actual model) as well as their safety properties to be specified in xUML. The safety properties are expressed as state machines that "observe" the behavior of the model, and issue an error signal if a safety property is violated.

The verification is carried out using the well-known technique called model checking where the entire state space of a formal model is exhaustively explored and checked against a property. In our case, the formal model is specified in the process algebra mCRL2 [21] and the exploration is done using the symbolic model checking tools of LTSmin [8]. In particular, safety violations are found by detecting the above-mentioned error signals in the mCRL2 model. An xUML model defines a generic model of object behaviour, but an mCRL2 model describes the behaviour of a concrete collection of interacting objects. Our tool chain therefore also takes as input an instance specification from which a model instance can be created. The verification is thus always carried out with respect to a particular model instance.

The mCRL2 model is obtained by automatically translating the xUML model, its safety properties and the instance specification into mCRL2. This automated translation is implemented using the Eclipse-based model transformation tools of Epsilon [30, 31]. The translation goes via an internal format called iUML. This iUML representation is a intermediate step between the hierarchical xUML model and the "flat" transition system specified in mCRL2. We have several reasons for using such an intermediate representation. First, a one-step translation from xUML to mCRL2 would be quite complicated to implement due to the significant differences between the two languages. In particular, expressions and actions that can be statically evaluated are transformed in the iUML rather than translated into mCRL2 and evaluated there. Second, the iUML allows us to perform static analysis tasks which are not easily cast as a model checking task. Third, the iUML is sufficiently general to support different variations of the translation into mCRL2, and we expect that it can serve as a basis for translations into other target languages such as Promela [25] and Event-B [1].

The functionalites of our tool chain are illustrated in Figure 1 of the next section. To summarise, the tool chain consists of the following three main steps: (1) Automated translation of the model, its safety properties and an instance specification from xUML into the formal specification language mCRL2. (2) Checking for safety violations by searching for error signals in the mRL2 model. (3) Visualisation of an error trace as an UML sequence diagram, in case a safety violation is found.

In the paper [22], we reported on the early developments of our approach. The main contributions of the present work with respect to [22] are:

1. A method for specifying safety properties as UML state machines.
2. A tool chain which realises a fully automated verification and feedback trajectory, where both input and output are expressed in UML.
3. Automation of the translation from xUML to mCRL2, and a more detailed description of the translation itself.
4. Investigations into the scalability of our approach by conducting verification of several different xUML interlocking models and more realistic track layouts (that yield model instances).

We believe that items 1 and 2 greatly improve the usability of our approach to modelling engineers and domain specialists, since only knowledge of UML is required, rather than familiarity with formal methods. We should point out that our current translation differs at some point from the translation in [22] (see end of Section 5).

The rest of this paper is structured as follows. In Section 2, we give a more detailed overview of our tool chain and its architecture. In Section 3, we introduce the subset of xUML supported by our tool chain, and discuss its syntax and semantics. In Section 4, iUML is introduced, which is our intermediate format between the input models and the mCRL2 specification used for model checking. Section 5 presents the translation schema and Section 6 outlines the verification methods we used, the challenges we faced, and the results obtained. Finally, in Section 7 we conclude by discussing related and future work.

2 Tool chain

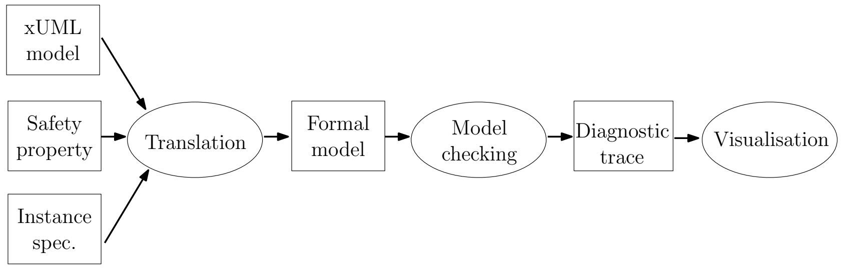


Fig. 1: Automatic verification of xUML models

The tool chain achieves its goal in a number of steps of which the details are hidden behind a rudimentary user interface. The core of this process is depicted in Figure 1. Next, we give more details on each part of the tool chain.

Input. The inputs of our tool chain comprise two xUML models (specifying the model itself and its safety properties) and an instance specification (defining the object identifiers and their roles in associations). The family of xUML models that can be used as input are described in Section 3. The input xUML models must be provided in an XMI format that is compatible with the Eclipse UML Tools v3.0 (which implement the UML 2.2 specification). Currently we support two ways of generating these XMI files: An xUML model created in Artisan Studio can be transformed into XMI by our tool chain, or an xUML model can be created in the UML modeling tool Papyrus [37] which is based on Eclipse just as several other parts of our tool chain. Finally, the instance specification must be supplied in a simple text-based format that we defined for this purpose.

Translation. The automated translation from xUML to mCRL2 is implemented in the Eclipse Modeling Framework [41] (Eclipse Galileo version 3.5.2). In the Eclipse Modeling Framework (EMF), metamodels define the abstract syntax of EMF models. In close analogy, the UML superstructure [36] provides a metamodel that defines the UML language elements and their relationships. But unlike UML, EMF models generally do not have a graphical representation. As already mentioned, the automated translation goes via an intermediate representation called iUML, and we thus have metamodels that define the structure of iUML models, and of their expressions and actions. We specify metamodels using the textual syntax of EMFText [23] which also provides a parser generator facility that we use to transform well-formed strings to EMF models. We access and manipulate EMF models using the Epsilon model transformation tools [30, 31]: The transformation from UML to iUML is implemented in the model-to-model transformation language ETL. Generating mCRL2 code from iUML models is carried out using the model-to-text transformation language EGL.

Model checking. The generated mCRL2 specification is verified using a combination of the mCRL2 [21] and LTSmin [8] model checking tool sets (revision 8543 and the next branch dated 3-3-2011, respectively). From the mCRL2 tool set we use a number of utilities that pre-process the mCRL2 specification. In this pre-processing, parallel behavior is turned into non-deterministic sequential behavior and the result is simplified by removing redundant and constant data parameters. The latter step potentially reduces the size of the generated state space. The actual state space exploration is achieved using the symbolic reachability tool from the LTSmin tool set, which also provides distributed and multi-core reachability tools, and a tool to reduce state spaces modulo branching bisimulation.

Visualisation In the case one of the specified safety properties is violated in the chosen instance of the xUML model, LTSmin generates a trace, that is, a sequence of actions leading to a violation of the property. This trace is visualized in Eclipse in the form of a UML message sequence diagram. If none of the safety properties are violated, a message is displayed reporting this result.

3 Executable UML: translation domain

In this section we describe the subset of Executable UML (xUML) [35] that is covered by our translation. For further information on the UML, we refer to [36].

3.1 Models, classes and state machines

Informally stated, an xUML model is a hierarchical structure that defines types of objects, their relationships and how they react to events in the system. In our subset of xUML, a *model* consists of signals, events, classes and associations. An event can be a *signal event* denoted simply by the signal name; a *change event*, denoted by $\mathsf{when(cond)}$ where cond is the change expression; or a (relative) time event, denoted by $\mathsf{after(n)}$ where n is the timeout delay. We do not include absolute time events. A *class* consists of properties, receptions and a state machine. A *property* can be an attribute, a generalisation or an association end. An attribute can be *derived* in which case it is defined by a Boolean expression. Derived attribute names start with a slash (/). A *generalisation* in a class C is a reference to another class. A class $\mathsf{C'}$ is called a superclass of C if $\mathsf{C'}$ can be reached from C via the transitive closure of the generalisation relation. We require that attribute names are unique within a class and its superclasses. The *receptions* declare the signals that the class will react to, and the state machine specifies how the class reacts to events. State machines are described below. An *association* is an n-ary relation between classes.

Classes and their associations can be graphically represented by a class diagram, as illustrated in Figure 2 which shows the class diagram of a toy example called *Micro interlocking*, kindly provided to us by KnowGravity[4]. Figure 2 shows that there are five classes: track, point, signal, route and HAL device where HAL device generalises track, point and signal. (The boxes with labels LCL and HAL can be ignored.) Furthermore, there are four binary associations, one between route and track, two between route and point, and one between route and signal. Note that a route instance should be linked to exactly one signal instance via the property entry_signal. Figure 2 also shows that in class point there are derived attributes called /at_left, /at_right and /is_locked, but their definitions are not shown. Similarly, the classes track and route also have derived attributes.

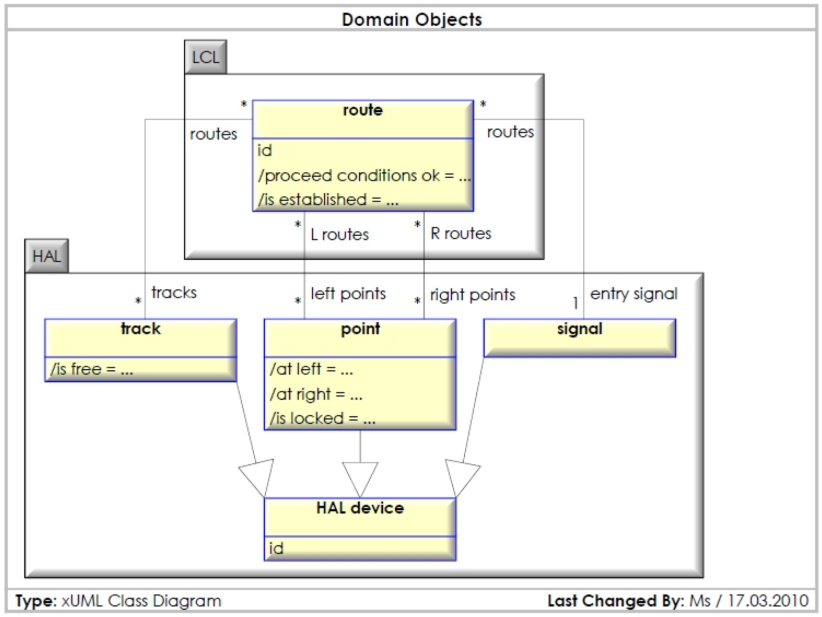


Fig. 2: Class diagram of Micro interlocking

A *state machine* consists of regions and states (alternatingly nested), and transitions between states. For a state s, we denote by $Regions(s)$ the regions immediately contained in s. Similarly, for a region r, we denote by $States(r)$ the states immediately contained in r. A *root region* is a region which is contained

[4] http://www.knowgravity.com

only in the state machine, but not in any state. A state may be *simple, composite* or *concurrent* meaning that it contains zero, at least one, or at least two regions, respectively. Furthermore, states may have *entry* and *exit* actions. The only pseudo-states we allow are *initial pseudo-states*, so in particular we exclude (deep) history, final, junction pseudo-states, and entry/exit points. The sets of all states and regions contained in the state machine of a class C are denoted by $\mathit{States}(\mathsf{C})$ and $\mathit{Regions}(\mathsf{C})$, respectively.

Example 1. The state machine diagram of point is found in Figure 3, and it shows, for example, that working and moving are composite, non-concurrent states (i.e. they contain one region), and the region of working contains the states left, right and moving. There are no concurrent states in point. All states except for startup and moving have entry actions. The tags, such as `<i>` or `<ic>`, that prefix signal names indicate a signal stereotype, but this is only a naming convention, and no UML semantics is derived from these tags.[5]

A *transition* t goes from a source state $\mathit{source}(t)$ to a target state $\mathit{target}(t)$, and is labelled with $\mathit{trg}[\mathit{grd}]/\mathit{eff}$ where $\mathit{trg}, \mathit{grd}, \mathit{eff}$ are the trigger, guard and effect of t, respectively. The trigger is an event, the guard is a Boolean expression, and the effect is a sequence of actions that should be carried out when the transition fires. We require that transition has a single event as trigger, but the guard and the effect can be omitted. Moreover, we require that for any state s there is at most one transition triggered by a time event with source s.

A transition t can be *internal* meaning that $\mathit{source}(t) = \mathit{target}(t)$ and firing t does not trigger exit or entry actions. In state machine diagrams, an internal transition in a state s is shown by placing the transition label $\mathit{trg}[\mathit{grd}]/\mathit{eff}$ in a box below the entry and exit actions of s. For example, in the point state machine there is an internal transition in the substate left of working with trigger `<ic> move right`.

If, for a transition t, $\mathit{source}(t)$ is an initial pseudo-state then $\mathit{target}(t)$ is called the *default entry state* of its enclosing region. If $\mathit{source}(t)$ is also directly contained in a root region, then t is called an *initial transition*. We require that initial transitions are unguarded.

A class inherits all properties, association ends, and so on, from its superclasses. In particular, a class also inherits state machines. So even though we only allow one state machine per class, when a class C is instantiated, the resulting object has all the state machines of C and its superclasses.

3.2 Expressions and actions

We now describe the expression language and the action language for our models. These languages correspond to certain subsets of the SIML-language from [29].

Boolean expressions occur as transition guards, as change conditions, and as definitions of derived attributes. Apart from the usual Boolean connectives,

[5] The version of Artisan Studio used to create this model does not properly support UML stereotypes.

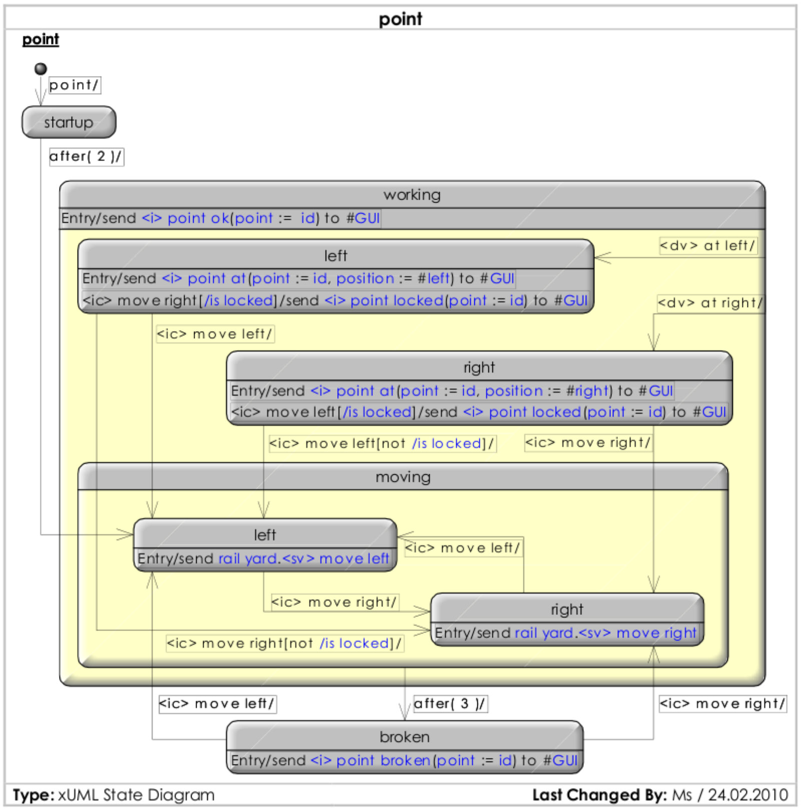

point
point
point/
startup
after(2)/
working
Entry/send <i> point ok(point := id) to #GUI
left
Entry/send <i> point at(point := id, position := #left) to #GUI
<ic> move right[/is locked]/send <i> point locked(point := id) to #GUI
<dv> at left/
<dv> at right/
<ic> move left/
right
Entry/send <i> point at(point := id, position := #right) to #GUI
<ic> move left[/is locked]/send <i> point locked(point := id) to #GUI
<ic> move left[not /is locked]/
<ic> move right/
moving
left
Entry/send rail yard.<sv> move left
<ic> move left/
<ic> move right/
right
Entry/send rail yard.<sv> move right
<ic> move right[not /is locked]/
<ic> move left/
after(3)/
<ic> move right/
broken
Entry/send <i> point broken(point := id) to #GUI
Type: xUML State Diagram
Last Changed By: Ms / 24.02.2010

Fig. 3: State machine diagram of point in Micro interlocking

they may include quantification over linked objects thereby referring to non-local state. Formally, Boolean expressions are generated by the following grammar.

$$\begin{aligned} \mathit{Bool} &::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ \mathit{Bool} \mid \mathit{Bool}\ \texttt{and}\ \mathit{Bool} \mid \mathit{Bool}\ \texttt{or}\ \mathit{Bool} \\ &\mid \texttt{in\_state(\#(}\mathit{State}\texttt{))} \mid \mathit{DAttr} \\ &\mid \texttt{forall}\ \mathit{ObjExpr}\ \texttt{is\_true}\ \mathit{Bool} \mid \texttt{exists}\ \mathit{ObjExpr}\ \texttt{is\_true}\ \mathit{Bool} \\ \mathit{ObjExpr} &::= \mathit{AssocEnd} \mid \mathit{AssocEnd}\ \texttt{UNION}\ \mathit{AssocEnd} \end{aligned}$$

where *DAttr* and *AssocEnd* are identifiers of a derived attribute and an association end, respectively, and *State* is a sequence of dot-separated state or region names that describes a state of a given state machine. For example, the substate left of the state moving in the point state machine (see Figure 3) is referred to by the expression `working.moving.left`. Expressions of type *ObjExpr* denote sets of objects. Such sets can be specified either as the collections of objects that can be referenced via association ends, or as a union of such collections. The `forall` ... and `exists` ... denote quantification over sets of objects.

Examples of Boolean expressions in Micro interlocking are:

in point: `in_state(#working.left)`
in point: `exists (L_routes UNION R_routes) is_true /established`
in route: `forall left_points is_true /at_left`

The are two types of *basic actions*: assignments and sending signals. An object can send signals to itself, to linked objects, or to some environment interface (referred to as `GUI`) and they may carry a list of parameters. All actions can be composed sequentially. Formally, the action language is generated by the following grammar.

$$
\begin{array}{rl}
\mathit{Act} ::= & \texttt{send}\ \mathit{Signal} \\
\mid & \texttt{send}\ \mathit{Signal}\ \texttt{to \#GUI} \\
\mid & \texttt{send}\ \mathit{ObjExpr}.\mathit{Signal} \\
\mid & \mathit{Attr}\ \texttt{:=}\ \mathit{Val} \\
\mid & \mathit{DAttr}\ \texttt{:=}\ \mathit{Expr} \\
\mid & \mathit{Act};\mathit{Act}
\end{array}
\qquad
\begin{array}{rl}
\mathit{Signal} ::= & \mathit{SignalName} \\
\mid & \mathit{SignalName}(\mathit{Params}) \\
 & \\
\mathit{Params} ::= & \mathit{ParamName}\ \texttt{:=}\ \mathit{ParamVal} \\
\mid & \mathit{Params}, \mathit{Params}
\end{array}
$$

where *ObjExpr* is defined as above, *Attr* is the name of an attribute, *DAttr* is the name of a derived attribute, *ParamName* is an identifier and *ParamVal* is an expression that is evaluated at the time the action takes place. An action `send` a means that the object is sending the signal a to itself, and `send` $Obs.a$ means that the signal a is sent to all objects in the set denoted by Obs.

3.3 UML semantics

An xUML model defines a generic model of communicating objects. A model instance is obtained by instantiating classes and associations For example, an instance of the Micro interlocking is defined based on a particular track layout which specifies a number of tracks, points, signals and routes and how they are linked (i.e. how the associations are instantiated).

UML semantics defines the operational behaviour of objects, that is, how objects react to events, and how they communicate with each other. Some of these aspects are defined by the UML specification [36], but for others the UML specification allows a choice between different interpretations in order to allow for flexible modeling. We give a brief, informal description of the semantics that we follow. We refer to [36, 15.3.12] for more details.

An *object of type* C is an instance of a class C together with an event pool. Due to inheritance, an object can have several state machines. These can be viewed as the concurrently executing regions of a single state machine. In the rest of this section, we let O denote an object of type C.

The set of states in which the object state machine currently resides is called the *active state configuration.* This is a set of states rather than a single state, due to the presence of concurrent regions. A transition is *enabled* if its trigger is available, its source state is active and its guard evaluates to true. When a transition fires, its source state is exited (possibly triggering exit actions), then the effect actions are carried out, and finally its target state is entered, which again can trigger entry actions. It is possible that several transitions are enabled at the same time. In this case, a maximal set of consistent, enabled transitions is fired. Informally stated, two transitions are consistent if executing one does not disable the other by exiting its source state. This is, in particular, the case if the transitions are contained in disjoint regions of the state machine.

The UML specifies a *run-to-completion (RTC) semantics* which means that an object must finish all the behaviour triggered by an event, before the next event can be processed. While an object is processing an event its state is considered undefined, hence other objects are not allowed to inspect its state at this point. The behaviour imposed by the RTC semantics can be described by the following *object execution cycle:*

O1: Let other objects read the currently active state configuration A, or choose an available event e for processing by the state machines of O. This marks the beginning of a run-to-completion (RTC) step in O.

O2: Let T be a maximal, consistent set of enabled transitions in O with trigger e. If there are several such Ts, then one is chosen nondeterministically. Fire all transitions in (the possibly empty) T (in arbitrary order). The RTC step is now completed. Go to step O1.

Note that in agreement with UML 2.2 semantics (see *Run-to-completion and concurrency* in [36, 15.3.12]) the RTC step applies simultaneously to all regions of the state machine. But the RTC steps of different objects may be interleaved.

We now define when events are available for processing: A *signal event* denotes the moment when a signal is sent. A signal event is available if it is in the event pool. A *change event* when(cond) is available whenever the change condition cond is true. This seems to conflict with [36, Sec.13.3.7] which says that a change event occurs when the change condition *becomes* true. But UML 2.2 does not specify when change events are detected, or whether they remain after the change condition becomes false again. In our interpretation change events are thus detected immediately, but they do not remain. A *time event* after(n), which triggers a transition t, is available whenever $source(t)$ is active. This semantics is based on the assumption that all transitions and actions take place in zero time.

The UML specification allows for different priority schemes when choosing an event for processing by an object, see [36, Sec.15.3.12]. We apply the following priority scheme: signals from the object to itself have priority over all others; signals from the environment, time events and change events have priority over signals coming from other objects; signals coming from other objects are processed on a FIFO basis (which we realise by implementing the event pool as a queue). Note that the semantics does not impose any fairness constraints: an available event is not guaranteed to be processed. In terms of transitions, it means that a transition may be enabled but never taken.

Objects communicate by sending signals to each other. We assume that signals are never lost or duplicated. The communication is one-to-one (i.e. no broadcast) and asynchronous (since signal events are stored in event pools).

4 The iUML representation

The iUML representation can be described as an intermediate step between UML and a labelled transition system (LTS) representation of object behaviour. The LTS states are active state configurations and labelled transitions are defined according to the semantics described in Section 3.3. Furthermore, we use the iUML to represent model instances.

4.1 Transitions in iUML

In order to associate a unique action sequence with iUML transitions it may be necessary to refine UML transitions into several iUML transitions. The reason is that a UML transition t can have a composite source state, hence the (exit) actions that should be executed when firing t may depend on which nested substates of *source*(t) are active. We illustrate the refinement of such a transition with the following simple example. Consider the transition t from A0 to B0 in Figure 4. (We have only drawn the elements relevant for the refinement of this t.) Assume that A0 is active, the trigger of t is available, and its guard is true.

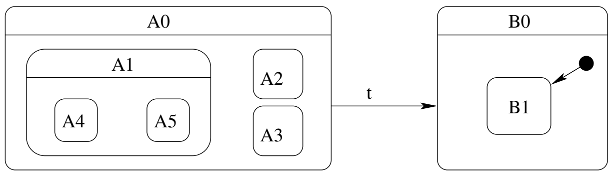


Fig. 4: UML transition with composite source state.

In order to determine which actions to carry out when firing t, we need to also inspect which of the states A1, A2, A3, A4 and A5 are active and whether they have exit actions defined. Suppose that A0, A4 have exit actions a_0, a_4, respectively. We then have two possible exit action sequences when t fires: if A4 is active, the exit action sequence is $a_4; a_0$ and if A4 is not active, it is a_0. This motivates the following definitions.

– An *active state predicate* is a pair (U, V) where $U, V \subseteq \mathit{States}(\mathsf{C})$. An active state configuration A *satisfies* (U, V) if $U \subseteq A$ and $V \cap A = \emptyset$. In other words, A satisfies (U, V) if all states in U are active, and no states in V are active. We denote by $[\![(U, V)]\!]$ the set of active state configurations of O that satisfies (U, V).

– An *iUML transition* consists of an active state predicate (instead of a source state), a target state, a trigger, a guard, an exit sequence, an effect sequence and an entry sequence. An iUML transition t is *enabled* if the currently active state configuration satisfies the active state predicate of t, the trigger is available and the guard is true.

Due to the nesting of A4 inside the composite state A1, the transition t from Figure 4 is refined into three iUML transitions t_1, t_2 and t_3 with the following active state predicates and exit sequences:

flat	active state predicate	exit sequence
t_1	$(\{\text{A0,A1,A4}\}, \emptyset)$	$a_4; a_0$
t_2	$(\{\text{A0,A1}\}, \{A4\})$	a_0
t_3	$(\{\text{A0}\}, \{\text{A1}\})$	a_0

Note that in iUML transitions, we keep as much of the high-level representation of the UML as we can. For example, in the above we have one iUML transition t_3 rather than two (where one requires A2 to be active, and the other one requires A3 to be active). This is facilitated by the use of active state predicates instead of source states. Note that the entry sequence of a UML transition does not depend on the currently active state configuration. For example, if in Figure 4 B0, B1 have entry actions b_0, b_1, respectively, then the entry sequence is $b_0; b_1$ for all t_1, t_2, t_3.

4.2 Transition selection

Recall from the object execution cycle (Section 3.3) that upon receiving an event e, an object must select a maximal, consistent set of enabled transitions for execution. In the iUML, transitions are grouped together in a way that reflects the transition selection algorithm. We first group an object's transitions by trigger and active state predicate. For each such transition group T, the consistent subsets of T that can fire depend also on the values of transition guards. These subsets are called multi-transitions. In the iUML representation, each object will contain a collection of transition groups, and each transition group will contain its multi-transitions.

We now give a formal definition. Let O be an object of type C. For an event e, we let $Tr(O, e)$ be the set of all transitions in O with trigger e. A *transition group with active state predicate* (U, V) *and trigger* e *(in object* O*)* is a non-empty, maximal subset T of $Tr(O, e)$ such that for all $t \in T$, $[\![(U, V)]\!] \subseteq [\![(U_t, V_t)]\!]$ where (U_t, V_t) is the active state predicate of t. Note that if e is a time event or a change event, then $Tr(O, e)$ is a singleton and equal to the only transition group with trigger e. Transition groups that contain several consistent transitions can result from transitions in concurrent regions.

For example, an object of type `point` has five transitions triggered by the signal `<ic> move left` (see Figure 3). These transitions will result in four transition groups. One group will contain the two transitions that both have source state working.right; the other three groups will be singletons. Note that the non-singleton transition group is not consistent (since firing one will disable the other by exiting working.right). However, these transitions will never be enabled at the same time, since their guards are complementary.

The subsets of a transition group that can actually fire depends on the consistency of transitions and the value of transition guards. We formalise these subsets as multi-transitions. A *multi-transition* of a transition group T is a consistent subset M of T such that $M = G \cup \mathit{ungrd}(T)$ where $G \subseteq \mathit{grd}(T)$, $\mathit{grd}(T)$ is the set of all guarded $t \in T$, and $\mathit{ungrd}(T) := T \setminus \mathit{grd}(T)$.

For example, if $T = \{t_0, t_1, t_2\}$ is a transition group in which all transitions are pairwise consistent, and $\mathit{grd}(T) = \{t_1, t_2\}$ where g_i is the guard of t_i for $i = 1, 2$, then the multi-transitions of T are $\{t_0, t_1, t_2\}$, $\{t_0, t_1\}$, $\{t_0, t_2\}$ and $\{t_0\}$ which correspond to the scenarios where g_1 and g_2 are both true, only g_1 is true, only g_2 is true, or g_1, g_2 were both false.

5 Translation from iUML to mCRL2

We sketch our translation from iUML into mCRL2. The mCRL2 specification language [21] extends the process algebra ACP [5] with facilities to specify abstract datatypes (ADTs), and includes built-in types such as Booleans, integers, and lists. The advantages of translating into mCRL2 are that the language has a formal semantics [21], and that it comes with powerful verification technology. The possibility of defining data types and using them in the behavioral specification is essential for the translation presented in this section. Process algebras without ADT support such as CSP [24] supported by the FDR2 toolset [15] are thus not directly applicable, while process algebras with ADT support such as E-LOTOS [26] and LOTOS-NT [40] supported by the CADP toolset [16] provide alternatives for mCRL2 in our setting. We refer to [21] and the website `http://www.mcrl2.org` for details regarding the syntax and operational semantics of mCRL2.

Our translation takes as input an iUML model instance $\mathcal{M}$. $\mathcal{M}$ describes a collection of objects that interact by sending signals to each other, but also by inspecting each others state. We translate each object into an mCRL2 process; the translation of the entire model $\mathcal{M}$ will then, roughly, be the parallel composition of the mCRL2 processes associated with the objects in $\mathcal{M}$, and the interaction between the objects is implemented by means of mCRL2's facility for (synchronous) communication between components.

We proceed to discuss the translation of the objects in $\mathcal{M}$ as *object processes*. Recall that each object has associated with it an event pool, and a collection of state machines describing its behaviour. An object process will consist of the parallel composition of a buffer process modelling the event pool, and a state machine process modelling the actual behaviour of the object. Below we will discuss the translation of the event pool and object behaviour in more detail, illustrating the translation on the object p1 of type point (see Fig. 3).

Event pool. Signals are specified using mCRL2's facility for defining abstract data types. The signals that are to be stored in the event pool of an object of type `C` are represented by an enumerated data type `C_Messages`. That is, `C_Messages` has a member for each reception in the class `C`. The buffer process of an object of type `C` receives and stores the signals that are sent to the object by implementing a queue of `C_Messages`.

Example 2. The specification of the enumerated type `point_Messages` is:

```
point_Messages = struct
   ic_move_right_point | ic_move_left_point |
   dv_at_right_point| dv_at_left_point ;
```

The event pool of the object `p1` is specified by the following mCRL2 process specification.

```
proc point_Buffer_p1(message_buffer: List(point_Messages))=
  % Messages received from other components
  (#message_buffer < max_buffer_size) ->
    sum m: point_Messages. sum sender : Identifiers.
      receive_from_system(sender, p1, m).
        point_Buffer_p1(message_buffer <| m)
+ % Messages sent to component
  (#message_buffer > 0) ->
    send_to_component(p1, head(message_buffer)).
      point_Buffer_p1(tail(message_buffer));
```

The specification above expresses that the process `point_Buffer_p1` consists of a choice (denoted by `+`) between the following options: either (if its buffer is not full) it receives some element `m` of sort `point_Messages` from some `sender` and then appends `m` to its `message_buffer`, or (if its buffer is not empty) it sends the first element in `message_buffer` to the associated state machine, removing this element from the `message_buffer`.

States, messages and buffers. The state machine process implements the concurrent composition of the state machines of the class `C`, and the process carries data parameters that encode the active state configuration by letting each such state parameter range over an enumerated data type that represents the states contained in a region r. More precisely, we declare for each state machine X in `C` and each region r in X, an enumerated data type `X__r_States` whose members represent the set $\mathit{States}(r)$. If r is not a root region, then `X__r_States` will also have a member (ending with `_nop`) that will be used to indicate that no state in r is currently active.

Example 3. The regions in the Point state machine shown in Figure 3 give rise to enumerated data types: `point_States`, `point__working_States`, and `point__working_moving_States`. Below we show the definitions of the first two:

```
point_States = struct                point__working_States = struct
    point__broken_substate               point__working_right_substate
  | point__startup_substate            | point__working_left_substate
  | point__working_substate ;          | point__working_moving_substate
                                       | point__working_nop ;
```

Recall from Figure 2 that Point is a specialisation of HAL device; the active state configurations of the latter are represented by the data type `HAL_device_States`. The behaviour defined by the state machines of the object `p1` is then expressed by a process definition of the form

```
proc point_p1(
  HAL_device_state : HAL_device_States,
  point_state : point_States,
  point__working_state : point__working_States,
  point__working_moving_state : point__working_moving_States
) = ...
```

We postpone the discussion of how the state machines of HAL device and Point give rise to the right-hand side of the above defining equation for `point_p1`. First, we explain how mCRL2's features of parallel composition, communication and blocking are used to combine `point_Buffer_p1` and `point_p1` specifying the behaviour of the object `p1`. The object `p1` is represented by the following expression:

```
proc point_Complex_p1
=  block({send_to_component, receive_from_buffer},
   comm({send_to_component|receive_from_buffer -> message_to_component},
      point_Buffer_p1([])
   || point_p1(HAL_device__normal_substate, point__startup_substate,
               point__working_nop, point__working_moving_nop)));
```

The process `point_Complex_p1` is defined as a parallel composition of instances of the processes `point_Buffer_p1` and `point_p1`. Initially, the active states are startup (in Point) and normal (in HAL device), and the buffer is empty; this explains the respective parameter values passed to `point_Buffer_p1` and `point_p1`. The operation `comm` expresses a communication between `point_p1` and `point_Buffer_p1`: both components may synchronise by simultaneously executing the action `receive_from_buffer` and `send_to_component`; the resulting event is denoted by `message_to_component`. The operation `block` declares, in fact, that the actions `send_to_component` and `receive_from_buffer` may not be executed in isolation; they may only occur as part of the aforementioned synchronisation.

State machine process. We proceed to explain how the behaviour of an object as expressed by a state machine is translated into mCRL2. The state machine process implements the object execution cycle (see page 9). Starting from a "stable state", which corresponds to *O1* in the object execution cycle, the process can either receive and process an event, or let other processes inspect its state. This inspection of states is implemented as a communication of state parameter values: a consumer action takes place in the object process that needs the data; the matching producer action takes place in the object process whose current state must be known to the consumer.

The possible behaviours in state *O1* (process event or send state data) are modelled as a nondeterministic choice (sum) over the different alternatives. As an example, part of the state machine process for the object p1 is sketched below. In the first two summands, a message is received from the buffer. In the next two summands, a message is received directly from the environment. The following two summands represent the two time event transitions in the Point state machine, one has source state startup and the other has source state working.moving. The last four summands are actions that produce data for the evaluation of change conditions in two Route objects r1 and r2. They show that in r1, there are two change conditions that require the expression in_state(#(working.right)) to be true in p1.

```
proc point_p1(<state params>)
  = receive_from_buffer(p1,ic_move_left_point).  ...
  + receive_from_buffer(p1,ic_move_right_point). ...
  + receive(p1,dv_at_left_point). ...
  + receive(p1,dv_at_right_point). ...
  + (point_state == point__startup_substate) ->
       tick(p1). ...
  + (point__working_state == point__working_moving_substate) ->
       tick(p1). ...
  + when_data_r1_1_p1_producer(p1,
       point__working_state == point__working_right_substate). ...
  + when_data_r1_2_p1_producer(p1,
       point__working_state == point__working_right_substate). ...
  + when_data_r2_1_p1_producer(p1,
       point__working_state == point__working_left_substate). ...
  + when_data_r2_2_p1_producer(p1,
       point__working_state == point__working_left_substate). ...
```

Recall that in our semantics, time event transitions can fire whenever the source state is active. The detection of a time event is specified by the action *tick*. The *tick* action does not actually place an event in the buffer. Change events are specified in a manner similar to time events which we describe towards the end of this section.

The part of the state machine process that follows an action that models the choice of an event for processing specifies the transition selection algorithm and the execution of the selected transitions. Recall now that in the iUML representation, we have a representation of the transition selection algorithm given by transition groups and multi-transitions. The mCRL2 specification that implements the transition selection algorithm consists of a nesting of conditional statements ranging over transition groups. For each transition group it is checked whether its active state predicate is satisfied by the currently active state configuration. If this check fails for all transition groups, then the process continues recursively with its state parameters unchanged. Otherwise, there is a transition group T for which the active state predicate is satisfied by the currently active state configuration. If the guards of transitions in T refer to the state of other objects, then the process carries out a number of consumer actions to retrieve this state information. Next, a conditional statement runs through the multi-transitions of T ordered decreasingly by size, until it finds a multi-transition M whose transition guards are all true, and then the action sequence associated with M is executed, and the state machine process continues recursively with its state parameters updated to reflect the state after firing the transitions in M.

The mCRL2 code for the transition group in p1 with trigger move_left and active state predicate ({working.right}, {working.left,working.moving.right,broken}) is shown in Figure 5. The transition group consists of two transitions t_{left} and t_{right} with guards not /is_locked and /is_locked, respectively. Such pairs of transitions with complementary guards result in two singleton multi-transitions. The derived attribute /is_locked in p1 refers to the state of route objects r1 and r2 which explains the communication with r1 and r2 in lines 2-4. These consumer actions are matched by producer actions in r1 and r2. In line 3, the active state predicate is checked. In line 7 the transition guard not /is_locked is evaluated using the received data values. Line 8 contains the action that results from firing the multi-transition $\{t_{\mathrm{left}}\}$. Lines 9-13 specify the updated state of the process after firing $\{t_{\mathrm{left}}\}$. If not /is_locked evaluates to false in line 7, then /is_locked must evaluate to true, and so the multi-transition $\{t_{\mathrm{right}}\}$ is fired, which is specified in lines 14-20.

```
 1  point_p1(...) =
 2   ...
 3   (point__working_state == point__working_right_substate) ->
 4    (sum r1_var: Bool. sum r2_var: Bool.
 5     condition_data_p1_1_consumer(r1,r1_var)
 6       | condition_data_p1_1_consumer(r2,r2_var).
 7     (!(r2_var || r1_var)) ->
 8      send_to_rail_yard(p1,sv_move_left_point_railyard).
 9      point_p1(
10          HAL_device_state,
11          point_state,
12          point__working_moving_substate,
13          point__working_moving_left_substate)
14     <>
15       send_to_environment(p1,i_point_locked_point_environment).
16       point_p1(
17          HAL_device_state,
18          point_state,
19          point__working_state,
20          point__working_moving_state)
21     )
```

Fig. 5: Example mCRL2 code for a transition group

A transition t_c triggered by a change event c can fire whenever the change condition is true, and $source(t_c)$ is active. If the change condition refers to data in other objects, then a sequence of consumer actions are executed in order to obtain the data, similarly to how transition guards are evaluated.

Difference with earlier translation. In our earlier work [22], we presented a slightly different translation from xUML to mCRL2. This translation was not formulated in terms of an intermediate format, as it was done by hand. There are, however, also semantic differences between the two translations:

- In the translation from [22], change events are detected by an additional monitor component of object processes, and when a change condition becomes true, the monitor places a message in the buffer. In particular, change events remain in the buffer even after the condition becomes false.
- In the translation from [22], change events, object-internal signals, and system-internal signals have equal priority (all go through the FIFO-buffer), and all these events take priority over external signals. In the current translation, object-internal signals have priority over all others, and change events, time events and external events are allowed to overtake system-internal signals.

Based on discussions with the UML modelling engineers, our current translation is more in line with their view on UML semantics (which is based on the CASSANDRA simulator [29]) than our previous translation. As a further advantage, we have found that the mCRL2 models resulting from our current translation are dealt with more easily by our model checking tools.

6 Verification

Our approach to verifying safety properties of xUML models is based on expressing the safety properties as UML state machines that observe the system state and send an error signal in case a violation is found. The model and its safety properties are both translated into mCRL2, as described in the previous section, and safety violations are detected by using the facility of our model checking tools that allow searching for a particular action, in this case, the *"send error signal"*-action.

Our motivation for expressing safety properties as state machines is two-fold. First, it allows UML modelling engineers to specify safety requirements without having to learn temporal logic. Second, the mCRL2 tools that provide (explicit-state) model checking of modal mu-calculus formulas were not able to deal with the sizes of our models. By turning the verification problem into a reachability problem, we were able to verify our models using the symbolic reachability tool from the LTSmin tool set [8]. This symbolic tool allows for varying exploration strategies, and reports some basic performance analytics.

In Section 6.1 we describe how safety properties are modelled as UML state machines. The verification proper is discussed in Sections 6.2 and 6.3, which investigate, respectively, the size of the models we are able to deal with, and ways to 'attack' larger models.

6.1 Safety properties as observer classes

In an internal document describing the Micro interlocking the following two safety requirements are given:

MS1: "A point that is locked by an established route shall never move."
MS2: "The entry signal of a route shall never display proceed when one of its tracks is not free."

The exact meaning of the railway signaling concepts mentioned in MS1 and MS2 is not so important for the present discussion, but one should think of a route being established as a requirement for letting a train pass (safely) over the route. What is relevant is that properties such as a point being locked, a route being established, and hence MS1 and MS2 themselves, can be expressed in terms of the system's state, that is, without reference to the ordering of events. We will use the term *state property* to refer to such safety properties.

Our approach to verifying state properties is based on the observation that state property violations can be detected in the system itself as certain change events. In order to detect such violations we define a collection of *observer classes* whose state machines will detect safety violations and send error signals. These observer classes are expressed in the same subset of xUML as the model we wish to verify, and we can therefore apply our automated translation to generate an mCRL2 specification of the "observed model".

Observer classes. The common structure of observer classes is modelled by the class StateObserver. The StateObserver class has three attributes: id, ObservedObject and ObservedClass, one derived attribute /triggered (which will be defined by a Boolean expression), and a state machine with a single transition with trigger when(/triggered) and effect `send <i>_violation(observer := id) to #GUI`. The purpose of the attribute ObservedClass is to define the context in which /triggered is evaluated. The attribute ObservedObject must be the name of an instance of ObservedClass. All attribute values are defined upon instantiation.

Specific state properties are modelled as specialisations of the class StateObserver. These specialisations are what we call *observer classes.* The state machine of an observer class consists of just one state and an initial transition to it. The definition of /triggered, and of any additional derived attributes that may aid the definition of /triggered, are assigned as the effect of the initial transition. We illustrate using the two examples MS1 and MS2 from above. It should be clear that MS1 and MS2 are both state properties.

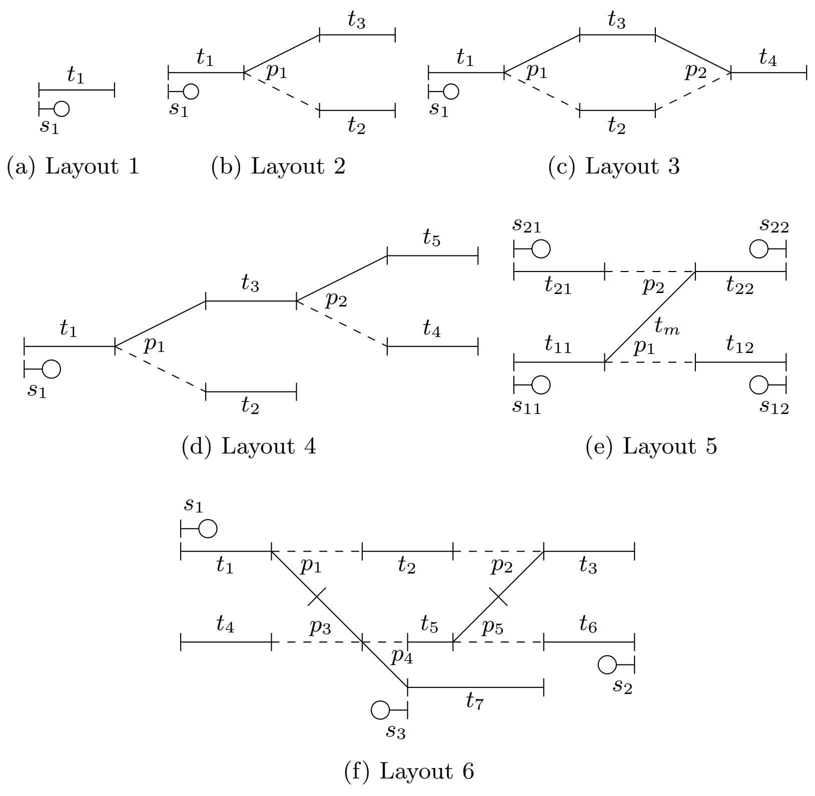
The property MS1 will be modelled as an observer for the class Point, that is, the attribute ObservedClass has value "Point". Hence, the definition of /triggered may use derived attributes from the Point class such as /is_locked which is defined as `/is_locked` := `exists (L_routes UNION R_routes) is_true /is_established`. In other words, /is_locked is true if the point is locked by an established route, and we define `/triggered := /is_locked and in_state(#working.moving)`.

The property MS2 is modelled as an observer for the class Route, and it defines two "auxiliary" derived attributes `/all_tracks_free` and `/proceed`. The initial transition of its state machine has effect:

```
/all_tracks_free := forall tracks is_true /is_free;
/proceed := entry_signal.in_state(#proceed);
/triggered := /is_established and /proceed and not /all_tracks_free;
```

6.2 Feasibility of verification

Given instances of the translated UML models, we would first of all like to know the size of models we can deal with. To this end we designed several track layouts for the Micro interlocking which are of increasing complexity. The layouts are presented in Figure 6 and correspond to simple configurations one might find in rail yards. Although not depicted in the figure, the possible routes in a layout are precisely all the maximal paths in starting from a signal and not passing both the left and right branch of a point (e.g., Layout 5 has six routes).



(a) Layout 1 (b) Layout 2 (c) Layout 3

(d) Layout 4 (e) Layout 5

(f) Layout 6

Fig. 6: Several track layouts used to test the feasibility of the verification task.

Using the mCRL2 and LTSmin model checking tool sets (see Section 2) we can next generate the state spaces for the depicted track layouts. The measurements as obtained with the BDD based symbolic model checker from the LTSmin tool set are depicted in Table 1. As can be seen in the table, model checking our simple track layouts is possible, but running times and memory consumption increase fast when introducing more routes (compare Layouts 4 and 5, where 4 has three routes and 5 has six).

Table 1: State spaces of the layouts from Figure 6, without observers, where each state machine is assumed to have at most one message in its event pool. Resource consumption is for the LTSmin symbolic model checker run on an Intel Xeon X5550 machine with 148 GB of internal memory; a saturation-like [9] exploration strategy was used. The running times exclude the time to load the model.

Layout	Components	Routes	States	Runtime (s)	Memory (MB)
1	2	1	1.7×10^{4}	0.01	61
2	5	2	1.3×10^{9}	0.25	76
3	7	2	4.9×10^{11}	7.73	86
4	8	3	8.9×10^{13}	19.39	115
5	11	6	6.8×10^{23}	2605.90	3133
6	15	8	$> 7.0\times10^{30}$	> 496 h	> 30 GB

Given the resources consumed already by Layout 5, we can conclude that it is impossible to generate complete states spaces for ‘realistic’ layouts, which usually consists of hundreds of components and routes. Hence, more advanced methods are required, which are discussed in the next section. Of course, if certain properties are violated, this might already be detectable when instantiating the UML models for the small layouts.

Remark 1. The figures in Table 1 make it clear that explicit state space generation is infeasible. Using the distributed model checker which is also part of the LTSmin tool set, already Layout 2 is too large to be dealt with on a cluster of 10 Intel Xeon E5335 machines each with 24 GB of internal memory.

Given the simplicity of track layouts and the state machines shown, it may come as a surprise that the state spaces are so huge. Note, however, that not all states are depicted in the state machines: First, event pools are not included. Second, as a sequence of actions that is carried out when a transition fires is not executed as an atomic block, additional ‘intermediate’ states exist between actions in the sequence. Third, the communication needed to exchange state parameters also introduces some additional states.

Remark 2. Bounded model checking (BMC) [6] is not a suitable model checking technique in the current context. The technique cannot prove the absence of errors, which is precisely what we are interested in, given the safety critical nature of interlockings.

6.3 Speeding and scaling up verification

Given our interest in proving the presence or absence of certain actions (see Section 6.1), it might be possible to speed up and scale up the verification. From the literature at least two symbolic model checking techniques are known that may help to achieve this:

– Compositional exploration of state spaces [33, 4]: The transition relation is split into several parts and only some of these are used in state space exploration. Selection of the employed parts is based on an analysis of the interaction between the parts and on the particular property one is interested in. Additional parts are selected in case the property could be neither proved nor disproved using the selected subset of the transition relation.
– Counterexample-guided abstraction refinement (CEGAR) [11]: Given a property, the transition relation is over-approximated (i.e., a relation is used of which the transition relation is a sub-relation). Next, refinement takes place based on violations of the property. Eventually either a violation is found that also holds given the transition relation (i.e., the non-over-approximated one) or an over-approximation is reached in which the property holds (implying it also holds given the original transition relation).

In [33, 4, 11] it is reported that with both techniques, speed up and scalability are achieved in case only part of the transition relation is needed to show that the considered property holds. Moreover, in case of CEGAR speed up and scalability are also achieved as the symbolic representation of the over-approximated state space is often smaller than the symbolic representation of the real state space.

Preliminary results. Thus far we have extended the symbolic model checker from the LTSmin tool set with the first of the aforementioned techniques and we are currently working on implementing the second technique.

Some preliminary results obtained with the implementation of the first technique mentioned above are shown in Table 2. The safety properties MS1 and MS2 (see Section 6.1) have been verified for Layout 2 from Figure 6. Both properties are violated by the Micro interlocking specification and, hence, an error trace (of a certain length) can be generated.

Table 2: Running times and trace lengths when searching for error actions using the LTSmin symbolic model checker run on an Intel Core 2 Duo machine with 4 GB of internal memory; a saturation-like [9] exploration strategy was used. The running times include the time to load the model and generate a trace.

Layout	Property	Runtime (s)		Trace Length	
		Default	Compositional	Default	Compositional
2	MS1	6.33	6.42	35	22
	MS2	7.33	6.78	41	41

It is impossible to draw any definite conclusions given the small sample. Nevertheless, we note that some speed up is obtained in the case of MS2. Furthermore, the reported lengths of the error traces are encouraging: the length of the error trace is substantially shorter in the case of MS1, and it is not longer in the case of MS2.

Remark 3. We do not automatically obtain the shortest trace possible, as we use a saturation-like strategy instead of breadth-first search. Using breadth-first increases running times for the layout 2 to 7.05 and 8.98 seconds for MS1 and MS2, respectively. For layout 2, the shortest error traces for MS1 and MS2 consist, respectively, of 22 and 28 steps.

7 Discussion and conclusion

We have presented a fully automated, translation-based approach to the verification of safety properties in xUML models. Since both the input and the output of our tool chain are expressed in UML, our verification technology can be used by engineers without a thorough background in process algebra, model checking, or modal mu-calculus.

Additional case studies. Our translation from xUML to mCRL2 has been further applied in two case studies: (i) a UML model of a controller for mixing hot and cold water, obtained from one of the industrial partners of the University of Twente; (ii) a UML model of the session setup protocol from the ISO/IEEE 11073-20601 [27] standard (a data exchange standard for the health care industry).

With regard to item (i), we were able to identify certain property violations. However, this required the use of temporal logic, as the specified properties were liveness properties (which we currently cannot capture using our observer state machine approach). Moreover, since the controller for mixing hot and cold water is intended to be implemented on a Programmable Logic Controller (PLC), which does not buffer incoming events, we modified our translation in this respect.

The second case study (ii) was carried out by a colleague from the Eindhoven University of Technology [28]. After resolving some ambiguity issues in the state machines provided, the UML model could be translated into mCRL2. The verification revealed that it is possible for the system setup to reach an unsafe state (where the communicating devices operate with different measurement units). However, it was also shown that no unsafe operational behaviour could occur, due to detection of the unsafe state before the first data exchange.

Related work. Formalisation of xUML models for the purpose of verification is a widely studied topic, and we mention just a few [2, 3, 12, 20, 34, 42, 44]. We briefly relate our approach to some of the aforementioned ones. Recall that we currently support signal, time and change events, but not call events.

In the UMC framework of [3], xUML models may be specified in the UMC specification language. UMC supports signal and call events, but no time or change events, nor exit/entry actions of states. Properties must be specified in a CTL-like logic, and are verified using on-the-fly model checking. Specifying properties of UMC models thus requires some knowledge of formal methods (and not just of UML), but on the other hand, the types of properties that can be expressed far exceeds what our observers can define. In [20], xUML models (with change, call and send events) are translated into a temporal logic that supports compositional specification, and verification takes the form of refinement proofs. Our approach is based on model checking which has the advantage that the verification process is fully automatic whereas theorem proving generally requires human interaction in order to find proofs. In [34], UML state machines are translated into timed automata using, as we do, model transformation technology. The resulting timed automaton is verified using the UPPAAL model checker. It is, however, not clear how the timing constraints of the timed automaton are derived from the input xUML model, in particular, since only signal events are allowed as transition triggers (see Rule 10 in [34]).

Formal verification of high-level interlocking specifications has been studied in [13, 43]. For verification of concrete interlocking systems, see e.g. [10, 14, 19]. One major source of challenges in the verification of xUML models lies in the asynchronous communication model. It has been observed earlier that synchronous specifications may be more amenable to automatic and exhaustive verification; see, e.g., [39], for a short experience report in the domain of railway interlockings.

Our approach to modelling safety properties of xUML models in xUML itself seems to be new, but similar ideas are found in [7, 17, 18, 32, 38]. The main difference with our work is that our observers monitor state changes whereas the above-mentioned observers monitor event sequences, and hence they are able to express temporal properties. The expression of liveness properties in our approach is still very much an open issue.

Future work. In the future, we intend to extend the subset of xUML covered in our translation. In particular, we would like to include synchronous calls, which are used in the more elaborate xUML models of railways interlockings that we have been presented with. We would also like to extend our approach to specifying safety properties in UML to include specification of temporal properties.

Moreover, we would like to give a formal operational semantics of our xUML models, so that we can make formal statements about the generic properties of models, as well as a formal comparison of different alternative approaches to the xUML semantics. In order to enhance the scalability of our verification techniques, we will continue our efforts to adopt compositional techniques as well as counterexample-guided abstraction refinement.

Acknowledgements The authors thank Louis Rose for his helpful comments on an earlier draft of this paper.

Funding This research is partially funded by the European Comission (EC), as a grant to the FP7 project INESS, grant agreement no. 218575. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of either the EC or the INESS consortium.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):273–303, 2001.
3. M. H. t. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119 – 135, 2011.
4. G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. *Formal Methods in System Design*, 21(2):225–244, 2002.
5. J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
6. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
7. J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In *Proc. Formal Approaches to Software Testing (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 125–139, 2005.
8. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *Computer Aided Verification, 22nd Int. Conf., CAV 2010, Edinburgh, UK, July 15-19, 2010.*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.
9. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.
10. A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, 10(4):361–380, 1998.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
12. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 55:81–155, 2005.
13. L.-H. Eriksson. Specifying railway interlocking requirements for practical use. In *Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP'96).* Springer, 1996.
14. W. Fokkink. Safety criteria for the vital processor interlocking at Hoorn-Kersenboogerd. In *5th Conference on Computers in Railways (COMPRAIL'96). Volume I: Railway Systems and Management*, 1996.

15. Formal Systems (Europe) Ltd. Failures-divergence refinement: FDR2 User Manual, 2010.
16. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. of the 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.
17. M. Geilen. On the construction of monitors for temporal logic properties. *Electr. Notes in Theor. Comp. Sci.*, 55(2), 2001.
18. M. Ghazel, A. Toguyéni, and P. Yim. State observer for DES under partial observation with timed petri nets. *Discrete Event Dynamic Systems*, 19(2):137–165, 2009.
19. S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. M. Amendola, and P. Marmo. An automatic SPIN validation of a safety critical railway control system. In *Proceedings of the 2000 Int. Conf. on Dependable Systems and Networks*, pages 119–124, Washington, DC, USA, 2000. IEEE Computer Society.
20. G. Graw and P. Herrmann. Transformation and verification of Executable UML models. In *Proceedings of the Workshop on the Compositional Verification of UML Models*, volume 101 of *Electr. Notes in Theor. Comp. Sci.*, pages 3–24, 2004.
21. J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems*, volume 06351 of *Dagstuhl Seminar Proceedings*, 2007.
22. H. H. Hansen, J. Ketema, B. Luttik, M. R. Mousavi, and J. van de Pol. Towards model checking Executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1-2):83–90, 2010.
23. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of the European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2009. See also `http://www.emftext.org` (last visit: 4 July 2011).
24. T. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
25. G. J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2003.
26. ISO/IEC. Enhancements to Lotos (E-Lotos), 2001. International Standard 15437:2001.
27. ISO/IEEE. ISO/IEEE 11073-20601: Health infomatics — personal health device communication — Part 20601: Application profile — optimized exchange protocol, Apr. 2010.
28. J. Keiren. Modelling session setup of IEEE Std 11073-20601, 2011. Personal communication.
29. KnowGravity. *Cassandra/xUML User's Guide*, 2008.
30. D. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management.* PhD thesis, University of York, United Kingdom, 2009. See also `http://www.eclipse.org/gmt/epsilon/` (last visit: 4 July 2011).
31. D. Kolovos, L. Rose, and R. Paige. The Epsilon Book. Available online at: `http://www.eclipse.org/gmt/epsilon/doc/book/` (last visit: 4 July 2011).
32. S. Lafortune, D. Teneketzis, M. Sampath, R. Sengupta, and K. Sinnamohideen. Failure diagnosis of dynamic systems: an approach based on discrete event systems. In *Proceedings of the American Control Conference, vol. 3*, pages 2058–2071, 2001.
33. J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. J. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

34. A. Mekki, M. Ghazel, and A. Toguyeni. Time-constrained systems validation using MDA model transformation. A railway case study. In *Proceedings of the 8th International Conference of Modeling and Simulation (MOSIM'10)*, 2010.
35. S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison Wesley, 2002.
36. Object Management Group. OMG Unified Modeling Language Superstructure Version 2.2, Feb. 2009.
37. Papyrus Developers. Papyrus: Open source tool for graphical UML2 modelling. `http://www.papyrusuml.org` (last visit: 4 July 2011).
38. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, 2000.
39. M. Sheeran and G. Stålmarck. A tutorial on Stålmarcks's proof procedure for propositional logic. In *Proceedings of the 2nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 1998.
40. M. Sighireanu. LOTOS NT user's manual. Technical report, INRIA Rhône-Alpes/VASY, 2008.
41. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, Boston, Massachusetts, 2008. See also `http://www.eclipse.org/modeling/emf/` (last visit: 4 July 2011).
42. E. Turner, H. Treharne, S. Schneider, and N. Evans. Automatic generation of CSP $\|$ B skeletons from xUML models. In *Proceedings of Theoretical Aspects of Computing (ICTAC 2008)*, pages 364–379, 2008.
43. K. Winter and N. J. Robinson. Modelling large railway interlockings and model checking small ones. In *ACSC '03: Proceedings of the 26th Australasian Comp. Sci. conference*, pages 309–316. Australian Computer Society, Inc., 2003.
44. W. L. Yeung, K. R. P. H. Leung, J. Wang, and W. Dong. Improvements towards formalizing UML state diagrams in CSP. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*. IEEE Computer Society, 2005.