Towards Bounded Model Checking for UML*

Artur Niewiadomski[1], Wojciech Penczek[1,2], and Maciej Szreter[2]

[1] Institute of Computer Science, University of Podlasie, Siedlce, Poland,
artur@iis.ap.siedlce.pl,
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland,
{penczek,mszreter}@ipipan.waw.pl

Abstract. The paper presents a preliminary version of BMCU - a Bounded Model Checker for UML. All the executions of an UML system (unfolded to a given depth) are encoded into a boolean propositional formula, satisfiability of which is checked using a SAT-solver. Contrary to other UML verification tools we do not use any of the existing model checkers as we do not translate UML specifications into an intermediate formalism. The paper provides also preliminary experimental results.

1 Introduction

Unified Modelling Language (UML) [1] is a graphical specification language widely used in development of various systems. The current version (2.1) consists of thirteen types of diagrams. Each diagram allows for describing a system from a different point of view, with many levels of abstractions. Nowadays, model-checking techniques that are able to verify crucial properties of systems, at a very early stage of the design process, are used in development of IT systems increasingly often. The current paper presents preliminary results of our work aiming at development of a novel symbolic verification method that avoids an intermediate translation and operates directly on systems specified in a subset of UML. The method is a version of a symbolic bounded model checking, designed especially for UML systems. All the possible executions of a system (unfolded to a given depth) are encoded into a boolean propositional formula satisfiability of which is checked using a SAT-solver. Contrary to other UML verification systems we do not make use of any existing model checker as we do not translate UML specifications into any intermediate formalism.

There have been a lot of attempts to verify UML state machines - all of them based on the same idea: translate an UML specification to the input language of some model checker, and then perform verification using this model checker. Some of the approaches [2, 3] translate UML to Promela language and then make use of the model checker Spin [4]. Other [5, 6] exploit timed automata as an intermediate formalism and use UPPAAL [7] for verification. The third group of tools [8–10] apply the symbolic model checkers SMV [11] or NuSMV [12] via translating UML to their input languages.

* Partly supported by the Ministry of Education and Science under the grant No. 3 T11C 011 28

An important advantage of our method consists in an efficient encoding of hierarchical state machines (SM), which is linear in the size of SM. Most of other methods, that can handle hierarchy, perform flattening of SM so they are likely to cause the state explosion of models generated. To the best of our knowledge only the paper [10] handles hierarchies directly without flattening. Another disadvantage of traditional methods follows from the fact that it is hard to reconcile UML semantics with intermediate formalism semantics. This results in a significant grow of the model size caused by adding special control structures that force execution w.r.t. UML semantics.

Our implementation has proven to be quite non-trivial, featuring advanced encodings of several UML constructs. In this paper we discuss a preliminary version, with the major parts of the formalism provided. The experience of the authors with symbolic model checking techniques [13–15] allows us to expect that there is a potential for many optimizations.

One of the most serious problems hindering the verification of UML is the lack of its formal semantics. The OMG standard [1] describes all the UML elements, but it deals with many of them informally. Moreover, there are numerous *semantics variation points* having several possible interpretations. Many papers on the semantics of UML have been published so far, but most of them skip some important issues. The interested reader is referred to the surveys [16, 17].

The approach of [10], which considers a similar subset of UML, is most close to our work. The paper [10] deals with variables, their types and the instructions allowed to be executed during firing of transitions in abstractions, but it does not support time events, internal transitions as well as entry and exit actions. Moreover, it simplifies the handling of concurrent transitions. On the other hand we do not consider choice pseudo-states and deferred events.

The rest of the paper is organised as follows. The next section describes the subset of UML considered and formalises the semantics as a labelled transition system. Then, we present the symbolic encoding and give the preliminary experimental results. Final remarks are given in the last section.

Notice that due to the lack of space a lot of details of the UML syntax and semantics, given in our report [18], have been omitted here.

2 Semantics of UML Subset

This section defines an UML subset considered and its operational semantics. Due to the space limitations we give only intuitive explanations of the concepts and the symbols used for defining the transition system. All the remaining details and formal definitions can be found in the report [18]. We assume also that the reader is familiar with basic UML state machine concepts.

The systems considered are specified by a single class diagram which defines k classes, a single object diagram which defines n objects, and k state machine diagrams, each one assigned to a different class of the class diagram.

The class diagram defines a list of attributes and a list of operations (possibly with parameters) for each class. The object diagram specifies the instances of

classes (objects) and (optionally) assigns the initial values to variables. All objects are visible globally, and the set of objects is constant during the life time of the system - dynamic object creation and termination is not allowed. We denote the set of all the variables by $\mathcal{V}$, the set of the integer variables by $\mathcal{V}^{int} \subseteq \mathcal{V}$ and the set of the object variables by $\mathcal{V}^{obj} \subseteq \mathcal{V}$. The values of object variables are restricted to the set of all objects defined in the object diagram, denoted by $\mathcal{O}$, and the special value $NULL$. Each object has been assigned an instance of a state machine that determines the behaviour of the object. A state machine diagram may consist of simple states composite states, final states, and initial pseudostates, as well as regions (the areas filling the composite states) and transitions connecting source and target states. The transitions are labelled with the expressions of the form $trigger[guard]/action$, where each of these components can be empty.

A transition can be fired if the source state is *active*, the guard (a Boolean expression) is satisfied, and the trigger matching event occurs. An event can be of the following three types: an operation call, a *completion event*, or a *time event*. In general, firing of a transition causes deactivation and activation of some states (depending on the type of the transition and the hierarchy of given state machine). We say that the state machine *configuration* changes then. More details can be found in [18].

A time event, defined by an expression of the form $after(\delta_1, \delta_2)$, where $\delta_1, \delta_2 \in \mathbb{N}$ and $\delta_1 \leq \delta_2$, can occur not earlier than after the flow of δ_1 time units and no later than before the flow of δ_2 time units. The time flow is measured from entering the *time state*, which is the source state of a transition with the trigger of the form $after(\delta_1, \delta_2)$.

The operation calls and the time events coming to the given object are put into the *event queue* of the object, and then, one at a time, they are handled. The event from the head of queue fires a transition (or many transitions) and is consumed, or is discarded, if it cannot fire any transition. The transitions that can fire due to the events taken from queues are called *triggered transitions*. We refer to the processing of a single event from the queue as to the *Run-To-Completion (RTC) step*. Next, an event can be handled only if the previous one has been fully processed, together with all the completion events which eventually have occurred.

A completion event (denoted by κ) occurs for a state that has completed all of its internal activities. The completion events fire the *completion transitions*, i.e., transitions without a trigger defined explicitly. The completion transitions have the priority over the triggered transitions.

The execution of the whole system follows the interleaving semantics, similar to [6]. During a single step only one object performs its RTC step. If more than one object can execute its step, then an object is chosen in a non-deterministic way. However, if none of the objects can perform an action, then the time flows. Note that this happens when all event queues are empty and all the completion events have been handled. The time flow can cause the occurrence of time events.

The time events are placed in the queues of respective objects and processed in next RTC steps.

There are two key notions of our semantics, namely, *global states* and *transition relation*. A global state is determined by the active configuration of all instances of state machines, the valuations of all the variables, the content of all the event queues, and the valuations of all the clocks measuring how long ago the system has entered the given state.

A *configuration* of the state machine of the i-th object ($\mathcal{SM}_i$, $i \in \{1, \ldots, n\}$) is a pair $\langle \mathcal{A}_i, \mathcal{C}_i \rangle$, where $\mathcal{A}_i$ is a set of *active states*, and $\mathcal{C}_i \subseteq \mathcal{A}_i$ is a set of *completed states*. We say that a state is *completed* when the completion event has occurred for the state, but has not been handled yet. The set of all the configurations for the i-th object is denoted by $\widehat{\mathcal{S}}_i$.

Let E and $\mathcal{Q}$ denote respectively the set of all the events and the set of all the event queues. Let $\Omega = \mathbb{Z} \cup \mathcal{O} \cup (E \setminus \{\kappa\})^\star \cup \widehat{\mathcal{S}}$, where $\mathbb{Z}$ is the set of integer numbers, $(E \setminus \{\kappa\})^\star$ is the set of all finite sequences of events (without completion events), and $\widehat{\mathcal{S}}$ is the set of all configurations of all the objects. Let us define the valuation function $v : \mathcal{V} \cup \mathcal{Q} \cup \mathcal{O} \longmapsto \Omega$ such that $v(\mathcal{V}^{int}) \subseteq \mathbb{Z}$, $v(\mathcal{V}^{obj}) \subseteq \mathcal{O} \cup \{NULL\}$, $v(\mathcal{Q}) \subseteq (E \setminus \{\kappa\})^\star$ and $v(\mathcal{O}) \subseteq \widehat{\mathcal{S}}$.

The function v assigns an integer number to each integer variable, an object or $NULL$ to each object variable, a sequence of events to each event queue, and an active configuration to each object. The active configuration of the i-th object for a given valuation v is denoted by $\langle \mathcal{A}_i^v, \mathcal{C}_i^v \rangle$, and $\vartheta(v, \alpha)$ denotes the valuation v' computed from v after the execution of the action α.

The *initial valuation* v^0 is the valuation that returns an empty sequence (ε) for all the event queues, the initial states marked as active and completed for all objects, and the initial values for all variables.

Let $\mathcal{S}$ be the set of all states from all instances of state machines. Let $\Gamma \subseteq \mathcal{S}$ be the set of all time states. The *clocks valuation* function $\mu : \mathcal{S} \longmapsto \mathbb{N}$ assigns a natural number to each time state and zero to any other state. For $s \in \Gamma$, the clock valuation $\mu(s)$ indicates how long ago the system has entered to the time state s, or how long ago the system has started, if s has not been active yet.

Let $\mu + \delta$ for $\delta \in \mathbb{N}$ denote the clocks valuation such that $\mu'(s) = \mu(s) + \delta$ for $s \in \Gamma$ and $\mu'(s) = 0$ for $s \notin \Gamma$. Let $\mu[Y := 0]$ for $Y \subseteq \mathcal{S}$ denote the clocks valuation μ' such that $\mu'(s) = 0$ for $s \in Y$ and $\mu'(s) = \mu(s)$ for $s \notin Y$. The valuation μ^0 such that $\forall_{s \in \mathcal{S}}\ \mu^0(s) = 0$ is called the *initial clocks valuation*. Formally the *global state* is a pair $g = \langle v, \mu \rangle \in \Omega^{\mathcal{O} \cup \mathcal{V} \cup \mathcal{Q}} \times \mathbb{N}^{\Gamma}$.

2.1 Behaviour of the system

At the state $g = \langle v, \mu \rangle$ the system can perform one of the following transitions (the ordering given follows the priorities of the transitions):

1. **Consumption of the completion events.** Removes all the completion events that cannot fire any completion transition for the i-th object in the state g. The elements of the set I_i^v are removed from $\mathcal{C}_i^v$ (we denote this by $discard(I_i^v)$). The set $I_i^v \subseteq \mathcal{C}_i^v$ contains the completed states of the i-th

object that are not the source states for the completion transitions enabled in the state g.

2. **Execution of a completion transition.** Handles one completion event κ causing the firing of one completion transition t_κ, and changes the valuation according to the sequence of actions $\lambda(t_\kappa)$, that is: exit actions and deactivation of leaving states, the transition action, the entry actions and activation of the entered states, producing completion events for some of the activated states, and clock resets for the entered timed states.
3. **Discarding of the event.** Discards the event from the head of i-th event queue, when it does not enable any transition. It is denoted by $cons(q_i)$.
4. **Execution of triggered transitions.** Firing of the set of non-conflicting triggered transitions enabled by the event in the head of the event queue. The resolution of conflicts is based on the nesting level of the source states of transitions and is described in detail in [18]. We deal with changes of the valuation in a way similar to 2, but additionally the event in the head of queue is consumed. Moreover, in the presence of orthogonal (concurrent) regions more than one transition can be fired in the single RTC step, so the action sequence $\lambda(\varphi)$ which changes the valuation contains the actions caused by all executed transitions (the set $\overline{\varphi}$).
5. **Time passage.** If all the event queues are empty and all completion events have been processed, then $x \in X \subset \mathbb{N}$ time units passes. We consider all the time transitions with guard expressions satisfied and with active states as sources. Then, we compute the set X by subtracting of the lower and upper bound of the time events specifications from the clock valuations for the active time states ($\mu(s) - \delta_1$ and $\mu(s) - \delta_2$). The set X is bounded by the starting time of the earliest time event and the earliest expiration time of the considered time events.
   A time transition *must* be fired not later than its expiration time indicates, but obviously it can be fired earlier, even at its starting time. Thus, in order to choose the set of objects $O_x \subseteq \mathcal{O}$ to the queues of which the time event will be added, we choose the transitions which expire after passing x time units and any subset of the remaining transitions that are enabled after this time. The sequence of actions which place the time events in the event queues of the objects from O_x is denoted by $\tau(O_x)$.

2.2 Transition system

The operational semantics of the systems specified in the selected UML subset is defined by the labelled transition system $\langle \mathcal{G}, g^0, \Sigma, \rightarrow \rangle$, where:

- $\mathcal{G} = \Omega^{\mathcal{O} \cup \mathcal{V} \cup \mathcal{Q}} \times \mathbb{N}^{\Gamma}$ is the set of states,
- $g^0 = \langle v^0, \mu^0 \rangle$ is the initial state,
- $\Sigma = \mathbb{N}$ is the set of labels corresponding to time units passing during transitions,
- $\rightarrow \subseteq \mathcal{G} \times \Sigma \times \mathcal{G}$ is the transition relation. Let $g = \langle v, \mu \rangle$, $g' = \langle v', \mu' \rangle$ and $\sigma \in \Sigma$. There exists a transition from the state g leading to the state g'

labelled with σ iff:

$$\exists_{i\in\{1,\dots,n\}}\ I_i^v \neq \emptyset \ \wedge\ \sigma = 0 \ \wedge\ v' = \vartheta\big(v,\, discard(I_i^v)\big) \ \wedge\ \mu' = \mu \quad (1)$$

or

$$\exists_{i\in\{1,\dots,n\}}\ \mathcal{C}_i^v \neq \emptyset \wedge\ I_i^v = \emptyset$$
$$\wedge\ \sigma = 0 \wedge\ v' = \vartheta\big(v,\, \lambda(t_\kappa)\big) \ \wedge\ \mu' = \mu\big[\Lambda(t_\kappa) \ :=\ 0\big] \quad (2)$$

or

$$\exists_{i\in\{1,\dots,n\}}\ \mathcal{C}_i^v = \emptyset \ \wedge\ v(q_i) \neq \varepsilon \ \wedge\ enabled(g, o_i) = \emptyset$$
$$\wedge\ \sigma = 0 \ \wedge\ v' = \vartheta\big(v,\, cons(q_i)\big) \ \wedge\ \mu' = \mu \quad (3)$$

or

$$\exists_{i\in\{1,\dots,n\}}\ \mathcal{C}_i^v = \emptyset \wedge\ v(q_i) \neq \varepsilon \wedge\ enabled(g, o_i) \neq \emptyset$$
$$\wedge\ \sigma\ = 0 \wedge\ v' = \vartheta\big(v,\, \lambda(\varphi)\big) \wedge\ \mu' = \mu\big[\Lambda(\overline{\varphi}) := 0\big] \quad (4)$$

or

$$\forall_{i\in\{1,\dots,n\}}\, \mathcal{C}_i^v = \emptyset \wedge v(q_i) = \varepsilon \wedge \sigma = x \wedge v' = \vartheta\big(v,\, \tau(O_x)\big) \wedge \mu' = \mu + x \quad (5)$$

where $v(q_i)$ is the content of the i-th event queue in the state g, $\Lambda(t_\kappa)$ is the set of states activated as a result of firing the transition t_κ, and $\Lambda(\overline{\varphi})$ is the set of states activated as a result of firing the set of transitions $\overline{\varphi}$.

3 Symbolic Encoding

In order to define a symbolic encoding of our UML semantics we have to first represent the global states by sequences of bits. To this aim each global state g is represented by n binary sequences, where each sequence stands for a state of one object. The representation of a single object consists of five binary sequences that encode respectively a set of active states, a set of completed states, a contents of the event queue, a valuation of the variables, and a valuation of the clocks.

3.1 Binary representation of the global states

Let $\mathcal{S}_i = \{s_i^0, \dots, s_i^{l_i}\}$. A set of active states is represented by a binary sequence of length $|\mathcal{S}_i|$ such that its jth element is equal to 1 iff the state $s_i^{l_j}$ is active in g. The second binary sequence representing a set of the completed states in g is defined in the similar way. The third binary sequence represents the contents of the event queue in g. A single event queue q_i is represented by a m-element cyclic buffer, and a pair of the indices of the first and the last event in the queue. A maximal size of a single element of a queue is equal to the maximal number of bits needed to represent the longest event (an operation with the maximal number of the parameters for a given class), denoted by $b(i)$. So we need a sequence of $m * b(i) + 2 * \lceil \log_2 m \rceil$ bits to encode the i-th event queue. The last two binary sequences in the representation of g are for the valuations of the variables and of the clocks in g. In our prototype implementation, we treat all the types of the

variables (including clocks) as integers. In order to keep our verification problem decidable, we assume that the domain of values for each variable is finite. For the integer variables we bound the domain to $\langle -maxint, maxint\rangle$. Then, the number of bits for encoding an integer variable is equal to $int_{size} = \lceil \log_2 maxint \rceil + 1$. So, the number of bits r needed to encode one global state is equal to:

$$r = \Sigma_{i=1}^{n}\Big(2 * |\mathcal{S}_i| + m * b(i) + 2 * \lceil \log_2 m \rceil + (|\mathcal{V}_i| + |\Gamma_i|) * int_{size}\Big)$$

From now on, we identify a global state with its binary representation.

3.2 A symbolic path

Our aim is to encode symbolically all the executions of length k (called k-paths) of a system by means of a propositional formula $path_k$. Then, we check satisfiability of the formula which is the conjunction of $path_k$ and some encoded property to be tested (e.g. a reachability property) using a SAT-solver. If the formula is satisfiable, then we obtain a valuation satisfying the formula, which can be interpreted as a concrete execution of the system. This valuation can be decoded as a sequence of global states leading to the state in which the property tested holds.

In order to construct the formula $path_k$ for a given system we deal with vectors of propositional variables, called *state variables*. Denote by $\mathcal{S}_v$ a set of state variables, containing the symbols **true** and **false**. Each state of a k-path can be symbolically represented as a valuation of a vector of state variables $\mathbf{w} = (w_1, \ldots, w_r)$. Let us define an valuation of state variables as $\mathcal{V} : \mathcal{S}_v \longmapsto \{0, 1\}$. Then, a valuation of vectors of r state variables $\mathcal{V} : {\mathcal{S}_v}^r \longmapsto \{0, 1\}^r$ is given as: $\mathcal{V}(w_1, \ldots, w_r) = (\mathcal{V}(w_1), \ldots, \mathcal{V}(w_r))$.

All the k-paths can be encoded over a symbolic k-path, i.e., $k + 1$ vectors of state variables $\mathbf{w}_j$ for $j = 0, \ldots, k$. Each vector $\mathbf{w}_j$ is used for encoding global states of a system. Specifically, $\mathbf{w}_0$ encodes the initial state (g^0) whereas $\mathbf{w}_k$ encodes the last states of the k-paths. A vector $\mathbf{w}_j$ consists of n sub-vectors of state variables $\mathbf{o}_i^{\mathbf{w}_j}$ for $i = 1, \ldots, n$, where $\mathbf{o}_i^{\mathbf{w}_j}$ encodes a state of the i-th object. A state of the i-th object is encoded over a sequence of five vectors of state variables: $\mathbf{a}_i^{\mathbf{w}_j}$, $\mathbf{c}_i^{\mathbf{w}_j}$, $\mathbf{q}_i^{\mathbf{w}_j}$, $\mathbf{v}_i^{\mathbf{w}_j}$, and $\tau_i^{\mathbf{w}_j}$ that represent a set of active states, a set of completed states, a contents of the event queue, a valuation of the variables, and a valuation of the clocks, respectively.

Let $\mathbf{w}$ and $\mathbf{w}'$ be vectors of state variables, and $\mathcal{V}$ - a valuation of state variables, as discussed above. Define the following formulae:

- $\mathfrak{I}(\mathbf{w})$ is a formula s.t. for every valuation $\mathcal{V}$ have we that $\mathcal{V}$ satisfies $\mathfrak{I}(\mathbf{w})$ iff $\mathcal{V}(\mathbf{w})$ is equal to the initial state g^0 of the transition system.
- $\mathfrak{T}(\mathbf{w}, \mathbf{w}')$ - a formula s.t. for every valuation $\mathcal{V}$ we have that $\mathcal{V}$ satisfies $\mathfrak{T}(\mathbf{w}, \mathbf{w}')$ iff $\mathcal{V}(\mathbf{w}) \xrightarrow{x} \mathcal{V}(\mathbf{w}')$, for $x \in \mathbb{N}$,

3.3 Symbolic transition relation

In Section 2.2 we defined five types of transitions that are executed according to their priorities. Here, we define the propositional formulae for transitions of types $1-4$ that encode their preconditions over the vector $\mathbf{w}$ for the i-th object: $pre_i^p(\mathbf{w})$, where $p \in \{1,2,3,4\}$. For all the types of the transitions we define the propositional formulae encoding an execution of these transitions over the vectors $\mathbf{w}, \mathbf{w}'$ for the i-th object: $post_i^p(\mathbf{w}, \mathbf{w}')$, where $p \in \{1,\dots,5\}$.

The transitions of types 1–4 are "local" i.e. their execution does not depend on which type of transition can be fired by other objects (on the contrary, the time transition of type 5 is "global", because it can be fired only if all the objects can execute no transition of type 1–4). Because of the lack of space we show some details describing the implementation of the pre- and postconditions only for transitions of the type 1.

Let us define a helper formula $compl(s, \mathbf{w})$ that holds true iff a state s is completed in a global state $\mathbf{w}$, and a helper formula $guard(t, \mathbf{w})$ that holds true iff the guard of a transition t is satisfied in a global state $\mathbf{w}$. Let Tc_s denote the set of completion transitions outgoing from the state s. Then, let the formula:

$$preCS1(s, \mathbf{w}) = compl(s, \mathbf{w}) \wedge \bigwedge_{t \in Tc_s} \neg guard(t, \mathbf{w}) \tag{6}$$

be a precondition for a transition of type 1 for some state s: it is true iff state s is completed in global state $\mathbf{w}$, and there exists no completion transition with guard satisfied outgoing from s. Now, if for some object o_i there exists a state s for which $preCS1(s, \mathbf{w})$ is satisfied, then the precondition for the transition of type 1 holds:

$$pre_i^1(\mathbf{w}) = \bigvee_{s \in \mathcal{S}_i} preCS1(s) \tag{7}$$

The postcondition for a transition of type 1 for object o_i is defined as follows:

$$post_i^1(\mathbf{w}, \mathbf{w}') = \bigwedge_{s \in \mathcal{S}_i} \Big(preCS1(s) \wedge \neg compl(s, \mathbf{w}') \tag{8}$$
$$\vee \neg preCS1(s) \wedge \big(compl(s, \mathbf{w}) \Leftrightarrow compl(s, \mathbf{w}') \big) \Big)$$

That is, for every potentially completed state, if the precondition holds true and the state is completed, we set the completion bit false for this state, while in the opposite case we simply copy the value of this bit.

Using the analogous formulas encoding postconditions for other "local" transitions we construct the formula describing the execution of the corresponding transitions, nesting the conditions for the consecutive levels. So that a transition of some level becomes enabled only if the transitions of the preceding levels cannot be executed:

$$
\begin{aligned}
post_i(\mathbf{w},\mathbf{w}') = {} & pre_i^1(\mathbf{w},\mathbf{w}') \wedge post_i^1(\mathbf{w},\mathbf{w}') \\
\vee\neg pre_i^1(\mathbf{w}) \wedge \Big(& pre_i^2(\mathbf{w},\mathbf{w}') \wedge post_i^2(\mathbf{w},\mathbf{w}') \\
\vee\neg pre_i^2(\mathbf{w}) \wedge \Big(& pre_i^3(\mathbf{w},\mathbf{w}') \wedge post_i^3(\mathbf{w},\mathbf{w}') \\
\vee\neg pre_i^3(\mathbf{w}) \wedge \big(& pre_i^4(\mathbf{w},\mathbf{w}') \wedge post_i^4(\mathbf{w},\mathbf{w}')\big)\Big)\Big)
\end{aligned} \tag{9}
$$

Finally, we combine all the transition levels to encode the transition relation:

$$
\mathfrak{T}(\mathbf{w},\mathbf{w}') = \bigvee_{i=1}^{n} post_i(\mathbf{w},\mathbf{w}') \vee \bigwedge_{i=1}^{n}\bigwedge_{p=1}^{4} \neg pre_i^p(\mathbf{w}) \wedge post^5(\mathbf{w},\mathbf{w}') \tag{10}
$$

Now, we can define $path_k$ over $\mathbf{w}_0,\ldots,\mathbf{w}_k$ as the following propositional formula:

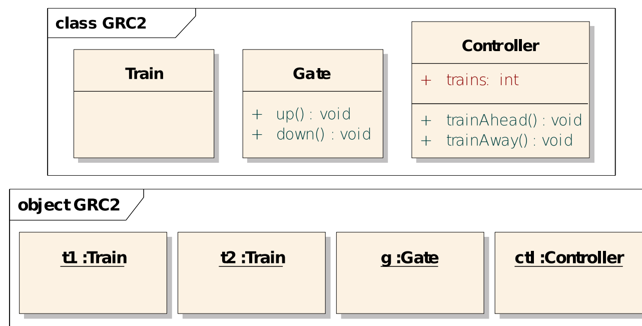
$$
path_k(\mathbf{w}_0,\ldots,\mathbf{w}_k) ::= \mathfrak{I}(\mathbf{w}_0) \bigwedge_{j=0}^{k-1} \mathfrak{T}(\mathbf{w}_j,\mathbf{w}_{j+1}) \tag{11}
$$

4 Experimental Results

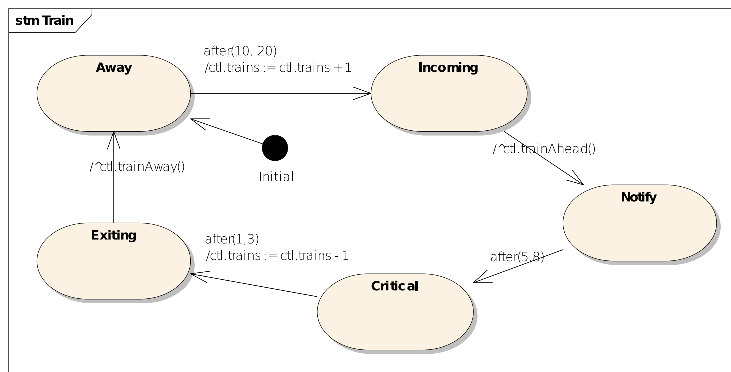
The prototype implementation has been tested on a well known Generalised Railroad Crossing (GRC) benchmark. The system, operating a gate at a railroad crossing, consists of a gate, a controller and N tracks which are occupied by trains. Each track is equipped with sensors that indicate a position of a train and send appropriate message to the controller. Depending on the track occupancy the controller can either open or close the gate.

Our version of GRC consits of 3 classes: Gate, Controller and Train, and $N{+}2$ objects: N instances of Train, one instance of Gate and one instance of Controller (Fig. 1).

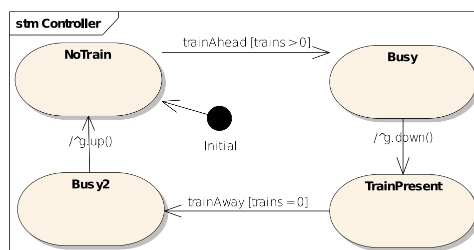
We have tested reachability of a global state in which some train is in the state *Critical*, whereas the gate is in the state *open*. Indeed the GRC specification contains a subtle error that allows this kind of behaviour. The obtained counterexample shows the situation when one train leaves the gate behind, the gate is in the state *opening* and the other train is approaching. Then, the message *down* sent to the gate by the controller is discarded, and the train crosses the railroad when the gate is open.



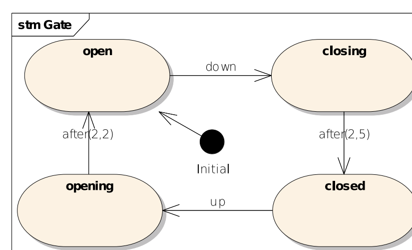
(a) Class and object diagrams



(b) State machine for class Train



(c) State machine for class Controller



(d) State machine for class Gate

Fig. 1. State machines of GRC system

Table 1 presents the preliminary results of GRC verification for 2 and 3 trains. The results are very encouraging. As it can be seen, quite long paths (of size 34) can be handled. In order to see how the tool is dealing with shorter counterexamples, we have simplified the GRC specification by merging the gate and controller into one class. It appears that the simplified specification also contains an error, in this case, reachable at a smaller depth, so more processes can be verified. The results of verification of the simplified GRC are presented in Table 2.

N	k	Obj.	States	Trans.	Vars	Clauses	Encoding [s]	SAT time[s]	SAT MB
2	30	4	22	22	198339	601217	5.872	41.067	93.30
3	34	5	28	28	287041	877738	9.416	347.662	201.12

Table 1. Preliminary results of verification of GRC system

The tests have been performed on the computer equipped with Pentium M 1.73 GHz CPU and 1.2 GB RAM running Linux. The maximum size of event queues has been set to 5 messages.

N	k	Obj.	States	Trans.	Vars	Clauses	Encoding [s]	SAT time[s]	SAT MB
2	7	3	11	11	40306	121949	1.064	0.896	19.52
10	15	11	43	43	307254	959121	10.34	9.629	147.70
20	25	21	83	83	1045219	3272066	43.835	95.026	553.09
22	27	23	91	91	1254636	3927279	54.532	130.336	693.20

Table 2. Preliminary results of verification of simplified GRC system

5 Final Remarks

In this paper we described a new approach to Bounded Model Checking for UML. Instead of dealing with a translation to a standard formalism of timed automata, we encoded the verification problem directly into SAT. We believe that this is a way in which symbolic methods can be used to handle advanced languages. Our preliminary results are very promising.

A plan for a future work is to add more functionality of the UML State Diagrams language, and to provide a detailed comparison with the existing tools for UML. The latter is not trivial given the lack of common semantics used by the tools. Another possible extension is to use a temporal logic as a specification formalism.

References

1. OMG: Unified Modeling Language. http://www.omg.org/spec/UML/2.1.2 (2007)
2. Lilius, J., Paltor, I.: vUML: A tool for verifying uml models. In: ASE. (1999) 255–258
3. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model checking dynamic and hierarchical UML state machines. In: MoDeV^{2}a. (2006) 94–110
4. Holzmann, G.J.: The SPIN Model Checker : Primer and Reference Manual. Addison-Wesley Professional (September 2003)
5. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: FTRTFT. (2002) 395–416
6. Diethers, K., Goltz, U., Huhn, M.: Model checking UML statecharts with time. In: Critical Systems Development with UML – Proceedings of the UML'02 workshop, Technische Universität München (2002) 35–52
7. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST. IEEE Computer Society (2006) 125–126
8. Compton, K., Gurevich, Y., Huggins, J., Shen, W.: An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan (2000)
9. Gutiérrez, M.E.B., Barrio-Solórzano, M., Quintero, C.E.C., de la Fuente, P.: Uml automatic verification tool with formal methods. Electr. Notes Theor. Comput. Sci. **127**(4) (2005) 3–16
10. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. Technical Report B24, Helsinki University of Technology, Laboratory for Theoretical Computer Science (2007)
11. Mcmillan, K.: The SMV system. Technical Report CMU-CS-92-131, School of computer Science, Carnegie Mellon University (1992)
12. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV, London, UK, Springer-Verlag (2002) 359–364
13. Penczek, W., Szreter, M.: SAT-based Unbounded Model Checking of Timed Automata. In: ACSD. (2007) 236–237
14. Kacprzak, M., Lomuscio, A., Niewiadomski, A., Penczek, W., Raimondi, F., Szreter, M.: Comparing BDD and SAT Based Techniques for Model Checking Chaum's Dining Cryptographers Protocol. Fundam. Inform. **72**(1-3) (2006) 215–234
15. Penczek, W., Pólrola, A.: Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach. Volume 20 of Studies in Computational Intelligence. Springer (2006)
16. Bhaduri, P., Ramesh, S.: Model Checking of Statechart Models: Survey and Research Directions. ArXiv Computer Science e-prints (July 2004)
17. Crane, M.L., Dingel, J.: On the semantics of uml state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen's University, Kingston, Ontario, Canada (2005)
18. Niewiadomski, A., Penczek, W., Szreter, M.: Semantyka operacyjna wybranych diagramów UML (in Polish). Technical Report 1009, ICS PAS (2008)