Fundamenta Informaticae ??? (2009) 1001–1015
IOS Press

A New Approach to Model Checking of UML State Machines[*]

Artur Niewiadomski
ICS, University of Podlasie, Poland, artur@iis.ap.siedlce.pl

Wojciech Penczek
ICS, University of Podlasie and Institute of Computer Science, PAS, Poland, penczek@ipipan.waw.pl

Maciej Szreter
Institute of Computer Science, PAS, Poland, mszreter@ipipan.waw.pl

Abstract. The paper presents a new approach to model checking of systems specified in UML. All the executions of an UML system (unfolded to a given depth) are encoded directly 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 method has been implemented as the (prototype) tool BMC4UML and some experimental results are presented.

Keywords: UML, Bounded Model Checking, symbolic verification

1. Introduction

Unified Modeling Language (UML) [15] 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 abstraction. 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 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

Address for correspondence: A. Niewiadomski, Institute of Computer Science, University of Podlasie, ul. Sienkiewicza 51, 08-110 Siedlce, Poland
[*]Partly supported by the Ministry of Education and Science under the grant No. N N516 370436

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 a UML specification to the input language of some model checker, and then perform verification using the underlying model checker. Some of the approaches [10, 12] translate UML to Promela and then make use of the model checker Spin [9]. Others [6, 11] exploit timed automata as an intermediate formalism and use UPPAAL [1] for verification. The third group of tools [4, 7, 8] apply the symbolic model checkers SMV [13] or NuSMV [3] via translating UML to their input languages.

An important advantage of our method consists in an efficient encoding of hierarchical state machines (HSM, for short). Most of other methods, that can handle hierarchy, perform flattening of HSM so they are likely to cause the state explosion of models generated. To the best of our knowledge only the paper [7] 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 growth of the model size caused by adding special control structures that force execution w.r.t. UML semantics.

One of the most serious problems hindering the verification of UML is the lack of its formal semantics. The OMG standard [15] describes all the UML elements, but it deals with many of them informally. Moreover, there are numerous *semantic 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 such as completion events or composite states. The interested reader is referred to the surveys [2, 5].

The approach of [7], which considers a similar subset of UML, is the closest to our work. The paper [7] deals with variables, their types, and the instructions allowed to be executed while firing transitions, but it does not support time events, internal transitions as well as entry and exit actions. Moreover, it simplifies handling of concurrent transitions. On the other hand we do not consider choice pseudostates and deferred events.

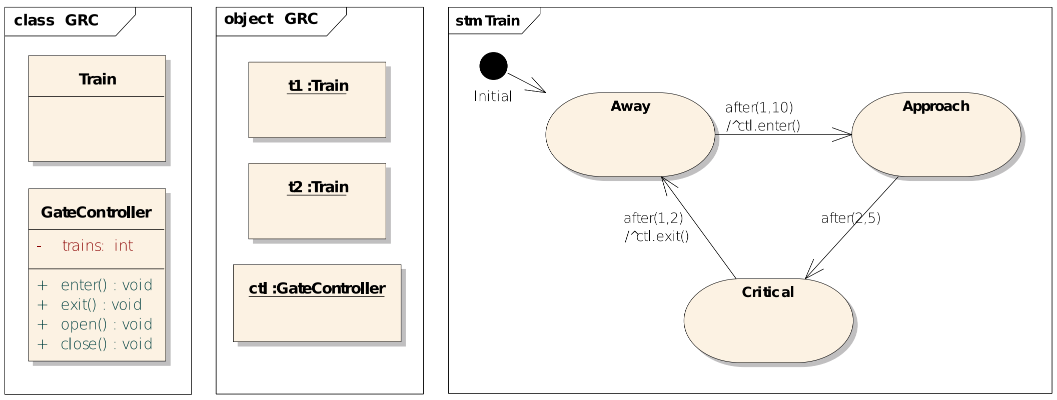
The rest of the paper is organised as follows. The next section describes the subset of UML considered and formalises its semantics as a labelled transition system. We present a symbolic encoding in Section 3, and discuss some preliminary experimental results in Section 4. Final remarks are given in the last section.

2. Syntax and Semantics of an UML Subset

This section defines an UML subset considered and its operational semantics. We give only an intuitive explanation of the concepts and the symbols used for defining the semantics. All the remaining details and formal definitions can be found in our Technical Report [14]. We assume also that the reader is familiar with basic UML state machine concepts.

2.1. Overview

We start with an overview of a syntax and a semantics of UML, while in the next section we give a formal operational semantics. The syntax is illustrated with the diagrams of the Generalized Railroad Crossing



(a) Class and object diagrams (b) State machine diagram of class Train

Figure 1. Specification of Generalized Railroad Crossing system

system, which is also used as a benchmark in Section 4.

The systems considered are specified by a single class diagram which defines k classes (e.g. see Fig. 1(a)), a single object diagram which defines n objects (e.g. in Fig. 1(a)), and k state machine diagrams (e.g. in Fig. 1(b), 2), 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 the 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 is assigned an instance of a state machine that determines the behavior of the object. An instance of a state machine assigned to ith object is denoted by $\mathcal{SM}_i$. A state machine diagram typically consists of states, regions and transitions connecting source and target states. The set of all states of $\mathcal{SM}_i$ is denoted by $\mathcal{S}_i$, whereas $\mathcal{S} = \bigcup_{i=1}^{n} \mathcal{S}_i$ is the set of all states from all instances of state machines. We consider several types of states, namely: simple states (e.g. *Away* in Fig. 1(b)), composite states, (e.g. *Main* in Fig. 2), final states, and initial pseudostates[1], (e.g. *Initial* in Fig. 2). For each object we define the set of *active* states $\mathcal{A}_i$, where $\mathcal{A}_i \subseteq \mathcal{S}_i$, $A_i \neq \emptyset$, and $i = 1, \ldots, n$. The areas filling the composite states are called *regions*. The regions contained in the same composite state are *orthogonal* (e.g. *Gate* and *Controller* in Fig. 2). The regions contain states and transitions, and thus introduce a *hierarchy* of state machines. We assume that a definition of the hierarchy relation is given, and we implicitly refer to this relation by using the terms ancestor and descendant. See [14] for more details.

The transitions are labelled with 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

[1]A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph, e.g. initial, choice, or history pseudostates.

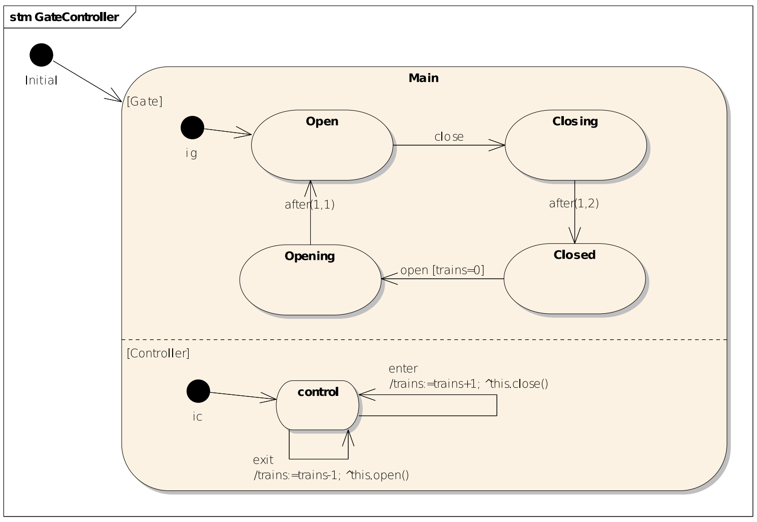


Figure 2. Specification of Generalized Railroad Crossing system - state machine diagram of class GateController

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 [14].

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 passing of δ_1 time units and no later than before passing of δ_2 time units. This is an extension of the standard $after(x)$ expression, which allows one to specify an interval of time in which a transition is enabled. However, we follow the discrete-time semantics where the clock valuations are natural numbers. For measuring time implicit natural variables, called *clocks*, are used. 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 set of all time states from $\mathcal{SM}_i$ is denoted by Γ_i, and the set of all time states from all instances of state machines is denoted by Γ, where $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$.

The operation calls 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 the queue either fires a transition (or many transitions) and is consumed, or it is discarded if it cannot fire any transition. The transitions with non-empty trigger are called *triggered transitions*. We refer to the processing of a single event from the queue or a time event 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 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 such a step, then an object is chosen in a non-deterministic way. However, if none of the objects can perform an *untimed action* (i.e., any action but a timed transition), then time flows. Note that this happens when all event queues are empty and all the completion events have been handled. The time flow causes occurrences of time events. The time events are processed in the next RTC steps.

2.2. Operational semantics

There are two key notions of our semantics, namely, *global states* and a *transition relation*. Below, we give several definitions, exploited later in the operational semantics.

Definition 2.1. (State machine configuration)
A set of states is *consistent* if for each pair of its distinct states these states either belong to orthogonal regions or one is an ancestor of the other. A state is *completed* if a completion event has occurred for this state, but has not been handled yet.
A *configuration* of the state machine of the i-th object is a pair $\langle \mathcal{A}_i, \mathcal{C}_i \rangle$, where $\mathcal{A}_i \subseteq \mathcal{S}_i$ is a *consistent* set of *active states*, and $\mathcal{C}_i \subseteq \mathcal{A}_i$ is a set of *completed states*. The set of all the configurations of the i-th object is denoted by $\widehat{\mathcal{S}_i}$ while $\widehat{\mathcal{S}}$ is the set of all the configurations of all the objects.

Definition 2.2. (Valuation)
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, and $(E \setminus \{\kappa\})^\star$ is the set of all finite sequences of events (without completion events).
A *valuation function* v is defined as: $v : \mathcal{V} \cup \mathcal{Q} \cup \mathcal{O} \longmapsto \Omega$, where $v(\mathcal{V}^{int}) \subseteq \mathbb{Z}$, $v(\mathcal{V}^{obj}) \subseteq \mathcal{O} \cup \{NULL\}$, $v(\mathcal{Q}) \subseteq (E \setminus \{\kappa\})^\star$ and $v(\mathcal{O}) \subseteq \widehat{\mathcal{S}}$. The function v assigns an integer 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 configuration of the i-th object for a given valuation v is denoted by $\langle \mathcal{A}_i^v, \mathcal{C}_i^v \rangle$, whereas $\vartheta(v, \alpha)$ denotes the valuation v' computed from v after the execution of the action α. The *initial valuation* v^0 is a valuation that returns an empty sequence (ε) for all the event queues, the initial states marked as active and completed for all the objects and the initial values for all the variables.

Definition 2.3. (Clocks valuation)
A *clocks valuation* function $\mu : \mathcal{S} \longmapsto \mathbb{N}$ assigns a natural number to each time state and zero to any other state. For $s \in \Gamma$, a clock valuation $\mu(s)$ indicates how long ago the system entered the time state s, or how long ago the system 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 $s \in \mathcal{S} \setminus \Gamma$. For $Y \subseteq \mathcal{S}$ let $\mu[Y := 0]$ denote the clocks valuation μ' such that $\mu'(s) = 0$ for $s \in Y$ and $\mu'(s) = \mu(s)$ for $s \in \mathcal{S} \setminus Y$. The valuation μ^0 such that $\forall_{s \in \mathcal{S}}\ \mu^0(s) = 0$ is called the *initial clocks valuation*. A pair $g = \langle v, \mu \rangle$ is called a *global state*. It is determined by the active configuration of all instances of state machines, the valuations of all the variables, the contents of all the event queues, and the valuations of all the clocks.

Definition 2.4. (Operational semantics)
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 a set of the global states,
- $g^0 = \langle v^0, \mu^0 \rangle$ is the initial state,
- $\Sigma = \mathbb{N}$ is a set of the labels corresponding to time units passing during transitions,

- $\rightarrow \subseteq \mathcal{G} \times \Sigma \times \mathcal{G}$ is the transition relation such that for $g = \langle v, \mu \rangle$, $g' = \langle v', \mu' \rangle$, and $\sigma \in \Sigma$ we have $g \xrightarrow{\sigma} g'$ iff one of the following conditions holds:

  1. $\exists_{i \in \{1,\ldots,n\}}\, I_i^v \neq \emptyset \;\wedge\; \sigma = 0 \;\wedge\; v' = \vartheta\big(v,\, discard(I_i^v)\big) \;\wedge\; \mu' = \mu$
  2. $\exists_{i \in \{1,\ldots,n\}}\, \mathcal{C}_i^v \neq \emptyset \wedge\; I_i^v = \emptyset \;\wedge\; \sigma = 0 \wedge\; v' = \vartheta\big(v,\, \lambda(t_\kappa)\big) \;\wedge\; \mu' = \mu\big[\Lambda(t_\kappa) \;:=\; 0\big]$
  3. $\exists_{i \in \{1,\ldots,n\}}\, \mathcal{C}_i^v = \emptyset \wedge$ *enabled*$(g, o_i) \neq \emptyset \wedge \sigma\; = 0 \wedge v' = \vartheta\big(v,\, \lambda(\varphi)\big) \wedge \mu' = \mu\big[\Lambda(\overline{\varphi}) := 0\big]$
  4. $\exists_{i \in \{1,\ldots,n\}}\, \mathcal{C}_i^v = \emptyset \wedge v(q_i) \neq \varepsilon\, \wedge$*enabled*$(g, o_i) = \emptyset \wedge \sigma = 0 \wedge v' = \vartheta\big(v,\, cons(q_i)\big) \wedge \mu' = \mu$
  5. $\forall_{i \in \{1,\ldots,n\}}\, \mathcal{C}_i^v = \emptyset \wedge v(q_i) = \varepsilon \wedge \sigma = x \wedge 0 < X_1 \leq x \leq X_2 \wedge v' = v \wedge \mu' = \mu + x$

  where:

  – the set $I_i^v \subseteq \mathcal{C}_i^v$ contains the completed states of the i-th object that are the source states for the completion transitions not enabled in the state g,

  – $discard(I_i^v)$ is the action of removing the elements of the set I_i^v from $\mathcal{C}_i^v$,

  – $\lambda(t_\kappa)$ is the sequence of actions w.r.t. the specification of the completion transition t_κ executed,

  – $\Lambda(t_\kappa)$ is the set of states activated as a result of firing the transition t_κ,

  – *enabled*(g, o_i) is the set of triggered transitions of i-th object enabled in the state g,

  – $\lambda(\varphi)$ is the sequence of actions w.r.t. the specifications of the sequence of triggered transitions φ executed,

  – $\Lambda(\overline{\varphi})$ is the set of states activated as a result of firing the set of transitions $\overline{\varphi} \subseteq$ *enabled*(g, o_i),

  – $v(q_i)$ is the content of the i-th event queue in the state g,

  – $cons(q_i)$ is the action of removing an event from the head of i-th event queue,

  – $X_1, X_2 \in \mathbb{N}$ are respectively the starting time of the earliest time event and the earliest expiration time of the considered time events.

It follows from Definition 2.4 that at a 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.

2. **Execution of a completion transition.** Handles one completion event κ causing a 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, and producing completion events for some of the activated states. Moreover the clocks of the entered timed states are reset.

3. **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 [14]. We deal with changes of the valuation in a way similar to the level 2. If a transition is triggered by an event from the queue, then it is additionally consumed. The second possibility is the firing of a timed transition

triggered by a time event. In this case the enabling condition depends rather on the clock valuation than the queue contents. Moreover, in the presence of orthogonal (concurrent) regions more than one transition can be fired in a single RTC step, so the action sequence $\lambda(\varphi)$, which changes the valuation, contains the actions caused by all executed transitions (the set $\overline{\varphi}$).

4. **Discarding of an event.** Discards an event from the head of the i-th event queue if the event does not enable any transition.

5. **Time flow.** If all the event queues are empty and all the completion events have been processed, then x time units passes, where $0 < X_1 \leq x \leq X_2$. We consider all the time transitions with guard expressions satisfied and with active states as the sources. Then, we compute the set of the allowed values of x by subtracting 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 is bounded by the starting time of the earliest time event (X_1) and the earliest expiration time of the considered time events (X_2). In other words, every number of time units lower than X_1 does not enable any timed transition, while every number of time units higher than X_2 causes an expiration of at least one timed transition.

3. Symbolic Encoding

Below we present a symbolic encoding of the operational semantics introduced in the previous section. First, the encoding of the global states is defined in order to give the symbolic transition relation. The most important details of the encoding of particular transition types can be found at the end of this section.

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 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 is decoded as a sequence of global states leading to the state in which the property tested holds.

3.1. Binary representation of the global states

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.

Observe that the following conditions hold:

1. The number of bits r needed to encode one global state is given as follows:
$r = \Sigma_{i=1}^{n}\Big(|\mathcal{S}_i| + |Cmpl(o_i)| + m * b(i) + 2 * \lceil \log_2 m \rceil + (|\mathcal{V}_i| + |Reg(\Gamma_i)|) * int_{size}\Big)$

2. The number of clocks sufficient for representing a state of the ith object is equal to the number of regions that directly contain time states.

Let $\mathcal{S}_i = \{s_1, \dots, s_k\}$ be a set of the UML states of the ith object. A set of active states of the ith object is represented by a binary sequence of length $|\mathcal{S}_i|$ such that its jth element is equal to 1 iff the state s_j is active in the state g. The second binary sequence representing a set of the completed states in g is defined in the similar way, but we consider only the *completion sensitive* states (from the set $Cmpl(o_i) \subseteq \mathcal{S}_i$), i.e., the states being the source states for completion transitions. The third binary sequence represents the contents of the event queue in g. A single event queue q_i is represented by an 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 needed to represent the valuation of the variables of the ith object is equal to $|\mathcal{V}_i| * int_{size}$, and to represent the valuation of the clocks of the ith object we need $|Reg(\Gamma_i)| * int_{size}$ bits, where $Reg(\Gamma_i)$ is the set of regions that directly contain time states.

A proof of the second condition follows from the observation that at most one UML state directly contained in a region can be active at the same time. So the time states contained in the same region can share the same clock.

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

3.2. A symbolic path

Now, we give an encoding of the symbolic transition relation. The description is structured in a top-down manner, i.e., first, we provide an encoding of the symbolic path, then the transition relation, and finally we give the detailed encoding of some particular transition types.

In order to encode all the executions of length k for a given system, as the formula $path_k$, 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 $\mathbf{w} = (w_1, \dots, w_r)$.

Definition 3.1. (Valuation of state variables)
Let us define a valuation of the state variables as $\mathcal{V} : \mathcal{S}_v \longmapsto \{0,1\}$. Then, a valuation of the vectors of r state variables $\mathcal{V} : \mathcal{S}_v{}^r \longmapsto \{0,1\}^r$ is given as: $\mathcal{V}(w_1, \dots, w_r) = (\mathcal{V}(w_1), \dots, \mathcal{V}(w_r))$.

All the k-paths can be encoded over a symbolic k-path, i.e., $k+1$ vectors of state variables $\mathbf{w}_j$ for $j = 0, \dots, k$. 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-path, for given k.

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 $\mathcal{V}$ we have 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 $x \in \mathbb{N}$.

Hence the formula encoding a symbolic k-path is defined as follows:

$$path_k(\mathbf{w}^0,\ldots,\mathbf{w}^k) \;=\; \mathfrak{I}(\mathbf{w}^0) \wedge \bigwedge_{i=0}^{k-1} \mathfrak{T}(\mathbf{w}^i,\mathbf{w}^{i+1}) \tag{1}$$

3.3. Symbolic transition relation

The following subsection introduces a set of helper formulae that encode enabling conditions and execution of transitions of types 1 - 5, given in Def. 2.4. These are necessary for defining an encoding of the symbolic transition relation. In order to improve readability, we apply the naming convention of the helper formulae, where their names consist of the letters E (from Enabling), X (from eXecution), C (from Class), O (from Object), S (from State), T (from UML Transition), and the numbers 1 – 5 (from the priorities of transitions). We define propositional formulae for the transitions of types 1 – 4 that encode their preconditions over the vector $\mathbf{w}$ for the object o: $EO1(o,\mathbf{w}), EO2(o,\mathbf{w}), EO3(o,\mathbf{w}), EO4(o,\mathbf{w})$. We also define the propositional formulae encoding an execution of these transitions over the vectors $\mathbf{w},\mathbf{w}'$ for the object o: $XO1(o,\mathbf{w},\mathbf{w}'), XO2(o,\mathbf{w},\mathbf{w}'), XO3(o,\mathbf{w},\mathbf{w}'), XO4(o,\mathbf{w},\mathbf{w}')$ and the formula encoding the time flow $X5(\mathbf{w},\mathbf{w}')$.

The transitions of types 1–4 are called *local* as 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 called *global*, because it can be fired only if all the objects can execute no untimed transition of type 1–4). The execution of local transitions for the object o over the vectors of state variables $\mathbf{w}$ and $\mathbf{w}'$ is encoded as:

$$\begin{aligned} XO(o,\mathbf{w},\mathbf{w}') = {} & EO1(o,\mathbf{w}) \wedge XO1(o,\mathbf{w},\mathbf{w}') \vee \neg EO1(o,\mathbf{w}) \wedge (EO2(o,\mathbf{w}) \wedge XO2(o,\mathbf{w},\mathbf{w}') \vee \\ & \neg EO2(o,\mathbf{w}) \wedge (EO3(o,\mathbf{w}) \wedge XO3(o,\mathbf{w},\mathbf{w}') \vee \neg EO3(o,\mathbf{w}) \wedge EO4(o,\mathbf{w}) \wedge XO4(o,\mathbf{w},\mathbf{w}'))) \end{aligned} \tag{2}$$

We ensure that a transition of some level becomes enabled only if the transitions of the preceding levels cannot be executed, by nesting the conditions for the consecutive levels. Then, iterating over the objects of class c, we encode the execution of local transitions for the class c:

$$XC(c,\mathbf{w},\mathbf{w}') \;=\; \bigvee_{o \in Objects(c)} XO(o,\mathbf{w},\mathbf{w}') \tag{3}$$

Now we are ready to give an encoding of the transition relation:

$$\mathfrak{T}(\mathbf{w},\mathbf{w}') \;=\; \bigvee_{c \in Classes} XC(c,\mathbf{w},\mathbf{w}') \vee E5(\mathbf{w}) \wedge X5(\mathbf{w},\mathbf{w}') \tag{4}$$

where the formula $E5(\mathbf{w})$ encodes the enabling conditions and $X5(\mathbf{w},\mathbf{w}')$ encodes the execution of the time flow transition.

3.4. Symbolic encoding: details

In this section we introduce a symbolic encoding of all of the transition types. We give the encoding of the preconditions of particular transition types as formulae beginning with E, and the encoding of postconditions (execution) as formulae beginning with X.

Transitions of type 1 (discarding of completion events). A precondition for a transition of level one for object o ($EO1$) is satisfied if in a symbolic state $\mathbf{w}$ there exists an UML state s that is completed and there does not exist any completion transition, outgoing from s, having the guard satisfied.

$$ES1(s, \mathbf{w}) = \big(compl(s, \mathbf{w}) \wedge \bigwedge_{t \in outCmpl(s)} \neg(guard(t, \mathbf{w})\big), \quad EO1(o, \mathbf{w}) = \bigvee_{s \in Cmpl(o)} ES1(s, \mathbf{w})$$

where $compl(s, \mathbf{w})$ is evaluated to **true** iff the state s is completed in $\mathbf{w}$, $outCmpl(s)$ is a set of all completion transitions outgoing from the state s, and the formula $guard(t, \mathbf{w})$ encodes the guard of transition t over the state variables from $\mathbf{w}$. The execution of this transition is encoded as the formula:

$$XO1(o, \mathbf{w}, \mathbf{w}') = \bigwedge_{s \in Cmpl(o)} \big(ES1(s, \mathbf{w}) \implies \neg compl(s, \mathbf{w}')\big) \wedge cpRest(\mathbf{w}, \mathbf{w}')$$

The states satisfying the precondition $ES1$ are marked as not completed in the state $\mathbf{w}'$, and the formula $cpRest(\mathbf{w}, \mathbf{w}')$ encodes copying of the state variables unchanged from $\mathbf{w}$ to $\mathbf{w}'$.

Transitions of type 2 (execution of completion transition). The enabling condition of this type of transition holds for an object o if there exists a completion transition, the guard of which is satisfied, outgoing from a state marked as completed:

$$ET2(t, \mathbf{w}) = compl(src(t), \mathbf{w}) \wedge guard(t, \mathbf{w}), \quad EO2(o, \mathbf{w}) = \bigvee_{s \in Cmpl(o)} \bigvee_{t \in outCmpl(s)} ET2(t, \mathbf{w})$$

$$XO2(o, \mathbf{w}, \mathbf{w}') = \bigvee_{s \in Cmpl(o)} \bigvee_{t \in outCmpl(s)} \big(ET2(t, \mathbf{w}) \wedge TEffect(t, \mathbf{w}, \mathbf{w}')\big) \wedge cpRest(\mathbf{w}, \mathbf{w}'))$$

where $TEffect$ encodes a change of a global state according to the specification of the transition executed (i.e., deactivation of the states left, activation of the states entered and the change of the variables and event queues if the transition's action is not empty).

Event queues and queue interface. Before we give an encoding of the transitions of types 3 and 4, which make use of event queues, we introduce some important details of a symbolic representation of the event queues. Under an assumption that at most one method can be called as a result of executing a single transition, we improved the encoding of the event queues by a kind of a symmetry optimization. For each step of the symbolic path we introduce a pair of additional event queues called *queue interface*. The operations of inserting and removing an event are encoded over the queue interface. Then, depending on the object which fires a triggered transition, the contents of the respective event queue is copied to the interface, the operation is performed and the modified contents of the queue interface is copied back to the queue of the appropriate object. In this way we avoid the encoding of complex operations over the event queue of each object. Below we use the formulae $msg(\mathbf{w})$ and $popMsg(\mathbf{w}, \mathbf{w}')$ that encode, respectively, the event in the head of the queue interface and the operation of removing the event from the queue interface.

Transitions of type 3 (execution of triggered transition). A transition of type 3 can be fired if its source state is active, the guard is satisfied and the trigger matches the event in the front of the queue, or if respective clock value is between the time constraints, in the case of a timed transition:

$$\begin{aligned} ET3(t, \mathbf{w}) &= active(src(t), \mathbf{w}) \wedge guard(t, \mathbf{w}) \wedge \begin{cases} match\big(msg(\mathbf{w}), trig(t)\big), \text{for } t \text{ untimed} \\ inTime(t, \mathbf{w}) \wedge \neg exp(t, \mathbf{w}), \text{for } t \text{ timed} \end{cases} \\ EO3(o, \mathbf{w}) &= \bigvee_{s \in States(o)} \bigvee_{t \in out(s)} ET3(t, \mathbf{w}) \end{aligned}$$

where $trig(t)$ is the trigger of the transition t, $match(message, trigger)$ is a formula evaluating to **true** iff $message$ matches the $trigger$, $States(o)$ is the set of all UML states of object o, and $out(s)$ is the set of all triggered transitions outgoing from state s. The formulae $inTime(t, \mathbf{w})$ and $exp(t, \mathbf{w})$ concern the timed transition and encode respectively that the transition t has entered its allowed time period and that the transition t is expired. An execution of this type of transition is similar to the type 2. Additionally, in the case of firing a triggered transition, the event is removed from the queue.

$$\begin{aligned} XT3(t, \mathbf{w}, \mathbf{w}') &= TEffect(t, \mathbf{w}, \mathbf{w}') \wedge cpRest(t, \mathbf{w}, \mathbf{w}') \wedge \begin{cases} popMsg(\mathbf{w}, \mathbf{w}'), \text{for } t \text{ untimed} \\ true, \text{for } t \text{ timed} \end{cases} \\ XO3(o, \mathbf{w}, \mathbf{w}') &= \bigvee_{s \in States(o)} \bigvee_{t \in out(s)} \big(ET3(t, \mathbf{w}) \wedge XT3(t, \mathbf{w}, \mathbf{w}')\big) \end{aligned}$$

Transitions of type 4 (implicit consumption of event). The transition of type 4 for an object o can be fired if the enabling conditions $EO3$, $EO2$ and $EO1$ (see formula 2) are not satisfied and the queue is not empty. The execution is simple: the event is removed from the queue and all the other state variables are copied to the next symbolic state $\mathbf{w}'$.

$$EO4(o, \mathbf{w}) = \neg queueEmpty(o, \mathbf{w}), \quad XO4(o, \mathbf{w}, \mathbf{w}') = popMsg(\mathbf{w}, \mathbf{w}') \wedge cpRest(o, \mathbf{w}, \mathbf{w}')$$

Transitions of type 5 (time flow). The enabling condition for the transition of type 5 is defined as:

$$E5(\mathbf{w}) = \bigwedge_{c \in Classes} \bigwedge_{o \in Objects(c)} \Big(queueEmpty(o, \mathbf{w}) \wedge \bigwedge_{s \in Cmpl(o)} \neg compl(s, \mathbf{w})\Big)$$

The definition of $E5$ follows from the fact that the transitions of type 1 and 2 can be fired only if there exists at least one state marked as completed, and the transitions of type 4 and untimed transitions of level 3 are enabled only if there exists at least one non empty event queue. Note that the time flow and firing of timed transitions can be interleaved. A time transition can be fired at any point of its time period (i.e., it fires immediately when it becomes enabled, or the firing can be delayed, but at most to the end of its time period). The execution of the time flow transition is encoded as:

$$X5(\mathbf{w}, \mathbf{w}') = incClocks(\mathbf{w}, \mathbf{w}') \wedge \bigwedge_{t \in timeTr} notExpire(t, \mathbf{w}, \mathbf{w}') \wedge \bigvee_{t \in timeTr} inTime(t, \mathbf{w}')$$

where $incClocks(\mathbf{w}, \mathbf{w}')$ is the formula that encodes incrementing all the clocks by some positive value x, and the formula:

$$notExpire(t, \mathbf{w}, \mathbf{w}') = \\ \Big(\neg exp(t, \mathbf{w}) \wedge guard(t, \mathbf{w}) \wedge active\big(src(t), \mathbf{w}\big)\Big) \implies \Big(exp(t, \mathbf{w}) \Leftrightarrow exp(t, \mathbf{w}')\Big)$$

bounds the maximum value of x, i.e., if a timed transition t is potentially enabled, then it will not expire after the time flow. The last part of the formula $X5$ bounds the minimum value of x, i.e., the time step must be big enough to enable at least one timed transition.

3.5. Discussion on complexity of the encoding

In this section we estimate the complexity of the encoding of the transition relation. To this purpose we estimate first the complexity of the encoding of the helper formulae. We base our assessments on the paper [16], which defines an encoding of arithmetic operations.

The complexity of the encoding of particular helper formulae is as follows:

- $guard(t, \mathbf{w})$, $TEffect(t, \mathbf{w}, \mathbf{w}')$ - polynomial w.r.t. the size of an input expression (the guard or the action of transition t), or exponential in the case when the input expression contains a multiplication.
- $compl(s, \mathbf{w})$, $active(s, \mathbf{w})$ - constant. Both the formulae are just one of the state variables.
- $cpRest(\mathbf{w}, \mathbf{w}')$ - quadratic w.r.t. the number of objects.
- $match(message, trigger)$ - linear w.r.t. the number of operations per class.
- $inTime(t, \mathbf{w})$, $exp(t, \mathbf{w})$, $incClocks(\mathbf{w}, \mathbf{w}')$ - polynomial w.r.t. the size of clock variables.
- the queue interface operations - polynomial w.r.t. the length of the event queue.

It follows from the above that the complexity of the encoding of the transition relation is polynomial except for the case when the multiplication is used in the specification of the system, which makes the encoding exponential.

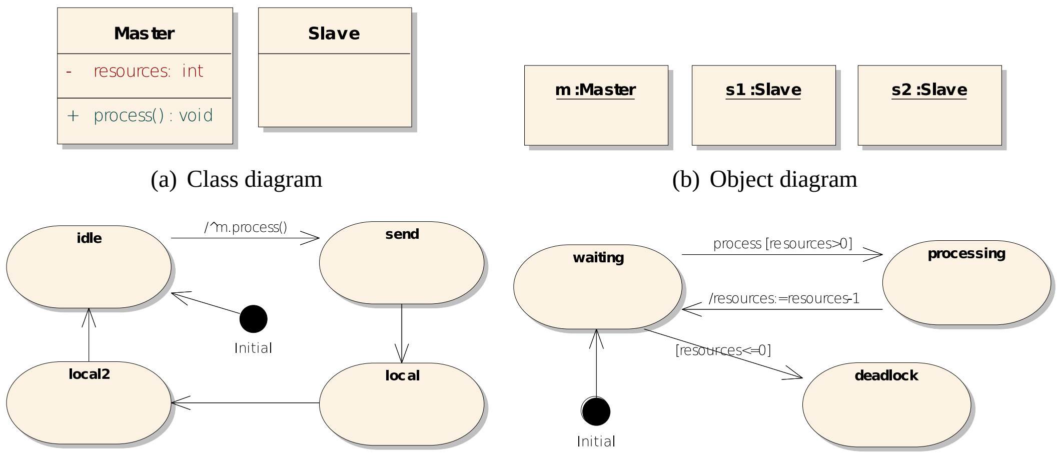
4. Experimental Results

Our prototype implementation has been tested using two example specifications. The first one – Master-Slave system (Fig. 3) – is an untimed specification of a simple system consisting of one instance of class Master and N instances of class Slave. The objects of type Slave send requests to the object of type Master (m) that handles the messages and decreases the variable $resources$. When the variable is equal to 0 the object m goes to the state $deadlock$.

Table 1 presents our experimental results of testing the reachability of the $deadlock$ state. The column k contains the depth of the symbolic path for SMUML and our tool. The variable $resources$ has been initially set to 4, the size of the event queue has been set to 3, and the integer variables have been

N	k	Hugo/Uppaal [s]	SMUML/NuSMV [s]	BMC4UML [s]
2	26	2.14	20.2	59.55
3	24	17.74	35.62	167.44
4	22	110.05	36.77	59.24
5	22	Out of memory	55.16	65.73
6	22	-	108.62	179.18
7	22	-	282.03	131.44
8	22	-	1 257.71	529.07
9	22	-	2 402.96	666.41

Table 1. The experimental results of verification of Master-Slave



(a) Class diagram (b) Object diagram (c) State machine of class Slave (d) State machine of class Master

Figure 3. Specification of Master-Slave system

N	k	Hugo/Uppaal [s]	BMC4UML [s]	BMC4UML* [s], k = 18
3	24	2.89	86.07	40.44
4	25	175.41	139.39	50.41
5	26	>2500	221.4	59.9
6	27	-	1354.89	75.21
7	-	-	-	92.6
10	-	-	-	152.36
15	-	-	-	279.61
20	-	-	-	448.64

Table 2. The experimental results of verification of GRC

encoded over 6 bits. The tests have been performed on the computer equipped with Pentium M 1.73 GHz CPU and 1.2 GB RAM running Linux.

The second specification tested is a variant of the well known Generalised Railroad Crossing (GRC) benchmark (Fig. 1, 2). 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 an appropriate message to the controller. Depending on the track occupancy the controller can either open or close the gate.

We were not able to use SMUML to verify the GRC system, because SMUML does not support timed specifications. On the other hand we had to model GRC in a way that minimises the semantic differences between Hugo and our tool, e.g., with the minimal number of completion transitions. The UML semantics implemented in Hugo allows to freely delay of handling of completion events, while our semantics (following the OMG standard) gives them the highest priority.

Table 2 presents the results of verification of GRC, where N denotes the number of trains, and k the depth of symbolic path at which the tested property is satisfiable. The results in the column marked with an asterisk concern the symbolic paths of length 18 that start from a non-initial state of the GRC system, but from the state where all trains are in the states *Away* and the object *ctl* is in the states *Main*, *Open*, and *control* (see Fig. 1, 2). In other words the paths have been made shorter by the “initialization part”.

Although this trick can be applied to all systems, it guarantees an improvement only for those where the initialisation part of all the objects *always* takes place before any other transitions. For other systems it can lead to increasing the length of the path or even incorrect results. In case of GRC the first non-completion transition, which can be fired, is a timed one going from the state *Away* to the state *Approach* in one of the *Train* objects. In order to fire such a transition the clocks have to be increased, so the time flow should happen first. It follows from the semantics that the time flow transition can be fired only if all untimed activities are done. Hence, this kind of a path reduction can be applied here, and the experimental results confirm it.

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 complex programming and specification languages. Our preliminary results are very promising. We have shown that our method is more efficient than two UML verifiers for some classes of systems and tested properties. A future work is to enlarge the subset of the UML state machines handled as well as to introduce more optimisations at the level of the symbolic encoding and the implementation.

References

[1] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST*, IEEE Computer Society, pages 125–126, 2006.

[2] P. Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *ArXiv Computer Science e-prints*, July 2004.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, London, UK, 2002. Springer-Verlag.

[4] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan, 2000.

[5] M. L. Crane and J. Dingel. On the semantics of UML state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen's University, Kingston, Ontario, Canada, 2005.

[6] K. Diethers, U. Goltz, and M. Huhn. Model checking UML statecharts with time. In *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 35–52. Technische Universität München, 2002.

[7] J. Dubrovin, T. Junttila, and K. Heljanko. Symbolic step encodings for object based communicating state machines. Technical Report B24, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2007.

[8] M. E. B. Gutiérrez, M. Barrio-Solórzano, C. E. C. Quintero, and P. de la Fuente. UML Automatic Verification Tool with Formal Methods. *Electr. Notes Theor. Comput. Sci.*, 127(4):3–16, 2005.

[9] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[10] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model checking dynamic and hierarchical UML state machines. In *MoDeV^{2}a*, pages 94–110, 2006.

[11] A. Knapp, S. Merz, and C. Rauh. Model checking - timed UML state machines and collaborations. In *FTRTFT*, pages 395–416, 2002.

[12] J. Lilius and I. Paltor. vUML: A tool for verifying uml models. In *ASE*, pages 255–258, 1999.

[13] K. Mcmillan. The SMV system. Technical Report CMU-CS-92-131, School of computer Science, Carnegie Mellon University, 1992.

[14] A. Niewiadomski, W. Penczek, and M. Szreter. Semantyka operacyjna wybranych diagramów UML (in Polish). Technical Report 1009, ICS PAS, 2008.

[15] OMG. Unified Modeling Language. http://www.omg.org/spec/UML/2.1.2, 2007.

[16] A. Zbrzezny. A boolean encoding of arithmetic operations. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'08)*, volume 225(3) of *Informatik-Berichte*, pages 536–547. Humboldt University, 2008.