
© 2008 IEEE.
Reprinted with permission.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Aalto Aalto University's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
Symbolic Model Checking of Hierarchical UML State Machines∗

Jori Dubrovin  
Helsinki University of Technology (TKK)  
Dept. of Information and Computer Science  
P.O.Box 5400, FI-02015 TKK, Finland  
Jori.Dubrovin@tkk.fi

Tommi Junttila  
Helsinki University of Technology (TKK)  
Dept. of Information and Computer Science  
P.O.Box 5400, FI-02015 TKK, Finland  
Tommi.Junttila@tkk.fi

Abstract

A compact symbolic encoding is described for the transition relation of systems modeled with asynchronously executing, hierarchical UML state machines that communicate through message passing and attribute access. This enables the analysis of such systems by symbolic model checking techniques, such as BDD-based model checking and SAT-based bounded model checking. Message reception, completion events, and run-to-completion steps are handled in accordance with the UML specification. The size of the encoding for state machine control logic is linear in the size of the state machine even in the presence of composite states, orthogonal regions, and message deferring. The encoding is implemented for the NuSMV model checker, and preliminary experimental results are presented.

1. Introduction

Model checking [7] is an automatic way of verifying that a hardware or software system fulfills its behavioral requirements. In symbolic model checking, the behavior of a system is analyzed by manipulating sets of states instead of individual states, often leading to remarkable speedup in verification time. A prerequisite for applying standard symbolic techniques is a symbolic encoding for the transition relation of the system.

The main contribution of this paper is a compact symbolic encoding for the control logic of communicating UML state machines [21]. This directly enables one to use state-of-the-art symbolic model checking techniques such as BDD-based [17] and bounded model checking [4] on UML systems composed of asynchronously executing, message passing state machines. The encoding is implemented in a tool that translates UML models to the input language of the symbolic model checker NuSMV [6]. In the perspective of Model Driven Engineering (MDE), the presented encoding is in principle sufficient for analyzing system models in early design phases when they do not yet contain too many data structures but only a communication skeleton. Analyzing models in later design phases requires in practice the use of model reduction techniques such as slicing and abstraction in addition.

The second contribution is that we give an accurate semantics for the subset of UML models we consider. The semantics is well suited for symbolic model checking because it (i) fixes the atomicity level, (ii) does not contain any pseudo-code but works directly on the state space level, and (iii) handles asynchronously executing objects, classes, and data (including signal parameters often omitted in similar works) in an abstract yet exact level without fixing the actual action description language or the type system. Special care is taken to formalize run-to-completion steps in accordance with [21, 23].

The main criterion for the selected UML subset is that it is suitable for describing systems composed of asynchronously executing objects communicating with message passing, such as communication protocols. Therefore, asynchronous signal events are included in the subset but synchronous call events are not. Of UML state machine features we support (i) state hierarchy, which is important for compactness and clarity of models, (ii) deferring of messages, (iii) completion events, (iv) concurrent composite states, and (v) initial and choice pseudostates. The only restriction is that we do not allow concurrent substates to react to the same signal because we feel that it is an inferior way of intra-object synchronization. As a result, the semantics becomes easier to understand and to use as a basis for model checking and code generation.

Note that one way to deal with hierarchical state machines is to flatten the hierarchy away as a preprocessing step. However, in the presence of concurrent composite states, this can result in an exponential blowup in the num-

*This work has been financially supported by Tekes (Finnish Funding Agency for Technology and Innovation), Nokia, Conformiq, Mipro, Helsinki Graduate School in Computer Science and Engineering, and the Academy of Finland (project 112016).
ber of states of the flattened state machine. If there are no concurrent composite states (or history pseudostates), then flattening will not add new states but the number of transitions can become quadratic in the size of the original state machine. The encoding presented here is not based on flattening but instead builds on the hierarchical structure of states. As a result, the control logic encoding can be represented in linear size w.r.t. the original state machine.

1.1. Related Work

A lot of research has already been done both on the semantics and on symbolic encodings of UML state machines, see e.g. [3, 8] for surveys. In the following we compare our work to the most relevant works in these areas.

The semantics presented in this paper is most closely related to that in [16]. The main difference is that while [16] presents a pseudo-code algorithm that executes UML models, the semantics in this paper is presented as a relation between successive configurations of the system. In our opinion such a relation based representation is a better basis for developing symbolic encodings as it makes it explicit what a single step of execution is.

A semantics for hierarchical UML state machines is presented in [14], but signal parameters and data manipulation are not discussed. In [24], the semantics is refined and a superlinear size symbolic encoding is presented. However, the processing of event queues is not thoroughly explained.

As a part of a larger EU project, [9] presents a symbolic transition relation for UML state machines (including call events which are not considered in this paper) but does not handle hierarchy or deferring of events.

In [2], a semantics for UML state machines and an approach for applying an interactive theorem prover is given. Choice pseudostates, signal parameters, and deferring of events are not supported, and the interaction between concurrently executing state machines is not discussed. A translation from UML state machines to NuSMV programs is sketched in [20], but completion events are not supported and signal parameters, deferring of events, handling of transition priorities, concurrent composite states, or the exact semantics of state machines are not discussed in detail.

The semantics in [13] is based on translating the full UML 2.0 state machine language to (superlinear size) “core state machines”. The granularity of the execution semantics is much finer than in this work, and we are not aware of symbolic encodings based on [13].

Furthermore, none of [13, 9, 14, 20, 24] seem to handle completion events in full accordance with [21, 23]; see Sect. 2.2 on “quiescent states” below.

2. A UML Subset and its Semantics

This section defines the class of UML systems considered in this paper. In a nutshell, a UML system is composed of a finite set of objects that are instances of classes in the underlying UML model. The objects can communicate with each other via asynchronous message passing and by accessing each other’s attributes. The behavior of each object is described by the hierarchical state machine associated with the class of the object.

We feel that a detailed formal definition such as the one below is justified and valuable in order to ensure correct handling of non-trivial features of UML state machines including hierarchy, deferring and implicit consumption of messages, message parameter reception, completion events, and quiescing.

2.1. Types, Signals, and Classes

As the focus of this article is on UML state machines, other relevant parts of UML models, e.g. data attribute manipulation, are defined only in a very abstract way.

To capture data types in UML models, a finite set $T$ of types is assumed, each type $T \in T$ being associated with a non-empty domain set $\text{dom}(T)$. In particular, the Boolean type $\mathbb{B}$ with $\text{dom}(\mathbb{B}) = \{\text{false}, \text{true}\}$ belongs to $T$. A typed variable is a name $x$ associated with a type $\text{type}(x) \in T$. The guards and effects appearing in state machines are expressed with a strongly typed action language $\mathcal{L}$ over the types; $\mathcal{L}_B \subseteq \mathcal{L}$ denotes the set of side-effect free Boolean valued expressions and $\mathcal{L}_{\text{stmt}} \subseteq \mathcal{L}$ the set of (possibly compound) statements. One example of such an action language is given in [10].

As usual, the set of all finite sequences over a set $X$ is denoted by $X^*$. If $a = \langle a_1, \ldots, a_k \rangle \in X^*$, $b = \langle b_1, \ldots, b_l \rangle \in X^*$, and $x \in X$, then $\text{dequeue}(a) = \langle a_2, \ldots, a_k \rangle$ (undefined if $k = 0$), $\text{append}(a, x) = \langle a_1, \ldots, a_k, x \rangle$, and $\text{concat}(a, b) = \langle a_1, \ldots, a_k, b_1, \ldots, b_l \rangle$.

Objects can communicate with each other by sending messages built over a finite set $\text{Sigs}$ of signals. Each signal $\text{sig} \in \text{Sigs}$ is associated with a list $\text{params}(\text{sig}) = \langle T_{\text{sig}, 1}, \ldots, T_{\text{sig}, k_{\text{sig}}} \rangle \in T^*$ of parameter types. A message is of the form $\text{sig}[v_1, \ldots, v_{k_{\text{sig}}}]$, where $\text{sig} \in \text{Sigs}$ and each $v_i \in \text{dom}(T_{\text{sig}, i})$: the set of all messages is denoted by $\text{Msgs}$. Message reception in state machines is denoted by $\text{signal triggers}$ of the form $\text{sig}(x_1, \ldots, x_{k_{\text{sig}}})$, where $\text{sig} \in \text{Sigs}$ and each $x_i$ is a typed variable with $\text{type}(x_i) = T_{\text{sig}, i}$. The set of all signal triggers is denoted by $\text{Trigs}$.

A class is a pair $C = \langle \text{attrs}, \text{sm} \rangle$, where $\text{Attrs}(C) = \text{attrs}$ is a finite set of typed variables called attributes and $\text{SM}(C) = \text{sm}$ is the state machine of the class.
2.2. State Machines

The behavior of an instance of a class (i.e. an object) is described by the associated state machine. Formally, a hierarchical UML state machine is a structure

\[ sm = (S, R, \text{top}, \text{container}, T, \text{defers}), \]

where

- \( S \) is a finite set of state vertices partitioned into simple states \( S_{si} \), composite states \( S_{co} \), final states \( S_{fi} \), initial pseudostates \( S_{ini} \), and choice pseudostates \( S_{ch} \);
- \( R \) is a finite set of regions (disjoint from \( S \));
- \( \text{top} \in R \) is the unique top region;
- \( \text{container} : (S \cup R \setminus \{\text{top}\}) \to (S \cup R) \) describes the state hierarchy of the state machine;
- \( T \) is a finite set of transitions; and
- \( \text{defers} : (S_{si} \cup S_{co}) \to 2^{S_{sig}} \) assigns each state a (possibly empty) set of deferrable signals.

For each \( v \in S \cup R \), define \( \text{descendants}(v) = \{v' \in S \cup R \setminus \{\text{top}\} \mid \exists s > 0 \colon \text{container}(v') = v\} \) and \( \text{children}(v) = \{v' \in S \cup R \setminus \{\text{top}\} \mid \text{container}(v') = v\} \).

The state hierarchy must be a connected tree, i.e. \( \text{descendants}(\text{top}) = S \cup R \setminus \{\text{top}\} \) must hold. It is required that the container of each non-top region is a composite state, and that the container of each state vertex is a region. Furthermore, each region must contain exactly one initial state, i.e. \( \forall r \in R : |\text{children}(r) \cap S_{ini}| = 1 \), and each composite state at least one region, i.e. \( \forall s \in S_{co} : \text{children}(s) \neq \emptyset \).

If a composite state contains more than one region, then it is called concurrent. Two state vertices \( s_1, s_2 \in S \) are orthogonal, denoted \( s_1 \perp s_2 \), if there are distinct regions \( r_1, r_2 \in R \), \( r_1 \neq r_2 \), such that \( \text{container}(r_1) = \text{container}(r_2) \), \( s_1 \in \text{descendants}(r_1) \), and \( s_2 \in \text{descendants}(r_2) \).

A set \( S \subseteq S \) of state vertices is consistent iff for any two distinct state vertices \( s_1, s_2 \in S \) either \( s_1 \perp s_2 \), \( s_1 \in \text{descendants}(s_2) \), or \( s_2 \in \text{descendants}(s_1) \).

As an example, consider the state machine in Fig. 1. \( A_2 \) is a concurrent composite state with \( \text{container}(A_2) = \text{top} \) and \( \text{children}(A_2) = \{r_1, r_2\} \), where \( r_1 \) and \( r_2 \) are regions. \( B_2 \) is a simple state with \( \text{defers}(B_2) = \{e\} \) and \( \text{container}(B_2) = r_1 \). The choice pseudostate \( B_3 \) and the final state \( C_4 \) are orthogonal. The state set \( \{A_2, B_1, C_2\} \) is consistent while \( \{A_3, D_2, D_3\} \) is not.

A transition \( t \) in the set \( T \) of transitions is a tuple

\[ (s, s', e) \in (S \setminus S_{fi}) \times (\text{Trigs}(\{\tau\}) \times L_\text{msg} \times X L_\text{sm}), \]

We define \( \text{source}(t) = s \), \( \text{guard}(t) = e \), \( \text{effect}(t) = e \), and \( \text{target}(t) = s' \). The container \( \text{container}(t) \) of \( t \) is the smallest (w.r.t. the partial order induced by \( \text{container} \)) region such that \( \{s, s'\} \subseteq \text{descendants}(r) \). If \( s = \tau \), we say that \( t \) is a completion transition and define \( \text{triggersig}(t) = \tau \). Otherwise, \( \sigma = \text{sig}(x_1, \ldots, x_{k_{sig}}) \) and we define \( \text{triggersig}(t) = \sigma \). In this work, the formal parameters \( x_1, \ldots, x_{k_{sig}} \) are required to be attributes in the associated class. In UML, only completion transitions can leave pseudostates: \( \text{source}(t) \in S_{ini} \cup S_{ch} \) implies \( \text{triggersig}(t) = \tau \). We require that transitions originating from orthogonal states are not triggered by the same signal: for all \( t_1, t_2 \in T \), if \( \text{triggersig}(t_1) = \text{triggersig}(t_2) \neq \tau \), then \( \text{source}(t_1) \perp \text{source}(t_2) \) must not hold.

In Fig. 1, \( t_{10} = (B_2, e(p), p > 3, x = 0;D_3) \) is a transition with \( \text{container}(t_{10}) = \text{top} \). The completion transition \( t_{15} \) has \( \text{container}(t_{15}) = r_2 \), \( \text{guard}(t_{15}) \) is implicitly true, and \( \text{effect}(t_{15}) = \text{skip} \), where \( \text{skip} \) is a pseudostate that does nothing.

**State Configurations.** A state configuration of the state machine is a pair

\[ SC = (A, Q), \]

where the set \( A \) of active state vertices is a maximal consistent subset of \( S \) and \( Q \subseteq A \) is a set of quiescent states. The intuition is that a state is in \( Q \) if it has already consumed its implicit completion event. A completion event of a state is consumed either by firing an outgoing completion transition or, if the guards of all completion transitions evaluate to false, by quiescing the state, after which the completion transitions will not become enabled even if the guards become true. This construction (from [16]) accurately models the requirement that the guards of completion transitions will not be evaluated again without re-entering the state. This requirement [21, p. 572] is made more explicit in [23, p. 659]. For example, state \( C_3 \) in Fig. 1 can become quiescent if \( x = 3 \) does not hold, and then \( t_{18} \) will not become enabled even if the value of \( x \) is changed to 3.

Because completion events are only relevant for states with outgoing completion transitions, we only define quiescence status for the set of completion sensitive states \( S_{CS} = \{s \in S_{ini} \cup S_{co} \mid \exists t \in T \text{ s.t. } \text{source}(t) = s \text{ and } \text{triggersig}(t) = \tau\} \). Thus, \( Q \) is always a subset of
A ∩ S_{CS}. A state $s \in S_{CS}$ is ready to consume its completion event in $sc$, denoted by $ceready(sc, s)$, if

1. it is active but not quiescent: $s \in A \setminus Q$, and
2. it is either (i) a simple state: $s \in S_{si}$, or (ii) a composite state with all its regions in final states: $s \in S_{co}$ and $\forall s' \in A : \text{container}(\text{container}(s')) = s \Rightarrow s' \in S_{R}$.

A state configuration $sc = \langle A, Q \rangle$ is

- in compound transition, denoted by $inct(sc)$, if it contains an active pseudostate: $A \cap (S_{in} \cup S_{ch}) \neq \emptyset$,
- in run-to-completion (RTC) step, denoted by $inrtc(sc)$, if (i) it is in compound transition, or (ii) there is a completion sensitive state that is ready to consume its completion event in it, and
- stable, denoted by $stable(sc)$, if it is not in RTC step.

A UML state machine consumes messages from its input queue only when it is in a stable state configuration.

Consider again the state machine in Fig. 1. The state $C_3$ is the only completion sensitive state in it. The pair $\{\{A_2, B_2, C_3\}, \{C_3\}\}$ is a stable state configuration, while the state configuration $\{\{A_2, B_3, C_3\}, \{C_3\}\}$ is in compound transition. The state configuration $\{\{A_2, B_2, C_3\}, \emptyset\}$ is in RTC step (but not in compound transition) because the state $C_3$ is ready to consume its completion event in it.

The default entry completion of a state vertex $s \in S$, denoted by $dec(s)$, is the smallest maximal consistent subset of $S_{co} \cup S_{in} \cup \{s\}$ such that $s \in dec(s)$. In the state machine in Fig. 1, $\text{dec}(A_2) = \{A_3, D_1\}$, $\text{dec}(D_3) = \{A_3, D_1\}$, $\text{dec}(C_3) = \{A_2, B_1, C_3\}$, and $\text{dec}(A_2) = \{A_2, B_1, C_1\}$.

Given a state configuration $sc = \langle A, Q \rangle$ and a transition $t \in T$ with $source(t) = A$, the $t$-successor of $sc$ is $\text{succ-conf}(sc, t) = \langle A', Q' \rangle$,

where $A' = (A \setminus D) \cup (\text{dec}(\text{target}(t)) \cap D)$, $Q' = Q \setminus D$, and $D = \text{descendants}(\text{container}(t))$. In the state machine in Fig. 1, the $t_{30}$-successor of the state configuration $\{\{A_2, B_2, C_3\}, \{C_3\}\}$ is $\{\{A_3, D_3\}, \emptyset\}$ while the $t_{11}$-successor of $\{\{A_2, B_3, C_3\}, \{C_3\}\}$ is $\{\{A_3, D_1\}, \emptyset\}$.

### 2.3. Systems and State Spaces

We consider a UML system to consist of a finite set of objects $O$, each object $o \in O$ being associated with the class $\text{Class}(o)$ that it is an instance of. A global configuration of the system is a tuple

$$gc = \langle \text{stateconf}_{gc}, \text{attrs}_{gc}, \text{input}_{gc}, \text{deferr}_{gc} \rangle,$$

where

- $\text{stateconf}_{gc}$ maps each object $o$ to the current state configuration of its state machine $\text{SM}(\text{Class}(o))$,
- $\text{attrs}_{gc}$ maps each object $o$ to a function giving each attribute $x \in \text{Attrs}(\text{Class}(o))$ its current value in $\text{dom}(\text{type}(x))$, and
- $\text{input}_{gc}, \text{deferr}_{gc} : O \rightarrow \text{Msgs}^*$ describe the contents of the input and deferred queues, respectively, of each object.

For convenience, let $\text{StateConf}(gc, o) = \text{stateconf}_{gc}(o)$, $\text{AttrVal}(gc, o, x) = \text{attrs}_{gc}(o)(x)$, $\text{InputQ}(gc, o) = \text{input}_{gc}(o)$, and $\text{DeferQ}(gc, o) = \text{deferr}_{gc}(o)$. The set of all global configurations is denoted by $\mathcal{GC}$.

Given a side-effect free Boolean expression $\phi$ in $\mathcal{L}_B$, $\text{eval}(gc, o, \phi)$ evaluates it in the context of a global configuration $gc$ and an object $o$, and returns $\text{false}$ or $\text{true}$. Given a statement $\gamma$ in $\mathcal{L}_{\text{sync}}$, $\text{exec}(gc, o, \gamma)$ executes it in the context of $gc$ and $o$, and returns a new global configuration $gc'$ with the restrictions that, for each $o' \in O$, (i) the state machine configuration is not modified: $\text{StateConf}(gc', o') = \text{StateConf}(gc, o')$, (ii) messages cannot be removed from the input queue: $\text{InputQ}(gc, o')$ is a prefix of $\text{InputQ}(gc', o')$, and (iii) the deferred queue is not modified: $\text{DeferQ}(gc', o') = \text{DeferQ}(gc, o')$.

UML requires that for each pseudostate, there is always at least one outgoing transition whose guard is true [21, p. 540]: $\forall gc \in \mathcal{GC}, \forall o \in O, \forall s \in S_{in} \cup S_{ch} : \exists t \in T$ s.t. $source(t) = s$ and $\text{eval}(gc, o, \text{guard}(t)) = \text{true}$.

### State Spaces

The actual semantics of a UML system is given by its state space that describes how the system may evolve from one global configuration to another. Each atomic step between global configurations corresponds to one object either firing one transition, deferring a message, or implicitly consuming a message or a completion event. The UML run-to-completion semantics for individual state machines is followed as messages can only be consumed in stable state configurations. Formally, the state space of a UML system is the tuple

$$\langle \mathcal{GC}, gc_{\text{init}}, \Delta \rangle,$$

where $gc_{\text{init}} \in \mathcal{GC}$ is the initial configuration, and $\Delta \subseteq \mathcal{GC} \times A \times \mathcal{GC}$ is the minimal transition relation defined by the following rules ($A$ being a set of possible annotations). Assume an object $o \in O$, that $\text{SM}(\text{Class}(o)) = \langle S, R, \text{top}, \text{container}, T, \text{defers} \rangle$ and let $sc = \langle A, Q \rangle = \text{StateConf}(gc, o)$.

- Signal Triggered Transitions. Assume a transition $t = \langle s, \text{sig}(x_1, \ldots, x_k), g, e, s' \rangle \in T$. The transition instance $(o, t)$ is enabled in $gc$, denoted by $\text{enabled}(gc, (o, t))$, if
– the state configuration is stable: \( \text{stable}(sc) \),
– the source state is active: \( s \in A \),
– \( \text{InputQ}(gc, o) = (\text{sig}|v_1, \ldots, v_k|, \ldots) \),
– \( \text{eval}(ge^*, o, g) = \text{true} \), where \( ge^* \) is equal to \( gc \) except that the message \( sig|v_1, \ldots, v_k| \) has been received: \( \text{InputQ}(gc^*, o) = \text{dequeue}(\text{InputQ}(gc, o)) \)

- no prioritized transition is enabled [21, p. 565]:
  \[ \exists \tau \in \mathcal{T} : \text{source}(\tau) \in \text{descendants}(s) \cap A \land \text{enabled}(gc, \langle o, t \rangle) \]
- the state configuration is stable: \( \text{stable}(sc) \),
- \( \text{InputQ}(gc, o) = (\text{sig}|v_1, \ldots, v_k|, \ldots) \),
- \( \forall 1 \leq i \leq k : \text{AttrVal}(ge^*, o, x_i) = v_i \),

- Completion transitions. Implicit completion events are consumed until a stable state configuration is reached. Formally, if \( t = (s, \sigma, g, e, s') \in \mathcal{T} \) and \( \sigma = \tau \), then the completion transition instance \( \langle o, t \rangle \) is enabled in \( gc \), denoted by \( \text{enabled}(gc, \langle o, t \rangle) \), if
  - the source is active: \( s \in A \),
  - either (i) the source is a pseudostate: \( s \in S_{\text{in}} \cup S_{\text{ch}} \) or (ii) the state configuration is in RTC step but not in compound transition and the source state \( s \) ready to consume its completion event: \( \neg \text{inct}(sc) \land \text{inrc}(sc) \land s \in S_{\text{in}} \cup S_{\text{co}} \land \text{ceready}(sc, s) \), and
  - the guard condition holds: \( \text{eval}(gc, o, g) = \text{true} \).

If \( \text{enabled}(gc, \langle o, t \rangle) \) holds, then \( \langle gc, \langle o, t \rangle, gc' \rangle \in \Delta \), where \( gc' \) is equal to \( gc'' = \text{exec}(gc, o, e) \) except that \( \text{StateConf}(gc', o) = \text{succ-conf}(gc, sc, t) \) and the deferred queue is flushed to the input queue: \( \text{DeferQ}(gc', o) = \emptyset \) and \( \text{InputQ}(gc', o) = \text{concat}(\text{DeferQ}(gc, o), \text{InputQ}(gc'', o)) \).

- **Deferring.** If no transition instance is enabled, then the first message in the input queue can be deferred. Formally, the deferring instance \( \langle o, \text{deferr} \rangle \) is enabled in \( gc \), denoted by \( \text{enabled}(gc, \langle o, \text{deferr} \rangle) \), if
  - the state configuration is stable: \( \text{stable}(sc) \),
  - \( \text{InputQ}(gc, o) = (\text{sig}|v_1, \ldots, v_k|, \ldots) \),
  - \( \exists t \in \mathcal{T} : \text{enabled}(gc, \langle o, t \rangle) \), and
  - there is an active state deferring the message: \( \exists s \in (S_{\text{in}} \cup S_{\text{co}}) \cap A : \text{sig} \in \text{deferr}(s) \).

If \( \text{enabled}(gc, \langle o, \text{deferr} \rangle) \) holds, then \( \langle gc, \langle o, \text{deferr} \rangle, gc' \rangle \in \Delta \), where \( gc' \) is equal to \( gc'' \) except that \( \text{InputQ}(gc', o) = \text{dequeue}(\text{InputQ}(gc, o)) \) and \( \text{DeferQ}(gc', o) = \text{append}(\text{DeferQ}(gc, o), \text{sig}|v_1, \ldots, v_k|) \).

- **Implicit consumption.** If the first message in the input queue is not consumed by a transition or deferred, it can be implicitly consumed. Formally, \( \text{enabled}(gc, \langle o, \text{impcons} \rangle) \) holds if
  - the state configuration is stable: \( \text{stable}(sc) \),
  - \( \text{InputQ}(gc, o) = (\text{sig}|v_1, \ldots, v_k|, \ldots) \),
  - \( \exists t \in \mathcal{T} : \text{enabled}(gc, \langle o, t \rangle) \), and
  - \( \text{enabled}(gc, \langle o, \text{deferr} \rangle) \) does not hold.

If \( \text{enabled}(gc, \langle o, \text{impcons} \rangle) \) holds, then \( \langle gc, \langle o, \text{impcons} \rangle, gc' \rangle \in \Delta \), where \( gc' \) is equal to \( gc'' = \text{dequeue}(\text{InputQ}(gc, o)) \).

- **Quiescing.** If a state \( s \in S_{\text{CS}} \) is ready to consume its completion event but no outgoing completion transition is enabled, the state can quiesce (i.e. implicitly consume the completion event). Formally, \( \text{enabled}(gc, \langle o, \text{quiesce} \rangle) \) holds if
  - the state configuration is in RTC step but not in compound transition: \( \text{inrc}(sc) \land \neg \text{inct}(sc) \),
  - the state is active and ready to consume its completion event: \( s \in S_{\text{CS}} \cap A \land \text{ceready}(sc, s) \), and
  - \( \exists t \in \mathcal{T} : \text{source}(t) = s \land \text{triggers}(t) = \tau \land \text{enabled}(gc, \langle o, t \rangle) \).

If \( \text{enabled}(gc, \langle o, \text{quiesce} \rangle) \) holds, then \( \langle gc, \langle o, \text{quiesce} \rangle, gc' \rangle \in \Delta \), where \( gc' \) is equal to \( gc \) except that \( \text{StateConf}(gc', o) = \langle A, Q \cup \{s\} \rangle \).

3. Encoding System Behavior

The symbolic encoding of the state space transition relation is based on constraints involving state variables, whose valuation represents a global configuration, next-state variables, whose valuation represents the global configuration after executing one step, input variables used to capture non-determinism and whose values are only limited by the constraints, and auxiliary derived functions defined over the variables. All variables and functions have Boolean values unless otherwise stated. To keep the state space finite, we assume that \( \text{dom}(T) \) is finite for each type \( T \) and restrict the analysis to bounded global configurations, where for each object, the total number of messages in its input and deferred queues is limited by the constant \( \text{FSIZE} \). This implies that all non-Boolean variables have finite domains and thus can be booleanized to enable the use of SAT- and BDD-based techniques.
More specifically, the encoding is such that, given a valuation for state variables that represents a bounded global configuration \(gc\), there exists a valuation for input variables that satisfies all the constraints if and only if the valuation for next-state variables represents a bounded global configuration \(gc'\) such that \((gc, a, gc') \in \Delta\) for some annotation \(a\).

3.1. State Variables

Let \(gc\) be a bounded global configuration. Let the state configuration of an object \(o \in O\) be \((A, Q) = StateConf(gc, o)\), the deferred queue \((M_0, \ldots, M_{d-1}) = DeferQ(gc, o)\), and the input queue \((M_d, \ldots, M_{n-1}) = InputQ(gc, o)\). Because \(gc\) is bounded, we have \(0 \leq d \leq n \leq \text{QSIZE}\). The set of state variables contains five kinds of elements, with values derived from \(gc\) as follows.

- **Active**\((o, s)\), where \(s\) is a state vertex in the state machine of \(o\), is true if and only if \(s\) is active, i.e. \(s \in A\).
- **Quiescent**\((o, s)\), where \(s\) is a completion sensitive state in the state machine of \(o\), is true if and only if \(s \in Q\).
- **AttrVal**\((o, x)\), where \(x \in \text{Attr}(\text{Class}(o))\), has domain \(\text{dom}(\text{type}(x))\) and value \(\text{AttrVal}(gc, o, x)\). These variables may appear in the definition of the \(\text{EvalGuard}(o, t)\) function below, as well as in the constraints defining the effects of transitions, but we omit all details of data handling.
- **Queue**\((o, k)\), where \(0 \leq k < \text{QSIZE}\), has domain \(\text{Msgs} \cup \{\text{none}\}\). Its value is \(M_k\) if \(k < n\), and \text{none} otherwise. In other words, the sequence \((\text{Queue}(o, 0), \ldots, \text{Queue}(o, \text{QSIZE} - 1))\) consists of the deferred queue, followed by the input queue, followed by zero or more \text{none} entries.
- **QPos**\((o)\) has domain \(\{0, 1, \ldots, \text{QSIZE}\}\) and value \(d\).

The corresponding next-state variables are denoted by \(\text{next}(\text{Active}(o, s))\), \(\text{next}(\text{Quiescent}(o, s))\), etc.

3.2. Queues and Messages

Let \(o \in O\) be an object. We restrict the model so that one transition can send at most one new message to each object (this can be circumvented by splitting non-complying transitions to segments and adding choice pseudostates between them). The function \(\text{QNewMsg}(o)\) with values in \(\text{Msgs} \cup \{\text{none}\}\) evaluates the message being sent to \(o\) during the step \((gc, a, gc')\), or \text{none} if no message is being sent. The definition of \(\text{QNewMsg}(o)\) depends on the action language statements, which are not considered here.

The function \(\text{CurrentMsg}(o)\) with domain \(\text{Msgs} \cup \{\text{none}\}\) contains the first message in the input queue, or \text{none} if the input queue is empty.

\[
\text{CurrentMsg}(o) :=
\begin{cases}
\text{QPos}(o) = 0 & : \text{Queue}(o, 0) \\
\vdots & : \\
\text{QPos}(o) = \text{QSIZE} - 1 & : \text{Queue}(o, \text{QSIZE} - 1) \\
\text{else} & : \text{none}.
\end{cases}
\]

The input variable \(\text{Dispatch}(o)\) with domain \(\text{Sigs} \cup \{\text{none}\}\) determines the signal in the message (if any) being consumed by \(o\). The signal must be present in \(\text{CurrentMsg}(o)\), so we add for each \(\text{sig} \in \text{Sigs}\) the constraint

\[
(\text{Dispatch}(o) = \text{sig}) \implies (\text{CurrentMsg}(o) = \text{sig}[\ldots]).
\]

The queues interface with other parts via the following operations: (i) removing the first message from the input queue and discarding it, triggered by the predicate \(\text{QRem}(o)\), (ii) moving the first message from the input queue to the deferred queue, triggered by \(\text{QDefer}(o)\), (iii) flushing the entire contents of the deferred queue to the input queue, triggered by \(\text{QFlush}(o)\), and (iv) adding the new message \(\text{QNewMsg}(o)\) to the input queue. Any combination of operations is allowed to occur in the same step as long as their order respects the list above, and cases (i) and (ii) do not occur in the same step. The functions \(\text{QRem}(o)\), \(\text{QDefer}(o)\), and \(\text{QFlush}(o)\) are defined in Sect. 3.3.

Operation (iii) above corresponds to resetting \(\text{QPos}(o)\) to zero, and operation (ii) corresponds to incrementing \(\text{QPos}(o)\), formalized by the constraint

\[
\text{next}(\text{QPos}(o)) = \begin{cases}
\text{QFlush}(o) & : 0 \\
\text{QDefer}(o) & : \text{QPos}(o) + 1 \\
\text{else} & : \text{QPos}(o).
\end{cases}
\]

Operation (i) removes the element at index \(\text{QPos}(o)\) from the queue, shifting the elements at \(\text{QPos}(o) + 1, \ldots, \text{QSIZE} - 1\) one position to the left. Operation (iv) adds the new message to the first free position, i.e. the position \(k\) such that \(\text{Queue}(o, k - 1) \neq \text{none}\) and \(\text{Queue}(o, k) = \text{none}\). However, if operation (i) has been performed in the same step, the position must be decremented by 1. The new queue contents for \(0 \leq k < \text{QSIZE}\) is thus determined by

\[
\text{next}(\text{Queue}(o, k)) =
\begin{cases}
\text{N}_o(k) & : \text{QNewMsg}(o) \\
\text{if} & : \text{QRem}(o) \land \text{QPos}(o) \leq k : \text{Queue}(o, k + 1) \\
\text{else} & : \text{Queue}(o, k),
\end{cases}
\]

where

\[
\text{N}_o(k) := (\text{QRem}(o) \iff (\text{Queue}(o, k) \neq \text{none})) \land
(\text{Queue}(o, k - 1) \neq \text{none}) \land (\text{Queue}(o, k + 1) = \text{none}).
\]
The boundaries are defined as we set $\text{Queue}(o, \text{QSIZE}) = \text{none}$ and $\text{Queue}(o, -1) \neq \text{none}$. To forbid transitions to global configurations that are not bounded, we prevent the queue from overflowing by setting the constraint

$$\left(\text{QNewMsg}(o) \neq \text{none}\right) \Rightarrow \left(\text{Queue}(o, \text{QSIZE} - 1) = \text{none}\right) \lor \text{QRem}(o).$$

(5)

3.3. Control Logic of State Machines

Next we describe a compact symbolic encoding for the semantics of UML state machines as described in Sect. 2.3. Let $o \in O$ be an object with the state machine $\langle S, R, \text{top}, \text{container}, T, \text{defers} \rangle$.

State Configuration Classification. As defined in Sect. 2.2, a completion sensitive state $s \in S_{CS}$ is ready to consume its completion event if it is active, not quiescent, and all its regions (if any) are in final states:

$$\text{CEReady}(o, s) := \text{Active}(o, s) \land \neg \text{Quiescent}(o, s) \land \bigwedge \{ F(o, r) \mid r \in \text{children}(s) \},$$

(6)

where $F(o, r) := \bigvee \{ \text{Active}(o, s') \mid s' \in \text{children}(r) \cap S_{R} \}$ is true iff the active child state of the region $r$ is a final state. Based on this, it is easy to define the predicates $\text{InCT}(o)$ and $\text{InRTC}(o)$ telling whether the state configuration is in compound transition or in RTC step, respectively:

$$\text{InCT}(o) := \bigvee \{ \text{Active}(o, s) \mid s \in S_{in} \cup S_{ch} \},$$

(7)

$$\text{InRTC}(o) := \text{InCT}(o) \lor \bigvee \{ \text{CEReady}(o, s) \mid s \in S_{CS} \}.$$  

(8)

Because an object can consume events from the input queue only if its state configuration is stable, we add the constraint

$$\text{InRTC}(o) \Rightarrow (\text{Dispatch}(o) = \text{none}).$$

(9)

Enabledness Conditions for Transitions. We associate with each transition $t \in T$ of the state machine an input variable $\text{Fire}(o, t)$ that determines whether $t$ is being fired in $o$. According to the semantics, firing a transition $t$ requires that (i) the source state is active, and (ii) the guard evaluates to true, captured by the constraint

$$\text{Fire}(o, t) \Rightarrow \text{Active}(o, \text{source}(t)) \land \text{EvalGuard}(o, t).$$

(10)

We assume that $\text{EvalGuard}(o, t)$ implements the function $\text{eval}(gc^*, o, \text{guard}(t))$ (or $\text{eval}(gc, o, \text{guard}(t))$ if $t$ is a completion transition) in Sect. 2.3, and omit its formula.

If $t$ is a completion transition whose source is a simple or composite state, it is also required that (i) the state configuration is not in a compound transition and (ii) the source state is ready to consume its completion event:

$$\text{Fire}(o, t) \Rightarrow \neg \text{InCT}(o) \land \text{CEReady}(o, \text{source}(t)).$$

(11)

If $t$ is not a completion transition, then we have to ensure that there is no enabled prioritized transition and no active descendant state can defer the message. For this purpose let $R_o(s, sig) := \{ t \in T \mid \text{source}(t) = s \land \text{triggers}(t) = \text{sig} \}$ and define an auxiliary function

$$\text{Feasible}(o, s, \text{sig}) := \text{Active}(o, s) \land$$

$$\neg \text{DescFeasible}(o, s, \text{sig}) \land \neg \text{DescDeferring}(o, s, \text{sig}) \land$$

$$\bigvee \{ \text{EvalGuard}(o, t) \mid t \in R_o(s, \text{sig}) \}$$

(12)

evaluating to true iff (i) there is at least one sig-triggered transition with source $s$ and guard satisfied, and (ii) there is neither an active descendant of $s$ that can defer sig nor a prioritized enabled sig-triggered transition. The function $\text{DescFeasible}(o, v, \text{sig})$ is defined simply by

$$\text{DescFeasible}(o, v, \text{sig}) := \bigvee \{ \text{Feasible}(o, v', \text{sig}) \lor$$

$$\text{DescFeasible}(o, v', \text{sig}) \mid v' \in \text{children}(v) \}.$$  

(13)

with $\text{Feasible}(o, v, \text{sig}) := \text{false}$ for $v \in S_{in} \cup S_{ch} \cup S_{R} \cup R$. The similar function $\text{DescDeferring}(o, s, \text{sig})$ is defined later. Now we only have to constrain that if a message with signal $\text{sig}$ is to be consumed from the input queue and there are enabled $\text{sig}$-triggered transitions leaving from an active state $s \in S_{in} \cup S_{ch}$, then one of them is fired:

$$\bigvee \{ \text{Fire}(o, t) \mid t \in R_o(s, \text{sig}) \} \Leftrightarrow$$

$$\{ \text{Dispatch}(o) = \text{sig} \} \land \text{Feasible}(o, s, \text{sig}).$$

(14)

Constraint (20) below ensures that at most one such transition is fired. Due to (9), a signal triggered transition can only be fired in a stable state configuration.

State Configuration Change due to Transition Firing. Perhaps the most complicated part in the symbolic encoding is the computation of successor state configurations as defined in Sect. 2.2. A state vertex $s \in S$ becomes active in the next configuration if it is entered by firing a transition, and it remains active if it is not exited by a transition:

$$\text{next}(\text{Active}(o, s)) \Leftrightarrow$$

$$\text{Enter}(o, s) \lor \{ \text{Active}(o, s) \land \neg \text{Exit}(o, s) \}.$$  

(15)

A non-initial state vertex is entered if it is the target of a transition being fired, or if it is a composite state whose region is being broken in by a transition. We say that a fired transition $t$ breaks in a region $r$ if the transition cuts in through the boundary of the region in the diagram, or formally, if $r \in \text{descendants}(\text{container}(t))$ and $\text{target}(t) \in$
descendants(r). In Fig. 1, the only such transition is t_{10}, which breaks in region r_3. For each vertex s ∈ S \ S_{in},

\[
Enter(o, s) := \bigvee \{ \text{Fire}(o, t) \mid t ∈ T \land \text{target}(t) = s \} \lor \bigvee \{ \text{Breakin}(o, r) \mid r ∈ \text{children}(s) \},
\]

and for each region r ∈ \mathcal{R},

\[
\text{Breakin}(o, r) := \bigvee \{ \text{Enter}(o, s) \mid s ∈ \text{children}(r) \setminus \text{S_{in}} \} \land \neg \bigvee \{ \text{Fire}(o, t) \mid t ∈ T \land \text{container}(t) = r \}.
\]

An initial pseudostate s_{in} ∈ S_{in} is entered if its containing state is entered but its containing region is not broken in:

\[
Enter(o, s_{in}) := \text{Enter}(o, \text{container}^2(s_{in})) \land \neg \text{Breakin}(o, \text{container}(s_{in})).
\]

As a special case, \text{Enter}(o, s_{in}) := false if s_{in} ∈ S_{in} ∩ \text{children}(top). For each s ∈ S, the value of \text{Exit}(o, s) is true iff s or one of its ancestors is being exited by a transition, defined by

\[
\text{Exit}(o, s) := \text{Exit}(o, \text{container}^2(s)) \lor \bigvee X_o(s),
\]

where \text{X}_o(s) is the set consisting of each \text{Fire}(o, t) such that t ∈ T and s ∈ \text{children}(\text{container}(t)) and \text{source}(t) ∈ \{ s \} ∪ \text{descendants}(s). Intuitively, \text{Fire}(o, t) is in the set \text{X}_o(s) iff s is the outermost state vertex exited when t is fired. For example in Fig. 1, \text{X}_{o(A_2)} = \{ \text{Fire}(o, t_{10}), \text{Fire}(o, t_{11}) \} and \text{X}_{o(B_2)} = \{ \text{Fire}(o, t_8) \}.

To avoid firing several transitions from the same state at the same time, for each s ∈ S the constraint

\[
\text{AtMostOne} \left( \{ \text{Exit}(o, \text{container}^2(s)) \} \cup X_o(s) \right)
\]

is added that allows at most one of the disjuncts in (19) to be true. A predicate of the form \text{AtMostOne}(P) evaluates to \text{true} if and only if zero or one of the predicates in set P evaluates to \text{true}. This can be expressed with \mathcal{O}(|P|) binary Boolean connectives. In (19) and (20), the term \text{Exit}(o, \text{container}^2(s)) is omitted if \text{container}(s) = top.

Quiescing. We associate with each completion sensitive state s ∈ S_{CS}, an input variable \text{Quiesce}(o, s) that indicates whether s is becoming quiescent. The state s becomes quiescent when \text{Quiesce}(o, s) is true, and it remains quiescent until it is exited:

\[
\text{next}(\text{Quiescent}(o, s)) ⇔ \neg \text{Exit}(o, s) \land (\text{Quiesce}(o, s) \lor \text{Quiescent}(o, s)).
\]

The following constraint ensures that quiescing can only happen when (i) there are no active pseudostates and (ii) the state is ready to consume its completion event but none of the outgoing completion transitions is enabled:

\[
\text{Quiesce}(o, s) \Rightarrow \neg \text{InCT}(o) \land \text{CEReady}(o, s) \land \neg \bigvee \{ \text{EvalGuard}(o, t) \mid t ∈ R_o(s, \tau) \}
\]

(22)

with R_o(s, \tau) := \{ t ∈ T \mid \text{source}(t) = s \land \text{triggersig}(t) = \tau \}.

Together with (8) this ensures that quiescing can only happen in an RTC step: \text{Quiesce}(o, s) implies \text{InRTC}(o).

Deferring. For each signal \text{sig} ∈ \text{Sigs} and each simple or composite state s ∈ S_{si} ∪ S_{co}, the function

\[
\text{Deferring}(o, v, \text{sig}) := \begin{cases} \text{Active}(o, s) & \text{if sig ∈ defers}(s), \\ \text{false} & \text{otherwise} \end{cases}
\]

(23)

is true iff s is active and can defer the signal \text{sig}. Define

\[
\text{Deferring}(o, v, \text{sig}) := \text{false} \quad \text{if} \quad v ∈ S_{in} ∪ S_{ch} ∪ S_{n} ∪ \mathcal{R}.
\]

Now for each v ∈ S ∪ \mathcal{R},

\[
\text{DescDeferring}(o, v, \text{sig}) := \bigvee \{ \text{Deferring}(o, v', \text{sig}) \lor \text{DescDeferring}(o, v', \text{sig}) \mid v' ∈ \text{children}(v) \}
\]

(24)

is true iff v has an active descendant that can defer \text{sig}.

The object o defers the first message in its input queue if the message is dispatched, there is an active deferring state, and no transition is consuming the message:

\[
\text{Defer}(o, \text{sig}) := \text{DescDeferring}(o, \text{top}, \text{sig}) \land \neg \text{DescFeasible}(o, \text{top}, \text{sig}) \land \text{Dispatch}(o) = \text{sig},
\]

\[
\text{QDefer}(o) := \bigvee \{ \text{Defer}(o, \text{sig}) \mid \text{sig} ∈ \text{Sigs} \}
\]

Note that \text{QDefer}(o) implies \text{Dispatch}(o) \neq \text{none} and thus (9) ensures that deferring can only happen in a stable state configuration, i.e. \text{QDefer}(o) implies \neg \text{InRTC}(o).

Putting it All Together. The constraints so far do not prevent simultaneous firing of completion transitions or quiescing of states. This is fixed by the constraint

\[
\text{AtMostOne} \left( C_o ∪ Q_o \right),
\]

(25)

where \text{C}_o = \{ \text{Fire}(o, t) \mid t ∈ T \land \text{triggersig}(t) = \tau \} and \text{Q}_o = \{ \text{Quiesce}(o, s) \mid s ∈ S_{CS} \}.

When examining the system as a whole, an object is scheduled if it is consuming a message, firing a completion transition, or quiescing a state. The object is ready for execution if one of these occurrences is enabled. The system is in a deadlock if no object is ready. These are formalized below.

\[
\text{Scheduled}(o) := (\text{Dispatch}(o) \neq \text{none}) \lor \bigvee (C_o ∪ Q_o),
\]

\[
\text{Ready}(o) := (\text{CurrentMsg}(o) \neq \text{none}) \lor \text{InRTC}(o),
\]

\[
\text{Deadlock()} := \neg \bigvee \{ \text{Ready}(o) \mid o ∈ O \}.
\]
The functions needed by input queue encoding, presented in the previous section, are defined as follows:

\[
QFlush(o) := \bigvee \{ \text{DescFeasible}(o, \text{top}, \text{sig}) \land (\text{Dispatch}(o) = \text{sig}) \mid \text{sig} \in Sigs \},
\]

\[
QRem(o) := (\text{Dispatch}(o) \neq \text{none}) \land \neg QDefer(o).
\]

To obtain an interleaving execution semantics, we must constrain that exactly one object is scheduled at a time:

\[
\bigvee \{ \text{Scheduled}(o) \mid o \in O \}, \text{ and } \quad (26)
\]

\[
\text{AtMostOne} (\{ \text{Scheduled}(o) \mid o \in O \}). \quad (27)
\]

Using this transition relation encoding, we can now use symbolic model checking tools such as NuSMV [6] to check properties of UML systems. For example, to check that a deadlock cannot be reached, we use a model checker to check that \( \neg \text{Deadlock}() \) is an invariant in all executions.

### 3.4. Analysis

Assuming that the state variables represent a bounded global configuration and all constraints are satisfied, there is by (26) and (27) a unique object \( o \) such that \( \text{Scheduled}(o) \) holds. There are five mutually exclusive cases: (i) \( QFlush(o) \) is true, consequently \( Q\text{Rem}(o) \) is true, there is a unique \( t \in T \) such that \( \text{Fire}(o, t) \) is true, and \( \text{triggersig}(t) = \text{Dispatch}(o) \neq \text{none} \). This corresponds to the step \( \langle gc, \langle o, t \rangle, gc' \rangle \in \Delta \). (ii) \( Q\text{Defer}(o) \) is true, corresponding to \( \langle gc, \langle o, \text{DEFER} \rangle, gc' \rangle \in \Delta \). (iii) \( Q\text{Flush}(o) \) and \( Q\text{Defer}(o) \) are both false and \( \text{Dispatch}(o) \neq \text{none} \), which corresponds to implicit consumption \( \langle gc, \langle o, \text{IMPCONS} \rangle, gc' \rangle \in \Delta \). (iv) \( \text{Dispatch}(o) = \text{none} \) and \( \text{Fire}(o, t) \) is true for some \( t \), in which case \( \text{triggersig}(t) = \tau \), corresponding to \( \langle gc, \langle o, t \rangle, gc' \rangle \in \Delta \). (v) \( \text{Dispatch}(o) = \text{none} \) and \( \text{Quiesce}(o, s) \) is true for a state \( s \in S_{CS} \), corresponding to the quiescing step \( \langle gc, \langle o, \text{QUIESCE}_l \rangle, gc' \rangle \in \Delta \).

The size of the encoding is linear in the size \( |M| \) of the input model, which includes the definition of all signals, state vertices, transitions, and deferrable signals. We assume that all trivial definitions have been eliminated, i.e. those derived functions that are vacuously \text{true} or \text{false} or equal to another function or variable, and constraints that are vacuously \text{true}. The queue encoding of Sect. 3.2 for a single object has size \( O(q\text{SIZE} \cdot W) \), where \( W \) is the maximum bit width of a message. Consider the control logic encoding of Sect. 3.3 for an object \( o \). The definitions (16), (17), and (19) are designed so that each variable \( \text{Fire}(o, t) \) appears in exactly one definition of each kind, so these are \( O(|S| + |T|) \) in size. The total size of nontrivial definitions of the form (12) and (13), summed over all \( s \in S \) and \( \text{sig} \in Sigs \), is \( O(|T|) \). The total size of (23) and (24) is \( O(\Sigma_{s \in S} [\text{defers}(s)]) \). Other definitions are \( O(|S| + |T| + |Sigs|) \) in size. Summing up, the total size of the encoding of a model without data is \( O(|O|q\text{SIZE} \cdot W + |M|) \).

### 3.5. Preliminary Evaluation

We have implemented the symbolic encoding described above. The implementation\(^1\) reads a UML model, stored in the XMI file format supported by the open-source metamodeling tool Coral [1], and translates it to the input language of the NuSMV symbolic model checking tool [6]. For the experiments here we use NuSMV version 2.4.3.

To evaluate whether allowing hierarchy in the encoding is helpful, we use a variant of the TV model in [18] that is one of the standard example systems in UML model checking literature. We try to measure only the hierarchy aspect by model checking a deadlock-freedom property of a modified model in which the property does not hold—the TV stops working after it has been switched off 50 times. The symbolic encoding is evaluated on both the original hierarchical model and the corresponding flattened model; the original state machine has 12 state vertices and 13 transitions while the flattened state machine has 11 state vertices and 25 transitions. The results in Table 1 show the minimum and maximum CPU times of ten runs of (i) a state-of-the-art incremental BMC algorithm [15, 5] when ZChaff (version 64bit.2007.3.12) [19] and Minisat2 (version 061208) [12] are applied as the SAT solver, and (ii) the basic BDD-based invariant checking algorithm. For this model it is clearly beneficial not to flatten the hierarchy as a preprocessing step. Our hypothesis for this behavior is that because flattening increases the number of transitions in the model, it also increases the search space of the SAT solver.

Furthermore, the experiments in [11] compare explicit-state model checking using Spin to BMC using the queue and control logic encoding presented here. The results indicate that symbolic model checking can complement explicit-state model checking even when analyzing asynchronous protocol models. Many symbolic model checking techniques also have the favorable feature that when a property violation is found, the returned counterexample execution

\(^1\)Available at http://www.tcs.hut.fi/Research/Logic/SMUML.shtml

| Table 1. Symbolic model checking of a hierarchical vs. flattened TV model. |
|----------------|----------------|----------------|
|                | BMC ZChaff     | BMC Minisat    | BDD Invar     |
|                | time           | time           | time          |
| original       | 16.49–54.60    | 29.29–90.00    | 0.63–0.67     |
| flattened      | 30.12–77.94    | 25.16–300.63   | 1.06–1.09     |
|                |                |                |               |
tion is short (or even of minimal length), making it simpler for the user to analyze it.

Obviously, a more thorough evaluation of efficiency and bottlenecks in symbolic model checking of asynchronous message passing UML systems is needed.

4. Conclusions

In this paper we have defined a semantics and a compact symbolic encoding for a class of UML models composed of asynchronously executing, message passing hierarchical state machines. This enables the use of state-of-the-art symbolic model checking techniques for the analysis of such models. Experimental results indicate that it may be beneficial to maintain the state machine hierarchy in the encoding instead of flattening the state machine before encoding.

The encoding presented in this paper uses the standard interleaving semantics, i.e. at most one object can fire at most one transition during one time step. It is also possible to extend the encoding to use so-called @-step semantics [22] so that independent transitions in several objects can be executed at the same time step; see [11] for such an encoding for non-hierarchical state machines.

There are probably many ways to optimize the state machines or the encoding to make symbolic model checking more efficient. For instance, transitions leaving the initial state of a region could in many cases be eliminated in the encoding phase. Furthermore, in order to analyze systems with more data attributes, reduction techniques such as slicing and data abstraction should also be applied.

References