Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 23
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 23
Espoo 2007 HUT-TCS-B23

SYMBOLIC MODEL CHECKING OF HIERARCHICAL UML STATE MACHINES

Jori Dubrovin and Tommi Junttila

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 23
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 23
Espoo 2007 HUT-TCS-B23

SYMBOLIC MODEL CHECKING OF HIERARCHICAL UML STATE MACHINES

Jori Dubrovin and Tommi Junttila

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK, FINLAND
Tel. +358 9 451 1
Fax. +358 9 451 3369
E-mail: lab@tcs.tkk.fi
URL: http://www.tcs.tkk.fi/

ISBN 978-951-22-9191-5
ISSN 0783-540X

Multiprint Oy
Espoo 2007

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.

Keywords: UML semantics, UML state machine, symbolic model checking, verification

CONTENTS

1 Introduction **1**
1.1 Related Work 2

2 A UML Subset and its Semantics **2**
2.1 Types, Signals, and Classes 3
2.2 State Machines 3
2.3 Systems and State Spaces 6

3 Encoding System Behavior **9**
3.1 State Variables 9
3.2 Queues and Messages 10
3.3 Control Logic of State Machines 11
3.4 Analysis 15
3.5 Preliminary Evaluation 16

4 Conclusions **17**

References **17**

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 such 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 [22]. This directly enables one to use state-of-the-art symbolic model checking techniques such as BDD-based [17] and bounded model checking [4] to 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 [22].

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.

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 number 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, making it explicit what a single step of execution is. Such a transition relation is required for applying symbolic model checking.

A semantics for hierarchical UML state machines is presented in [14], but signal parameters and data manipulation are not discussed. In [23], 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 do not know of symbolic encodings based on [13].

Furthermore, none of [13, 9, 14, 20, 23] seem to handle completion events in full accordance with [22]; 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 report 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 $\mathcal{T}$ of *types* is assumed, each type $T \in \mathcal{T}$ being associated with a non-empty *domain* set $dom(T)$. In particular, the Boolean type $\mathbb{B}$ with $dom(\mathbb{B}) = \{\mathbf{false}, \mathbf{true}\}$ belongs to $\mathcal{T}$. A *typed variable* is a name x associated with a type $type(x) \in \mathcal{T}$. The guards and effects appearing in state machines are expressed with a strongly typed *action language* $\mathcal{L}$ over the types; $\mathcal{L}_{\mathbb{B}} \subset \mathcal{L}$ denotes the set of side-effect free Boolean valued expressions and $\mathcal{L}_{\mathrm{Stmt}} \subset \mathcal{L}$ 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 $append(a, x) = \langle a_1, \ldots, a_k, x \rangle$, $dequeue(a) = \langle a_2, \ldots, a_k \rangle$ (undefined if $k = 0$), and $concat(a, b) = \langle a_1, \ldots, a_k, b_1, \ldots, b_l \rangle$.

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

A *class* is a pair $C = \langle attrs, sm \rangle$, where $Attrs(C) = attrs$ is a finite set of typed variables called *attributes* and $SM(C) = sm$ 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

$$sm = \langle \mathcal{S}, \mathcal{R}, top, container, \mathcal{T}, defers \rangle,$$

where

- $\mathcal{S}$ is a finite set of *state vertices* partitioned into *simple states* $\mathcal{S}_{si}$, *composite states* $\mathcal{S}_{co}$, *final states* $\mathcal{S}_{fi}$, *initial pseudostates* $\mathcal{S}_{in}$, and *choice pseudostates* $\mathcal{S}_{ch}$;
- $\mathcal{R}$ is a finite set of *regions* (disjoint from $\mathcal{S}$);
- $top \in \mathcal{R}$ is the unique *top region*;
- $container : (\mathcal{S} \cup \mathcal{R} \setminus \{top\}) \to (\mathcal{S} \cup \mathcal{R})$ describes the *state hierarchy* of the state machine;
- $\mathcal{T}$ is a finite set of *transitions*; and

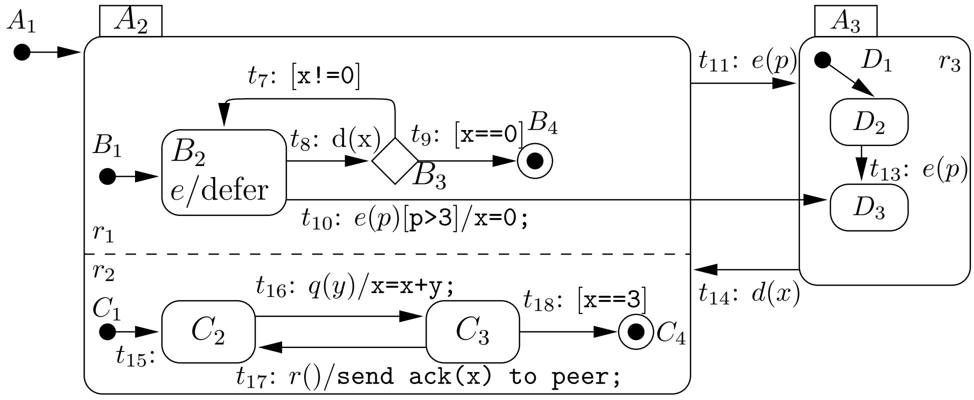

Figure 1: A UML state machine.

- *defers* : $(\mathcal{S}_{si} \cup \mathcal{S}_{co}) \to 2^{Sigs}$ assigns each state a (possibly empty) set of *deferrable signals*.

For each state vertex or region $v \in \mathcal{S} \cup \mathcal{R}$ of the state machine, define the functions $children(v) = \{v' \in \mathcal{S} \cup \mathcal{R} \setminus \{top\} \mid container(v') = v\}$ and $descendants(v) = \big\{v' \in \mathcal{S} \cup \mathcal{R} \setminus \{top\} \mid \exists i > 0 : container^i(v') = v\big\}$. 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. The state hierarchy must be a connected tree, i.e. $descendants(top) = \mathcal{S} \cup \mathcal{R} \setminus \{top\}$ must hold. Furthermore, each region must contain exactly one initial state, i.e. $\forall r \in \mathcal{R} : |children(r) \cap \mathcal{S}_{in}| = 1$, and each composite state at least one region, i.e. $\forall s \in \mathcal{S}_{co} : children(s) \neq \emptyset$. If a composite state contains more than one region, then it is called *concurrent*. Two state vertices $s_1, s_2 \in \mathcal{S}$ are *orthogonal*, denoted $s_1 \perp s_2$, if there are distinct regions $r_1, r_2 \in \mathcal{R}$, $r_1 \neq r_2$ such that $container(r_1) = container(r_2)$, $s_1 \in descendants(r_1)$, and $s_2 \in descendants(r_2)$. A set $S \subseteq \mathcal{S}$ of states is *consistent* iff for any two distinct state vertices $s_1, s_2 \in S$ either $s_1 \perp s_2$, $s_1 \in descendants(s_2)$, or $s_2 \in descendants(s_1)$.

As an example, consider the state machine in Fig. 1. A_2 is a concurrent composite state with $container(A_2) = top$ and $children(A_2) = \{r_1, r_2\}$, where r_1 and r_2 are regions. B_2 is a simple state with $defers(B_2) = \{e\}$ and $container(B_2) = r_1$. 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 $\mathcal{T}$ of transitions is a tuple

$$\langle s, \sigma, g, e, s' \rangle \in (\mathcal{S} \setminus \mathcal{S}_{fi}) \times (Trigs \cup \{\tau\}) \times \mathcal{L}_{\mathbb{B}} \times \mathcal{L}_{\mathrm{Stmt}} \times (\mathcal{S} \setminus \mathcal{S}_{in}).$$

We define $source(t) = s$, $guard(t) = g$, $effect(t) = e$, and $target(t) = s'$. The container $container(t)$ of t is the smallest (w.r.t. the partial order induced by $container$) region such that $\{s, s'\} \subseteq descendants(r)$. If $\sigma = \tau$, we say that t is a *completion transition* and define $triggersig(t) = \tau$. Otherwise, $\sigma = sig(\ldots)$ and we define $triggersig(t) = sig$. In UML only completion transitions can leave pseudostates: $source(t) \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}$ implies $triggersig(t) = \tau$. We require that transitions originating from orthogonal states are not triggered by the same signal: for all $t_1, t_2 \in \mathcal{T}$, if

$triggersig(t_1) = triggersig(t_2) \neq \tau$, then $source(t_1) \perp source(t_2)$ must not hold.

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

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

$$sc = \langle A, Q\rangle,$$

where the set A of *active state vertices* is a maximal consistent subset of $\mathcal{S}$ 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 [22, p. 659] that the guards of completion transitions will not be evaluated again without re-entering the state. 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* $\mathcal{S}_{CS} = \{s \in \mathcal{S}_{si} \cup \mathcal{S}_{co} \mid \exists t \in \mathcal{T} \text{ s.t. } source(t) = s \text{ and } triggersig(t) = \tau\}$. Thus, Q is always a subset of $A \cap \mathcal{S}_{CS}$. We say that a state $s \in \mathcal{S}_{CS}$ 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: $s \in \mathcal{S}_{si}$, or (ii) a composite state with all its regions in final states: $s \in \mathcal{S}_{co}$ and $\forall s' \in A : container^2(s') = s \Rightarrow s' \in \mathcal{S}_{fi}$.

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

- *in compound transition*, denoted by $inct(sc)$, if it contains an active pseudostate: $A \cap (\mathcal{S}_{in} \cup \mathcal{S}_{ch}) \neq \emptyset$,
- *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 $\langle\{A_2, B_2, C_3\}, \{C_3\}\rangle$ is a stable state configuration, while the state configuration $\langle\{A_2, B_3, C_3\}, \{C_3\}\rangle$ is in compound transition. The state configuration $\langle\{A_2, B_2, C_3\}, \emptyset\rangle$ 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 $s \in \mathcal{S}$, denoted by $dec(s)$, is the smallest maximal consistent subset of $\mathcal{S}_{co} \cup \mathcal{S}_{in} \cup \{s\}$ such that $s \in dec(s)$. In the state machine in Fig. 1, $dec(A_3) = \{A_3, D_1\}$, $dec(D_3) = \{A_3, D_3\}$, $dec(C_3) = \{A_2, B_1, C_3\}$, and $dec(A_2) = \{A_2, B_1, C_1\}$.

Given a state configuration $sc = \langle A, Q \rangle$ and a transition $t \in \mathcal{T}$ with $source(t) \in A$, the *t-successor* of sc is

$$succ\text{-}conf(sc, t) = \langle A', Q' \rangle,$$

where $A' = (A \setminus D) \cup (dec(target(t)) \cap D)$, $Q' = Q \setminus D$, and D is a shorthand for $descendants(container(t))$. In the state machine in Fig. 1, the t_{10}-successor of the state configuration $sc = \langle \{A_2, B_2, C_3\}, \{C_3\} \rangle$ is $\langle \{A_3, D_3\}, \emptyset \rangle$ while the t_{11}-successor of sc is $\langle \{A_3, D_1\}, \emptyset \rangle$.

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 $Class(o)$ that it is an instance of. A *global configuration* of the system is a tuple

$$gc = \langle stateconf_{gc}, attrvals_{gc}, inputq_{gc}, deferq_{gc} \rangle,$$

where

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

For convenience, let $StateConf(gc, o) = stateconf_{gc}(o)$, $AttrVal(gc, o, x) = attrvals_{gc}(o)(x)$, $InputQ(gc, o) = inputq_{gc}(o)$, and let $DeferQ(gc, o) = deferq_{gc}(o)$. The set of all global configurations is denoted by $\mathcal{GC}$.

Given a side-effect free Boolean expression ϕ in $\mathcal{L}_{\mathbb{B}}$, $eval(gc, o, \phi)$ evaluates it in the context of a global configuration gc and an object o, and returns **false** or **true**. Given a statement γ in $\mathcal{L}_{\text{Stmt}}$, $exec(gc, o, \gamma)$ 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 configuration is not modified: $StateConf(gc', o') = StateConf(gc, o')$, (ii) messages cannot be removed from the input queue: $InputQ(gc, o')$ is a prefix of $InputQ(gc', o')$, and (iii) the deferred queue is not modified: $DeferQ(gc', o') = DeferQ(gc, o')$.

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

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

- **Signal Triggered Transitions.** Let $t = \langle s, sig(x_1, \ldots, x_k), g, e, s' \rangle \in \mathcal{T}$ be a transition. The *transition instance* $\langle o, t \rangle$ is *enabled in gc*, denoted by $enabled(gc, \langle o, t \rangle)$, if

  - the state configuration is stable: $stable(sc)$,
  - the source state is active: $s \in A$,
  - $InputQ(gc, o) = \langle sig[v_1, \ldots, v_k], \ldots \rangle$,
  - $eval(gc^\star, o, g) = \mathbf{true}$, where $gc^\star$ is equal to gc except that in o, the message $sig[v_1, \ldots, v_k]$ has been received: $InputQ(gc^\star, o) = dequeue(InputQ(gc, o))$ and $\forall 1 \leq i \leq k : AttrVal(gc^\star, o, x_i) = v_i$ (the formal parameters in the signal trigger are required to be attributes in the class of o),
  - no prioritized transition is enabled [22, p. 608], i.e. there is no enabled transition deeper in the hierarchy: $\nexists t' \in \mathcal{T} : source(t') \in descendants(s) \cap A \wedge enabled(gc, \langle o, t' \rangle)$, and
  - the message is not being deferred by a state at a deeper level: $\nexists s'' \in descendants(s) \cap (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \cap A : sig \in defers(s'')$.

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

- **Deferring.** If no transition instance is enabled, then the first message in the input queue can be deferred. Formally, the *deferring instance* $\langle o, \text{DEFER} \rangle$ is *enabled* in *gc*, denoted by $enabled(gc, \langle o, \text{DEFER} \rangle)$, if

  - the state configuration is stable: $stable(sc)$,
  - $InputQ(gc, o) = \langle sig[v_1, \ldots, v_k], \ldots \rangle$,
  - $\nexists t \in \mathcal{T} : enabled(gc, \langle o, t \rangle)$, and
  - there is an active state deferring the message: $\exists s \in (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \cap A : sig \in defers(s)$.

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

- **Implicit consumption.** If the first message in the input queue is not consumed by a transition or deferred, it can be implicitly consumed. Formally, $enabled(gc, \langle o, \text{IMPCONS}\rangle)$ holds if

  – the state configuration is stable: $stable(sc)$,

  – $InputQ(gc, o) = \langle sig[v_1, \ldots, v_k], \ldots\rangle$,

  – $\nexists t \in \mathcal{T} : enabled(gc, \langle o, t\rangle)$, and

  – $enabled(gc, \langle o, \text{DEFER}\rangle)$ does not hold.

  If $enabled(gc, \langle o, \text{IMPCONS}\rangle)$ holds, then $\langle gc, \langle o, \text{IMPCONS}\rangle, gc'\rangle \in \Delta$, where gc' is equal to gc except that the first message has been consumed: $InputQ(gc', o) = dequeue(InputQ(gc, o))$.

- **Completion transitions.** Implicit completion events are consumed until a stable state configuration is reached. Formally, for each transition $t = \langle s, \sigma, g, e, s'\rangle \in \mathcal{T}$ such that $\sigma = \tau$, the *completion transition instance* $\langle o, t\rangle$ is *enabled in gc*, denoted by $enabled(gc, \langle o, t\rangle)$, if

  – the source is active: $s \in A$,

  – either (i) the source is a pseudostate: $s \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}$ 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: $\neg inct(sc) \wedge inrtc(sc) \wedge s \in \mathcal{S}_{si} \cup \mathcal{S}_{co} \wedge ceready(sc, s)$, and

  – the guard condition holds: $eval(gc, o, g) = \textbf{true}$.

  If $enabled(gc, \langle o, t\rangle)$ holds, then $\langle gc, \langle o, t\rangle, gc'\rangle \in \Delta$, where gc' is equal to $exec(gc, o, e)$ except that the state configuration is updated: $StateConf(gc', o) = succ\text{-}conf(sc, t)$.

- **Quiescing.** If a completion sensitive state $s \in \mathcal{S}_{CS}$ 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, $enabled(gc, \langle o, \text{QUIESCE}_s\rangle)$ holds if

  – the state configuration is in RTC but not in compound transition: $inrtc(sc) \wedge \neg inct(sc)$,

  – the state is active and ready to consume its completion event: $s \in \mathcal{S}_{CS} \cap A \wedge ceready(sc, s)$, and

  – $\nexists t \in \mathcal{T} : source(t) = s \wedge triggersig(t) = \tau \wedge enabled(gc, \langle o, t\rangle)$.

  If $enabled(gc, \langle o, \text{QUIESCE}_s\rangle)$ holds, then $\langle gc, \langle o, \text{QUIESCE}_s\rangle, gc'\rangle \in \Delta$, where gc' is equal to gc except that state s has become quiescent: $StateConf(gc', o) = \langle A, Q \cup \{s\}\rangle$.

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 $dom(T)$ 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 QSIZE. 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, we define an encoding such that, given a valuation for state variables that represents a bounded global configuration gc and an arbitrary valuation for next-state variables, 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 $\langle gc, a, gc' \rangle \in \Delta$ 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 $\langle A, Q \rangle = \mathit{StateConf}(gc, o)$, and let the numbers d and n and messages $M_0, \ldots, M_{n-1}$ be such that the deferred queue of o is $\langle M_0, \ldots, M_{d-1} \rangle = \mathit{DeferQ}(gc, o)$ and the input queue is $\langle M_d, \ldots, M_{n-1} \rangle = \mathit{InputQ}(gc, o)$. Because gc is bounded, we have $0 \leq d \leq n \leq \mathrm{QSIZE}$. For each object, the set of state variables contains five kinds of elements, with values derived from gc as follows.

- $\mathsf{Active}(o, s)$, where s is a state vertex in the state machine of o, is true if and only if s is active, i.e. $s \in A$.

- $\mathsf{Quiescent}(o, s)$, where s is a completion sensitive state in the state machine of o, is true if and only if $s \in Q$.

- $\mathsf{AttrVal}(o, x)$, where $x \in \mathit{Attrs}(\mathit{Class}(o))$, has domain $dom(x)$ and value $\mathit{AttrVal}(gc, o, x)$. We omit all details of data handling and do not reference these variables in the encoding.

- $\mathsf{Queue}(o, k)$, where $0 \leq k < \mathrm{QSIZE}$, has domain $Msgs \cup \mathbf{none}$. Its value is M_k if $k < n$, and **none** otherwise. In other words, the sequence $\langle \mathsf{Queue}(o, 0), \ldots, \mathsf{Queue}(o, \mathrm{QSIZE} - 1) \rangle$ consists of the deferred queue, followed by the input queue, followed by zero or more **none** entries.

- $\mathsf{QPos}(o)$ has domain $\{0, 1, \ldots, \mathrm{QSIZE}\}$ and value d.

The corresponding next-state variables are denoted by $\mathit{next}\,(\mathsf{Active}(o, s))$, $\mathit{next}\,(\mathsf{Quiescent}(o, s))$, 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 $\mathsf{QNewMsg}(o)$ with values in $\mathit{Msgs} \cup \{\mathbf{none}\}$ evaluates to the message being sent to o during the step $\langle gc, a, gc' \rangle$, or **none** if no message is being sent. The definition of $\mathsf{QNewMsg}(o)$ depends on the action language statements, which are not considered here.

The function $\mathsf{CurrentMsg}(o)$ with domain $\mathit{Msgs} \cup \{\mathbf{none}\}$ contains the first message in the input queue, or **none** if the input queue is empty:

$$\mathsf{CurrentMsg}(o) \;:=\; \mathit{if} \begin{cases} \mathsf{QPos}(o) = 0 & : \; \mathsf{Queue}(o, 0) \\ \quad\vdots & \quad\quad\vdots \\ \mathsf{QPos}(o) = \text{QSIZE} - 1 & : \; \mathsf{Queue}(o, \text{QSIZE} - 1) \\ \mathit{else} & : \; \mathbf{none}. \end{cases} \tag{1}$$

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

$$(\mathsf{Dispatch}(o) = \mathit{sig}) \Rightarrow (\mathsf{CurrentMsg}(o) = \mathit{sig}[\ldots])\,. \tag{2}$$

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 $\mathsf{QRem}(o)$, (ii) moving the first message from the input queue to the deferred queue, triggered by $\mathsf{QDefer}(o)$, (iii) flushing the entire contents of the deferred queue to the input queue, triggered by $\mathsf{QFlush}(o)$, and (iv) adding the new message $\mathsf{QNewMsg}(o)$ 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 $\mathsf{QRem}(o)$, $\mathsf{QDefer}(o)$, and $\mathsf{QFlush}(o)$ are defined in Sect. 3.3.

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

$$\mathit{next}\,(\mathsf{QPos}(o)) = \mathit{if} \begin{cases} \mathsf{QFlush}(o) & : \; 0 \\ \mathsf{QDefer}(o) & : \; \mathsf{QPos}(o) + 1 \\ \mathit{else} & : \; \mathsf{QPos}(o). \end{cases} \tag{3}$$

Operation (i) removes the element at index $\mathsf{QPos}(o)$ from the queue, shifting the elements at $\mathsf{QPos}(o) + 1, \ldots, \text{QSIZE} - 1$ one position to the left. Operation (iv) adds the new message to the first free position, i.e. the position k such that $\mathsf{Queue}(o, k-1) \neq \mathbf{none}$ and $\mathsf{Queue}(o, k) = \mathbf{none}$. 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

$$\mathit{next}\,(\mathsf{Queue}(o, k)) \;=\; \mathit{if} \begin{cases} N_o(k) & : \; \mathsf{QNewMsg}(o) \\ \mathsf{QRem}(o) \wedge \mathsf{QPos}(o) \leq k & : \; \mathsf{Queue}(o, k+1) \\ \mathit{else} & : \; \mathsf{Queue}(o, k), \end{cases} \tag{4}$$

where

$$N_o(k) := (\mathsf{QRem}(o) \Leftrightarrow (\mathsf{Queue}(o,k) \neq \mathbf{none})) \wedge \\ (\mathsf{Queue}(o,k-1) \neq \mathbf{none}) \wedge (\mathsf{Queue}(o,k+1) = \mathbf{none})\,.$$

The boundaries are defined as we substitute **true** for both $\mathsf{Queue}(o, \mathrm{QSIZE}) = \mathbf{none}$ and $\mathsf{Queue}(o,-1) \neq \mathbf{none}$. To forbid transitions to global configurations that are not bounded, we prevent the queue from overflowing by setting the constraint

$$(\mathsf{QNewMsg}(o) \neq \mathbf{none}) \Rightarrow (\mathsf{Queue}(o, \mathrm{QSIZE}-1) = \mathbf{none}) \vee \mathsf{QRem}(o). \quad (5)$$

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 $\langle \mathcal{S}, \mathcal{R}, top, container, \mathcal{T}, defers \rangle$.

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

$$\mathsf{CEReady}(o,s) := \mathsf{Active}(o,s) \wedge \neg\mathsf{Quiescent}(o,s) \wedge \\ \bigwedge \{F(o,r) \mid r \in children(s)\}, \quad (6)$$

where $F(o,r) := \bigvee \{\mathsf{Active}(o,s') \mid s' \in children(r) \cap \mathcal{S}_{fi}\}$ 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 $\mathsf{InCT}(o)$ and $\mathsf{InRTC}(o)$ telling whether the state configuration is in compound transition or in RTC step, respectively:

$$\mathsf{InCT}(o) := \bigvee \{\mathsf{Active}(o,s) \mid s \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}\}, \text{ and} \quad (7)$$

$$\mathsf{InRTC}(o) := \mathsf{InCT}(o) \vee \bigvee \{\mathsf{CEReady}(o,s) \mid s \in \mathcal{S}_{CS}\}. \quad (8)$$

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

$$\mathsf{InRTC}(o) \Rightarrow (\mathsf{Dispatch}(o) = \mathbf{none})\,. \quad (9)$$

Enabledness Conditions for Transitions. We associate with each transition $t \in \mathcal{T}$ of the state machine an input variable $\mathsf{Fire}(o,t)$ 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

$$\mathsf{Fire}(o,t) \Rightarrow \mathsf{Active}(o, source(t)) \wedge \mathsf{EvalGuard}(o,t). \quad (10)$$

We assume that $\mathsf{EvalGuard}(o,t)$ encodes the function $eval(gc^\star, o, guard(t))$ (or $eval(gc, o, guard(t))$ 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:

$$\mathsf{Fire}(o,t) \Rightarrow \neg\mathsf{InCT}(o) \wedge \mathsf{CEReady}(o, \mathit{source}(t)). \tag{11}$$

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 we define an auxiliary function

$$\begin{aligned}&\mathsf{Feasible}(o,s,\mathit{sig}) := \mathsf{Active}(o,s) \wedge \\ &\quad \neg\mathsf{DescFeasible}(o,s,\mathit{sig}) \wedge \neg\mathsf{DescDeferring}(o,s,\mathit{sig}) \wedge \\ &\quad \bigvee \{\mathsf{EvalGuard}(o,t) \mid t \in R(s,\mathit{sig})\}\end{aligned} \tag{12}$$

with $R(s,\mathit{sig}) = \{t{\in}\mathcal{T} \mid \mathit{source}(t){=}s \wedge \mathit{triggersig}(t){=}\mathit{sig}\}$. $\mathsf{Feasible}(o, s, \mathit{sig})$ evaluates 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 $\mathsf{DescFeasible}(o,v,\mathit{sig})$ is defined simply by

$$\begin{aligned}\mathsf{DescFeasible}(o,v,\mathit{sig}) := \bigvee \big\{&\mathsf{Feasible}(o,v',\mathit{sig}) \vee \\ &\mathsf{DescFeasible}(o,v',\mathit{sig}) \mid v' \in \mathit{children}(v)\big\},\end{aligned} \tag{13}$$

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

$$\begin{aligned}&\bigvee \big\{\mathsf{Fire}(o,t) \mid t \in R_o(s,\mathit{sig})\big\} \Leftrightarrow \\ &\qquad (\mathsf{Dispatch}(o) = \mathit{sig}) \wedge \mathsf{Feasible}(o,s,\mathit{sig}).\end{aligned} \tag{14}$$

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 $s \in \mathcal{S}$ 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:

$$\mathit{next}\,(\mathsf{Active}(o,s)) \Leftrightarrow \mathsf{Enter}(o,s) \vee (\mathsf{Active}(o,s) \wedge \neg\mathsf{Exit}(o,s))\,. \tag{15}$$

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 $r \in \mathit{descendants}(\mathit{container}(t))$ and $\mathit{target}(t) \in \mathit{descendants}(r)$. In Fig. 1, the only such transition is t_{10}, which breaks in region r_3. For each vertex $s \in \mathcal{S} \setminus \mathcal{S}_{in}$,

$$\mathsf{Enter}(o, s) := \bigvee \{\mathsf{Fire}(o, t) \mid t \in \mathcal{T} \wedge \mathit{target}(t) = s\} \vee \bigvee \{\mathsf{BreakIn}(o, r) \mid r \in \mathit{children}(s)\}, \tag{16}$$

and for each region $r \in \mathcal{R}$,

$$\mathsf{BreakIn}(o, r) := \bigvee \{\mathsf{Enter}(o, s) \mid s \in \mathit{children}(r) \backslash \mathcal{S}_{in}\} \wedge \neg \bigvee \{\mathsf{Fire}(o, t) \mid t \in \mathcal{T} \wedge \mathit{container}(t) = r\}. \tag{17}$$

An initial pseudostate $s_{in} \in \mathcal{S}_{in}$ is entered if its containing state is entered but its containing region is not broken in:

$$\mathsf{Enter}(o, s_{in}) := \mathsf{Enter}(o, \mathit{container}^2(s_{in})) \wedge \neg\mathsf{BreakIn}(o, \mathit{container}(s_{in})). \tag{18}$$

As a special case, $\mathsf{Enter}(o, s_{in}) :=$ **false** if $s_{in} \in \mathcal{S}_{in} \cap \mathit{children}(\mathit{top})$. For each $s \in \mathcal{S}$, the value of $\mathsf{Exit}(o, s)$ is true iff s or one of its ancestors is being exited by a transition, defined by

$$\mathsf{Exit}(o, s) := \mathsf{Exit}(o, \mathit{container}^2(s)) \vee \bigvee X_o(s), \tag{19}$$

where $X_o(s)$ is the set consisting of each $\mathsf{Fire}(o, t)$ such that $t \in \mathcal{T}$ and $s \in \mathit{children}(\mathit{container}(t))$ and $\mathit{source}(t) \in \{s\} \cup \mathit{descendants}(s)$. Intuitively, $\mathsf{Fire}(o, t)$ is in the set $X_o(s)$ iff s is the outermost state vertex exited when t is fired. For example in Fig. 1, $X_o(A_2) = \{\mathsf{Fire}(o, t_{10}), \mathsf{Fire}(o, t_{11})\}$ and $X_o(B_2) = \{\mathsf{Fire}(o, t_8)\}$.

To avoid firing several transitions from the same state at the same time, for each $s \in \mathcal{S}$ the constraint

$$\mathit{AtMostOne}\left(\{\mathsf{Exit}(o, \mathit{container}^2(s))\} \cup X_o(s)\right) \tag{20}$$

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

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

$$\mathit{next}\,(\mathsf{Quiescent}(o, s)) \Leftrightarrow \neg\mathsf{Exit}(o, s) \wedge (\mathsf{Quiesce}(o, s) \vee \mathsf{Quiescent}(o, s))\,. \tag{21}$$

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:

$$\mathsf{Quiesce}(o, s) \Rightarrow \neg\mathsf{InCT}(o) \wedge \mathsf{CEReady}(o, s) \wedge \neg \bigvee \{\mathsf{EvalGuard}(o, t) \mid t \in R_\tau(s)\} \tag{22}$$

with $R_\tau(s) = \{t \in \mathcal{T} \mid \mathit{source}(t) = s \wedge \mathit{triggersig}(t) = \tau\}$. Together with (8) this ensures that quiescing can only happen in an RTC step: $\mathsf{Quiesce}(o, s)$ implies $\mathsf{InRTC}(o)$.

Deferring. For each signal $\mathit{sig} \in \mathit{Sigs}$ and each simple or composite state $s \in \mathcal{S}_{si} \cup \mathcal{S}_{co}$, the function

$$\mathsf{Deferring}(o, s, \mathit{sig}) := \begin{cases} \mathsf{Active}(o, s) & \text{if } \mathit{sig} \in \mathit{defers}(s), \\ \mathbf{false} & \text{otherwise} \end{cases} \tag{23}$$

is true iff s is active and can defer the signal sig. Define $\mathsf{Deferring}(o, v, \mathit{sig}) :=$ **false** if $v \in \mathcal{S}_{in} \cup \mathcal{S}_{ch} \cup \mathcal{S}_{fi} \cup \mathcal{R}$. Now for each $v \in \mathcal{S} \cup \mathcal{R}$,

$$\mathsf{DescDeferring}(o, v, \mathit{sig}) := \bigvee \big\{\mathsf{Deferring}(o, v', \mathit{sig}) \vee \mathsf{DescDeferring}(o, v', \mathit{sig}) \mid v' \in \mathit{children}(v)\big\} \tag{24}$$

is true iff v has an active descendant that can defer 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:

$$\begin{aligned} \mathsf{Defer}(o, \mathit{sig}) &:= \mathsf{DescDeferring}(o, \mathit{top}, \mathit{sig}) \wedge \\ &\qquad \neg\mathsf{DescFeasible}(o, \mathit{top}, \mathit{sig}) \wedge (\mathsf{Dispatch}(o) = \mathit{sig}), \\ \mathsf{QDefer}(o) &:= \bigvee \{\mathsf{Defer}(o, \mathit{sig}) \mid \mathit{sig} \in \mathit{Sigs}\}. \end{aligned}$$

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

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

$$\mathit{AtMostOne}\,(C_o \cup Q_o)\,, \tag{25}$$

where $C_o = \{\mathsf{Fire}(o, t) \mid t \in \mathcal{T} \wedge \mathit{triggersig}(t) = \tau\}$ corresponds to o firing a completion transition and $Q_o = \{\mathsf{Quiesce}(o, s) \mid s \in \mathcal{S}_{CS}\}$ corresponds to o quiescing a state.

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.

$$\begin{aligned} \mathsf{Scheduled}(o) &:= (\mathsf{Dispatch}(o) \neq \mathbf{none}) \vee \bigvee (C_o \cup Q_o)\,, \\ \mathsf{Ready}(o) &:= (\mathsf{CurrentMsg}(o) \neq \mathbf{none}) \vee \mathsf{InRTC}(o), \\ \mathsf{Deadlock}() &:= \neg \bigvee \{\mathsf{Ready}(o) \mid o \in O\}. \end{aligned}$$

The functions needed by input queue encoding, presented in the previous section, are defined as follows:

$$\begin{aligned}\mathsf{QFlush}(o) &:= \bigvee \{\mathsf{DescFeasible}(o, top, sig) \wedge \\ &\qquad (\mathsf{Dispatch}(o) = sig) \mid sig \in Sigs\},\\ \mathsf{QRem}(o) &:= (\mathsf{Dispatch}(o) \neq \mathbf{none}) \wedge \neg\mathsf{QDefer}(o).\end{aligned}$$

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

$$\bigvee \{\mathsf{Scheduled}(o) \mid o \in O\}, \text{ and} \tag{26}$$

$$AtMostOne\left(\{\mathsf{Scheduled}(o) \mid o \in O\}\right). \tag{27}$$

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 $\neg\mathsf{Deadlock}()$ 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 $\mathsf{Scheduled}(o)$ holds. There are five mutually exclusive cases: (i) $\mathsf{QFlush}(o)$ is true, consequently $\mathsf{QRem}(o)$ is true, there is a unique $t \in \mathcal{T}$ such that $\mathsf{Fire}(o, t)$ is true, and $triggersig(t) = \mathsf{Dispatch}(o) \neq \mathbf{none}$. This corresponds to the step $\langle gc, \langle o, t\rangle, gc'\rangle \in \Delta$. (ii) $\mathsf{QDefer}(o)$ is true, corresponding to $\langle gc, \langle o, \text{DEFER}\rangle, gc'\rangle \in \Delta$. (iii) $\mathsf{QFlush}(o)$ and $\mathsf{QDefer}(o)$ are both false and $\mathsf{Dispatch}(o) \neq \mathbf{none}$, which corresponds to implicit consumption $\langle gc, \langle o, \text{IMPCONS}\rangle, gc'\rangle \in \Delta$. (iv) $\mathsf{Dispatch}(o) = \mathbf{none}$ and $\mathsf{Fire}(o, t)$ is true for some t, in which case $triggersig(t) = \tau$, corresponding to $\langle gc, \langle o, t\rangle, gc'\rangle \in \Delta$. (v) $\mathsf{Dispatch}(o) = \mathbf{none}$ and $\mathsf{Quiesce}(o, s)$ is true for a state $s \in \mathcal{S}_{CS}$, corresponding to the quiescing step $\langle gc, \langle o, \text{QUIESCE}_s\rangle, gc'\rangle \in \Delta$.

The size of the encoding is linear in the size $|\mathcal{M}|$ 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 **true** or **false** or equal to another function or variable, and constraints that are vacuously **true**. The queue encoding of Sect. 3.2 for a single object has size $\mathcal{O}(\text{QSIZE} \cdot W)$, 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 $\mathsf{Fire}(o, t)$ appears in exactly one definition of each kind, so these are $\mathcal{O}(|\mathcal{S}| + |\mathcal{T}|)$ in size. The total size of nontrivial definitions of the form (12) and (13), summed over all $s \in \mathcal{S}$ and $sig \in Sigs$, is $\mathcal{O}(|\mathcal{T}|)$. The total size of (23) and (24) is $\mathcal{O}(\Sigma_{s\in\mathcal{S}}|defers(s)|)$. Other definitions are $\mathcal{O}(|\mathcal{S}| + |\mathcal{T}| + |Sigs|)$ in size. Summing up, the total size of the encoding of a model without data is $\mathcal{O}(|O|(\text{QSIZE} \cdot W + |\mathcal{M}|))$.

Table 1: Symbolic model checking of a hierarchical vs. flattened TV model.

	BMC ZChaff time	BMC MiniSat time	BDD Invar 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

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 meta-modeling 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. Symbolic model checking also has the favorable feature that when a property violation is found, the returned counterexample execution is always short, making it simpler for the user to analyze.

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

[1] Available at `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`

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 [21] 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.

Acknowledgements

This work has been financially supported by Tekes (Finnish Funding Agency for Technology and Innovation), Nokia, Conformiq, Mipro, and Helsinki Graduate School in Computer Science and Engineering.

REFERENCES

[1] Marcus Alanen and Ivan Porres. Coral: A metamodel kernel for transformation engines. In *Proc. Second European Workshop on Model Driven Architecture (MDA)*, number 17-04 in Tech. Report, pages 165–170. Computing Laboratory, Univ. of Kent, Sep 2004.

[2] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive verification of UML state machines. In *ICFEM'04*, volume 3308 of *LNCS*, pages 343–448. Springer, 2004.

[3] Purandar Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. Computing Research Repository, arXiv:cs/0407038v1 [cs.SE], 2004.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[5] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5), 2006.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *CAV'02*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[7] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[8] Michelle L. Crane and Juergen 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.

[9] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 55:81–115, 2005.

[10] Jori Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2006.

[11] Jori Dubrovin, Tommi Junttila, and Keijo Heljanko. Symbolic step encodings for object based communicating state machines. Technical Report B24, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2007.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[13] Harald Fecher and Jens Schönborn. UML 2.0 state machines: Complete formal semantics via core state machines. In *FMICS and PDMC 2006*, volume 4346 of *LNCS*, pages 244–260. Springer, 2007.

[14] Stefania Gnesi, Diego Latella, and Mieke Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *The Journal of Logic and Algebraic Programming*, 51:43–75, 2002.

[15] Keijo Heljanko, Tommi Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In *CAV 2005*, volume 3576 of *LNCS*, pages 98–111. Springer, 2005.

[16] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *Proc. MoDeV²a: Model Development, Validation and Verification*, pages 94–110, 2006.

[17] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in Promela/Spin. In *WIFT'98*, pages 90–101. IEEE Computer Society, 1998.

[19] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535. ACM, 2001.

[20] E. Mota, E. Clarke, A. Groce, W. Oliveira, M. Falcão, and J. Kanda. Veriagent: an approach to integrating UML and formal verification tools. *Electronic Notes in Theoretical Computer Science*, 95:111–129, 2004.

[21] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12–13):1031–1080, 2006.

[22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.

[23] Sara Van Langenhove. *Towards the Correctness of Software Behavior in UML — A Model Checking Approach based on Slicing*. PhD thesis, Universiteit Gent, May 2006.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
TECHNICAL REPORTS

HUT-TCS-B10	Kari J. Nurmela, Patric R. J. Östergård Constructing Covering Designs by Simulated Annealing. January 1993.
HUT-TCS-B11	Peter Grönberg, Mikko Tiusanen, Kimmo Varpaaniemi PROD – A PrT–Net Reachability Analysis Tool. June 1993.
HUT-TCS-B12	Kimmo Varpaaniemi On Computing Symmetries and Stubborn Sets. April 1994.
HUT-TCS-B13	Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, Tino Pyssysalo PROD Reference Manual. August 1995.
HUT-TCS-B14	Tuomas Aura Modelling the Needham-Schröder authentication protocol with high level Petri nets. September 1995.
HUT-TCS-B15	Eero Lassila ReFlEx — an Experimental Tool for Special-Purpose Processor Code Generation. March 1996.
HUT-TCS-B16	Markus Malmqvist Methodology of Dynamical Analysis of SDL Programs using Predicate/Transition Nets. April 1997.
HUT-TCS-B17	Tero Jyrinki Dynamical Analysis of SDL Programs using Predicate/Transition Nets. April 1997.
HUT-TCS-B18	Tommi Syrjänen Implementation of Local Grounding for Logic Programs With Stable Model Semantics. October 1998.
HUT-TCS-B19	Marko Mäkelä, Jani Lahtinen, Leo Ojala Performance Analysis of a Traffic Control System Using Stochastic Petri Nets. December 1998.
HUT-TCS-B20	Eero Lassila A Tree Expansion Formalism for Generative String Rewriting. June 2001.
HUT-TCS-B21	Annikka Aalto Automatic Translation of SDL into High Level Petri Nets. November 2004.
HUT-TCS-B22	Maarit Hietalahti, Mikko Särelä, Antti Tuominen, Pekka Orponen Security Topics and Mobility Management in Hierarchical Ad Hoc Networks (Samoyed): Final Report. December 2007.
HUT-TCS-B23	Jori Dubrovin, Tommi Junttila Symbolic Model Checking of Hierarchical UML State Machines. December 2007.

ISBN 978-951-22-9191-5
ISSN 0783-540X