Helsinki University of Technology Laboratory for Theoretical Computer Science
Technical Reports 24
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 24
Espoo 2007 HUT-TCS-B24

SYMBOLIC STEP ENCODINGS FOR OBJECT BASED COMMUNICATING STATE MACHINES

Jori Dubrovin, Tommi Junttila, and Keijo Heljanko

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 24
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tekninen raportti 24
Espoo 2007 HUT-TCS-B24

SYMBOLIC STEP ENCODINGS FOR OBJECT BASED COMMUNICATING STATE MACHINES

Jori Dubrovin, Tommi Junttila, and Keijo Heljanko

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 978-951-22-9193-9
ISSN 0783-540X

Multiprint Oy
Espoo 2007

Abstract: In this work, novel symbolic step encodings of the transition relation for object based communicating state machines are presented. This class of systems is tailored to capture the essential data manipulation features of UML state machines when enriched with a Java-like object oriented action language. The main contribution of the work is the generalization of the $\exists$-step semantics approach, which Rintanen has used for improving the efficiency of SAT based AI planning, to a much more complex class of systems. Furthermore, the approach is extended to employ a dynamic notion of independence. To evaluate the encodings, UML state machine models are automatically translated into NuSMV models and then symbolically model checked with NuSMV. Especially in bounded model checking (BMC), the $\exists$-step semantics often significantly outperforms the traditional interleaving semantics without any substantial blowup in the BMC encoding as a SAT formula.

Keywords: UML state machine, bounded model checking, step semantics, verification

CONTENTS

1 INTRODUCTION

Bounded model checking (BMC) [3] has been introduced as an alternative to binary decisions diagrams (BDDs) to implement symbolic model checking. The main contributions of our work are symbolic step encodings of the transition relation for object based communicating state machines. Similarly to partial order reduction techniques such as stubborn, ample, persistent, or sleep sets for explicit state model checking (see e.g., [22]) the idea is to exploit the concurrency available in the analyzed system to make the model checking for it more efficient. The main idea in the above mentioned partial order reduction methods is to try to generate a reduced state space of the system by removing some of the edges of the state space while still preserving the property (such as the existence of a deadlock state) being verified. However, the considerations for BMC are usually quite different from explicit state model checking. Instead of removing edges from the state space trying to minimize the size of the reduced state space, the idea is to try to minimize the bound needed to reach each state of the system. Our approach will not remove any edges but will instead add a number of "shortcut edges" to the state space of the analyzed system, intuitively executing several actions at the same time step, as allowed by the concurrency of the system being analyzed. The hope is that by making more states reachable with smaller bounds, the worst case exponential behavior of the bounded model checker wrt. the bound k can be alleviated by allowing bugs to be found with smaller values of k. The decrease in the bound k needs of course to be balanced against the size of the transition relation encoding as well as the efficiency of the SAT checker in solving the generated BMC instances.

As the system model we consider object based communicating state machines, a model tailored to capture the essential data manipulation features of UML state machines when enriched with a Java-like object oriented action language. The work aims at analyzing object oriented data communications protocol software designed using UML state machines, see [16]. The model checking tool we have developed can also handle other aspects of UML state machines not covered in this paper due to space considerations, see Sect. 4 for details.

This work can be seen as a generalization of the approach of Rintanen [21] on artificial intelligence planning, where the notion of $\exists$-step semantics for planning problems is employed. The intuitive idea is that a set of actions can be executed at the same time step in $\exists$-step semantics if there exists at least one order in which they can be executed. This is to be contrasted with the classical $\forall$-step semantics which allows the concurrent execution only in the case all permutations of the set of actions of a step are executable [21, 10]. In [21] the $\exists$-step approach has been shown to dramatically speed up SAT based AI planning over all other suggested semantics mainly due to significant reductions in the number of time steps needed to reach the goal state of the plan. The setup here differs from the AI planning domain mainly by having a much more complex class of systems with object oriented and other non-trivial data handling features such as asynchronous communication through message queues. Our approach also introduces the novel use of a dynamic dependency relation to optimize the encoding even further. We show that all of these extensions can be efficiently encoded, and thus the approach can be extended to a more realistic model of software systems.

Related Work

In the area of SAT based BMC, Heljanko was the first to consider exploiting the concurrency in encoding the transition relation [10]. That paper considers BMC for 1-bounded Petri nets using the $\forall$-step semantics (the classical Petri net step semantics, see e.g. [2]). The intuitive idea is that a set of actions can be executed at the same time step in $\forall$-step semantics if they can be executed in all possible orders. In the area of SAT based AI planning already the early papers of Kautz and Selman used $\forall$-step semantics [19] (see also [21]). The work of Dimopoulos et al. was the first one to use an $\exists$-step semantics like approach in hand optimized planning encodings of [6], the idea of which was later formalized and automated in the SAT based planning system of Rintanen [21]. On the BMC side Ogata et al. [20] and Jussila et al. [14, 17] both show other approaches to obtaining an optimized transition relation encoding for 1-bounded Petri nets and labeled transition systems (LTSs), respectively. A nice overview of many optimized transition relation encodings for LTSs is the doctoral thesis of Jussila [15]. We are currently not aware of any published work employing classical partial order reductions methods such as stubborn, ample, or persistent sets to improve the efficiency of SAT based symbolic model checking.

2 SYSTEMS AND SEMANTICS

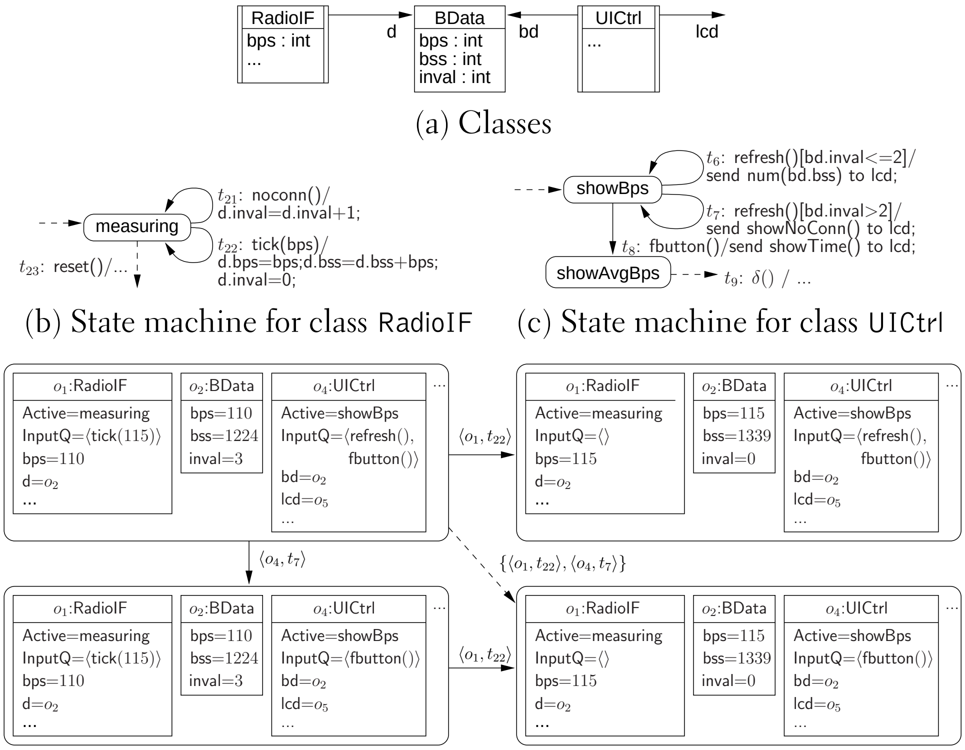
In this paper we consider a class of systems which are composed of a finite set of asynchronously executing objects communicating with each other through message passing and data attribute access. The behavior of each object is defined by a state machine. For instance, Figures 1(a)-(c) show a part of a simple heart beat monitor system described in UML with Java-like action language annotations. The dynamic behavior of a system is captured by its *interleaving state space*

$$M = \langle C, c_{\text{init}}, \Delta \rangle,$$

where C is the set of all *global configurations*, $c_{\text{init}} \in C$ is an *initial global configuration*, and the *transition relation* $\Delta \subseteq C \times A \times C$ describes how configurations may evolve to others: $\langle c, a, c' \rangle \in \Delta$ iff the configuration c can change to c' by executing an action $a \in A$. As an example, Fig. 1(d) shows a part of the state space (ignore the dashed arrow for a while) of the system in Figures 1(a)-(c); if the action $\langle o_1, t_{22} \rangle$ (corresponding to the object o_1 firing its transition t_{22}) is executed in the top left configuration, it changes to the top right one.

We say that a configuration c'' is *reachable* from a configuration c if there exist $a_1, \ldots, a_k$ and $c_0, c_1, \ldots, c_k$ for some $k \in \mathbb{N}$ such that (i) $c = c_0$, (ii) $\forall 1 \leq i \leq k : \langle c_{i-1}, a_i, c_i \rangle \in \Delta$, and (iii) $c_k = c''$.

The basic idea in $\exists$-step semantics exploited in this paper is to augment the state space with "shortcut edges" so that, under certain conditions, several actions can be executed "at the same time step". Formally, the *$\exists$-step state space* corresponding to the interleaving state space $M = \langle C, c_{\text{init}}, \Delta \rangle$ is the



(a) Classes

(b) State machine for class `RadioIF` (c) State machine for class `UICtrl`

(d) A part of the state space with three actions and one shortcut edge

Figure 1: A part of a simple heart beat monitor system

tuple

$$M_\exists = \langle C, c_{\text{init}}, \Delta_\exists \rangle$$

where the *transition relation* $\Delta_\exists \subseteq C \times 2^A \times C$ consists of all tuples $\langle c, S, c' \rangle$ such that

1. the step $S = \{a_1, \ldots, a_k\}$ is finite and non-empty, and
2. there is a total ordering $a_1 \prec \cdots \prec a_k$ of S and a sequence of configurations $c_0, c_1, \ldots, c_k \in C$ such that (i) $c = c_0$, (ii) $\forall 1 \leq i \leq k$: $\langle c_{i-1}, a_i, c_i \rangle \in \Delta$, and (iii) $c_k = c'$.

Continuing the running example, the dashed arrow in Fig. 1(d) denotes the $\exists$-step $S = \{\langle o_1, t_{22}\rangle, \langle o_4, t_7\rangle\}$. Note that the actions in the step can be executed in the order $\langle o_4, t_7\rangle \prec \langle o_1, t_{22}\rangle$ but not in $\langle o_1, t_{22}\rangle \prec \langle o_4, t_7\rangle$ as executing $\langle o_1, t_{22}\rangle$ disables $\langle o_4, t_7\rangle$.

By definition it holds that the $\exists$-step state space includes the interleaving state space in the sense that $\langle c, a, c' \rangle \in \Delta$ implies $\langle c, \{a\}, c' \rangle \in \Delta_\exists$. Conversely, if a $\langle c, S, c' \rangle$ belongs to $\Delta_\exists$, then there is a finite sequence of configurations leading from c to c' in the interleaving state space. Therefore, the set of configurations reachable from a given configuration is equal for the interleaving and the $\exists$-step state space. Note that the definition of $\exists$-step semantics is a truly semantic one; in the symbolic encoding given later, not all possible non-unit steps will be considered but only those that follow conveniently without complicating and growing the size of the encoding too much w.r.t. the one for the interleaving semantics. For example, similarly to [21], we require all actions of a step to be enabled already in the current configuration $c = c_0$. However, all unit steps will be included in order to preserve the soundness and completeness of the resulting encoding for the purpose of checking the reachability of desired/unwanted configurations. For complexity results on the semantic definition of ∃-step semantics in the AI planning domain, see [21], which also similarly only soundly underapproximates the ∃-step semantics in its implementation.

2.1 Object Based State Machine Models

We next give a brief description of the class of systems analyzed in this paper. Refer to Appendix A for the formal definitions.

We consider systems composed of a finite set O of *objects*. Each object is an instance of a *class*, and each class is composed of a finite set of typed *attributes* and a *state machine*. A state machine consists of a finite set of *states*, one of which is the *initial state*, and a finite set of *transitions*. Each transition has a *source state* and a *target state*, a *trigger*, a *guard*, which is an action language expression of Boolean type, and an *effect*, which is a list of action language statements. A trigger is of the form $sig(x_1, \ldots, x_k)$, where sig is a *signal* from a finite set $Sigs$, and the x_i are attributes of the class owning the state machine. The number and types of the x_i must match the predefined parameter types of the signal. A special signal $\delta \in Sigs$ with no parameters models spontaneously triggered transitions. For example, state measuring is both the source and the target of transition t_{22} in Fig. 1(c). The trigger of t_{22} is tick(bps), and the guard of t_{22} is implicitly `true`.

Objects can send and receive *messages* of the form $sig[v_1, \ldots, v_k]$, where $sig \neq \delta$ is a signal and the values v_i are message *arguments*. The number and types of arguments correspond to the parameter types of sig. We denote the set of all messages by $Msgs$. During execution, messages sent to an object are placed in a queue, and an object can consume a message from its queue either by firing a transition triggered by the corresponding signal or by *implicit consumption*, which means discarding a message that is not triggering any transitions. Spontaneously triggered transitions are fired without consuming messages.

A global configuration c of the system consists of, for each object o, (i) the currently active state of o, which is one of the states in the state machine of the class of o, (ii) the value of the instance $o.x$ of each attribute x of the class of o, (iii) the contents of the input queue of o, which is a sequence of messages. We will call the frontmost (oldest) message in the queue the *current* message of o. The initial configuration c_{init} is such that all objects are in their initial states and all input queues are empty.

The actions of the system are the *transition instances* $\langle o, t \rangle$ and the *implicit consumption actions* $\langle o, \mathrm{IMPCONS} \rangle$, where o is an object and t is a transition in the state machine of o. A transition instance $\langle o, t \rangle$ is *enabled* in a global configuration c if (i) the source state of t is active in o, (ii) the guard of t evaluates to true in the context of o, and (iii) either the trigger of t is $\delta()$ or o has a current message whose signal matches the trigger signal of t. Of the global configurations in Fig. 1(d), $\langle o_4, t_6 \rangle$ is only enabled in the top right one, in which the current message of o_4 is refresh() and o_2.inval ≤ 2.

An enabled action can be *executed* in a global configuration. Firing transition t in object o, or more formally, executing an enabled transition instance $\langle o, t\rangle$ in a global configuration c, leads to a new global configuration c' that is obtained from c by (i) assigning the argument values of the current message of o to the attributes mentioned in the trigger of t, (ii) removing the current message from the input queue of o, (iii) executing the effect of t in the context of o, and (iv) making the target of t the new active state of o. If t is a spontaneously triggered transition, i.e. the trigger of t is $\delta()$, then points (i) and (ii) above are not performed.

An implicit consumption action $\langle o, \text{IMPCONS}\rangle$ is enabled if (i) the input queue of o is not empty, and (ii) there is no enabled transition instance $\langle o, t\rangle$ such that the trigger of t is not $\delta()$. Executing an enabled implicit consumption action $\langle o, \text{IMPCONS}\rangle$ in a global configuration c leads to a global configuration c' that is equal to c except that the current message of o is removed.

The interleaving state space is now defined as the tuple $M = \langle C, c_{\text{init}}, \Delta\rangle$, where C is the set of all global configurations, $c_{\text{init}} \in C$ is the initial configuration, and the transition relation $\Delta \subseteq C \times A \times C$ consists of the triples $\langle c, a, c'\rangle$ such that the transition instance or implicit consumption action a is enabled in c and executing a in c leads to c' by the above rules.

Action Language

In the sequel, we fix a simple Java-based action language [7]. The type system consists of the Boolean type $\mathbb{B}$, the 32-bit signed integer type, and for each class C the reference type $\mathbb{T}_C$ with domain $\{o \in O \mid \text{class of } o \text{ is } C\} \cup \{\textbf{null}\}$. As in Java, static typing rules apply. The supported expressions are: (i) literals `true`, `false`, `null`, and 32-bit signed integer literals, (ii) the `this` reference, (iii) attribute access expressions of the form $\mathit{refexpr}$`.`x, where the type of $\mathit{refexpr}$ is $\mathbb{T}_C$ and x is an attribute of class C, and (iv) infix expressions $\mathit{leftexpr}\ \ \mathit{op}\ \ \mathit{rightexpr}$, where op is one of `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `>`, `<`, `>=`, `<=`, `==`, or `!=`, with Java semantics [9]. The only data accessible to expressions is attribute values reachable by following references. In particular, an expression cannot read the active state or the input queue of any object. An unqualified attribute name x is shorthand for `this.`x.

Statements of the language are: (i) assignments of the form $\mathit{refexpr}$`.`x `=` $\mathit{rhsexpr}$`;`, (ii) send statements of the form `send` sig`(`$\mathit{arg}_1, \ldots, \mathit{arg}_k$`) to` $\mathit{targetexpr}$`;`. When a send statement is executed, the input queue of the object referred by $\mathit{targetexpr}$ is appended with the message $\mathit{sig}[v_1, \ldots, v_k]$, where each v_i is the value of arg_i, and (iii) assertions of the form `assert` $\mathit{condexpr}$`;`. We want to check that $\mathit{condexpr}$ is never false when an assertion is executed.

The effect of a transition is an arbitrary list of statements. However, we require that for each transition t and class C, there is at most one send statement in the effect of t whose $\mathit{targetexpr}$ has type $\mathbb{T}_C$. The reason for this is that the symbolic encoding relies on the fact that in one step, at most one message is sent to each object.

3 SYMBOLIC ENCODING

The encoding of a 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 a step, *input variables* whose values are only limited by the constraints, and *derived functions* that are defined over the variables. The basic idea is that all constraints are satisfied if and only if there is a step from the configuration represented by the values of state variables to the configuration represented by the next-state variables. The desired properties of the system are encoded as a set of *invariants* that are to be verified using model checking. Many of the variables and functions have values in the Boolean domain. Non-Boolean variables and functions have a finite domain and thus can be booleanized to enable the use of SAT- and BDD-based techniques.

To keep the state space finite, we restrict the analysis to *bounded global configurations*, setting an upper limit QSIZE to the number of messages in any queue. Let $M = \langle C, c_{\text{init}}, \Delta\rangle$ be an interleaving state space, and let C^B be the set of configurations $c \in C$ such that the length of the input queue of o in c is at most QSIZE for all objects o. The *bounded interleaving state space* is $M^B = \langle C^B, c_{\text{init}}, \Delta^B\rangle$, where $\Delta^B = \{\langle c, a, c'\rangle \in \Delta \mid c, c' \in C^B\}$.

Fixed Ordering of Actions in Steps

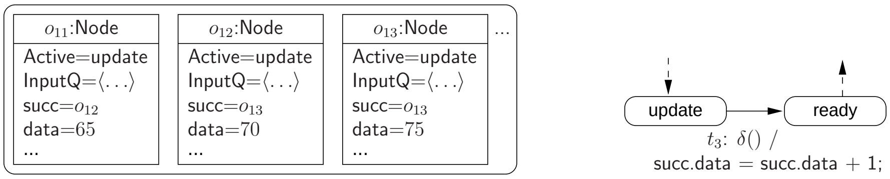
We fix an arbitrary total order $\prec$ of the set of actions A. The intuitive idea is that instead of considering all possible orderings of actions as the $\exists$-step semantic definition allows, $\prec$ gives us one static order in which the executability of actions at a time step will be guaranteed. To do this, the encoding must capture the state changes done by each action, and disallow all steps where a value modified by some action is (explicitly or implicitly) read by an action that is greater wrt. the order $\prec$. By doing this, we ensure that all of the $\exists$-steps allowed by the encoding are executable in the order given by $\prec$.

We will thus get a different encoding for each choice of $\prec$. The only requirement is that if $o_1, o_2 \in O$ and t_2 is a transition in the state machine of o_2, then $\langle o_1, \text{IMPCONS}\rangle \prec \langle o_2, t_2\rangle$ must hold. In words, all implicit consumption actions must precede all transition instances in the total order. The reason for this will be discussed in Sect. 3.4.

The choice of a good total order is an interesting question that we have left for further study. A set of actions is *independent* if no action reads any part of the system state modified by another action. Notice that an independent set of enabled actions can always be executed in all orders, and thus the $\exists$-step semantics always allows a set of independent actions to form an $\exists$-step for any selection of the total order $\prec$.

Three Alternative Encodings

We present three different symbolic encodings, namely an *interleaving* encoding, a *static* $\exists$*-step* encoding and a *dynamic* $\exists$*-step* encoding. All three encodings contain all interleaving steps in the sense that if $\langle c, a, c'\rangle \in \Delta^B$ and the valuations of state and next-state variables represent c and c' respectively, then there is a valuation of input variables such that the constraints are satisfied. Conversely, nothing but $\exists$-steps that respect the order $\prec$ are



(a) Global configuration (b) State machine of class `Node`

Figure 2: An example for illustrating static and dynamic $\exists$-steps

allowed by the encodings. Let $M^B_\exists = \langle C^B, c_{\text{init}}, \Delta^B_\exists \rangle$ be the state space obtained from M^B by the definition of $\exists$-step semantics with the further restriction that the total ordering of each step S respects the fixed order $\prec$. If the valuation of state variables represents a bounded configuration $c \in C^B$ and the constraints are satisfied, then the valuation of next-state variables represents a bounded configuration $c' \in C^B$ and there is a step $S \subseteq A$ such that $\langle c, S, c' \rangle \in \Delta^B_\exists$. Furthermore, in the interleaving encoding, S only contains one action, and thus every step is an interleaving step.

The definitions of the three encodings overlap for the most part, and the differences are stated explicitly. The difference between static and dynamic steps is that in static steps, whether actions a_1 and a_2 can be executed concurrently depends only on a_1 and a_2. In dynamic steps, it also depends on the current global configuration, in particular, on the values of attributes. Consider the global configuration in Fig. 2(a) consisting of three objects of class `Node`. Transition t_3 is enabled both in object o_{11} and in o_{12}. Because the action $\langle o_{11}, t_3 \rangle$ increments attribute `data` in o_{12} and $\langle o_{12}, t_3 \rangle$ increments `data` in o_{13}, neither action reads a value written by the other. The dynamic step encoding allows $\langle o_{11}, t_3 \rangle$ and $\langle o_{12}, t_3 \rangle$ to be executed concurrently in this global configuration. However, in some other global configuration the `succ` attribute in o_{11} and o_{12} might refer to the same object, so the two actions might read and modify the same attribute. As a safe statically computed approximation, the static step encoding never allows executing $\langle o_{11}, t_3 \rangle$ and $\langle o_{12}, t_3 \rangle$ concurrently.

3.1 State Variables

The set of state variables contains three kinds of elements for each object o.

1. $\mathsf{Active}(o, s)$, where s is a state in the state machine of o, is true iff s is the current active state in o.

2. $\mathsf{AttrVal}(o, x)$, where x is an attribute of the class of o, determines the current value of $o.x$ and has the same domain as the type of x.

3. $\mathsf{InputQ}(o)$ with domain $Msgs^0 \cup \cdots \cup Msgs^{\text{QSIZE}}$ determines the contents of the input queue of o.

Given a bounded global configuration, the values of state variables can be derived in the obvious way. The corresponding next-state variables are denoted by $\mathit{next}\,(\mathsf{Active}(o, s))$, $\mathit{next}\,(\mathsf{AttrVal}(o, x))$, and $\mathit{next}\,(\mathsf{InputQ}(o))$, respectively.

3.2 State Machines and Queues

This sections gives a rough overview of the encoding of state machine control logic. A more detailed definition can be found in Appendix B.

The control logic constraints are responsible for ensuring that in the context of a single object $o \in O$, (i) at most one transition instance or implicit consumption action is executed, (ii) an action is executed only if it is enabled in the global configuration represented by the state variables, and (iii) the variables $\mathit{next}\,(\mathsf{Active}(o, s))$ correctly reflect the active state after the step.

Let $o \in O$ be an object. The input variable $\mathsf{Dispatch}(o)$ with domain $\mathit{Sigs} \cup \{\mathbf{none}\}$ determines which message $\mathit{sig}[\ldots]$, if any, is being consumed by o. For each transition t in the state machine of o, the input variable $\mathsf{Fire}(o, t)$ determines whether t is being fired in o. We define the current step $S \subseteq A$ as the set consisting of all transition instances $\langle o, t\rangle$ such that $\mathsf{Fire}(o, t)$ is true, and all implicit consumption actions $\langle o, \text{IMPCONS}\rangle$ such that $\mathsf{Fire}(o, t)$ is false for all t but $\mathsf{Dispatch}(o) \neq \mathbf{none}$.

For example in the state machine of Fig. 1(c), the next-state variable related to state $\mathsf{showAvgBps}$ is fixed by the constraint

$$\begin{aligned}\mathit{next}\,(\mathsf{Active}(o, \mathsf{showAvgBps})) \Leftrightarrow \\ \mathsf{Fire}(o, t_8) \vee \big(\neg\mathsf{Fire}(o, t_9) \wedge \mathsf{Active}(o, \mathsf{showAvgBps})\big),\end{aligned}$$

and the enabledness check for the action $\langle o, t_8\rangle$ is encoded in the constraint

$$\mathsf{Fire}(o, t_8) \Leftrightarrow (\mathsf{Dispatch}(o) = \mathsf{fbutton}) \wedge \mathsf{Active}(o, \mathsf{showBps}).$$

Object o is *scheduled* if it is consuming a signal, formalized in the function definition

$$\mathsf{Scheduled}(o) := \big(\mathsf{Dispatch}(o) \neq \mathbf{none}\big). \tag{1}$$

To rule out an empty step, we require that at least one object is scheduled in each step by the constraint

$$\bigvee \big\{\mathsf{Scheduled}(o) \mid o \in O\big\}. \tag{2}$$

Furthermore, there are queue constraints ensuring that consumed messages are removed from the front of the input queue and received messages are added to the back of the queue. The input variable $\mathsf{NewMsg}(o)$ with domain $\mathit{Msgs} \cup \{\mathbf{none}\}$ denotes the message possibly being sent to o. Queue overflows are prevented by specialized constraints, disallowing transitions into global configurations that are not in C^B.

3.3 Effects and Data

Expressions
All data manipulation in the effects of transitions is based on evaluating expressions. We define a function $\mathsf{Eval}(\mathit{expr})$ that gives the value of expression expr. For clarity, we leave out the context in which the expression is evaluated; a more rigorous treatment can be found in Appendix B. Evaluation of

constant and infix expressions is straightforward, given the encodings for all infix operators. In the context of an object o, the value of $\mathsf{Eval}(\texttt{this})$ is o. Attribute access expressions of the form $\mathit{refexpr}\,.\,\hat{x}$ are encoded as case switches over all objects of the type of $\mathit{refexpr}$. For example, the expression $\mathsf{succ.data}$ in Fig. 2(b) translates in the context of o_{11} to the formula

$$\mathsf{Eval}(\mathsf{succ.data}) := \mathit{if} \begin{cases} \mathsf{AttrVal}(o_{11}, \mathsf{succ}) = o_{11} & : \ \mathsf{AttrVal}(o_{11}, \mathsf{data}) \\ \mathsf{AttrVal}(o_{11}, \mathsf{succ}) = o_{12} & : \ \mathsf{AttrVal}(o_{12}, \mathsf{data}) \\ \mathsf{AttrVal}(o_{11}, \mathsf{succ}) = o_{13} & : \ \mathsf{AttrVal}(o_{13}, \mathsf{data}) \\ \mathit{else} & : \ 0. \end{cases}$$

Effects

The effects of transitions can modify the global configuration by assigning values to attributes and by sending messages to objects. For each transition instance $\langle o, t\rangle$, each object $\hat{o}$, and each attribute $\hat{x}$ of the class of $\hat{o}$, we define functions $\mathsf{Send}_{o,t}(\hat{o})$ and $\mathsf{Write}_{o,t}(\hat{o}, \hat{x})$ that evaluate to true if the transition instance is executed and sends a message to $\hat{o}$ or assigns to $\hat{o}.\hat{x}$, respectively. Assignment can occur explicitly by an assignment statement or implicitly by message argument reception.

These functions are used for determining the next global configuration in the following way. If the effect of a transition t in the state machine of a class C contains a send statement $\texttt{send}\ \mathit{sig}\texttt{(}\mathit{arg}_1, \ldots, \mathit{arg}_m\texttt{)}\ \texttt{to}\ \mathit{targetexpr}\texttt{;}$, we add for each object o of class C and each object $\hat{o}$ of the type of $\mathit{targetexpr}$ the constraint

$$\mathsf{Send}_{o,t}(\hat{o}) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathit{sig}[\mathsf{Eval}(\mathit{arg}_1), \ldots, \mathsf{Eval}(\mathit{arg}_m)]\big), \tag{3}$$

which fixes the message received by $\hat{o}$ in case $\langle o, t\rangle$ is executed. Similarly, the value assigned to $\hat{o}.\hat{x}$ by a transition instance $\langle o, t\rangle$ is fixed by the constraint

$$\mathsf{Write}_{o,t}(\hat{o}, \hat{x}) \Rightarrow \big(\mathit{next}\,(\mathsf{AttrVal}(\hat{o}, \hat{x})) = \mathsf{Temp}_{o,t}(\hat{o}, \hat{x})\big), \tag{4}$$

where $\mathsf{Temp}_{o,t}(\hat{o}, \hat{x})$ evaluates to the value of $\hat{o}.\hat{x}$ after executing the effect of t in object o. This is defined using the Eval function. For example in Fig. 2(b),

$$\mathsf{Temp}_{o_{11},t_3}(o_{12}, \mathsf{data}) := \mathit{if} \begin{cases} \mathsf{AttrVal}(o_{11}, \mathsf{succ}) = o_{12} & : \ \mathsf{Eval}(\mathsf{succ.data}\ \texttt{+ 1}) \\ \mathit{else} & : \ \mathsf{AttrVal}(o_{12}, \mathsf{data}). \end{cases}$$

For each transition instance $\langle o, t\rangle$ and each assertion $\texttt{assert}\ \mathit{condexpr}\texttt{;}$ in the effect of t, we check that the invariant $\mathsf{Fire}(o, t) \Rightarrow \mathsf{Eval}(\mathit{condexpr})$ holds.

Frame Conditions

Let $\Theta \subseteq A$ be the set of all transition instances. The only situation when $\mathsf{NewMsg}(\hat{o})$ is not fixed by (3) is when $\mathsf{Send}_{o,t}(\hat{o})$ is false for all transition instances $\langle o, t\rangle \in \Theta$. Similarly, $\mathit{next}\,(\mathsf{AttrVal}(\hat{o}, \hat{x}))$ is not fixed when all functions $\mathsf{Write}_{o,t}(\hat{o}, \hat{x})$ are false. To fix these, we add the constraints

$$\neg \bigvee \big\{\mathsf{Send}_{o,t}(\hat{o}) \mid \langle o, t\rangle \in \Theta\big\} \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \textbf{none}\big), \tag{5}$$

$$\neg \bigvee \big\{\mathsf{Write}_{o,t}(\hat{o}, \hat{x}) \mid \langle o, t\rangle \in \Theta\big\} \Rightarrow \big(\mathit{next}\,(\mathsf{AttrVal}(\hat{o}, \hat{x})) = \mathsf{AttrVal}(\hat{o}, \hat{x})\big). \tag{6}$$

3.4 Step Constraints

In the interleaving encoding, at most one object is scheduled at a time, as required by the constraint

$$AtMostOne\left(\{\mathsf{Scheduled}(o) \mid o \in O\}\right). \tag{7}$$

Consequently, at most one action is executed in each step. A predicate of the form $AtMostOne\,(P)$ 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 the $\exists$-step encodings, (7) is replaced by more liberal constraints as follows. We require that an action must not send a message to an object if a preceding action has already done so, and it must not read an attribute that a preceding action has written. This is formalized in the constraints

$$\bigvee\left\{\mathsf{Send}_{o^-,t^-}(\hat{o}) \mid \langle o^-, t^-\rangle \prec \langle o,t\rangle\right\} \Rightarrow \neg\mathsf{Send}_{o,t}(\hat{o}), \tag{8}$$

$$\bigvee\left\{\mathsf{Write}_{o^-,t^-}(\hat{o},\hat{x}) \mid \langle o^-, t^-\rangle \prec \langle o,t\rangle\right\} \Rightarrow \neg\mathsf{Read}_{o,t}(\hat{o},\hat{x}). \tag{9}$$

Constraints (9) say that a transition instance $\langle o,t\rangle$ that reads an attribute $\hat{o}.\hat{x}$ can only be executed (denoted by $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$) if that attribute is not modified by any concurrently executed transition instance that precedes $\langle o,t\rangle$ in the total order. This means that in the global configuration that would result from executing the preceding transition instances, the value of $\hat{o}.\hat{x}$ is still the same as in the starting configuration represented by the state variables. This justifies the use of $\mathsf{AttrVal}(\hat{o},\hat{x})$ in evaluating the expressions in the effect of $\langle o,t\rangle$. Also notice that (4) forbids executing two transition instances that would assign a different value to the same attribute, and (8) prevents two transition instances from sending to the same object.

Implicit consumption actions cannot send messages or modify attributes. However, an implicit consumption action $\langle o, \text{IMPCONS}\rangle$ can implicitly *read* an attribute because the enabledness of $\langle o, \text{IMPCONS}\rangle$ might depend on the enabledness of a transition instance $\langle o,t\rangle$, which in turn might depend on an attribute that is mentioned in the guard of t. By setting the requirement that implicit consumption actions precede all transition instances in the total order, we rule out the possibility of $\langle o, \text{IMPCONS}\rangle$ implicitly reading an attribute that has been written by a preceding action.

Static and Dynamic Steps

The difference between the two $\exists$-step encodings is in the definitions of Send, Write, and Read. In dynamic steps, these functions are evaluated accurately using the input and state variables. For example, $\mathsf{Send}_{o,t}(\hat{o})$ is defined as $\mathsf{Fire}(o,t) \wedge \left(\mathsf{Eval}(targetexpr) = \hat{o}\right)$, and $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$ is true iff $\mathsf{Fire}(o,t)$ is true and $\mathsf{Eval}(refexpr) = \hat{o}$ is true for any subexpression of the form $refexpr.\hat{x}$ in the guard or effect of t. The same definitions are used in the interleaving encoding.

In static steps, overapproximations are used. If the guard or effect of transition t does not contain $\hat{x}$ in any subexpression, then $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$ is trivially **false** for all o and $\hat{o}$. Otherwise, $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$ is defined as $\mathsf{Fire}(o,t)$. Conceptually this means that when $\langle o,t\rangle$ is executed, it reads the attribute $\hat{x}$ of *all* objects that contain it. Equivalently, $\mathsf{Send}_{o,t}(\hat{o})$ is defined as $\mathsf{Fire}(o,t)$ if the effect of t contains any send statement to an object of the same class as $\hat{o}$ and **false** otherwise, and similarly for Write. This approximation strengthens the step constraints (8) and (9) and also makes the constraints static in the sense that they no longer refer to the state variables. As an optimization, if transition t only accesses $\hat{x}$ using the expression $\mathtt{this}.\hat{x}$, then it is known that the action $\langle o, t\rangle$ does not read $\hat{o}.\hat{x}$ if $\hat{o} \neq o$, and therefore $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$ is defined as **false** in these cases. The same optimization is applied to Send and Write.

Because the function $\mathsf{Send}_{o,t}(\hat{o})$ is an approximation in the static step encoding, it may evaluate to **true** even though no message is being sent to $\hat{o}$. For this reason, in static steps we replace (3) with two constraints

$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathit{sig}[\ldots]\big), \quad (10)$$

$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \neg\big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathbf{none}\big). \quad (11)$$

A similar correction does not need to be made to (4) because $\mathsf{Temp}_{o,t}(\hat{o},\hat{x})$ is evaluated accurately even in the static step encoding.

Consider again the setting of Fig. 2. Assuming that $\langle o_{11}, t_3\rangle \prec \langle o_{12}, t_3\rangle$ are the two first transition instances in the total order, we get from (9) the step constraint

$$\mathsf{Write}_{o_{11},t_3}(\hat{o}, \mathsf{data}) \Rightarrow \neg\mathsf{Read}_{o_{12},t_3}(\hat{o}, \mathsf{data}) \quad (12)$$

instantiated for each $\hat{o} \in \{o_{11}, o_{12}, o_{13}\}$. In dynamic steps, these expand to

$$\mathsf{Fire}(o_{11}, t_3) \wedge \big(\mathsf{AttrVal}(o_{11}, \mathsf{succ}) = \hat{o}\big) \Rightarrow \\ \neg\big(\mathsf{Fire}(o_{12}, t_3) \wedge (\mathsf{AttrVal}(o_{12}, \mathsf{succ}) = \hat{o})\big),$$

allowing, for example, executing $\langle o_{11}, t_3\rangle$ and $\langle o_{12}, t_3\rangle$ concurrently in the global configuration of Fig. 2(a). In static steps, all three instantiations of (12) reduce to the same constraint $\mathsf{Fire}(o_{11}, t_3) \Rightarrow \neg\mathsf{Fire}(o_{12}, t_3)$, which disallows concurrent execution of the two actions in any global configuration. In this example, static $\exists$-step semantics yields a smaller encoding, but dynamic $\exists$-step semantics permits more concurrency in a single step.

Steps and Bounded Queues

In some cases, the constraints presented above allow a step that has an interleaving in the interleaving state space M but not in the bounded interleaving state space M^B. In particular, there may be a step $\{a_1, a_2\}$ where action a_1 sends a message to a queue that already contains QSIZE messages, action a_2 consumes a message from the same queue, and $a_1 \prec a_2$. To remove spurious steps like this, we add extra constraints in the step encodings. The details are in Appendix B.

3.5 Size of the Encodings

Let $|M|$ be the size of the model, containing the definition of every class, attribute, signal, and state machine, and the textual definitions of guards and effects. Assuming that common subformulas are shared between constraints, the size of all three encodings is $\mathcal{O}(|M|(\text{QSIZE} \cdot |O| + |O|^2) \log |O|)$. The term $\log |O|$ is the required number of bits to represent the values of attributes and expressions of reference type. The term $|O|^2$ appears because objects can refer to each other arbitrarily, and it seems unavoidable in the presence of dynamic references. The total size of queue and state machine encodings without data or transition effects is $\mathcal{O}(|M| \cdot \text{QSIZE} \cdot |O| \log |O|)$.

The worst-case size for the data and step encoding can be seen in (4). For each assignment statement (quantity bounded by $|M|$), there are $\mathcal{O}(|O|^2)$ instantiations of (4), each of size $\mathcal{O}(\log |O|)$ because of the comparison of object references. Thus the total size sums up to $\mathcal{O}(|M| \cdot |O|^2 \log |O|)$ even in the interleaving encoding. In the $\exists$-step encodings, there are $\mathcal{O}(|M| \cdot |O|^2)$ additional step constraints, but they do not seem to dominate the total encoding size. We also point out that the left-hand sides of the step constraints (8) and (9) do not add extra size to the step encodings because they are already present as subformulas of the frame constraints (5) and (6).

4 EXPERIMENTAL RESULTS

We have implemented the symbolic encoding described above. The implementation[1] assumes the system models to be described with a subset of UML, the main additions to the above encoding being that state machines also support (i) hierarchy, (ii) completion events via the busy-quiescent construction given in [16], (iii) deferring of events, and (iv) initial and choice pseudostates (see [8] for a symbolic *interleaving* semantics encoding of such extended state machines). Currently the tool supports model checking queries for deadlocks, implicit consumption of messages, assertion violations, and action language run-time errors. The encoder is implemented in the Python programming language, uses the Coral toolkit [1] to read the UML models given in XMI format, and outputs the symbolic encoding as a NuSMV [5] program. The following experiments were run on a PC machine with a 2 GHz AMD Athlon 64 processor, 2 GB of memory, and Debian Linux operating system. We used version 2.4.3 of NuSMV and limited the available memory to 1.5 GB and time to ten minutes. The width of integer attributes in the encoding was 32 bits and the input queue size was two.

We used the following models. (i) `SCP` is a simple communication protocol with three active objects (environment, sender, and receiver) and three cycling phases (connection establishment, data transfer, connection release). (ii) `travel` is an essentially sequential resource allocation process modeling a travel agent accessing a database, involving 4 active objects. (iii) `mtravel` is a variant of `travel` with 3 competing travel agents and databases organized in a ring. (iv) `giop1_2` has been adapted from the General Inter-ORB Protocol model [18], with an uninitialized variable introduced in the adaptation.

Table 1 shows our preliminary results. The column "sem." gives the semantics (interleaving, static $\exists$-steps, or dynamic $\exists$-steps) and the length of the shortest counterexamples under it, "SBMC zchaff" gives the minimum and maximum running time (in seconds) of 10 runs of the incremental BMC algorithm [11, 4] (the NuSMV command `check_ltlspec_sbmc_inc`) when ZChaff is used as the SAT solver, "SBMC minisat" is the same with MiniSat as the SAT solver, and "BDD invar" gives the running times of a BDD-based

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

Table 1: Results

model +	sem.	k	SBMC zchaff	SBMC minisat	BDD invar	Spin DFS		Spin DFS -i	
property			time	time	time	time	|cex|	time	|cex|
travel deadlock	interl.	15	0.59–0.64	0.46–0.55	1.19–1.20	0.01–0.01	17–24	0.03–0.05	15–15
	s.step	11	0.32–0.36	0.30–0.34	1.91–1.93				
	d.step	11	0.31–0.35	0.31–0.34	1.69–1.71				
mtravel deadlock	interl.	36	T.O.(21–22)	T.O.(23–24)	M.O.	0.01–43.58	71–103576	T.O.	
	s.step	14	5.87–10.73	3.26–5.03	M.O.				
	d.step	11	2.01–2.26	1.56–1.67	M.O.				
SCP deadlock	interl.	13	2.17–2.70	1.21–1.51	174.05–174.89	0.02–0.03	13–104	0.04–0.74	13–13
	s.step	7	0.37–0.41	0.37–0.40	107.04–107.42				
	d.step	6	0.38–0.40	0.37–0.40	T.O.				
SCP implicit cons.	interl.	7	0.46–0.49	0.42–0.45	n.a.	0.01–0.01	12–24	0.02–0.04	7–7
	s.step	6	0.38–0.41	0.37–0.41	n.a.				
	d.step	5	0.37–0.39	0.36–0.40	n.a.				
giop1_2 runtime errors	interl.	14	293.09–338.47 [79.06–124.39]	250.15–261.90 [36.30–48.49]	n.a.	0.29–580.96	30–64	T.O.	
	s.step	9	162.59–174.98 [6.71–9.73]	156.94–165.81 [2.16–3.34]	n.a.				
	d.step	8	156.57–162.53 [4.04–5.13]	154.23–158.20 [1.19–2.30]	n.a.				

invariant checking algorithm (`check_invar` in NuSMV). The times shown in square brackets are the times used by SAT solvers instead of the total running times of NuSMV (especially on the model giop1_2, the total running time is dominated by preprocessing overhead). M.O. means that all runs exceeded the memory limit, and T.O.(x–y) means that all runs timed out; x and y give the minimum and maximum, respectively, of the bounds that were reached before timeout. We check (i) deadlocks of the models SCP, mtravel, and travel, (ii) whether implicit consumption of messages is possible in the model SCP, and (iii) if the model giop1_2 has run-time errors. The latter two properties are only checked with BMC but not with BDDs because the invariant involves input variables; this is not accepted by the NuSMV BDD invariant checking command.

Analyzing the sizes of SAT instances generated by NuSMV shows that the proportion of $\exists$-step constraints in an instance is between 4 and 8 % when static steps are used, and between 5 and 15 % when dynamic steps are used, depending on the model. This verifies the presumption that using step semantics does not substantially increase the size of the encoding.

From the results we see that using $\exists$-step semantics instead of interleaving can (i) drop the bound required to find a counterexample, and (ii) more importantly, quite radically reduce the running times of BMC algorithms. This is especially true with models that contain lots of concurrency. E.g. on the model mtravel using interleaving semantics, BMC could not reach the required bound 36 even if we gave 1 hour of time, while using step semantics a corresponding counter-example is found within seconds. For BDDs the situation is not at all that clear; it seems in fact that the interleaving semantics works best with BDDs. We also experimented with the BDD-based breadth-first enumeration of reachable states in NuSMV and the preliminary findings were that the step semantics indeed covered more states than the interleaving semantics with the same number of iterations but it took more time to do so.

We also ran tests with the state-of-the-art explicit state model checker Spin [13] that is designed especially for the analysis of this kind of models. The UML models were automatically translated to the input language of Spin by a translation based on that of [16]. The last two columns in Table 1 give the running times and lengths of produced counter-examples of Spin with (i) the default depth-first search mode ("Spin DFS"), and (ii) the same with counter-example minimization option "-i" enabled. Partial order reductions and the state compression ("COLLAPSE") were enabled in both modes. As expected, Spin is superior on models with relatively small state spaces (`SCP` and `travel`). On the models `mtravel` and `giop1_2` containing more concurrency Spin sometimes consumed more time and, more importantly, produced counter-examples that are substantially longer than the minimal ones produced by BMC based methods; such long counter-examples are not as useful in debugging the system as it becomes much harder to locate the real source of the bug.

To sum up, symbolic model checking, especially with step semantics, can potentially provide a competitive approach for model checking of communication protocols, which has traditionally been the strong area of explicit state model checkers.

5 CONCLUSIONS AND FUTURE WORK

We have shown how to exploit the concurrency in the transition relation encoding for object based communicating state machines. Especially in bounded model checking, the proposed $\exists$-step semantics significantly outperform the traditional interleaving semantics approach, without any considerable blowup in the encoding as a SAT formula.

One avenue for further study is the use of SAT modulo theories (SMT) solvers to improve the performance of bounded model checking of systems containing data. Our encoding can be fairly easily adjusted to do that. Also requiring further study are the details of the way the $\exists$-semantics needs to be restricted in order to soundly accommodate the model checking of liveness properties along the lines of [12, 4]. And as the choice of the total ordering between actions in the encoding affects which steps are considered, how to statically choose a good ordering needs further investigation.

Acknowledgements

This work has been financially supported by Tekes (Finnish Funding Agency for Technology and Innovation), Nokia, Conformiq Software, Mipro, Helsinki Graduate School in Computer Science and Engineering, the Academy of Finland (projects 112016, 211025, and 213113), and Technology Industries of Finland Centennial Foundation.

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] Eike Best and Raymond R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[4] 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.

[5] 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.

[6] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *ECP'97*, volume 1348 of *LNCS*, pages 169–181. Springer, 1997.

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

[8] Jori Dubrovin and Tommi Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report B23, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2007.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[10] Keijo Heljanko. Bounded reachability checking with process semantics. In *CONCUR'01*, volume 2154 of *LNCS*, pages 218–232. Springer, 2001.

[11] Keijo Heljanko, Tommi Junttila, and Timo Latvala. Incremental and complete bounded model checking for full PLTL. In *CAV'05*, volume 3576 of *LNCS*, pages 98–111. Springer, 2005.

[12] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4&5):519–550, 2003.

[13] Gerard J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2004.

[14] Toni Jussila. BMC via dynamic atomicity analysis. In *ACSD'04*, pages 197–206. IEEE Computer Society, 2004.

[15] Toni Jussila. *On Bounded Model Checking of Asynchronous Systems*. Doctoral dissertation, Helsinki University of Technology, 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] Toni Jussila, Keijo Heljanko, and Ilkka Niemelä. BMC via on-the-fly determinization. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101, 2005.

[18] Moataz Kamel and Stefan Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.

[19] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI'96/IAAI'96, Vol. 2*, pages 1194–1201. AAAI Press, 1996.

[20] Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno. SAT-based verification of safe Petri nets. In *ATVA'04*, volume 3299 of *LNCS*, pages 79–92. Springer, 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] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.

A FORMAL EXECUTION SEMANTICS

We next give a formal definition of the class of systems analyzed in this paper. We consider systems composed of a finite set O of objects. Each object $o \in O$ is an instance of a class $\mathit{Class}(o)$ and each class C is composed of a finite set of attributes $\mathit{Attrs}(C)$ and a state machine $\mathit{SM}(C)$. An attribute $x \in \mathit{Attrs}(C)$ is a typed variable, i.e. is associated with a type $\mathit{Type}(x)$ in a set $\mathcal{T}$ of types. Each type $T \in \mathcal{T}$ has a non-empty domain set $\mathit{Dom}(T)$ and a default value $\mathit{default}(T) \in \mathit{Dom}(T)$. In particular, the Boolean type $\mathbb{B} \in \mathcal{T}$ has $\mathit{Dom}(\mathbb{B}) = \{\mathbf{false}, \mathbf{true}\}$. A strongly typed action language $\mathcal{L}$ over the types is assumed; $\mathcal{L}_{\mathbb{B}} \subset \mathcal{L}$ denotes the set of side-effect free Boolean valued expressions and $\mathcal{L}_{\text{Stmt}} \subset \mathcal{L}$ the set of (possibly compound) statements.

The asynchronous passing of messages builds on a finite set Sigs of signals, each $\mathit{sig} \in \mathit{Sigs}$ having a list $\mathit{params}(\mathit{sig}) = \langle T_{\mathit{sig},1}, \ldots, T_{\mathit{sig},k_{\mathit{sig}}} \rangle \in \mathcal{T}^*$ of parameter types. We assume a special signal $\delta \in \mathit{Sigs}$ with $\mathit{params}(\delta) = \langle\rangle$ to model spontaneous transitions. A *trigger* is of the form $\mathit{sig}(x_1, \ldots, x_{k_{\mathit{sig}}})$ such that $\mathit{sig} \in \mathit{Sigs}$ and $x_1, \ldots, x_{k_{\mathit{sig}}}$ are distinct typed variables with $\mathit{Type}(x_i) = T_{\mathit{sig},i}$ for $1 \leq i \leq k_{\mathit{sig}}$. The set of all triggers is denoted by $\mathit{Triggers}$. A *message* has the form $\mathit{sig}[v_1, \ldots, v_{k_{\mathit{sig}}}]$ such that $\mathit{sig} \in \mathit{Sigs} \setminus \{\delta\}$ and each $v_i \in \mathit{Dom}(T_{\mathit{sig},i})$; the set of all messages is denoted by Msgs.

A state machine $\mathit{SM}(C)$ is composed of

1. a non-empty, finite set $\mathit{States}(\mathit{SM}(C))$ of states,
2. an initial state $\mathit{InitialState}(\mathit{SM}(C)) \in \mathit{States}(\mathit{SM}(C))$, and
3. a finite set $\mathit{Trans}(\mathit{SM}(C)) \subseteq \mathit{States}(\mathit{SM}(C)) \times \mathit{Triggers} \times \mathcal{L}_{\mathbb{B}} \times \mathcal{L}_{\text{Stmt}} \times \mathit{States}(\mathit{SM}(C))$ of transitions.

For each transition $t = \langle s, \mathit{sig}(x_1, \ldots, x_k), g, e, s' \rangle \in \mathit{Trans}(\mathit{SM}(C))$, we define $\mathit{Source}(t) = s$, $\mathit{TriggerSig}(t) = \mathit{sig}$, $\mathit{Trigger}(t) = \mathit{sig}(x_1, \ldots, x_k)$, $\mathit{Guard}(t) = g$, $\mathit{Effect}(t) = e$, and $\mathit{Target}(t) = s'$. Parameters of triggers must be attributes of the owning class: $\langle s, \mathit{sig}(x_1, \ldots, x_k), g, e, s' \rangle \in \mathit{Trans}(\mathit{SM}(C)) \Rightarrow \forall 1 \leq i \leq k : x_i \in \mathit{Attrs}(C)$.

A *global configuration* c of a system maps each object $o \in O$ to a triple $\langle \mathit{Active}, \mathit{AttrVal}, \mathit{InputQ} \rangle$, where (i) $c(o).\mathit{Active} \in \mathit{States}(\mathit{SM}(\mathit{Class}(o)))$ is the current active state, (ii) $c(o).\mathit{AttrVal}$ is a function that maps each attribute $x \in \mathit{Attrs}(\mathit{Class}(o))$ to its current value in $\mathit{Dom}(\mathit{Type}(x))$, and (iii) $c(o).\mathit{InputQ} \in \mathit{Msgs}^*$ describes the current contents of the input queue. The action language $\mathcal{L}$ is always interpreted in the context of a global configuration c and an object o. For a side-effect free Boolean expression ϕ in $\mathcal{L}_{\mathbb{B}}$, $\mathit{eval}(c, o, \phi)$ evaluates it in the context of c and o, and returns **false** or **true**. Given a statement γ in $\mathcal{L}_{\text{Stmt}}$, $\mathit{execute}(c, o, \gamma)$ executes it in the context of c and o, and returns a new global configuration c'.

The interleaving state space of a system is the tuple $M = \langle C, c_{\text{init}}, \Delta \rangle$, where C is the set of all global configurations, $c_{\text{init}} \in C$ is the initial configuration, and the transition relation $\Delta \subseteq C \times A \times C$ is the minimal relation fulfilling the following rules:

1. Case: spontaneous transitions.

   If $t = \langle s, \delta(), g, e, s' \rangle \in \mathit{Trans}(\mathit{SM}(\mathit{Class}(o)))$ for an object $o \in O$, then the "spontaneous transition instance" $\langle o, t \rangle$ is enabled in c, denoted by $\mathit{enabled}(c, \langle o, t \rangle)$, if (i) the source is active: $c(o).\mathit{Active} = s$, and (ii) the guard holds: $\mathit{eval}(c, o, g) = \mathbf{true}$.

   If $\mathit{enabled}(c, \langle o, t \rangle)$ holds, then $\langle c, \langle o, t \rangle, c' \rangle \in \Delta$, where c' is equal to $\mathit{execute}(c, o, e)$ except that $c'(o).\mathit{Active} = s'$.

2. Case: message triggered transitions.

   If $t = \langle s, \mathit{sig}(x_1, \ldots, x_k), g, e, s' \rangle \in \mathit{Trans}(\mathit{SM}(\mathit{Class}(o)))$ for an object o and a normal signal $\mathit{sig} \neq \delta$, then the "message triggered transition instance" $\langle o, t \rangle$ is enabled in c, denoted by $\mathit{enabled}(c, \langle o, t \rangle)$, if (i) $c(o).\mathit{Active} = s$, (ii) $c(o).\mathit{InputQ} = \langle \mathit{sig}[v_1, \ldots, v_k], \ldots \rangle$, and (iii) $\mathit{eval}(c^\star, o, g) = \mathbf{true}$, where $c^\star$ is equal to c except that the first message has been received: $c^\star(o).\mathit{InputQ} = \mathit{dequeue}(c(o).\mathit{InputQ})$, and $\forall 1 \leq i \leq k : c^\star(o).\mathit{AttrVal}(x_i) = v_i$. We define $\mathit{dequeue}(\langle m_1, \ldots, m_n \rangle) = \langle m_2, \ldots, m_n \rangle$.

   If $\mathit{enabled}(c, \langle o, t \rangle)$ holds, then $\langle c, \langle o, t \rangle, c' \rangle \in \Delta$, where c' is equal to $\mathit{execute}(c^\star, o, e)$ except that $c'(o).\mathit{Active} = s'$.

3. Case: implicit consumption of messages.

   If no message triggered transition is enabled for an object o, the first message in its input queue can be implicitly consumed. Formally, the "implicit consumption action" $\langle o, \text{IMPCONS} \rangle$ is enabled in c, denoted by $\mathit{enabled}(c, \langle o, \text{IMPCONS} \rangle)$, if (i) $c(o).\mathit{InputQ} \neq \langle\rangle$, and (ii) there is no $t \in \mathit{Trans}(\mathit{SM}(\mathit{Class}(o)))$ for which $\mathit{enabled}(c, \langle o, t \rangle)$ holds and $\mathit{Trigger}(t) \neq \delta()$.

   If $\mathit{enabled}(c, \langle o, \text{IMPCONS} \rangle)$ holds, then $\langle c, \langle o, \text{IMPCONS} \rangle, c' \rangle \in \Delta$, where c' is equal to c except that $c'(o).\mathit{InputQ} = \mathit{dequeue}(c(o).\mathit{InputQ})$.

The initial configuration c_{init} is such that all state machines are in their initial states and all input queues are empty.

B DETAILED ENCODING

The three symbolic encodings of the transition relation of state machine models are shown below in more detail. The definitions and constraints already included in Sect. 3 are repeated here for completeness, and some are reformulated with more detail.

B.1 State Variables

Let $c \in C^B$ be a bounded global configuration. The set of state variables contains three elements for each object $o \in O$, with values derived from c as follows.

1. $\mathsf{Active}(o, s)$, where $s \in \mathit{States}(\mathit{SM}(\mathit{Class}(o)))$, determines whether s is the current active state in o, i.e. $\mathsf{Active}(o, s) = \big(c(o).\mathit{Active} = s\big)$.
2. $\mathsf{AttrVal}(o, x)$ with domain $\mathit{Dom}(\mathit{Type}(x))$, where $x \in \mathit{Attrs}(\mathit{Class}(o))$, determines the current value of x: $\mathsf{AttrVal}(o, x) = c(o).\mathit{AttrVal}(x)$.
3. $\mathsf{InputQ}(o)$ with domain $\mathit{Msgs}^0 \cup \cdots \cup \mathit{Msgs}^{\mathrm{QSIZE}}$ determines the contents of the input queue: $\mathsf{InputQ}(o) = c(o).\mathit{InputQ}$.

The corresponding next-state variables are denoted by $\mathit{next}\,(\mathsf{Active}(o, s))$, $\mathit{next}\,(\mathsf{AttrVal}(o, x))$, and $\mathit{next}\,(\mathsf{InputQ}(o))$, respectively.

B.2 Queues and Messages

Let $o \in O$ be an object. The input variable $\mathsf{Dispatch}(o)$ with domain $\mathit{Sigs} \cup \{\mathbf{none}\}$ determines which message $\mathit{sig}[\ldots]$ is being consumed by o. For each signal $\mathit{sig} \in \mathit{Sigs} \setminus \{\delta\}$, we add the constraint

$$\big(\mathsf{Dispatch}(o) = \mathit{sig}\big) \Rightarrow \big(\mathsf{CurrentSig}(o) = \mathit{sig}\big), \tag{13}$$

where $\mathsf{CurrentSig}(o)$ with domain $(\mathit{Sigs} \setminus \{\delta\}) \cup \{\mathbf{none}\}$ holds the first signal of $\mathsf{InputQ}(o)$, or the value **none** if $\mathsf{InputQ}(o)$ is empty. Let the queue after possible dequeuing be

$$\mathsf{ConsumedQ}(o) := \mathit{if} \begin{cases} \mathsf{Dequeue}(o) & : \ \mathit{dequeue}(\mathsf{InputQ}(o)) \\ \mathit{else} & : \ \mathsf{InputQ}(o), \end{cases} \tag{14}$$

where

$$\mathsf{Dequeue}(o) := \bigvee \big\{ \mathsf{Dispatch}(o) = \mathit{sig} \mid \mathit{sig} \in \mathit{Sigs} \setminus \{\delta\} \big\}. \tag{15}$$

The domain of $\mathsf{ConsumedQ}(o)$ is $\mathit{Msgs}^0 \cup \cdots \cup \mathit{Msgs}^{\mathrm{QSIZE}}$. Let the input variable $\mathsf{NewMsg}(o)$ with domain $\mathit{Msgs} \cup \{\mathbf{none}\}$ denote the message possibly being sent to o. Using the shorthands $\mathsf{Recv}(o) := (\mathsf{NewMsg}(o) \neq \mathbf{none})$ and $\mathsf{Space}(o) := (\mathit{length}(\mathsf{ConsumedQ}(o)) < \mathrm{QSIZE})$, the contents of the queue in the next configuration is fixed by the constraint

$$\mathit{next}\,(\mathsf{InputQ}(o)) = \mathit{if} \begin{cases} \mathsf{Recv}(o) \wedge \mathsf{Space}(o) & : \ \mathsf{ConsumedQ}(o) + \mathsf{NewMsg}(o) \\ \mathit{else} & : \ \mathsf{ConsumedQ}(o), \end{cases} \tag{16}$$

where $+$ denotes list concatenation. If $\mathsf{Recv}(o)$ is true but $\mathsf{Space}(o)$ is not, the capacity of the queue is being exceeded. We rule out this case with the constraint

$$\mathsf{Recv}(o) \Rightarrow \mathsf{Space}(o). \tag{17}$$

B.3 Control Logic of State Machines

Let $o \in O$ be an object and let $States = States(SM(Class(o)))$ be its set of states and $Trans = Trans(SM(Class(o)))$ its set of transitions. For each transition $t \in Trans$, an input variable $\mathsf{Fire}(o,t)$ determines whether t is being fired in o.

A state $s \in States$ 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, i.e.

$$next\,(\mathsf{Active}(o,s)) \Leftrightarrow \mathsf{Enter}(o,s) \vee \big(\neg\mathsf{Exit}(o,s) \wedge \mathsf{Active}(o,s)\big), \tag{18}$$

$$\mathsf{Enter}(o,s) := \bigvee \big\{\mathsf{Fire}(o,t) \mid t \in Trans \text{ and } Target(t) = s\big\}, \tag{19}$$

$$\mathsf{Exit}(o,s) := \bigvee \big\{\mathsf{Fire}(o,t) \mid t \in Trans \text{ and } Source(t) = s\big\}. \tag{20}$$

To avoid firing several transitions in o at the same time, the constraint

$$AtMostOne\,(\{\mathsf{Fire}(o,t) \mid t \in Trans \text{ and } Source(t) = s\}) \tag{21}$$

is added that allows at most one of the disjuncts in (20) to be true.

Let $R_o(s, sig) = \{t \in Trans \mid Source(t) = s \text{ and } TriggerSig(t) = sig\}$ be the set of transitions leaving state $s \in States$ with signal $sig \in Sigs$. For each nonempty $R_o(s, sig)$, define

$$\mathsf{Feasible}(o,s,sig) := \mathsf{Active}(o,s) \wedge \bigvee \big\{\mathsf{EvalGuard}(o,t) \mid t \in R_o(s,sig)\big\}. \tag{22}$$

The function $\mathsf{EvalGuard}(o,t)$ is defined later in Sect. B.4. The constraint

$$\bigvee \big\{\mathsf{Fire}(o,t) \mid t \in R_o(s,sig)\big\} \Leftrightarrow \big(\mathsf{Dispatch}(o) = sig\big) \wedge \mathsf{Feasible}(o,s,sig) \tag{23}$$

for each nonempty $R_o(s, sig)$ ensures that if sig is consumed and some transition in $R_o(s, sig)$ is enabled, then one of the transitions will be fired. To prevent a non-enabled transition from firing, we add for all $t \in Trans$ the constraint

$$\mathsf{Fire}(o,t) \Rightarrow \mathsf{EvalGuard}(o,t). \tag{24}$$

To avoid implicit consumption of the special signal used in the trigger of spontaneous transitions, we add the constraint

$$\big(\mathsf{Dispatch}(o) = \delta\big) \Rightarrow \bigvee \big\{\mathsf{Feasible}(o,s,\delta) \mid s \in States \wedge R_o(s,\delta) \neq \emptyset\big\}, \tag{25}$$

which says that if δ is consumed, then a spontaneous transition must be fired.

Object o is *scheduled* if it is consuming a signal, and the object is *ready* for execution if there is a signal it can consume. The system is in a *deadlock* if no object is ready.

$$\mathsf{Scheduled}(o) := \big(\mathsf{Dispatch}(o) \neq \mathbf{none}\big), \tag{26}$$

$$\begin{aligned}\mathsf{Ready}(o) := {}& \big(\mathsf{CurrentSig}(o) \neq \mathbf{none}\big) \vee \\ & \bigvee \big\{\mathsf{Feasible}(o, s, \delta) \mid s \in \mathit{States} \wedge R_o(s, \delta) \neq \emptyset\big\},\end{aligned} \tag{27}$$

$$\mathsf{Deadlock}() := \neg \bigvee \big\{\mathsf{Ready}(o) \mid o \in O\big\}. \tag{28}$$

We require that at least one object is scheduled in each step by the constraint

$$\bigvee \big\{\mathsf{Scheduled}(o) \mid o \in O\big\}. \tag{29}$$

B.4 Effects and Data

Consider a transition instance $\langle o, t\rangle$ with $\mathit{Effect}(t) = \mathit{stmt}_1 \ldots \mathit{stmt}_n$. For each stmt_i, each object $\hat{o}$, and each attribute $\hat{x} \in \mathit{Attrs}(\mathit{Class}(\hat{o}))$, the function $\mathsf{Assign}^i_{o,t}(\hat{o}, \hat{x})$ evaluates to **true** if stmt_i writes to $\hat{o}.\hat{x}$ in case the transition instance is executed. Because the written value may be used later in the same effect, we define the function $\mathsf{Temp}^i_{o,t}(\hat{o}, \hat{x})$ with domain $\mathit{Dom}(\mathit{Type}(\hat{x}))$ that holds the intermediate value of attribute $\hat{x}$ in $\hat{o}$ right after stmt_i has been executed. Similarly, function $\mathsf{Eval}^i_{o,t}(\mathit{expr})$ gives the value of expression expr evaluated in the context of object o right after stmt_i has been executed.

Message arguments
Let $\mathit{Trigger}(t) = \mathit{sig}(x_1, \ldots, x_k)$. If $\mathsf{CurrentSig}(o) = \mathit{sig}$, then let the first message in $\mathsf{InputQ}(o)$ be $\mathit{sig}[v_1, \ldots, v_k]$. Otherwise, let each v_i have the dummy value $\mathit{default}(\mathit{Type}(x_i))$. We view the implicit assignment of arguments $v_1, \ldots, v_k$ to attributes $x_1, \ldots, x_k$ as the zeroth statement of the effect and define

$$\mathsf{Assign}^0_{o,t}(\hat{o}, \hat{x}) := \begin{cases} \mathbf{true} & \text{if } \hat{o} = o \text{ and } \hat{x} = x_i \text{ for some } i = 1, \ldots, k, \\ \mathbf{false} & \text{otherwise,} \end{cases} \tag{30}$$

$$\mathsf{Temp}^0_{o,t}(\hat{o}, \hat{x}) := \begin{cases} v_i & \text{if } \hat{o} = o \text{ and } \hat{x} = x_i \text{ for some } i = 1, \ldots, k, \\ \mathsf{AttrVal}(\hat{o}, \hat{x}) & \text{otherwise.} \end{cases} \tag{31}$$

According to the definition of executing message triggered transitions (Appendix A), guards are evaluated *after* the implicit assignment of message arguments, therefore

$$\mathsf{EvalGuard}(o, t) := \mathsf{Eval}^0_{o,t}(\mathit{Guard}(t)). \tag{32}$$

Expressions
If expr is one of the expressions `true`, `false`, `null`, `this`, or an integer literal, then $\mathsf{Eval}^i_{o,t}(\mathit{expr})$ has the constant value **true**, **false**, **null**, o, or the value of the integer, respectively. If expr is an infix expression $\mathit{leftexpr}\ \mathit{op}\ \mathit{rightexpr}$, let $\circ_{\mathit{op}}$ be the semantic equivalent for operator op (e.g. signed addition modulo 32) and define

$$\mathsf{Eval}^i_{o,t}(\mathit{expr}) := \mathsf{Eval}^i_{o,t}(\mathit{leftexpr}) \circ_{\mathit{op}} \mathsf{Eval}^i_{o,t}(\mathit{rightexpr}). \tag{33}$$

If $expr = refexpr\,.\hat{x}$ is an attribute access expression and $Type(refexpr) = \mathbb{T}_{\widehat{C}}$, then let $\hat{o}_1, \ldots, \hat{o}_m$ be the objects of class $\widehat{C}$ and define

$$\mathsf{Eval}^i_{o,t}(expr) := \mathit{if} \begin{cases} \mathsf{Eval}^i_{o,t}(refexpr) = \hat{o}_1 & : \ \mathsf{Temp}^i_{o,t}(\hat{o}_1, \hat{x}) \\ \vdots & \\ \mathsf{Eval}^i_{o,t}(refexpr) = \hat{o}_m & : \ \mathsf{Temp}^i_{o,t}(\hat{o}_m, \hat{x}) \\ else & : \ default(Type(\hat{x})). \end{cases} \tag{34}$$

The step encodings rely on information about which attributes are being read or written by transition instances. We use a function $\mathsf{MayRefer}^i_{o,t}(refexpr, \hat{o})$ to see whether the expression $refexpr$, evaluated after the ith statement in the effect of the transition instance $\langle o, t\rangle$, refers to object $\hat{o}$. In the interleaving encoding and the dynamic $\exists$-step encoding, we evaluate this accurately by setting

$$\mathsf{MayRefer}^i_{o,t}(refexpr, \hat{o}) := \big(\mathsf{Eval}^i_{o,t}(refexpr) = \hat{o}\big). \tag{35}$$

In the static $\exists$-step encoding, we use a statically evaluated over-approximation

$$\mathsf{MayRefer}^i_{o,t}(refexpr, \hat{o}) := \begin{cases} \textbf{false} & \text{if } refexpr = \texttt{this} \text{ and } \hat{o} \neq o, \\ \textbf{true} & \text{otherwise.} \end{cases} \tag{36}$$

Assignment Statements

If $stmt_i$ is an assignment statement of the form $refexpr\,.\hat{x}$ `=` $rhsexpr$`;`, then for all objects $\hat{o}$ such that $Type(refexpr) = \mathbb{T}_{Class(\hat{o})}$

$$\mathsf{Assign}^i_{o,t}(\hat{o}, \hat{x}) := \mathsf{MayRefer}^i_{o,t}(refexpr, \hat{o}), \tag{37}$$

$$\mathsf{Temp}^i_{o,t}(\hat{o}, \hat{x}) := \mathit{if} \begin{cases} \mathsf{Eval}^{i-1}_{o,t}(refexpr) = \hat{o} & : \ \mathsf{Eval}^{i-1}_{o,t}(rhsexpr) \\ else & : \ \mathsf{Temp}^{i-1}_{o,t}(\hat{o}, \hat{x}). \end{cases} \tag{38}$$

If $stmt_i$ is not an assignment statement or $\hat{o}$ is not an object of the class referenced by $refexpr$, then trivially $\mathsf{Assign}^i_{o,t}(\hat{o}, \hat{x}) :=$ **false** and $\mathsf{Temp}^i_{o,t}(\hat{o}, \hat{x}) := \mathsf{Temp}^{i-1}_{o,t}(\hat{o}, \hat{x})$.

Send Statements

Let $\hat{o}$ be an object and $\widehat{C}$ its class. We restrict the model so that at most one statement among $stmt_1, \ldots, stmt_n$ sends a signal to class $\widehat{C}$. If there is no such statement, we set $\mathsf{Send}_{o,t}(\hat{o}) :=$ **false**. Otherwise, let $stmt_i$ be such a statement `send` sig`(`$arg_1, \ldots, arg_m$`) to` $targetexpr$`;`, and define

$$\mathsf{Send}_{o,t}(\hat{o}) := \mathsf{Fire}(o, t) \wedge \mathsf{MayRefer}^{i-1}_{o,t}(targetexpr, \hat{o}). \tag{39}$$

Let M be the message $sig[\mathsf{Eval}^{i-1}_{o,t}(arg_1), \ldots, \mathsf{Eval}^{i-1}_{o,t}(arg_m)]$. In the interleaving and dynamic $\exists$-step encoding, $\mathsf{Send}_{o,t}(\hat{o})$ is true if and only if the transition instance $\langle o, t\rangle$ is executed and it sends M to $\hat{o}$. Therefore, we set the constraint

$$\mathsf{Send}_{o,t}(\hat{o}) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = M\big). \tag{40}$$

In the static $\exists$-step encoding, $\mathsf{MayRefer}^{i-1}_{o,t}(targetexpr, \hat{o})$ is only an approximation. We take the actual value of $targetexpr$ into account by introducing two constraints

$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \left(\mathsf{Eval}^{i-1}_{o,t}(\mathit{targetexpr}) = \hat{o}\right) \Rightarrow \left(\mathsf{NewMsg}(\hat{o}) = M\right), \tag{41}$$
$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \neg\left(\mathsf{Eval}^{i-1}_{o,t}(\mathit{targetexpr}) = \hat{o}\right) \Rightarrow \left(\mathsf{NewMsg}(\hat{o}) = \mathbf{none}\right). \tag{42}$$

In all three encodings, $\mathsf{NewMsg}(\hat{o})$ is fixed by a constraint when $\mathsf{Send}_{o,t}(\hat{o})$ is true.

Assert Statements
If stmt_i is a statement of the form `assert` *condexpr*`;`, then we add the following to the set of invariants to be checked:

$$\mathsf{Fire}(o,t) \Rightarrow \mathsf{Eval}^{i-1}_{o,t}(\mathit{condexpr}). \tag{43}$$

Attribute Access
An attribute $\hat{x}$ in object $\hat{o}$ is written if the transition instance is executed and the attribute is assigned either by an assignment statement or by implicit assignment of message parameters. Therefore we define

$$\mathsf{Write}_{o,t}(\hat{o},\hat{x}) := \mathsf{Fire}(o,t) \wedge \bigvee \left\{\mathsf{Assign}^{i}_{o,t}(\hat{o},\hat{x}) \mid i = 0,\ldots,n\right\} \tag{44}$$

and add the constraint

$$\mathsf{Write}_{o,t}(\hat{o},\hat{x}) \Rightarrow \left(\mathit{next}\,(\mathsf{AttrVal}(\hat{o},\hat{x})) = \mathsf{Temp}^{n}_{o,t}(\hat{o},\hat{x})\right). \tag{45}$$

Even though $\mathsf{Assign}^{i}_{o,t}(\hat{o},\hat{x})$ may be an approximation and $\mathsf{Write}_{o,t}(\hat{o},\hat{x})$ may be true even if the attribute is not assigned, the constraint (45) fixes the correct value for the attribute in the next configuration because $\mathsf{Temp}^{n}_{o,t}(\hat{o},\hat{x})$ is evaluated accurately.

In both ∃-step encodings, we need to track reading as well as writing. We define a function that is true (at least) if $\hat{x}$ in $\hat{o}$ is being read by the transition instance, either in the guard or in a statement:

$$\mathsf{Read}_{o,t}(\hat{o},\hat{x}) := \mathsf{Fire}(o,t) \wedge \bigvee P_{o,t}(\hat{o},\hat{x}), \tag{46}$$

where the set $P_{o,t}(\hat{o},\hat{x})$ consists of (i) the functions $\mathsf{MayRefer}^{0}_{o,t}(\mathit{refexpr},\hat{o})$ for every expression *refexpr* such that $\mathit{refexpr}\,.\,\hat{x}$ is a subexpression of *Guard*(t) and (ii) the functions $\mathsf{MayRefer}^{i-1}_{o,t}(\mathit{refexpr},\hat{o})$ for every *refexpr* and i such that $\mathit{refexpr}\,.\,\hat{x}$ is a subexpression of stmt_i.

Frame Conditions
Let Θ be the set of all transition instances. The only situation when $\mathsf{NewMsg}(\hat{o})$ is not fixed by one of the constraints (40), (41), or (42), is when $\mathsf{Send}_{o,t}(\hat{o})$ is false for all $\langle o,t\rangle \in \Theta$. Similarly, $\mathit{next}\,(\mathsf{AttrVal}(\hat{o},\hat{x}))$ is not fixed when all functions $\mathsf{Write}_{o,t}(\hat{o},\hat{x})$ are false. To fix these, we add the constraints

$$\neg\bigvee\left\{\mathsf{Send}_{o,t}(\hat{o}) \mid \langle o,t\rangle \in \Theta\right\} \Rightarrow \left(\mathsf{NewMsg}(\hat{o}) = \mathbf{none}\right), \tag{47}$$
$$\neg\bigvee\left\{\mathsf{Write}_{o,t}(\hat{o},\hat{x}) \mid \langle o,t\rangle \in \Theta\right\} \Rightarrow \left(\mathit{next}\,(\mathsf{AttrVal}(\hat{o},\hat{x})) = \mathsf{AttrVal}(\hat{o},\hat{x})\right). \tag{48}$$

B.5 Step Constraints

In the interleaving encoding, at most one object is scheduled at a time:

$$AtMostOne\left(\{\mathsf{Scheduled}(o) \mid o \in O\}\right). \tag{49}$$

In the $\exists$-step encodings, this is replaced by other constraints as follows. We require that a transition instance must not send a message to an object if a preceding transition instance has already done so, and it must not read an attribute that a preceding transition instance has written. Hence the constraints

$$\bigvee \left\{\mathsf{Send}_{o^-,t^-}(\hat{o}) \mid \langle o^-, t^-\rangle \prec \langle o, t\rangle\right\} \Rightarrow \neg\mathsf{Send}_{o,t}(\hat{o}), \tag{50}$$

$$\bigvee \left\{\mathsf{Write}_{o^-,t^-}(\hat{o}, \hat{x}) \mid \langle o^-, t^-\rangle \prec \langle o, t\rangle\right\} \Rightarrow \neg\mathsf{Read}_{o,t}(\hat{o}, \hat{x}). \tag{51}$$

In addition, a message cannot be consumed from a queue whose capacity has already been exceeded by a preceding transition instance. Let $\langle o, t\rangle$ be a transition instance such that $\mathit{TriggerSig}(t) \neq \delta$. In the dynamic $\exists$-step encoding, we add the constraint

$$\bigvee \left\{\mathsf{Send}_{o^-,t^-}(o) \mid \langle o^-, t^-\rangle \prec \langle o, t\rangle\right\} \wedge \mathsf{QueueFull}(o) \Rightarrow \neg\mathsf{Fire}(o, t), \tag{52}$$

where $\mathsf{QueueFull}(o) := (\mathit{length}(\mathsf{InputQ}(o)) = \text{QSIZE})$. In the static $\exists$-step encoding, we replace $\mathsf{QueueFull}(o)$ with a static approximation **true**, and the constraint strengthens to

$$\bigvee \left\{\mathsf{Send}_{o^-,t^-}(o) \mid \langle o^-, t^-\rangle \prec \langle o, t\rangle\right\} \Rightarrow \neg\mathsf{Fire}(o, t). \tag{53}$$

B.6 Analysis

Assume that the state variables represent a configuration $c \in C^B$ and that the input variables and next-state variables fulfill all constraints. We define the current step $S \subseteq A$ as the set consisting of all transition instances $\langle o, t\rangle$ such that $\mathsf{Fire}(o, t)$ is true, and all implicit consumption actions $\langle o, \text{IMPCONS}\rangle$ such that o consumes a message without firing a transition, i.e. $\mathsf{Dispatch}(o) \neq$ **none** but $\mathsf{Fire}(o, t)$ is false for all $t \in \mathit{Trans}(\mathit{SM}(\mathit{Class}(o)))$.

Constraints (13)–(25) allow an object o to appear in at most one action $a \in S$. If $\langle o, t\rangle \in S$ for some t, then $\mathit{enabled}(c, \langle o, t\rangle)$ holds, assuming that $\mathit{eval}(c^\star, o, \mathit{Guard}(t)) = \mathsf{EvalGuard}(o, t)$. By (18), $\mathit{next}\left(\mathsf{Active}(o, s)\right)$ holds if and only if $s = \mathit{Target}(t)$. If S contains an implicit consumption action $\langle o, \text{IMPCONS}\rangle$, then $\mathsf{Dispatch}(o)$ cannot be δ because that would violate (23) or (25). Also by (23), $\mathsf{Feasible}(o, s, \mathsf{Dispatch}(o))$ must be false for all s and thus $\mathit{enabled}(c, \langle o, t\rangle)$ is false for all transitions t and therefore $\mathit{enabled}(c, \langle o, \text{IMPCONS}\rangle)$ is true. The constraints also allow the case where no action is executed in object o, i.e. $\mathsf{Dispatch}(o) =$ **none** and $\mathsf{Fire}(o, t) =$ **false** for all t.

Correctness of the step encodings is based on the following arguments. (i) Constraint (29) forbids an empty step. (ii) Step constraints (49)–(53) cannot be broken in any encoding if the step S contains only one action. (iii)

The encoding ensures that $a \in S$ implies *enabled*(c, a). (iv) In $\exists$-steps, constraints (50) and (51) make sure that if $a_1, \ldots, a_k \in S$ and $a_1 \prec \cdots \prec a_k$, then *enabled*$(c, a_k)$ is still true if c is replaced by the configuration that results from executing $a_1, \ldots, a_{k-1}$. This property requires that all implicit consumption actions precede transition instances in the order $\prec$. (v) If the capacity QSIZE of the input queue of some object o would be exceeded by executing a step S, then constraint (17) is violated. Constraints (52) and (53) forbid the case where (17) is fixed again by augmenting S with a transition instance $\langle o, t \rangle$ that consumes a message from the queue. (vi) In all cases, the next-state variables are fixed when the state ant input variables are fixed, and they reflect the changes made by executing all actions $a \in S$, where S is obtained from the input variables as described above.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
TECHNICAL REPORTS

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.

HUT-TCS-B24 Jori Dubrovin, Tommi Junttila, Keijo Heljanko
Symbolic Step Encodings for Object Based Communicating State Machines. December 2007.

ISBN 978-951-22-9193-9
ISSN 0783-540X