Symbolic Step Encodings for Object Based Communicating State Machines [*]

Jori Dubrovin, Tommi Junttila, and Keijo Heljanko

Helsinki University of Technology (TKK)
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
{Jori.Dubrovin, Tommi.Junttila, Keijo.Heljanko}@tkk.fi

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 ∃-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 ∃-step semantics often significantly outperforms the traditional interleaving semantics without any substantial blowup in the BMC encoding as a SAT formula.

1 Introduction

This paper describes a method that allows UML state machine models enriched with a Java-like object oriented action language to be efficiently model checked with symbolic model checking techniques. We describe the theoretical background of a tool that encodes the transition relation of these object based communicating state machines in the NuSMV [1] model checker input language in a novel way that, in particular, makes the search for short counterexamples competitive with the state-of-the-art explicit state model checker Spin [2].

The model checking approach our encoding is best suited for is bounded model checking (BMC) [3] that 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 semantics* encodings of the transition relation for object based communicating state machines. Similarly to partial

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

order reduction techniques such as stubborn, ample, persistent, or sleep sets for explicit state model checking (see e.g., [4]) 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 here 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 [5]. The model checking tool we have developed can also handle other aspects of UML state machines not covered in this paper for the sake of clarity, see Sect. 4 for details.

Our encoding is a significant generalization of the approach of Rintanen [6] on AI planning, where the notion of ∃-step semantics for planning problems was first systematically employed. There are the following substantial differences to this earlier work. First of all, our setup employs a UML state machine based model of concurrency enriched with asynchronous message passing and full object based data handling features such as dynamic object references. We show how the ∃-step semantics can still be efficiently encoded with a full Java-like action language. The main challenge is indeed the handling of the complex action language parts of the encoding, something that is non-existent in the AI planning domain. Secondly, our approach also introduces the novel use of a dynamic dependency relation to optimize the encoding even further. This allows for example concurrent attribute access of different instances of the same object at the same time step. The approach of Rintanen is based on a static dependency relation.

Other Related Work. In the area of SAT based BMC, Heljanko was the first to consider exploiting the concurrency in encoding the transition relation [7]. That paper considers BMC for 1-bounded Petri nets using the ∀-step semantics (the classical Petri net step semantics, see e.g., [8]). The intuitive idea is that a set of actions can be executed at the same time step in ∀-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 ∀-step semantics [9] (see also [6]). The work of Dimopoulos et al. was the first one to use an ∃-step semantics like approach in hand optimized planning encodings of [10], the idea of which was later formalized and automated in the SAT based planning system of Rintanen et al. [6, 11]. On the BMC side, Ogata et al. [12] and Jussila et al. [13, 14] 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].

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, Figs. 1(a)–(c) show a part of a simple heart beat monitor system described in UML state machines with Java-like action language annotations. The dynamic behavior of a system is captured by its *interleaving state space*

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

where C is the set of all *global configurations*, $c_{\mathrm{init}} \in C$ is an *initial global configuration*, and the *transition relation* $\mathit{\Delta} \subseteq C \times A \times C$ describes how configurations may evolve to others: $\langle c, a, c' \rangle \in \mathit{\Delta}$ 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 Figs. 1(a)–(c); if the action $\langle o_1, t_{22} \rangle$ (corresponding to object o_1 firing its transition t_{22}) is executed in configuration c_1, it changes to c_2. 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) $\forall 1 \leq i \leq k : \langle c_{i-1}, a_i, c_i \rangle \in \mathit{\Delta}$, and (iii) $c_k = c''$.

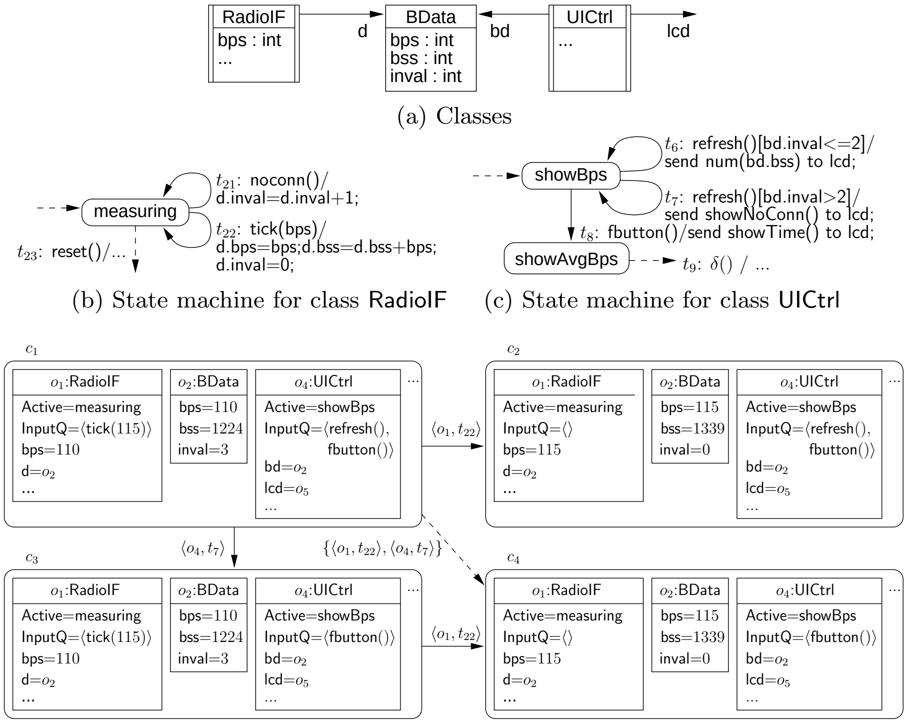
The basic idea in ∃-step semantics exploited in this paper is to augment the state space with "shortcut edges" so that, under certain conditions, several actions can be executed "at the same time step". This is formalized in the definition below.

Definition 1. *The* ∃-step state space *corresponding to the interleaving state space* $M = \langle C, c_{\mathrm{init}}, \mathit{\Delta} \rangle$ *is the tuple*

$$M_\exists = \langle C, c_{\mathrm{init}}, \mathit{\Delta}_\exists \rangle$$

where the transition relation $\mathit{\Delta}_\exists \subseteq C \times 2^A \times C$ *contains a* step $\langle c, S, c' \rangle$ *iff*

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



(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

Fig. 1. A part of a simple heart beat monitor system

Continuing the running example, the dashed arrow in Fig. 1(d) denotes the step $\langle c_1, S, c_4\rangle$ with $S = \{\langle o_1, t_{22}\rangle, \langle o_4, t_7\rangle\}$. 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 ∃-step state space includes the interleaving state space in the sense that $\langle c, a, c'\rangle \in \Delta$ implies that the *unit step* $\langle c, \{a\}, c'\rangle$ is in $\Delta_\exists$. Conversely, if $\langle c, S, c'\rangle$ 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 ∃-step state space. Note that the definition of ∃-step semantics is a purely 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 [6], 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 $\exists$-step semantics in the AI planning domain, see [6], which also similarly soundly underapproximates the $\exists$-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 the technical report version of this paper [16] for the formal definitions.

We consider systems composed of a fixed and 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(b). 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, and (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* $\langle o, \text{IMPCONS} \rangle$, 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), the transition instance $\langle o_4, t_6 \rangle$ is only enabled in c_2, 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 $\langle o, \textsc{impcons} \rangle$ 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 $\langle o, \textsc{impcons} \rangle$ 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 [17] for expressing the guards and effects of transitions. 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 *refexpr*`.`x, where the type of *refexpr* is $\mathbb{T}_C$ and x is an attribute of class C, and (iv) infix expressions *leftexpr op rightexpr*, where *op* is one of `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `>`, `<`, `>=`, `<=`, `==`, or `!=`, with Java semantics [18]. 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.

The kinds of statements of the language are: (i) assignments of the form *refexpr*`.`x `=` *rhsexpr*`;`, (ii) send statements of the form `send` *sig*`(`$arg_1, \ldots, arg_k$`)` `to` *targetexpr*`;`. When a send statement is executed, the input queue of the object referred by *targetexpr* is appended with the message $sig[v_1, \ldots, v_k]$, where each v_i is the value of arg_i, and (iii) assertions of the form `assert` *condexpr*`;`. We want to check that *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 *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.

Method calls are not directly supported, but a non-recursive call can emulated by either inlining it at the call site or by modeling it as a pair of asynchronous message transmissions, one for the invocation and one for the return.

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 (in a subset of $\Delta_\exists$) 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 $\Delta^B = \left\{\langle c, a, c'\rangle \in \Delta \mid c, c' \in C^B\right\}$. As the semantics of Sect. 2.1 allows queues to grow indefinitely, by this restriction we effectively disregard all executions of the system where any queue at any point contains more than QSIZE messages. Consequently, some reachable configurations of the system may be omitted in the analysis. Whether or not such omission occurs could be detected by adding a suitable invariant to check, and if needed, QSIZE could be incremented to cover a larger set of the reachable configurations.

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 in a 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 $\prec$.

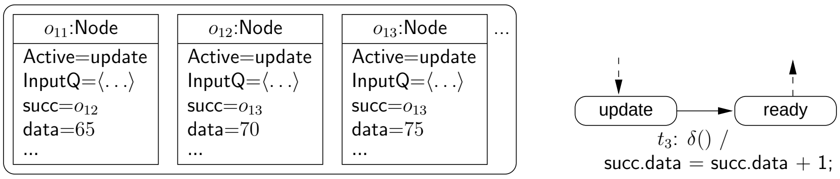
Step Encoding Overview. At a very coarse level, the encoding of $\exists$-steps contains three sets of constraints as follows.

1. For all actions a and all attributes $\hat{x}$: if a is executed and it writes to $\hat{x}$, then the value of $\hat{x}$ in the next global configuration is equal to the written value as if a was executed alone.
2. For all attributes $\hat{x}$: if no executed action writes to $\hat{x}$, then its value remains unchanged in the next global configuration.
3. For all actions a^- and a, and all attributes $\hat{x}$: if both a^- and a are executed and $a^- \prec a$ and a^- writes to $\hat{x}$, then a does not read from $\hat{x}$.

Constraint sets 1 and 2 together correspond to an encoding of *parallel* execution of actions. Any non-conflicting set of enabled actions can be executed at the same time, and the written values are computed independently for each action. Several actions are allowed to write to the same attribute at the same step, but only if the value written by each action is the same. For example, if action a_1 has the effect `o.x = o.y;` and action a_2 has the effect `o.x = o.y + 1;`, they cannot both be executed because there is no next-state value of `o.x` that satisfies the constraints. However, the parallel encoding allows the concurrent execution of a_1 together with another action a_3 whose effect is `o.y = o.x;`. Such a step swaps the values of `o.x` and `o.y`. This does not correspond to the sequential execution of a_1 and a_3 in either order, and thus we want to rule out non-$\exists$-steps like this.

For this reason, we add the further constraint set 3 of *step constraints*. These ensure that for any two concurrently executed actions $a^- \prec a$, the action that is greater w.r.t. the total order does not depend on any values written by the other action, and thus parallel execution leads to the same global configuration as executing first a^- and then a.

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 unit 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 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 the set of actions S in each step (the ordering in item 2 of Definition 1) 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 set of actions $S \subseteq A$ such that the step $\langle c, S, c' \rangle$



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

Fig. 2. An example for illustrating static and dynamic ∃-steps

belongs to $\Delta_{\exists}^{B}$. Furthermore, in the interleaving encoding, S only contains one action, and thus every step is a unit 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 $\mathit{Msgs}^{0} \cup \cdots \cup \mathit{Msgs}^{\text{QSIZE}}$ 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 section gives a rough overview of the encoding of state machine control logic. A more detailed definition can be found in [16].

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 in one step, (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 actions $S \subseteq A$ of the current step 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 $\langle o,\textsc{impcons}\rangle$ 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 $\mathit{next}\,(\mathsf{Active}(o,\mathsf{showAvgBps})) \Leftrightarrow \mathsf{Fire}(o,t_8) \vee \big(\neg\mathsf{Fire}(o,t_9) \wedge \mathsf{Active}(o,\mathsf{showAvgBps})\big)$, and the enabledness check for the action $\langle o,t_8\rangle$ is encoded in the constraint $\mathsf{Fire}(o,t_8) \Leftrightarrow \big((\mathsf{Dispatch}(o) = \mathsf{fbutton}) \wedge \mathsf{Active}(o,\mathsf{showBps})\big)$.

Object o is *scheduled* if it is consuming a signal or firing a spontaneously triggered transition, formalized in the function definition

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

To rule out an empty step with no actions, 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 [16]. 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 $\mathsf{Eval}(\mathtt{this})$ 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 `send` sig`(`$\mathit{arg}_1, \ldots, \mathit{arg}_m$`)` `to` $\mathit{targetexpr}$`;`, 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),

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

For each transition instance $\langle o, t\rangle$ and each assertion `assert` $\mathit{condexpr}$`;` 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

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

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

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

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

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

In the implementation, we employ maximal sharing of subformulas in the left-hand sides of (5), (6), (8), and (9), resulting in an encoding that is linear instead of quadratic in the number of transition instances. Furthermore, the left-hand sides of (8) and (9) are “free” in the sense that they are already present as subformulas of (5) and (6), and can therefore be shared.

The function $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$, which appears in constraint (9), is defined so that it evaluates to true if the transition instance $\langle o,t\rangle$ is executed and reads the attribute $\hat{o}.\hat{x}$. The constraint says that a transition instance $\langle o,t\rangle$ that reads an attribute $\hat{o}.\hat{x}$ can only be executed 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 $\langle o,\textsc{impcons}\rangle$ can implicitly *read* an attribute because the enabledness of $\langle o,\textsc{impcons}\rangle$ 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 $\langle o,\textsc{impcons}\rangle$ implicitly reading an attribute that has been written by a preceding action. No extra constraints are thus needed.

In order to strictly confine the analysis to the bounded transition relation $\Delta_{\exists}^{B}$, additional step constraints are placed that prevent a queue from growing past its bound and shrinking back within a single step. The details are in [16].

Static and Dynamic Steps. The difference between the two ∃-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 $\mathsf{Fire}(o,t) \wedge \big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big)$, and $\mathsf{Read}_{o,t}(\hat{o},\hat{x})$ is true iff $\mathsf{Fire}(o,t)$ is true and $\mathsf{Eval}(\mathit{refexpr}) = \hat{o}$ is true for any subexpression of the form $\mathit{refexpr}.\hat{x}$ 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 $\texttt{this}.\hat{x}$, 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

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

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

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

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

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 ∃-step semantics yields a smaller encoding, but dynamic ∃-step semantics permits more concurrency in a single step.

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 dominate the total encoding size in the experiments.

4 Experimental Results

We have implemented the symbolic encoding described above.[1] The tool 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 [5], (iii) deferring of events, and (iv) initial and choice pseudostates (see [19] for a symbolic *interleaving* semantics encoding of such extended state machines). It outputs the encoding as a NuSMV [1] program and currently supports model checking queries for deadlocks, implicit consumption of messages, assertion violations, and action language run-time errors. The tool chooses the fixed ordering of actions arbitrarily but deterministically based on the identifiers in the input file. 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 [20], with an uninitialized variable introduced in the adaptation.

Table 1 shows the results. The column “sem.” gives the semantics (interleaving, static $\exists$-steps, or dynamic $\exists$-steps) and the smallest bound for a counterexample under it, “SBMC zchaff” gives the minimum and maximum running time

[1] The source code and the models used in the experiments are 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.				

(in seconds) of 10 runs of the incremental BMC algorithm [21, 22] 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 invariant checking. The numbers in square brackets are the times used by the 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 as 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 ∃-step constraints in an instance is 4–8 % when static steps are used, and 5–15 % 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 ∃-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 counterexample is found within seconds. With BDDs, it seems in fact that the interleaving semantics is quite competitive with the step semantics. This is an interesting finding requiring further study. We also experimented with the BDD-based breadth-first enumeration of reachable states in NuSMV and the 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 [2] 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 in [5]. The last two columns in Table 1 give the running times and lengths of produced counterexamples of Spin with (i) the default depth-first search mode ("Spin DFS"), and (ii) the same with counterexample minimization option "-i" enabled. Partial order reductions and state compression ("COLLAPSE") were enabled in both modes. Different runs on the same model produced diverse results as the order of variable and process declarations in the Spin model varied between runs of the translator program due to some non-determinism in the libraries used. 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 consumes more time. More importantly, in cases it manages to find a counterexample, the produced counterexamples are often much longer than the minimal ones produced by BMC based methods. These very long counterexamples are not as useful in debugging the system as it becomes much harder for the user to locate the real source of the bug.

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 ∃-step semantics significantly outperform the traditional interleaving semantics approach, without any considerable blowup in the encoding as a SAT formula.

Our experimental results show that when searching for short counterexamples, symbolic model checking, especially with ∃-step semantics, can provide a competitive approach to model checking of asynchronous message passing protocols. This has traditionally believed to be a field where only explicit state model checkers can be efficient. We show that by using bounded model checking with our step encodings, we can find much shorter counterexamples than Spin does, and achieve this with competitive running times.

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 ∃-step semantics needs to be restricted in order to soundly accommodate the model checking of liveness properties along the lines of [23, 22]. And as the choice of the total ordering between actions in the encoding affects which steps are considered, a statically chosen good ordering might improve performance, but this needs further investigation.

References

1. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An opensource tool for symbolic model checking. In: CAV'02. Volume 2404 of LNCS., Springer (2002) 359–364
2. Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2004)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS'99. Volume 1579 of LNCS., Springer (1999) 193–207
4. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS. Springer (1998) 429–528
5. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model checking dynamic and hierarchical UML state machines. In: Proc. MoDeV^{2}a: Model Development, Validation and Verification. (2006) 94–110
6. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence **170**(12-13) (2006) 1031–1080
7. Heljanko, K.: Bounded reachability checking with process semantics. In: CONCUR'01. Volume 2154 of LNCS., Springer (2001) 218–232
8. Best, E., Devillers, R.R.: Sequential and concurrent behaviour in Petri net theory. Theoretical Computer Science **55**(1) (1987) 87–136
9. Kautz, H.A., Selman, B.: Pushing the envelope: Planning, propositional logic and stochastic search. In: AAAI'96/IAAI'96, Vol. 2, AAAI Press (1996) 1194–1201
10. Dimopoulos, Y., Nebel, B., Koehler, J.: Encoding planning problems in nonmonotonic logic programs. In: ECP'97. Volume 1348 of LNCS., Springer (1997) 169–181
11. Wehrle, M., Rintanen, J.: Planning as satisfiability with relaxed ∃-step plans. In: AI'07. Volume 4830 of LNCS., Springer (2007) 244–253
12. Ogata, S., Tsuchiya, T., Kikuno, T.: SAT-based verification of safe Petri nets. In: ATVA'04. Volume 3299 of LNCS., Springer (2004) 79–92
13. Jussila, T.: BMC via dynamic atomicity analysis. In: ACSD'04, IEEE Computer Society (2004) 197–206
14. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. International Journal on Software Tools for Technology Transfer **7**(2) (2005) 89–101
15. Jussila, T.: On Bounded Model Checking of Asynchronous Systems. Doctoral dissertation, Helsinki Univ. of Technology (2005)
16. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. Technical Report B24, Helsinki Univ. of Technology, Lab. for Theoretical Computer Science (2007)
17. Dubrovin, J.: Jumbala — An action language for UML state machines. Research Report A101, Helsinki Univ. of Technology, Lab. for Theoretical Computer Science (2006)
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Third edn. Addison-Wesley (2005)
19. Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical UML state machines. (2008) To appear in ACSD'08.
20. Kamel, M., Leue, S.: Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. International Journal on Software Tools for Technology Transfer **2**(4) (2000) 394–409
21. Heljanko, K., Junttila, T., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: CAV'05. Volume 3576 of LNCS., Springer (2005) 98–111
22. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science **2**(5:5) (2006)

23. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming **3**(4&5) (2003) 519–550