Publication IV

Jori Dubrovin and Tommi Junttila and Keijo Heljanko. Exploiting Step Semantics for Efficient Bounded Model Checking of Asynchronous Systems. Accepted for publication in *Science of Computer Programming*, Special Issue on Automated Verification of Critical Systems, 27 pages, available online 23 July 2011.

ARTICLE IN PRESS

Science of Computer Programming ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Exploiting step semantics for efficient bounded model checking of asynchronous systems☆

Jori Dubrovin*, Tommi Junttila, Keijo Heljanko

Aalto University, School of Science, Department of Information and Computer Science, PO Box 15400, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Bounded model checking
Step encodings
Process semantics
Asynchronous systems
SMT

ABSTRACT

This paper discusses bounded model checking (BMC) for asynchronous systems. Bounded model checking is a technique that employs the power of efficient SAT and SMT solvers for model checking. The main contribution of this paper is the presentation of a simple modeling formalism independent way of translating an asynchronous system into a transition formula for three partial order semantics: the ∃-step semantics, its generalization, the relaxed ∃-step semantics, and a novel variant that combines the latter with the idea of process semantics. Step and process semantics have been introduced in earlier works for different low level asynchronous system formalisms to improve the efficiency of BMC. However, this paper is the first one showing how to translate the semantics for any asynchronous system modeling formalism including high-level data manipulation operations while encoding the independence of actions in a dynamic fashion. Thus, the approaches have been extended to cover a larger class of modeling formalisms. The technical approach uses the notion of a *coherent encoding* of the transition relation, making for a simple and elegant translation of the partial order semantics in question. The presented translations have been implemented and we present extensive empirical results comparing the efficiency of the different translations to each other as well as to an explicit state model checker DiVinE on its own BEEM benchmark suite.

1. Introduction

Model checking [1,2] is a widely applied technique to validate safety critical systems. One of the most influential model checking techniques has been symbolic model checking using binary decision diagrams (BDDs) [3]. The key idea in symbolic model checking is that the state space of the system under validation is represented and manipulated symbolically, allowing one to handle enormous numbers of states succinctly. Due to the symbolic representation of states and inputs of a system, symbolic model checking methods are particularly good at handling non-determinism originating from the use of external, uncontrollable data; consequently, symbolic model checking has been very successful especially in validating hardware designs. Later, bounded model checking (BMC) [4] was introduced as an alternative to binary decision diagrams (BDDs) to implement symbolic model checking. It employs the power of efficient propositional satisfiability (SAT) [5] and SAT modulo theories (SMT) [6] solvers for model checking. BMC is mainly used for its effectiveness in bug finding. In its basic form, bounded model checking limits the search for counter-examples up to some specified bound k, which is usually initialized to 0, and slowly increased until either a counterexample is found or a user specified time or memory limit is exceeded. An interesting observation is that the experiments with BDD and BMC based model checking algorithms done in [7] for bug

☆ Work financially supported by Academy of Finland (projects 122399, 126860, 128050, and 139402), Hecse (Helsinki Graduate School in Computer Science and Engineering), the Emil Aaltonen Foundation, and the Foundation of Nokia Corporation.

* Corresponding author.
E-mail addresses: Jori.Dubrovin@tkk.fi (J. Dubrovin), Tommi.Junttila@tkk.fi (T. Junttila), Keijo.Heljanko@tkk.fi (K. Heljanko).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.07.005

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

finding show that the two approaches are quite complementary, as there seems to be little correlation between the run times of the approaches.

In addition to symbolic model checkers, there are also the so-called explicit-state model checkers such as the established SPIN [8] and the more recent DiVinE [9], which are tailored especially for asynchronous systems. Instead of symbolic, logic based methods, they explicitly represent and explore the state space of the system under validation. In order to alleviate the state space explosion problem, i.e. to handle the vast number of global states the concurrently executing processes can have, a number of techniques have been proposed in the literature. Perhaps the most important such techniques are the *partial order reduction* methods (see e.g. [1, Chapter 10]). In the context of explicit-state model checking, the common idea in all the partial order reduction methods is to generate a reduced (sub-)state space of a system that preserves the truth value of the property being model checked. This reduction is carried out by exploiting the independence between the different actions in the asynchronous system so that only a subset of actions is considered for execution in a given state. There are different variants of such partial order reductions: ample sets (applied in SPIN), stubborn sets, persistent sets, and the sleep set method. Please refer to [10] for a survey.

A natural question that arises is whether the independence of actions exploited by partial order reduction algorithms in explicit-state model checking can also be exploited to make bounded model checking of asynchronous systems more efficient. The answer is positive and there have been several proposals for making BMC more efficient for asynchronous systems, roughly falling in two different categories. In the first one, one aims at mimicking the explicit-state partial order methods by adding constraints to the BMC transition formulas so that only a subset of enabled actions can be executed in a state; the hope is that the formulas become easier to solve. In the second category of approaches, which is the main subject of this paper, the idea is rather orthogonal: the transition formulas that characterize what the system can do within one time step are modified to allow *more actions to be executed simultaneously within a step*. The main observation behind this idea is that the run time of bounded model checking usually grows very rapidly, even exponentially, when the bound (i.e., the maximum length of possible counter-examples) is increased. Therefore, techniques that allow one to decrease the bound required to find counter-examples (without significantly increasing the size or the complexity of the transition formulas) can be quite beneficial to improve the performance of BMC. One sound way of doing this is to exploit *partial order semantics*, for example the ∀-step semantics [11,12] (also called step semantics), which allows any set of enabled pairwise independent actions to be executed concurrently in a *step*, thus introducing so called *shortcut* edges to the state space of the system. This was first done in the BMC context in [12] for 1-safe Petri nets. In the context of AI-planning, a field similar to bounded model checking, Rintanen was the first to systematically employ an even more aggressive semantics, called the ∃-*step semantics* [13]. This semantics allows all of the ∀-steps, as well as for some additional steps.

In this paper, we consider speeding up BMC for asynchronous systems by employing two variants of the ∃-step semantics. The first, simply called ∃-step semantics, allows a set of actions to be executed in a single step as long as they are all enabled and could be executed sequentially in some order. The second, which we call *relaxed* ∃-*step semantics*, removes the constraint that all actions of a step need to be enabled at the start of the step. Furthermore, we will use a more constrained "process semantics" version of the latter, restricting the number of allowed steps while ensuring that the same reachable states are still covered. We extend the previous state-of-the-art by allowing the asynchronous systems to incorporate non-trivial data manipulation operations in their actions. In order to keep our approach formal and precise but not to restrict ourselves to a single system modeling language, we devise a novel, abstract framework for capturing the essential features of actions, namely enabledness and information about which variables are read or written when the action executes. This framework, which we call *coherent operational encoding* of actions, allows us to capture the independence between actions to the extent needed in developing symbolic transition formulas for interleaving, ∃-step, relaxed ∃-step, and process semantics considered in this paper. As both of the ∃-step semantics are hard to capture precisely, we follow the common practice where they are soundly under-approximated in the transition formula translation. The resulting transition formulas for both of the ∃-step semantics are surprisingly concise and elegant; although they usually allow many more actions to be executed in a single step, they are not significantly larger than the transition formulas for the interleaving semantics.

In order to eliminate some semantic redundancy in the bounded model checking translations caused by step semantics, we also present and prove the correctness of a way to add some additional constraints when the relaxed ∃-step transition formulas are applied. These constraints only allow bounded executions in a *process semantics* normal form, thus hopefully making the BMC translations easier to solve for the SAT/SMT solver; in a sense this approach combines the explicit-state partial order reduction idea of only executing some of the enabled transitions with the symbolic partial order semantics idea of executing as many transitions as possible within a single time step.

All the proposed translations have been implemented and we present extensive empirical results comparing the efficiency of the different translations to each other as well as to an explicit-state model checker DiVinE on its own BEEM benchmark suite. The results show that using either of the ∃-step semantics results usually in very significant run time reductions, the relaxed ∃-step semantics being the better one of the two.

1.1. Related work

In the area of SAT based BMC, Heljanko was the first to consider exploiting the independence between actions in encoding the transition relation [12]. That paper considers BMC for 1-bounded Petri nets using the ∀-step semantics (the classical Petri net step semantics; see e.g., [11]). The intuitive idea is that a set of actions can be executed at the same time step in ∀-step

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

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 [14] (see also [13]). In [15], a form of ∀-step semantics is applied to bounded model checking of timed systems, with relaxation techniques concerning the use of clock variables.

The work of Dimopoulos et al. was the first one to use an ∃-step semantics like approach in hand optimized planning encodings of [16], the idea of which was later formalized and automated in the SAT based planning system of Rintanen et al. [13]. The semantic definition used in [13] matches what we call the relaxed ∃-step semantics in this work, while the actual corresponding AI planning encoding to SAT is a close match to our (non-relaxed) ∃-step semantics. The treatment of data values in [13] is restricted to the Boolean domain, and the ∃-step constraints in the SAT encoding are special cased to the two possible truth values of state variables. In contrast, we allow arbitrary data domains, and our ∃-step constraints are not statically tied to separate data values.

The approach of [13] is generalized to relaxed ∃-steps in [17], while restricting to systems with more limited data handling. Unlike [17], our relaxed ∃-step semantics also allows a variable to change its value several times in a step, further compressing the bound. This corresponds to the BMC translation presented by Ogata et al. in [18], where the system formalism is 1-bounded Petri nets. In this work, we extend the ideas of [18] to the unified framework of abstract actions. Jussila et al. [19,20] show other approaches to obtaining an optimized transition formula for labeled transition systems, with an overview of different encodings in the doctoral thesis of Jussila [21]. The notion of independence between actions in ∃-step semantics has been made dynamic in previous work by the authors [22], which focused on BMC for UML state machine models.

Another thread of work in exploiting the independence of actions is the peephole partial order reduction [23] implemented on top of BMC by adding a constraint that each pair of independent actions can occur at consecutive time steps only in one predefined order. A generalization of this, called monotonic partial order reduction [24], claims to capture exactly one execution per each Mazurkiewicz trace of the system being analyzed. An earlier approach with the same property but based on the ∀-step semantics is the process semantics BMC encoding employed in [12]. Another form of partial order reduction is obtained in the CheckFence tool [25], where individual processes are coupled together by modeling the potential causal relations of global operations. A similar approach is defined for a more general high-level Petri net formalism in [26].

In addition to methods exploiting independence between the actions in model checking for asynchronous systems listed above, there are also approaches based on the partial order semantics of systems. For example, the *unfolding method* [27, 28] of McMillan is a complete model checking method based on partial order semantics. This method can offer exponential space saving compared to traditional explicit-state model checking in some cases.

1.2. Outline

The rest of the article is organized as follows. First, in Section 2 we define an abstract class of asynchronous systems as well as the different semantics (interleaving, ∃-step, and relaxed ∃-step) applied in the rest of the work. In Section 3, we briefly describe symbolic model checking, especially bounded model checking, to the extent needed in this work and also formulate a baseline symbolic transition formula that captures the interleaving semantics behavior of asynchronous systems. Next, in Section 4, we devise and analyze the novel symbolic transition formulas for the step semantics, with experimental evaluation in Section 5. After this, the process semantics and the associated experiments are considered in Section 6. Finally, Section 7 concludes the work with some remarks on possible future work topics.

2. Semantics

In this section we formally define some central concepts needed in the rest of the paper. We first give rather standard definitions of concurrent systems and their semantics via interleaving state spaces. After that we introduce two alternative semantic models for systems that allow more actions to be executed "at the same time" in one step.

2.1. Systems and interleaving state spaces

First, we assume a set of *types*, each type T being associated with a non-empty domain set $\mathrm{dom}(T)$. A *typed variable* x is a variable associated with a type $\mathrm{type}(x)$. Given a set X of typed variables, a valuation for X is a mapping τ that associates each $x \in X$ with an element $\tau(x) \in \mathrm{dom}(\mathrm{type}(x))$. As usual, we write $\tau[x \leftarrow c]$ to denote the valuation that is the same as τ except that the variable x has the value c.

In the rest of the paper, we assume systems that are composed of

1. a finite set of typed state variables $V = \{v_1, \ldots, v_n\}$,
2. a finite, non-empty set of *actions* Acts, and
3. an initial state predicate I.

Thus a *state* s of the system is a valuation for V and the set of all states is denoted by S. The semantics of the actions and the initial state predicate I describe the behaviors of the system, formally captured by its *interleaving state space*

$$\mathcal{S} = \langle S, s_{\mathrm{init}}, \Delta_{\mathrm{int}} \rangle,$$

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

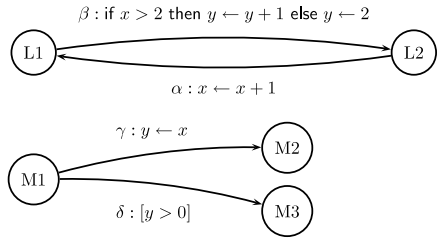


Fig. 1. An example system.

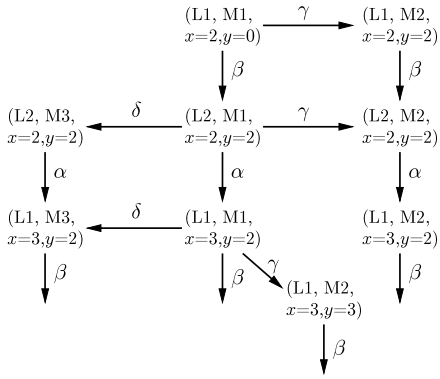


Fig. 2. A part of the interleaving state space of the system in Fig. 1.

where $s_{\text{init}} \in S$ is the *initial state* fulfilling the initial state predicate, and the *interleaving transition relation* $\Delta_{\text{int}} \subseteq S \times \text{Acts} \times S$ describes how states may evolve to others: $\langle s, \text{act}, s' \rangle \in \Delta_{\text{int}}$ iff executing the enabled action $\text{act} \in \text{Acts}$ in the state s leads to the state s'. An action act is *enabled* in a state s if there exists a state s' such that $\langle s, \text{act}, s' \rangle \in \Delta_{\text{int}}$. A state s' is *reachable* from a state s if there exist actions $\text{act}_1, \ldots, \text{act}_k \in \text{Acts}$ and states $s_0, s_1, \ldots, s_k$ for some $k \in \mathbb{N}$ such that (i) $s = s_0$, (ii) $\forall 1 \leq i \leq k : \langle s_{i-1}, \text{act}_i, s_i \rangle \in \Delta_{\text{int}}$, and (iii) $s_k = s'$.

Example 1. Fig. 1 represents a system with two processes, L and M. The variables are $V = \{pc_L, pc_M, x, y\}$ with domains {L1, L2}, {M1, M2, M3}, $\mathbb{Z}$, and $\mathbb{Z}$, respectively. The actions are $\text{Acts} = \{\alpha, \beta, \gamma, \delta\}$, and the initial state predicate $(pc_L = \text{L1}) \wedge (pc_M = \text{M1}) \wedge (x = 2) \wedge (y = 0)$. Fig. 2 shows a part of the interleaving state space of the system.

2.2. ∃-step semantics

As already mentioned in the Introduction, 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 distinct 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 $\mathcal{S} = \langle S, s_{\text{init}}, \Delta_{\text{int}} \rangle$ is the tuple

$$\mathcal{S}_\exists = \langle S, s_{\text{init}}, \Delta_\exists \rangle,$$

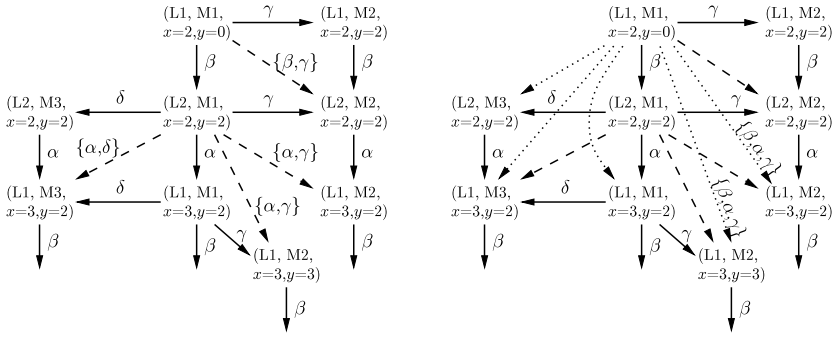
where the *transition relation* $\Delta_\exists \subseteq S \times 2^{\text{Acts}} \times S$ contains a *step* $\langle s, ex, s' \rangle$ if and only if

1. the set of actions $ex = \{\text{act}_1, \ldots, \text{act}_k\} \subseteq \text{Acts}$ is non-empty,
2. all the actions in ex are enabled in s, and
3. there is a total ordering $\text{act}_1 \prec \cdots \prec \text{act}_k$ of ex and a set of states $s_0, s_1, \ldots, s_k \in S$ such that (i) $s = s_0$, (ii) $\forall 1 \leq i \leq k : \langle s_{i-1}, \text{act}_i, s_i \rangle \in \Delta_{\text{int}}$, and (iii) $s_k = s'$.

If condition 3 above is fulfilled, we say that the set ex is linearizable w.r.t. $\prec$.

Example 2. Consider again the system in Fig. 1. A part of its ∃-step state space is given in Fig. 3(a). The solid arrows depict the unit steps also present in the interleaving state space (recall Fig. 2), while the dashed arrows illustrate the non-unit steps that are added in the ∃-step state space. Observe that in the ∃-step state space all the states in the figure can be reached with at most two steps from the initial state ($pc_L = \text{L1}, pc_M = \text{M1}, x = 2, y = 0$), whereas three transitions are needed in

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005



(a) ∃-steps. (b) Relaxed ∃-steps.

Fig. 3. Parts of the ∃-step and relaxed ∃-step state spaces.

the interleaving state space. Also observe that depending on the execution order of the actions in a step, a different state can be obtained. As an example, consider the state ($pc_L = \mathrm{L2}, pc_M = \mathrm{M1}, x = 2, y = 2$) in the middle of Fig. 3(a) and the step $\{\alpha, \gamma\}$. If α is executed before γ, then the state ($pc_L = \mathrm{L1}, pc_M = \mathrm{M2}, x = 3, y = 3$) is obtained while executing γ before α results in the state ($pc_L = \mathrm{L1}, pc_M = \mathrm{M2}, x = 3, y = 2$).

By definition, it holds that the ∃-step state space includes the interleaving state space in the sense that $\langle s, \mathsf{act}, s'\rangle \in \Delta_{\mathrm{int}}$ implies that the *unit step* $\langle s, \{\mathsf{act}\}, s'\rangle$ is in $\Delta_{\exists}$. Conversely, if $\langle s, ex, s'\rangle$ belongs to $\Delta_{\exists}$, then there is a finite sequence of states leading from s to s' in the interleaving state space. Therefore, the sets of states reachable from a given state are equal for the interleaving and the ∃-step state spaces:

Proposition 1. *A state s' is reachable from a state s in the interleaving state space if and only if it is in the ∃-step state space.*

Note that the definition of ∃-step semantics is a purely semantic one; in the symbolic encodings 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 symbolic transition relation formula (discussed later) too much w.r.t. the one for the interleaving semantics. For instance, given a system, we will fix a total order $\prec$ on Acts and only consider steps that are linearizable w.r.t. this ordering. 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 [13], which also similarly soundly under-approximates the ∃-step semantics in its implementation.

2.3. Relaxed ∃-step semantics

To formally capture the extended version of the Ogata et al. approach [18] proposed in this paper, we will, in addition to the classic step semantics presented above, also consider a relaxed version where the actions in a step do not have to be all enabled in the first state. Thus even more "shortcut edges" are added to the interleaving state space, the goal being to allow the state space to be covered with even less steps. This is formalized in the definition below.

Definition 2. The *relaxed ∃-step state space* corresponding to the interleaving state space $\mathcal{S} = \langle S, s_{\mathrm{init}}, \Delta_{\mathrm{int}}\rangle$ is the tuple

$$\mathcal{S}_{\exists\mathrm{relax}} = \langle S, s_{\mathrm{init}}, \Delta_{\exists\mathrm{relax}}\rangle,$$

where the *transition relation* $\Delta_{\exists\mathrm{relax}} \subseteq S \times 2^{\mathsf{Acts}} \times S$ contains a *step* $\langle s, ex, s'\rangle$ if and only if

1. the set of actions $ex = \{\mathsf{act}_1, \ldots, \mathsf{act}_k\} \subseteq \mathsf{Acts}$ is non-empty, and
2. there is a total ordering $\mathsf{act}_1 \prec \cdots \prec \mathsf{act}_k$ of ex and a set of states $s_0, s_1, \ldots, s_k \in S$ such that (i) $s = s_0$, (ii) $\forall 1 \leq i \leq k : \langle s_{i-1}, \mathsf{act}_i, s_i\rangle \in \Delta_{\mathrm{int}}$, and (iii) $s_k = s'$.

The only difference from Definition 1 is that the actions in ex are not required to be enabled in s. Again, the definition of relaxed ∃-step semantics is purely semantic one. By definition it holds that the relaxed ∃-step state space includes both the interleaving and ∃-step state spaces. Therefore, it is not difficult to see that a reachability property similar to Proposition 1 holds.

Proposition 2. *A state s' is reachable from a state s in the interleaving state space if and only if it is in the relaxed ∃-step state space.*

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

Example 3. A part of the relaxed ∃-step state space for our running example system in Fig. 1 is given in Fig. 3(b). In addition to the unit steps (solid arrows) and ∃-step state space steps (dashed arrows), the relaxed ∃-step state space also includes the steps depicted with the dotted arrows. Contrasted to the interleaving and ∃-step state spaces, in the relaxed ∃-step state space all the states in the figure can be reached within one step from the initial state ($pc_L = \mathrm{L1}, pc_M = \mathrm{M1}, x = 2, y = 0$). Reaching any state not shown in the figure requires executing the action β at least twice, and thus cannot be done in a single relaxed ∃-step from the initial state.

3. Symbolic model checking

In symbolic model checking (see e.g. [1]), the state space $\mathcal{S}$ is not constructed explicitly but (quantifier-free) first-order logic formulas are used to represent the transition relation and to manipulate the state space symbolically. In the following, we use standard concepts and notations of propositional and first-order logics, and assume that the formulas are interpreted modulo some background theory $\mathcal{T}$ such as linear arithmetics, theory of bit vectors, theory of lists, or their combinations (see e.g. [6] and the references therein). We use the term *variable* to denote both constants (nullary function symbols) and propositional symbols (nullary predicate symbols) usually used in the logic literature.

The transition relation (under interleaving, ∃-step, or relaxed ∃-step semantics) is represented by a symbolic transition relation formula $\tilde{\Delta}(\tilde{V}, \tilde{C}, \tilde{V}')$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$, or *transition formula* for short, where

1. $\tilde{V} = \{\tilde{v}_1, \ldots, \tilde{v}_n\}$ is a set of *state encoding variables* corresponding to the state variables $\{v_1, \ldots, v_n\}$,
2. $\tilde{V}' = \{\tilde{v}'_1, \ldots, \tilde{v}'_n\}$ are their *next state* versions, and
3. $\tilde{C}$ is a finite set of *choice encoding variables* that are used to fix the set of actions being executed and the results of nondeterministic choices.

As in the SMT-LIB standard [29] for expressing first order logic formulas modulo background theories, we assume that the logic is typed (i.e., sorted). An *interpretation* $\mathcal{I}$ for a formula ϕ associates each variable in ϕ with an element in the domain of its type (e.g., an integer) and each non-nullary function and predicate symbol in ϕ with a construct over the domains of the argument types. An interpretation *satisfies* a formula ϕ if it is a model of the background theory $\mathcal{T}$ and its homomorphic extension evaluates the formula to true; in such a case, the interpretation is called a *model* of ϕ and we may write $\mathcal{I} \models \phi$. For instance, if $\mathcal{T}$ is the theory of integer linear arithmetics, then the interpretation $\mathcal{I} = \{x \mapsto 0, y \mapsto 2\}$ (together with the standard interpretations for the symbols $+$ and $\leq$ given by $\mathcal{T}$) is extended so that $\mathcal{I}(x+1) = 1, \mathcal{I}(x+1 \leq y) = \mathbf{T}$, and thus $\mathcal{I} \models (x+1 \leq y)$ but $\mathcal{I} \not\models (x+3 \leq y)$. A formula ϕ is *satisfiable* if it has a model and *unsatisfiable* otherwise.

In order to map states of the system to models of formulas and vice versa, we define the following notation. If $\mathcal{I}$ is an interpretation that gives values to the state encoding variables $\tilde{V} = \{\tilde{v}_1, \ldots, \tilde{v}_n\}$ (in other words, $\mathcal{I}$ is an interpretation over $\tilde{V}$), we define

$$\mathrm{state}_{\mathcal{I}} : v_i \mapsto \mathcal{I}(\tilde{v}_i) \tag{1}$$

to denote the state encoded in $\mathcal{I}$. Similarly, if $\mathcal{I}$ is an interpretation over the next state encoding variables $\tilde{V}' = \{\tilde{v}'_1, \ldots, \tilde{v}'_n\}$, we use

$$\text{next-state}_{\mathcal{I}} : v_i \mapsto \mathcal{I}(\tilde{v}'_i) \tag{2}$$

to denote the state encoded in the values of $\tilde{V}'$.

We denote the actions by $\mathsf{Acts} = \{\mathsf{act}^1, \ldots, \mathsf{act}^K\}$ and define the propositional choice encoding variables $\tilde{f}^1, \ldots, \tilde{f}^K \in \tilde{C}$, where $\tilde{f}^i$ means selecting act^i for execution. When $\mathcal{I}$ is an interpretation over the choice encoding variables, we decode from $\mathcal{I}$ the set $\mathrm{ex}_{\mathcal{I}} \subseteq \mathsf{Acts}$ of actions being executed by

$$\mathrm{ex}_{\mathcal{I}} := \left\{\mathsf{act}^i \;\middle|\; 1 \leq i \leq K \text{ and } \mathcal{I} \text{ satisfies } \tilde{f}^i\right\}. \tag{3}$$

In order to be correct, a transition formula must only allow transitions that are part of the transition relation (soundness), and conversely, all transitions must be allowed (completeness).

Definition 3. Let $\Delta \subseteq S \times 2^{\mathsf{Acts}} \times S$ be a transition relation and let $\tilde{\Delta}$ be a transition formula. We say that $\tilde{\Delta}$ is *sound with respect to* Δ if, for every interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ that satisfies $\tilde{\Delta}$,

$$\langle \mathrm{state}_{\mathcal{I}}, \mathrm{ex}_{\mathcal{I}}, \text{next-state}_{\mathcal{I}} \rangle \in \Delta.$$

The formula $\tilde{\Delta}$ is *complete with respect to* Δ if, for every transition $\langle s, ex, s' \rangle \in \Delta$, there is an interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ such that $\mathrm{state}_{\mathcal{I}} = s$, $\mathrm{ex}_{\mathcal{I}} = ex$, $\text{next-state}_{\mathcal{I}} = s'$, and $\mathcal{I}$ satisfies $\tilde{\Delta}$.

Above, Δ can be one of the ∃-step transition relations or the interleaving transition relation $\Delta_{\mathrm{int}} \subseteq S \times \mathsf{Acts} \times S$. In the latter case, we regard Δ_{int} as a subset of $S \times 2^{\mathsf{Acts}} \times S$ by abusing the notation and identifying the tuple $\langle s, \mathsf{act}, s' \rangle$ with $\langle s, \{\mathsf{act}\}, s' \rangle$.

For the actual model checking, e.g. to check whether the given system has a reachable state where some proposition over the state variables does not hold, one can use several different techniques. If $\tilde{\Delta}$ is propositional or can be translated

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

into an equivalent propositional formula, one can use the well-known BDD-based model checking methods (see e.g. [1]) as well as the bounded model checking (BMC) method with SAT-solvers [4]. If using Booleans is impossible or inefficient, one can still apply BMC by using SMT-solvers supporting background theories such as linear arithmetic over reals, lists, arrays and so on. It should be noted that in addition to BMC, the model checking technique used in this paper, there are also other symbolic model checking techniques for systems that have non-Boolean variables and for parameterized systems, including e.g. [30–36]. Whether the step and process semantics used in this paper could be incorporated in these techniques is an open question.

The basic idea in BMC is that one tries to find counter-examples of bounded length k by unrolling the transition formula k times. The unrolling is augmented with the initial state predicate I and a predicate capturing the negation of the property under verification so that the satisfying truth assignments for the resulting formula correspond exactly to the executions of k steps violating the property. As the simplest example, consider that we want to verify that a property P holds in all reachable states. To apply BMC, we make "timed" copies of the state encoding variables so that the variables $\tilde{V}^i = \{\tilde{v}_1^i, \ldots, \tilde{v}_n^i\}$ represent the ith state in the bounded execution. Similar timed copies are also made of the choice encoding variables $\tilde{C}$. Now consider the formula

$$\tilde{I}(\tilde{V}^0) \wedge \bigwedge_{t=0}^{k-1} \tilde{\Delta}(\tilde{V}^t, \tilde{C}^t, \tilde{V}^{t+1}) \wedge \neg \tilde{P}(\tilde{V}^k), \tag{4}$$

where $\tilde{I}$ is an encoding of the initial state predicate, $\tilde{\Delta}(\tilde{V}, \tilde{C}, \tilde{V}')$ is a sound and complete transition formula, and $\tilde{P}$ is an encoding of the invariance property P. It is satisfiable if and only if the system has an execution of k steps ending in a state violating the property P. And if the formula is satisfiable, then a system execution leading from an initial state to the property violating state (i.e., a counter-example) can be decoded from the satisfying interpretation. In BMC the bound k is gradually increased until a counter-example is found, a completeness condition is satisfied, or one runs out of resources (time or memory). For more information about encoding more complex temporal logic properties, using incremental solving techniques, and defining completeness conditions, please refer to e.g. [7] and the references therein.

3.1. Encoding actions

To construct the transition formula for a system, we need to encode the behavior of the actions in first order logic modulo the chosen theory. Such an encoding is usually extracted (automatically) from the description of the system. The exact details of this process are modeling language dependent and thus not formalized in this work; however, see Section 5.1 for a semi-formal description of an encoding for the BEEM modeling language. Here we take a higher level viewpoint and assume that the actions are encoded in a set of expressions that capture two essential features of actions: when they are enabled and how they modify variables. These expressions will be used as building blocks of the transition formula.

Each action $\mathrm{act}^i \in \{\mathrm{act}^1, \ldots, \mathrm{act}^K\}$ is encoded in the following expressions, where v ranges over the state variables:

- formula $enabled^i$ ("act^i is enabled"),
- formulas $write_v^i$ ("act^i writes v"), and
- expressions $newvalue_v^i$ ("the value assigned to v").

The free variables of these expressions may only include state encoding variables $\tilde{V}$ and possibly action-specific choice encoding variables $\tilde{A}^i \subset \tilde{C}$ for modeling nondeterminism in act^i. We assume that $\tilde{A}^i$ is disjoint from $\{\tilde{f}^1, \ldots, \tilde{f}^K\}$ and from $\tilde{A}^j$ when $i \neq j$. When unrolling the transition formula, a new copy of $\tilde{A}^i$ is allocated for each time step.

Example 4. The action $\mathrm{act}^i = \delta$ in Fig. 1, changing the control location from M1 to M3 if the guard condition $y > 0$ holds, can be encoded as below. As the action is deterministic, no choice encoding variables are needed.

$$enabled^i := (\tilde{pc}_M = \mathrm{M1}) \wedge (\tilde{y} > 0)$$
$$write^i_{pc_L} := write^i_x := write^i_y := \mathbf{F}$$
$$write^i_{pc_M} := \mathbf{T}$$
$$newvalue^i_{pc_M} := \mathrm{M3}$$

Example 5. Consider an action act^j that moves from control location L7 to L8 and nondeterministically assigns a value between 1 and 9 to x. We can encode this using a choice encoding variable $\tilde{a} \in \tilde{A}^j$ by setting

$$enabled^j := (\tilde{pc}_L = \mathrm{L7}) \wedge (\tilde{a} \geq 1) \wedge (\tilde{a} \leq 9)$$
$$write^j_{pc_L} := write^j_x := \mathbf{T}$$
$$newvalue^j_{pc_L} := \mathrm{L8}$$
$$newvalue^j_x := \tilde{a}$$

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

The correctness of the encoding of actions is defined in terms of the interleaving transition relation as follows. Informally, evaluating the expressions under all possible interpretations must result in exactly those transitions that are in the transition relation.

Definition 4. Given a system with actions $\{\mathsf{act}^1, \ldots, \mathsf{act}^K\}$ and interleaving transition relation Δ_{int}, the expressions $enabled^i$, $write^i_v$, and $newvalue^i_v$ form an *operational encoding* iff

$$\Delta_{\text{int}} = \left\{ \langle \text{state}_{\mathcal{I}}, \mathsf{act}^i, \text{succ}^i_{\mathcal{I}} \rangle \mid 1 \le i \le K,\ \mathcal{I} \in E^i \right\},$$

where E^i is the set of interpretations over $\tilde{V} \cup \tilde{A}^i$ that satisfy $enabled^i$, and $\text{succ}^i_{\mathcal{I}}$ is the state defined by

$$\text{succ}^i_{\mathcal{I}}(v) = \begin{cases} \mathcal{I}(newvalue^i_v) & \text{if } \mathcal{I} \text{ satisfies } write^i_v, \\ \mathcal{I}(\tilde{v}) & \text{otherwise.} \end{cases}$$

For instance, the encodings in Examples 4 and 5 are operational with respect to the interleaving transition relations induced by the usual semantics of the applied programming language and control flow constructs (cf Fig. 2).

3.2. Interleaving transition formula

As a baseline for comparing the ∃-step transition formulas, we first define a transition formula corresponding to the interleaving state space. Assume that an encoding of the actions $\{\mathsf{act}^1, \ldots, \mathsf{act}^K\}$ is given.

Definition 5. The *interleaving transition formula* $\tilde{\Delta}_{\text{int}}$ over the encoding variables $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ is the conjunction of the following formulas.

$$\bigvee_{i=1}^{K} \tilde{f}^i \tag{5}$$

$$\left(\bigvee_{i=1}^{j-1} \tilde{f}^i \right) \Rightarrow \neg \tilde{f}^j \qquad \text{for } 2 \le j \le K \tag{6}$$

$$\tilde{f}^i \Rightarrow enabled^i \qquad \text{for } 1 \le i \le K \tag{7}$$

$$(\tilde{f}^i \wedge write^i_v) \Rightarrow (\tilde{v}' = newvalue^i_v) \qquad \text{for } v \in V,\ 1 \le i \le K \tag{8}$$

$$\left(\neg \bigvee_{i=1}^{K} \tilde{f}^i \wedge write^i_v \right) \Rightarrow (\tilde{v}' = \tilde{v}) \qquad \text{for } v \in V \tag{9}$$

These constraints enforce the following requirements: at least one action is executed (5), at most one action is executed (6), only enabled actions are executed (7), the variables being written get their next values from the action (8), and the variables not being written keep their old values (9).

The definition of operational encoding almost directly entails the correctness of $\tilde{\Delta}_{\text{int}}$ with respect to the interleaving transition relation. The remaining proof steps are stated below.

Proposition 3. *The interleaving transition formula $\tilde{\Delta}_{\text{int}}$ constructed from an operational encoding is sound and complete with respect to the interleaving transition relation Δ_{int}.*

Proof. Soundness. Take any interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ that satisfies (5)–(9). By (5) and (6), there is exactly one j such that $\mathcal{I}$ satisfies $\tilde{f}^j$. Thus, $\text{ex}_{\mathcal{I}} = \{\mathsf{act}^j\}$. By (7), $\mathcal{I}$ also satisfies $enabled^j$, and by Definition 4, the tuple $\langle \text{state}_{\mathcal{I}}, \mathsf{act}^j, \text{succ}^j_{\mathcal{I}} \rangle$ is in Δ_{int}. We need to show that $\text{next-state}_{\mathcal{I}} = \text{succ}^j_{\mathcal{I}}$, that is, for all state variables v,

$$\mathcal{I}(\tilde{v}') = \begin{cases} \mathcal{I}(newvalue^j_v) & \text{if } \mathcal{I} \text{ satisfies } write^j_v, \\ \mathcal{I}(\tilde{v}) & \text{otherwise.} \end{cases}$$

The claim follows from (8) and (9), since $\mathcal{I}(\tilde{f}^i \wedge write^i_v)$ is equivalent to $(i = j) \wedge \mathcal{I}(write^i_v)$.

Completeness. Take a transition $\langle s, \text{act}, s' \rangle \in \Delta_{\text{int}}$, where $\text{act} = \mathsf{act}^j$ for some j. By Definition 4, there is an interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{A}^j$ that satisfies $enabled^j$ such that $\text{state}_{\mathcal{I}} = s$ and $\text{succ}^j_{\mathcal{I}} = s'$. We extend the interpretation to $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ by setting $\mathcal{I}(\tilde{f}^j) = \mathbf{T}$, $\mathcal{I}(\tilde{f}^i) = \mathbf{F}$ when $i \ne j$, and $\text{next-state}_{\mathcal{I}} = s'$. For $\tilde{a} \in \tilde{A}^i$ where $i \ne j$, we can choose $\mathcal{I}(\tilde{a})$ arbitrarily. Then, $\text{ex}_{\mathcal{I}} = \{\mathsf{act}^j\}$, and by a straightforward check, $\mathcal{I}$ satisfies all formulas (5)–(9). □

3.2.1. Size of the interleaving transition formula

Because the transition formula $\tilde{\Delta}$ is instantiated k times in the bounded model checking formula (4), its size is very relevant in bounded model checking. In estimating the formula size, we assume that multiple occurrences of the same sub-

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

expression share their representation in the overall description of the formula. For example, for a fixed $v \in V$ and $1 \leq i \leq K$, the sub-formula $write_v^i$ occurs in both (8) and (9), but we only count it once toward the size of the transition formula. This is a realistic standpoint, because we can always replace all occurrences of $write_v^i$ with a fresh propositional variable $\tilde{a}$ and add the extra constraint $\tilde{a} \Leftrightarrow write_v^i$, ending up with an equisatisfiable formula where $write_v^i$ occurs just once.

Moreover, we use simple constant folding to eliminate true and false constants under logical operators. For example, formulas $(\mathbf{F} \Rightarrow \phi)$, $(\phi \Rightarrow \mathbf{T})$, and $(\phi \wedge \mathbf{T})$ are reduced to $\mathbf{T}$, $\mathbf{T}$, and ϕ, respectively. In particular, if $write_v^i$ is the constant false formula $\mathbf{F}$, then the constraint (8) reduces to a tautology, and we can also omit the index i in the disjunction $\bigvee_{i=1}^{K} \tilde{f}^i \wedge write_v^i$ in (9). This is a common case in practice, since most actions access only a fraction of the state variables.

The left-hand side $\bigvee_{i=1}^{j-1} \tilde{f}^i$ of (6) can be expressed as $(\bigvee_{i=1}^{j-2} \tilde{f}^i) \vee \tilde{f}^{j-1}$. This enables sharing the sub-formulas for different values of j, and allows representing the conjunction of constraints (6) for $2 \leq j \leq K$ in size $\mathcal{O}(K)$.

Let L be the total size of the encoding expressions $enabled^i$ for $1 \leq i \leq K$ and $write_v^i$ and $newvalue_v^i$ for $1 \leq i \leq K$ and $v \in V$ such that $write_v^i$ is not $\mathbf{F}$. Assuming that constant folding, sub-expression sharing, and the above representation for the disjunction $\bigvee_{i=1}^{j-1} \tilde{f}^i$ are used, we see that the size of the interleaving transition formula is linear in L.

4. Transition formulas with ∃-step semantics

Our realization of step semantics in symbolic model checking is based on the following desirable properties of the transition formula.

1. It is sound with respect to the relaxed ∃-step transition relation $\Delta_{\exists\text{relax}}$.
2. It is complete with respect to the interleaving transition relation Δ_{int}.
3. It allows a large number of the steps in $\Delta_{\exists\text{relax}}$.
4. It is not much larger than the interleaving transition formula $\tilde{\Delta}_{\text{int}}$.

Properties 1 and 2 guarantee correctness in model checking invariant properties: nothing but relaxed ∃-steps are allowed, and all unit steps are allowed. Properties 3 and 4 address efficiency, trying to avoid unwieldy formulas resulting from a complicated transition relation or extensive transition formula unrolling in the context of BMC. They are conflicting goals, however. Even the interleaving symbolic transition relation fulfills properties 1, 2, and 4. On the other hand, we could write a formula that allows all relaxed ∃-steps and is thus complete with respect to $\Delta_{\exists\text{relax}}$, but it would be likely to become unmanageable in size. This is because we would have to take into account all $K!$ potential execution orders of the actions $\mathsf{act}^1, \ldots, \mathsf{act}^K$ within a step.

In Sections 4.2 and 4.3, we will define two ∃-step transition formulas, one based on parallel and the other on serial execution of the actions within a step. Regarding property 3 above, the formulas allow a significant share of non-unit steps to be executed. In particular, this includes all steps consisting of only pairwise independent actions. The ∃-step transition formulas are variations of the interleaving transition formula, with only moderate increase in size.

Parallel ∃-step transition formula. In a parallel ∃-step, we allow several actions to make changes to the state variables at the same time, with extra constraints that guarantee the linearizability of the actions. We take the interleaving transition formula and remove the constraint that just one action is executed. The resulting semantics corresponds to the parallel execution of a set of actions in each step. Without extra constraints, such steps are not always legal ∃-steps. For example, parallel execution of the assignments $x \leftarrow y$ and $y \leftarrow x$ swaps the values of x and y, which does not correspond to sequential execution in any order. To obtain the parallel ∃-step transition formula, we ensure linearizability by adding restrictions based on the dependencies between read and write operations of actions. First, we predefine a total order $\mathsf{act}^1 \prec \cdots \prec \mathsf{act}^K$ of all actions. Then, we add constraints that only allow those parallel steps that contain no two actions $\mathsf{act}^i \prec \mathsf{act}^j$ such that act^i writes to a state variable that act^j reads. The parallel execution of such a step corresponds to sequential execution in the order given by $\prec$, because no action is affected by any of the preceding actions. The confinement to a fixed action ordering keeps the size of the transition formula manageable. The resulting formula is sound with respect to the ∃-step transition relation $\Delta_\exists$ in Definition 1.

Serial ∃-step transition formula. Again, we fix a total order $\mathsf{act}^1 \prec \cdots \prec \mathsf{act}^K$. The serial ∃-step transition formula allows a subset of the actions to be executed in a step. When executing act^i, conceptually, any state changes made by the previous actions $\mathsf{act}^1, \ldots, \mathsf{act}^{i-1}$ in the step are already in effect. With serial ∃-steps, act^i may be affected by the preceding actions. The resulting formula is sound with respect to the relaxed ∃-step transition relation $\Delta_{\exists\text{relax}}$. In fact, the transition formula allows precisely those steps of Definition 2 where the ordering of the actions respects the predefined total order $\prec$. Ogata et al. first proposed this approach for bounded model checking of safe Petri nets [18].

With both parallel and serial ∃-steps, a different choice of the order $\prec$ results in a different transition formula. The choice also affects which steps of the ∃-step transition relation are allowed by the formula. This aspect will be discussed in Section 5.3.

4.1. Encoding read operations of actions

The expressions $enabled^i$, $write_v^i$, and $newvalue_v^i$, introduced in Section 3.1, determine the possible changes that action act^i can make to state variables in each state. Their correctness with respect to the state space is formulated in the definition

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

of an operational encoding. For the parallel ∃-step encoding, we also need to formalize the dependence of act^i on the value of each state variable. Therefore we include in the encoding the formulas $read^i_v$ over $\tilde{V} \cup \tilde{A}^i$, where $1 \le i \le K$ and $v \in V$. The meaning is that $read^i_v$ evaluates to true if action act^i reads variable v.

Example 6. Action $\mathsf{act}^i = \beta$ in Fig. 1 moves from location L1 to L2 and executes the statement

$$\mathsf{if}\ x > 2\ \mathsf{then}\ y \leftarrow y + 1\ \mathsf{else}\ y \leftarrow 2.$$

Whether β is enabled depends on the value of the program counter variable pc_L, and the values of x and y are used to compute the outcome. We therefore set $read^i_{pc_L}$, $read^i_x$, and $read^i_y$ to the constant true expression. As the variable pc_M is irrelevant, $read^i_{pc_M}$ is set to false. We get the following encoding for β.

$$enabled^i := (\tilde{pc}_L = \mathrm{L1})$$
$$read^i_{pc_M} := \mathbf{F}$$
$$read^i_{pc_L} := read^i_x := read^i_y := \mathbf{T}$$
$$write^i_{pc_M} := write^i_x := \mathbf{F}$$
$$write^i_{pc_L} := write^i_y := \mathbf{T}$$
$$newvalue^i_{pc_L} := \mathrm{L2}$$
$$newvalue^i_y := \mathbf{if}\ \tilde{x} > 2\ \mathbf{then}\ \tilde{y} + 1\ \mathbf{else}\ 2$$

Example 7. Our notion of reading can also be sensitive to the current state. In action β above, the value of variable y is only used in the "then" branch, therefore β only depends on y when $x > 2$. It is allowable to over-approximate the dependency relation, so we can either decide that β reads y regardless of the values of other variables, or that β reads y only when $x > 2$. The former, static case corresponds to Example 6, and the latter, dynamic case corresponds to modifying the encoding by setting $read^i_y := (\tilde{x} > 2)$.

The $read^i_v$-formulas are over-approximated in practice since computing exact dependences is expensive and would make the transition formula more complex. On the other hand, over-approximation introduces spurious dependences and may reduce the set of ∃-steps allowed by the transition formula.

The correctness requirement on $read^i_v$ is defined in terms of the rest of the encoding: intuitively, $read^i_v$ must evaluate to true whenever the expressions $enabled^i$, $write^i_v$, or $newvalue^i_v$ depend on $\tilde{v}$. If the requirement holds, we say that the encoding is *coherent*. For the formal definition, we need additional notation. Given an interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{A}^i$, we denote the sets of state variables being read and written by act^i by

$$R^i(\mathcal{I}) := \left\{ v \in V \mid \mathcal{I} \text{ satisfies } read^i_v \right\}, \text{ and}$$
$$W^i(\mathcal{I}) := \left\{ v \in V \mid \mathcal{I} \text{ satisfies } write^i_v \right\}.$$

Generally, these sets are not statically defined but may depend on the values of state and choice encoding variables. For $1 \le i \le K$ and two interpretations $\mathcal{I}$ and $\mathcal{I}'$ over $\tilde{V} \cup \tilde{A}^i$, we say that $\mathcal{I}'$ *i-conforms to* $\mathcal{I}$ iff $\mathcal{I}'(\tilde{a}) = \mathcal{I}(\tilde{a})$ for all $\tilde{a} \in \tilde{A}^i$ and $\mathcal{I}'(\tilde{v}) = \mathcal{I}(\tilde{v})$ for all $v \in R^i(\mathcal{I})$. That is, the i-conforming interpretations are those that only differ from $\mathcal{I}$ on those variables $\tilde{v} \in \tilde{V}$ for which $\mathcal{I}(read^i_v)$ is false.

Definition 6. The expressions $enabled^i$, $read^i_v$, $write^i_v$, and $newvalue^i_v$ form a *coherent encoding* iff for all $1 \le i \le K$, all interpretations $\mathcal{I}$ that satisfy $enabled^i$, and all interpretations $\mathcal{I}'$ that i-conform to $\mathcal{I}$:

1. $\mathcal{I}'$ satisfies $enabled^i$ (the action is also enabled under $\mathcal{I}'$),
2. $R^i(\mathcal{I}') = R^i(\mathcal{I})$ (the same variables are read),
3. $W^i(\mathcal{I}') = W^i(\mathcal{I})$ (the same variables are written), and
4. for all $v \in W^i(\mathcal{I})$, $\mathcal{I}'(newvalue^i_v) = \mathcal{I}(newvalue^i_v)$ (the written values are the same).

The definition places a requirement on the expressions $read^i_v$: any dependence of act^i on a variable v must be reflected in $read^i_v$ evaluating to true. We can always make an encoding coherent by maximal over-approximation, setting $read^i_v$ to the constant expression $\mathbf{T}$ for all i and v. A more practical policy is based on syntax: if a state encoding variable $\tilde{v}$ does not occur in the formula $enabled^i$ nor any of the expressions $write^i_u$ or $newvalue^i_u$ for any $u \in V$, then the action does not depend on v and we can set $read^i_v$ to constant $\mathbf{F}$. Otherwise, set $read^i_v$ to $\mathbf{T}$. This approach was taken in Example 6 above. All encodings of Examples 6 and 7 are coherent for the case of action β.

4.2. Parallel ∃-step transition formula

Given a coherent operational encoding of a system and an ordering $\mathsf{act}^1 \prec \cdots \prec \mathsf{act}^K$ of its actions, let us define a transition formula that captures the parallel ∃-steps discussed above. The idea is to modify the interleaving transition

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

formula to allow also non-unit steps, while disallowing steps where some variable is first written and then read by another action later in the order $\prec$.

Definition 7. The *parallel ∃-step transition formula* $\tilde{\Delta}_{\text{par}}$ is the conjunction of the following formulas.

$$\bigvee_{i=1}^{K} \tilde{f}^i \tag{10}$$

$$\left(\bigvee_{i=1}^{j-1} \tilde{f}^i \wedge write_v^i\right) \Rightarrow \neg(\tilde{f}^j \wedge read_v^j) \qquad \text{for } v \in V,\ 2 \le j \le K \tag{11}$$

$$\tilde{f}^i \Rightarrow enabled^i \qquad \text{for } 1 \le i \le K \tag{12}$$

$$(\tilde{f}^i \wedge write_v^i) \Rightarrow (\tilde{v}' = newvalue_v^i) \qquad \text{for } v \in V,\ 1 \le i \le K \tag{13}$$

$$\left(\neg \bigvee_{i=1}^{K} \tilde{f}^i \wedge write_v^i\right) \Rightarrow (\tilde{v}' = \tilde{v}) \qquad \text{for } v \in V \tag{14}$$

Technically, the constraints are the same as in the interleaving transition formula except that the at-most-one constraint (6) is replaced with the requirement (11) that no variable is written by action act^i and read by action act^j if $i < j$. Constraints (12)–(14) describe steps where zero or more actions are executed in parallel: every executed action is enabled (12), the values written by different actions are not contradictory (13), and the variables not written to keep their old values (14). Constraint (10) forbids the empty step, and (11) ensures that the parallel step is linearizable in at least one order, namely the one that respects $\prec$.

In the following examples we illustrate the properties of parallel ∃-step transition formulas under coherent encodings. In all examples, we use our running example system in Fig. 1 and its ∃-step state space in Fig. 3(a).

Example 8. Let us first consider the case of mutually independent actions by using the actions $\alpha : x \leftarrow x+1$ and $\delta : [y > 0]$ occurring in different processes. Executing them in parallel in the state $s = (pc_L = \text{L2}, pc_M = \text{M1}, x = 2, y = 2)$ results in the state $s' = (pc_L = \text{L1}, pc_M = \text{M3}, x = 3, y = 2)$. If they were executed sequentially in s, we would obtain the same state s' irrespective whether α is executed before δ or vice versa. In the intuitive coherent encoding for these actions we have

$$read_x^\alpha := write_x^\alpha := write_{pc_L}^\alpha := read_{pc_L}^\alpha := \mathbf{T}$$
$$read_y^\alpha := write_y^\alpha := write_{pc_M}^\alpha := read_{pc_M}^\alpha := \mathbf{F}$$
$$read_y^\delta := write_{pc_M}^\delta := read_{pc_M}^\delta := \mathbf{T}$$
$$write_y^\delta := read_x^\delta := write_x^\delta := write_{pc_L}^\delta := read_{pc_L}^\delta := \mathbf{F}$$

and therefore the parallel ∃-step transition formula allows, irrespective of the chosen ordering $\prec$, the step $\{\alpha, \delta\}$ whenever the actions are both enabled.

Example 9. Let us next observe that the chosen order $\prec$ can make a difference in which non-unit steps are allowed by the parallel ∃-step transition formula if the actions are not totally mutually independent. For this consider the actions $\alpha : x \leftarrow x + 1$ and $\gamma : y \leftarrow x$. If α and γ are executed in parallel in the state $s = (pc_L = \text{L2}, pc_M = \text{M1}, x = 2, y = 2)$, the result is the state $s' = (pc_L = \text{L1}, pc_M = \text{M2}, x = 3, y = 2)$.

If the order $\prec$ sets $\gamma \prec \alpha$, then the corresponding linearization of first executing γ and then α in the state s results in the desired state s'. When the intuitive coherent encoding of α and γ is used, the parallel ∃-step transition formula allows this step as α does not read the variable y written by γ.

However, if the order $\prec$ sets $\alpha \prec \gamma$, then the corresponding linearization of first executing α and then γ in the state s would not produce the desired state s' but $s'' = (pc_L = \text{L1}, pc_M = \text{M2}, x = 3, y = 3)$. This linearization is also forbidden by the parallel ∃-step transition formula when a coherent encoding is used as one must have $write_x^\alpha = read_x^\gamma = \mathbf{T}$ and thus Eq. (11) forbids firing α and γ at the same time under this ordering.

Example 10. Finally, let us illustrate that the preciseness of the read operation encoding may also affect how many actions can be executed simultaneously in a step. Consider the state $s = (pc_L = \text{L2}, pc_M = \text{M1}, x = 2, y = 0)$ and the step $\{\beta, \gamma\}$ of the actions β : if $x > 2$ then $y \leftarrow y + 1$ else $y \leftarrow 2$ and $\gamma : y \leftarrow x$ under an ordering that sets $\gamma \prec \beta$. If the state-insensitive over-approximation $read_y^\beta := \mathbf{T}$ of $read_y^\beta$ is applied (recall Example 6), then the parallel ∃-step transition formula does not allow the step in the state s as $\gamma \prec \beta$ and $write_y^\gamma := \mathbf{T}$. However, if the state-sensitive version $read_y^\beta := (\tilde{x} > 2)$ given in Example 7 is applied, the step is allowed as $(\tilde{x} > 2)$ does not hold in s.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

The following propositions establish the correctness of the parallel ∃-step transition formula. That is, the formula allows executing any unit step and does not allow executing anything but ∃-steps.

Proposition 4. *The parallel ∃-step transition formula* $\tilde{\Delta}_{\text{par}}$ *constructed from a coherent operational encoding is complete with respect to the interleaving transition relation* Δ_{int}.

Proof. Every interpretation that satisfies the interleaving transition formula $\tilde{\Delta}_{\text{int}}$ of Definition 5 also satisfies the parallel ∃-step transition formula. In particular, (6) implies (11). The claim follows from the completeness of $\tilde{\Delta}_{\text{int}}$ with respect to the interleaving transition relation. □

Proposition 5. *The parallel ∃-step transition formula* $\tilde{\Delta}_{\text{par}}$ *constructed from a coherent operational encoding is sound with respect to the ∃-step transition relation* $\Delta_{\exists}$.

To avoid repeated arguments, we will prove Proposition 5 using a result in Section 4.3 below. The proof is postponed until that section.

4.2.1. Size of the parallel ∃-step transition formula

The new constraint in the parallel ∃-step transition formula is (11), repeated below.

$$\left(\bigvee_{i=1}^{j-1} \tilde{f}^i \wedge write_v^i\right) \Rightarrow \neg(\tilde{f}^j \wedge read_v^j) \quad \text{for } v \in V,\ 2 \le j \le K$$

The disjunction on the left-hand side states that "variable v is written before action act^j in the step". Increasing j to $j+1$ adds one new disjunct $\tilde{f}^j \wedge write_v^j$ to the left-hand side, which allows reusing the previous disjunction $\bigvee_{i=1}^{j-1} \tilde{f}^i \wedge write_v^i$ as a shared sub-formula. Thus, the total size of these constraints is linear in the number of actions K. The possibility of compact representation by nesting these "written-before" sub-formulas is one of the main reasons why we investigate ∃-step semantics with a fixed ordering of actions. We also point out that using the disjunctions $\bigvee_{i=1}^{j-1} \tilde{f}^i \wedge write_v^i$ is "free" in terms of the formula size since they occur as sub-formulas in the frame constraint (14), which is present already in the interleaving transition formula.

When $read_v^j$ is the constant false expression **F**, the constraint (11) can be omitted as it reduces to **T** by constant folding. Altogether, the size of the parallel ∃-step transition formula is, like the interleaving transition formula, linear in the size of the expressions encoding the actions. This time however, the size of the encoding also includes the expressions $read_v^j$ that are not **F**.

4.3. Serial ∃-step transition formula

The serial ∃-step transition formula is based on an ordering $\text{act}^1 \prec \cdots \prec \text{act}^K$ of the actions and an operational encoding with the expressions $enabled^i$, $write_v^i$, and $newvalue_v^i$. The $read_v^i$ expressions of Section 4.1 are not used. The idea is that any nonempty subsequence of $[\text{act}^1, \ldots, \text{act}^K]$ can be selected for execution in a single step, as long as the subsequence is executable starting from the current state. Conceptually, this is the same as putting K interleaving steps one after another, with the restriction that at the ith step, either action act^i is executed or nothing happens.

Unlike in the previous section, the selected actions need not be enabled in the current state, and the value of each state variable may change several times during the sequence. The intermediate states will be encoded in the expressions $temp_v^i$, where $0 \le i \le K$ and v is a state variable. The value of $temp_v^i$ is the intermediate value of v after executing those actions $\text{act}^1, \ldots, \text{act}^i$ that are in the step but before executing any of the actions $\text{act}^{i+1}, \ldots, \text{act}^K$. In particular, $temp_v^0$ corresponds to the current state and $temp_v^K$ to the next state.

To determine whether an action act^i is enabled, we need to evaluate $enabled^i$ in the $i-1$st intermediate state instead of the current state. Therefore, we substitute the expression $temp_u^{i-1}$ for each state encoding variable $\tilde{u}$ in $enabled^i$. The same applies to evaluating $write_v^i$ and $newvalue_v^i$. For $1 \le i \le K$ and $v \in V$, we will denote by $temp\text{-}enabled^i$, $temp\text{-}write_v^i$, or $temp\text{-}newvalue_v^i$ the expression that is obtained from $enabled^i$, $write_v^i$, or $newvalue_v^i$, respectively, by replacing (non-recursively) every occurrence of each $\tilde{u} \in \tilde{V}$ by the expression $temp_u^{i-1}$.

The intermediate state expressions are defined by

$$temp_v^0 := \tilde{v} \qquad \text{for } v \in V, \text{ and} \tag{15}$$

$$temp_v^i := \textbf{if } \tilde{f}^i \wedge temp\text{-}write_v^i \textbf{ then } temp\text{-}newvalue_v^i \textbf{ else } temp_v^{i-1} \qquad \text{for } v \in V,\ 1 \le i \le K. \tag{16}$$

That is, the ith intermediate state is obtained by executing act^i in the $i-1$st intermediate state if the choice encoding variable $\tilde{f}^i$ is true, and applying the changes to all variables that are written.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

Definition 8. The *serial* ∃*-step transition formula* $\tilde{\Delta}_{\text{ser}}$ is the conjunction of the following formulas.

$$\bigvee_{i=1}^{K} \tilde{f}^i \tag{17}$$

$$\tilde{f}^i \Rightarrow \textit{temp-enabled}^i \qquad \text{for } 1 \le i \le K \tag{18}$$

$$\tilde{v}' = \textit{temp}_v^K \qquad \text{for } v \in V \tag{19}$$

The constraints state that at least one action is executed (17), only enabled actions are executed (18), and the next state corresponds to the last intermediate state after executing all actions (19). Although the last constraint technically contains all expressions of the form $\textit{temp}_v^i$ as sub-expressions, the formula is in practice smaller because many of these can be simplified away, as later discussed in Section 4.3.1.

A strength of the serial ∃-step transition formula is that, unlike the previous transition formulas, it allows steps corresponding to potentially long paths in the control flow graphs of the processes of the system. This is because an action may enable further actions to be executed later in the same step, as shown in the examples below.

Example 11. Let the actions of Fig. 1 be ordered so that $\mathsf{act}^1 = \alpha$ and $\mathsf{act}^2 = \beta$. Action α moves to control location L1, encoded by

$$\textit{write}^1_{pc_L} := \mathbf{T}, \text{ and}$$
$$\textit{newvalue}^1_{pc_L} := \text{L1}.$$

The enabledness condition of β is $\textit{enabled}^2 \equiv (\tilde{pc}_L = \text{L1})$, which now becomes

$$\begin{aligned}\textit{temp-enabled}^2 &\equiv \left(\textit{temp}^1_{pc_L} = \text{L1}\right)\\ &\equiv \left((\textbf{if } \tilde{f}^1 \wedge \mathbf{T} \textbf{ then } \text{L1} \textbf{ else } \tilde{pc}_L) = \text{L1}\right),\end{aligned}$$

which simplifies to $\tilde{f}^1 \vee (\tilde{pc}_L = \text{L1})$. Thus, β can be enabled by either executing α in the same step, or by the location L1 being active already in the beginning of the step.

Example 12. Consider the initial state $s = (pc_L = \text{L2}, pc_M = \text{M1}, x = 2, y = 0)$ of the system in Fig. 1. Order the actions so that $\alpha \prec \beta \prec \delta \prec \gamma$. Now the serial ∃-step transition formula allows the steps $\{\beta\}$, $\{\gamma\}$, $\{\beta, \gamma\}$, and $\{\beta, \delta\}$ to be executed in the state and thus four different states can be reached from s in one step (instead of two in the interleaving case and at most three in the parallel ∃-step case). If the ordering $\beta \prec \delta \prec \gamma \prec \alpha$ is used instead, the steps $\{\beta\}$, $\{\gamma\}$, $\{\beta, \gamma\}$, $\{\beta, \gamma, \alpha\}$, $\{\beta, \alpha\}$, $\{\beta, \delta\}$, and $\{\beta, \delta, \alpha\}$ are allowed and seven different states can be reached from s in one step.

In the rest of this section, we will establish the correctness of the serial ∃-step transition formula. The formula is complete with respect to the interleaving transition relation and sound with respect to the *relaxed* ∃-step transition relation, where all executed actions need not be enabled in the beginning of the step. Thus, all unit steps and no spurious transitions are allowed by the formula. First however, we will show that when the order $\prec$ is fixed, the parallel ∃-step transition formula entails the serial ∃-step transition formula. In other words, every parallel ∃-step is also a serial ∃-step. We will use this as an auxiliary result in proving the correctness. A further consequence is that a fixed number of serial ∃-steps covers at least all the states covered in the same number of parallel ∃-steps.

Proposition 6. *Given a coherent encoding, if an interpretation over* $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ *satisfies the parallel* ∃*-step transition formula* $\tilde{\Delta}_{\text{par}}$*, then it also satisfies the serial* ∃*-step transition formula* $\tilde{\Delta}_{\text{ser}}$*.*

Proof. Let $\mathcal{I}$ be an interpretation over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ that satisfies the parallel ∃-step transition formula of Definition 7 built on a coherent encoding. Define interpretations $\mathcal{I}^0, \ldots, \mathcal{I}^K$ over $\tilde{V} \cup \tilde{C}$ by

$$\mathcal{I}^j(\tilde{v}) := \mathcal{I}(\textit{temp}_v^j) \qquad \text{for } v \in V, \text{ and} \tag{20}$$

$$\mathcal{I}^j(\tilde{a}) := \mathcal{I}(\tilde{a}) \qquad \text{for } \tilde{a} \in \tilde{C}. \tag{21}$$

Then, $\text{state}_{\mathcal{I}^j}$ corresponds to the jth intermediate state, and every $\mathcal{I}^j$ gives the same interpretation to the choice variables. By definition, for all $1 \le j \le K$ and $v \in V$,

$$\mathcal{I}(\textit{temp-enabled}^j) = \mathcal{I}^{j-1}(\textit{enabled}^j), \tag{22}$$

$$\mathcal{I}(\textit{temp-write}_v^j) = \mathcal{I}^{j-1}(\textit{write}_v^j), \text{ and} \tag{23}$$

$$\mathcal{I}(\textit{temp-newvalue}_v^j) = \mathcal{I}^{j-1}(\textit{newvalue}_v^j). \tag{24}$$

Let us prove by induction in j that for all $0 \le j \le K$ and $v \in V$,

$$\mathcal{I}^j(\tilde{v}) = \begin{cases} \mathcal{I}(\tilde{v}') & \text{if } \mathcal{I} \text{ satisfies } \bigvee_{i=1}^{j} \tilde{f}^i \wedge \textit{write}_v^i,\\ \mathcal{I}(\tilde{v}) & \text{otherwise.}\end{cases} \tag{25}$$

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

That is, an intermediate state of a parallel ∃-step coincides with the next state on those variables that have already been written, and with the current state on those variables that have not.

In the base case $j = 0$, the disjunction $\bigvee_{i=1}^{0} \tilde{f}^i \wedge write_v^i$ is **F**, so the right-hand side of (25) is $\mathcal{I}(\tilde{v}) = \mathcal{I}(temp_v^0) = \mathcal{I}^0(\tilde{v})$.

For the inductive step, let $1 \leq j \leq K$. The inductive hypothesis is

$$\mathcal{I}^{j-1}(\tilde{v}) = \begin{cases} \mathcal{I}(\tilde{v}') & \text{if } \mathcal{I} \text{ satisfies } \bigvee_{i=1}^{j-1} \tilde{f}^i \wedge write_v^i, \\ \mathcal{I}(\tilde{v}) & \text{otherwise.} \end{cases} \tag{26}$$

If $\mathcal{I}$ does not satisfy $\tilde{f}^j$, then the right-hand sides of (25) and (26) evaluate to the same for all v. On the other hand, $\mathcal{I}^j(\tilde{v}) = \mathcal{I}(temp_v^j) = \mathcal{I}(temp_v^{j-1}) = \mathcal{I}^{j-1}(\tilde{v})$ by (16).

Assume that $\mathcal{I}$ satisfies $\tilde{f}^j$. We rely on the coherence of the encoding (Definition 6), so let us verify that $\mathcal{I}^{j-1}$ j-conforms to $\mathcal{I}$. That is, $\mathcal{I}^{j-1}(\tilde{a}) = \mathcal{I}(\tilde{a})$ for all $\tilde{a} \in \tilde{A}^j$, and $\mathcal{I}^{j-1}(\tilde{v}) = \mathcal{I}(\tilde{v})$ for all $v \in R^j(\mathcal{I})$. The former follows directly from (21), and the latter from (26) and the fact that $\mathcal{I}$ satisfies (11) and $\tilde{f}^j$. Also, $\mathcal{I}$ satisfies $enabled^j$ by (12). From Definition 6, we have

1. $\mathcal{I}^{j-1}$ satisfies $enabled^j$,
2. $R^j(\mathcal{I}^{j-1}) = R^j(\mathcal{I})$,
3. $W^j(\mathcal{I}^{j-1}) = W^j(\mathcal{I})$, and
4. for all $v \in W^j(\mathcal{I})$, $\mathcal{I}^{j-1}(newvalue_v^j) = \mathcal{I}(newvalue_v^j)$.

From item 3 and (23), we have $\mathcal{I}(write_v^j) = \mathcal{I}^{j-1}(write_v^j) = \mathcal{I}(temp\text{-}write_v^j)$ for all $v \in V$. If v is such that $\mathcal{I}$ satisfies $write_v^j$ and hence also $\mathcal{I}(temp\text{-}write_v^j)$, then by (16), (24), item 4, and (13), $\mathcal{I}^j(\tilde{v}) = \mathcal{I}(temp_v^j) = \mathcal{I}(temp\text{-}newvalue_v^j) = \mathcal{I}^{j-1}(newvalue_v^j) = \mathcal{I}(newvalue_v^j) = \mathcal{I}(\tilde{v}')$, so (25) is fulfilled. If $\mathcal{I}$ does not satisfy $write_v^j$, then the right-hand sides of (25) and (26) again evaluate to the same value, and the left-hand side of (25) becomes $\mathcal{I}^j(\tilde{v}) = \mathcal{I}(temp_v^j) = \mathcal{I}(temp_v^{j-1}) = \mathcal{I}^{j-1}(\tilde{v})$ by (16) because $\mathcal{I}$ does not satisfy $\mathcal{I}(temp\text{-}write_v^j)$. This concludes the inductive proof.

Now $\mathcal{I}$ satisfies the serial ∃-step transition formula. Namely, the constraint (17) is the same as (10). If j is such that $\mathcal{I}$ satisfies $\tilde{f}^j$, then by item 1 above and (22), $\mathcal{I}$ also satisfies $temp\text{-}enabled^j$ and thus (18) is fulfilled. By (25) and (14), $\mathcal{I}^K(\tilde{v}) = \mathcal{I}(\tilde{v}')$ for all $\tilde{v} \in \tilde{V}$, so (19) is satisfied. □

Proposition 7. *The serial ∃-step transition formula $\tilde{\Delta}_{\text{ser}}$ constructed from an operational encoding is complete with respect to the interleaving transition relation Δ_{int}.*

Proof. We rely on the completeness of the parallel ∃-step transition formula. Extend the operational encoding to a coherent operational encoding by setting, for example, $read_v^i := \mathbf{T}$ for $1 \leq i \leq K$ and $v \in V$. By Proposition 4, for every unit step in the interleaving transition relation, there is a corresponding interpretation $\mathcal{I}$ that satisfies the parallel ∃-step transition formula. According to Proposition 6, $\mathcal{I}$ also satisfies the serial ∃-step transition formula. □

Proposition 8. *The serial ∃-step transition formula $\tilde{\Delta}_{\text{ser}}$ constructed from an operational encoding is sound with respect to the relaxed ∃-step transition relation $\Delta_{\exists\text{relax}}$.*

Proof. Take any interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ that satisfies (17)–(19). We will show that $\langle \text{state}_{\mathcal{I}}, \text{ex}_{\mathcal{I}}, \text{next-state}_{\mathcal{I}} \rangle \in \Delta_{\exists\text{relax}}$ (see (1), (2), (3), and Definition 2) with the ordering that respects the order $\text{act}^1 \prec \cdots \prec \text{act}^K$.

Define interpretations $\mathcal{I}^0, \ldots, \mathcal{I}^K$ over $\tilde{V} \cup \tilde{C}$ as in the proof of Proposition 6 by (20) and (21). Then, the identities (22)–(24) hold as well. The path from $\text{state}_{\mathcal{I}}$ to $\text{next-state}_{\mathcal{I}}$ in the interleaving state space will be formed from the states encoded in $\mathcal{I}^0, \ldots, \mathcal{I}^K$. By (15), $\text{state}_{\mathcal{I}^0} = \text{state}_{\mathcal{I}}$, and we see from (19) that $\text{state}_{\mathcal{I}^K} = \text{next-state}_{\mathcal{I}}$.

Definition 2 requires that $\text{ex}_{\mathcal{I}}$ is nonempty. This is fulfilled by (17). For fulfilling the linearizability requirement, we show that $\text{state}_{\mathcal{I}^j} = \text{state}_{\mathcal{I}^{j-1}}$ when $\text{act}^j \notin \text{ex}_{\mathcal{I}}$, and $\langle \text{state}_{\mathcal{I}^{j-1}}, \text{act}^j, \text{state}_{\mathcal{I}^j} \rangle \in \Delta_{\text{int}}$ when $\text{act}^j \in \text{ex}_{\mathcal{I}}$. In the case $\text{act}^j \notin \text{ex}_{\mathcal{I}}$, $\mathcal{I}$ does not satisfy $\tilde{f}^j$, so by (16), $\text{state}_{\mathcal{I}^j} = \text{state}_{\mathcal{I}^{j-1}}$. In the case $\text{act}^j \in \text{ex}_{\mathcal{I}}$, as $\mathcal{I}$ satisfies $\tilde{f}^j$ and (18), $\mathcal{I}^{j-1}$ satisfies $enabled^j$ by (22). The claim then follows by applying Definition 4 with the interpretation $\mathcal{I}^{j-1}$ and verifying from (16), (23), and (24) that $\text{succ}_{\mathcal{I}^{j-1}}^j = \text{state}_{\mathcal{I}^j}$. □

Using the above results, we are ready to give a short proof to Proposition 5 (Section 4.2), which states that the parallel ∃-step transition formula $\tilde{\Delta}_{\text{par}}$ constructed from a coherent operational encoding is sound with respect to the ∃-step transition relation $\Delta_{\exists}$.

Proof of Proposition 5. Take any interpretation $\mathcal{I}$ over $\tilde{V} \cup \tilde{C} \cup \tilde{V}'$ that satisfies the parallel ∃-step transition formula. As required by Definition 3, we will show that $\langle \text{state}_{\mathcal{I}}, \text{ex}_{\mathcal{I}}, \text{next-state}_{\mathcal{I}} \rangle \in \Delta_{\exists}$. According to Proposition 6, $\mathcal{I}$ satisfies the serial ∃-step transition formula. By Proposition 8, the tuple $\langle \text{state}_{\mathcal{I}}, \text{ex}_{\mathcal{I}}, \text{next-state}_{\mathcal{I}} \rangle$ is in the relaxed ∃-step transition relation $\Delta_{\exists\text{relax}}$ of Definition 2. Thus, to show that the requirements of Definition 2 are fulfilled as well, we only need to check that every $\text{act}^j \in \text{ex}_{\mathcal{I}}$ is enabled in $\text{state}_{\mathcal{I}}$.

Let $\text{act}^j \in \text{ex}_{\mathcal{I}}$, that is, $\mathcal{I}$ satisfies $\tilde{f}^j$. By (12), $\mathcal{I}$ also satisfies $enabled^j$, and by Definition 4, the tuple $\langle \text{state}_{\mathcal{I}}, \text{act}^j, \text{succ}_{\mathcal{I}}^j \rangle$ is in the interleaving transition relation, which means that act^j is enabled in $\text{state}_{\mathcal{I}}$. □

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

4.3.1. Size of the serial ∃-step transition formula

Each encoding expression $enabled^i$, $write^i_v$, and $newvalue^i_v$ occurs once in the serial ∃-step transition formula, thus the transition formula size is again linear in the size of the encoding. However, in a serial ∃-step, data no longer flows directly from the current to the next state, but through K intermediate states. Consider two actions, say act^3 and act^4, that both contain the statement $x \leftarrow x+10$. In the interleaving or parallel ∃-step transition formula, the statement is always evaluated in the current state, so the expression $\tilde{x}+10$ is shared among the logic of act^3 and act^4. In the serial ∃-step transition formula, we get two different copies, $temp\text{-}newvalue^3_x \equiv temp^2_x + 10$ and $temp\text{-}newvalue^4_x \equiv temp^3_x + 10$, that cannot be shared. This duplication of logic might become significant in a model where several actions perform the same complex operation such as removing a message from a communication buffer.

The expressions $temp^0_v, \ldots, temp^K_v$ give the intermediate values of a variable v in a serial ∃-step, but generally, only a fraction of these are explicitly represented in the transition formula. Namely, if $write^i_v$ is the constant false expression **F**, then the definition (16) of $temp^i_v$ reduces to $temp^i_v := temp^{i-1}_v$, and these two expressions share their representation. Thus, the part of the ith intermediate state that is known not to change from the previous intermediate state does not contribute to the size of the transition formula.

Let L^i be the total size of the encoding expression $enabled^i$ together with $write^i_v$ and $newvalue^i_v$ for each $v \in V$ such that $write^i_v$ is not **F**. Then, the size of the serial ∃-step transition formula $\tilde{\Delta}_{\text{ser}}$ is linear in $\sum_{i=1}^{K} L^i$. In contrast to the interleaving transition formula, the encoding of each action is in the worst case counted separately toward the size of $\tilde{\Delta}_{\text{ser}}$, even if the encodings of different actions contain identical sub-expressions.

5. Experiments with ∃-step semantics

For experimental evaluation, we use the benchmark set BEEM (Benchmarks for Explicit Model checkers) [37]. As its name suggests, the benchmark set is tailored for explicit-state model checkers and is thus not necessarily an ideal test set for demonstrating the strength of symbolic methods. However, BEEM provides a large number of benchmarks laid out in a uniform framework that facilitates automatic translation, outlined in Section 5.1. For every benchmark, we employ bounded model checking to look for a witness for a given reachability property (or equivalently, a counter-example to an invariant property) with successive bounds 0, 1, 2, and so on. In this mode, it is only possible to give a definite answer if the reachability property holds. In Section 5.2, we compare the performance of BMC using the two ∃-step transition formulas to using the interleaving transition formula, and also contrast to explicit-state model checking. The effect of the ordering of actions on performance is discussed in Section 5.3.

5.1. Encoding system features

The BEEM benchmark set consists of 57 models that are parameterized so that one can choose, for example, the number of distributed components or whether the system should contain a bug or not. The set defines a number of instantiations of these parameterized models, and one or more reachability properties for most of the models. In total, there are 344 instances consisting of a system and a reachability property. The systems are specified in a subset of the DVE language, which is the input language of the DiVinE model checker [9]. We encode the systems as transition formulas for BMC using an automatic translation tool written in the Python programming language and exploiting the parser from the DiVinE tool. The translation covers all the features used in the 344 instances.

Below, we discuss ways to encode common constructs occurring in asynchronous systems, and DVE constructs in particular.

Control logic. A DVE system is specified using a fixed set of concurrent processes, each consisting of a finite number of control locations and transitions between them. We map individual transitions directly to actions of the system.

DVE also supports rendezvous communication channels, where one transition sends a message to the channel, while a transition in another process must simultaneously receive the message. To fit this kind of synchronous communication in an asynchronous formalism, we define a separate action for each possible sender–receiver pair of transitions. In the worst case, the number of actions is thus quadratic in the number of transitions, but in practice, we never get more actions than roughly twice the number of transitions in the BEEM benchmarks. Alternatively, we could define for each channel an action that includes the encoding of all transitions that synchronize through the channel. Action-specific choice variables would then be used to choose one sender transition and one receiver transition for execution. In this approach, we would lose the simplicity of knowing statically the control location changes involved with each action.

For each control location, we use a propositional state variable denoting whether the location is active. We optimize the transition formula with an extra constraint that exactly one location is active in every process. For the location variables, the parallel ∃-step constraint (11) reduces to forbidding the execution of an action if its source location has been made inactive by a preceding action. This optimized formula is semantically equivalent to having a single program counter variable for each process, but gives slightly better solver performance.

Variables and expressions. DVE supports only 16-bit signed and 8-bit unsigned integer variables, which are directly represented as state variables using the theory of bit-vectors. Intermediate values are computed as in the C language, with

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

32-bit modular arithmetic and the usual operators. To preserve correctly the semantics of overflows, truncation, and so on, these are encoded into 32-bit bit-vector operations.

The effects of transitions in DVE are just sequences of assignments. There are no conditional statements or loop constructs, so in the case of scalar variables, we simply define that a variable is written if it occurs as the target of an assignment, and a variable is read if it occurs in the action anywhere except as the target of an assignment.

Arrays. Arrays in DVE have a fixed, finite length. We use a conventional translation where separate state variables are defined for all array elements. The read and write dependences are evaluated dynamically: an action reads an array A at index k iff the action description contains an expression $A[\phi]$ such that the index expression ϕ evaluates to k. Thus, reading or writing the same array does not make two actions dependent or prevent including them in the same parallel ∃-step, unless the accessed elements also happen to overlap in the current state.

Because of the expansion to separate variables, the encoding grows linearly in the size of the array. However, if the array is read or written using an index expression that is constant (common in BEEM), then by constant folding, the encoding reduces to reading or writing the array element as if it was a scalar variable.

Our translation follows no meaningful semantics if array bounds are violated—in production use, boundary checks could be included in the invariant property.

Example 13. Consider a system with an array variable A of length 3 and an action act^i whose effect is the sequence $x \leftarrow A[y]; A[x] \leftarrow 0; A[y] \leftarrow 1$. The array is represented with state variables A_0, A_1, and A_2, and the action is encoded as follows.

$$
\begin{aligned}
&\mathit{read}^i_x := \mathit{read}^i_y := \mathit{write}^i_x := \mathbf{T} \\
&\mathit{write}^i_y := \mathbf{F} \\
&\mathit{newvalue}^i_x := \textbf{if } \tilde{y} = 0 \textbf{ then } \tilde{A}_0 \textbf{ else if } \tilde{y} = 1 \textbf{ then } \tilde{A}_1 \textbf{ else } \tilde{A}_2 \\
&\mathit{read}^i_{A_k} := (\tilde{y} = k) && \text{for } 0 \le k \le 2 \\
&\mathit{write}^i_{A_k} := (\mathit{newvalue}^i_x = k) \vee (\tilde{y} = k) && \text{for } 0 \le k \le 2 \\
&\mathit{newvalue}^i_{A_k} := \textbf{if } (\tilde{y} = k) \textbf{ then } 1 \textbf{ else } 0 && \text{for } 0 \le k \le 2.
\end{aligned}
$$

With large arrays, this translation may result in a large number of constraints in the parallel ∃-step formula (11). To avoid separate constraints for each array element, we could take an alternative approach and treat the array essentially as a monolithic variable, defining only a single read^i_A and write^i_A formula for each action, with the meaning "act^i reads/writes *some* element of array A". The contents would still be represented using one (bit-vector) state variable per array element. This corresponds to the "static" ∃-step translation described in [22]. Such an approach disregards the independency of actions that access disjoint array elements, and also requires that every action that writes to an array element explicitly writes the old values back to the unchanged elements. Thus, an assignment $A[0] \leftarrow x$ would be essentially encoded as $A[0] \leftarrow x; A[1] \leftarrow A[1]; A[2] \leftarrow A[2]; \ldots; A[N] \leftarrow A[N]$.

If the theory of arrays [6] is supported by the solver, then the monolithic encoding can be applied without expanding the elements, with constant-size read and write operations regardless of the length of the array. This approach also works if the length is unbounded.

Queues. A fairly commonly occurring data type is a first-in-first-out queue. Queues can occur, for example, as communication buffers, where each queue element represents a message that is waiting to be dispatched. Although the DVE language supports buffered communication channels, the language subset chosen for BEEM only includes rendezvous channels that do not buffer messages.

The queue representation can be built on top of an array in a number of different ways [38]. An interesting property of queues is that operations on the front of the queue (querying the front-most element or removing it) are independent from operations on the back of the queue (adding a new element) unless the queue is empty or full. This is exploited in an ∃-step transition formula in [22] in a restricted setting, where at most one element can be removed and at most one element added to each queue in one step. Efficient ∃-step formulations of actions with arbitrary sequences of queue operations are a topic left for future work.

5.2. Results with ∃-step semantics

We compare the performance of bounded model checking with the parallel and serial ∃-step transition formulas to two baselines: BMC with the interleaving transition formula and explicit-state model checking using the DiVinE tool.

All experiments were run using two SMT solvers, Yices 2.0 [39] and Boolector 1.2 [40], both of which were successful in the SMT-COMP 2009 competition. The theory of bit-vectors was used to follow the DVE semantics precisely, and other theories were not needed. These two solvers have very similar performance on our benchmarks, and unless otherwise noted, only the results for Yices are shown. We point out that the SMT-LIB 1.2 interface used by these solvers does not support incremental solving, which has been shown to increase SAT-based BMC performance significantly [41]. In preliminary experiments using

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

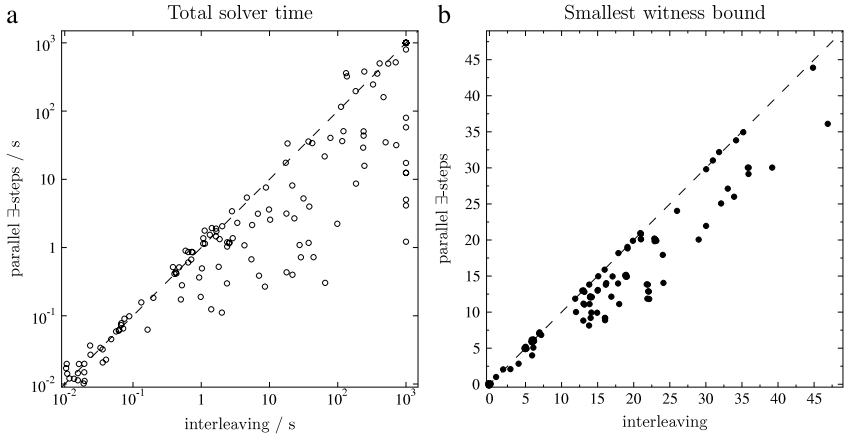


Fig. 4. BMC efficiency with parallel ∃-step vs. interleaving transition formulas. Comparing (a) BMC run times and (b) required BMC bounds to find executions witnessing reachability properties of BEEM benchmark instances.

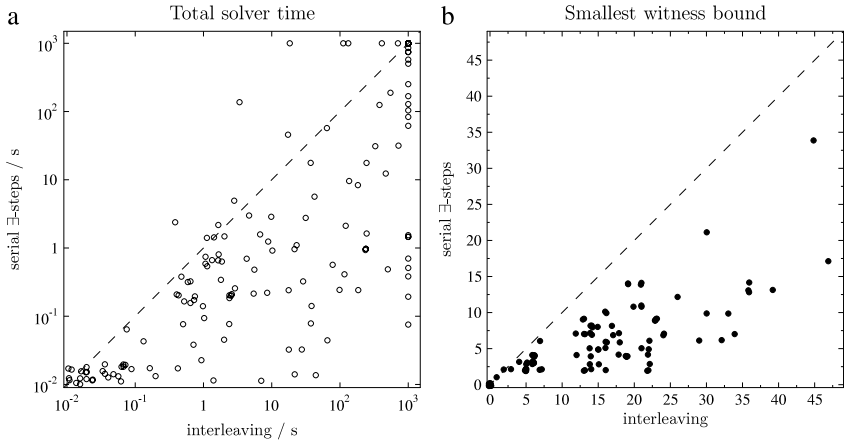


Fig. 5. BMC efficiency with serial ∃-step vs. interleaving transition formulas.

the MiniSat solver on the same benchmark set, we observed that incrementality usually improves SAT solving time. In half of the cases, the time to find a witness was reduced by 50% or more. For final experiments however, we used Yices and Boolector for their better overall scalability on these benchmarks.

The experiments were run on a single core of an Intel Xeon 5130 processor with a timeout limit of 1000 s and a memory limit of 1.5 gigabytes. Exceeding the memory limit did not occur with either of the SMT solvers, only with explicit-state model checking.

5.2.1. The effect of ∃-step semantics

We measure the efficiency of witnessing reachability properties (in other words, finding counter-examples to invariant properties) with BMC using the ∃-step transition formulas compared to the base case of using the interleaving transition formula. In Figs. 4(a) and 5(a), each marker denotes a BEEM benchmark instance for which a witness is found. The logarithmic *x*-axis shows the cumulative solver CPU time with the interleaving transition formula from bound 0 up to and including the bound *k* with which a witness is found, i.e. the total run time of *k* unsatisfiable and the final satisfiable SMT call. Markers at the 1000 s limit denote timeouts. The *y*-axis shows the run time with the parallel ∃-step transition formula in Fig. 4(a), and with the serial ∃-step transition formula in Fig. 5(a). We do not include the time taken to construct the transition formulas because the speed of the prototype translator would not be a meaningful measure. Nonetheless, the translation task is straightforward and all real model checking work is done in the solver.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

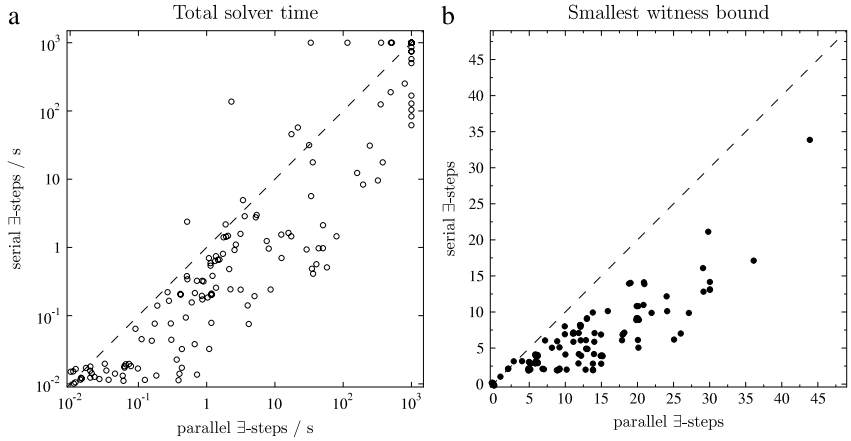


Fig. 6. BMC efficiency with serial vs. parallel ∃-step transition formulas.

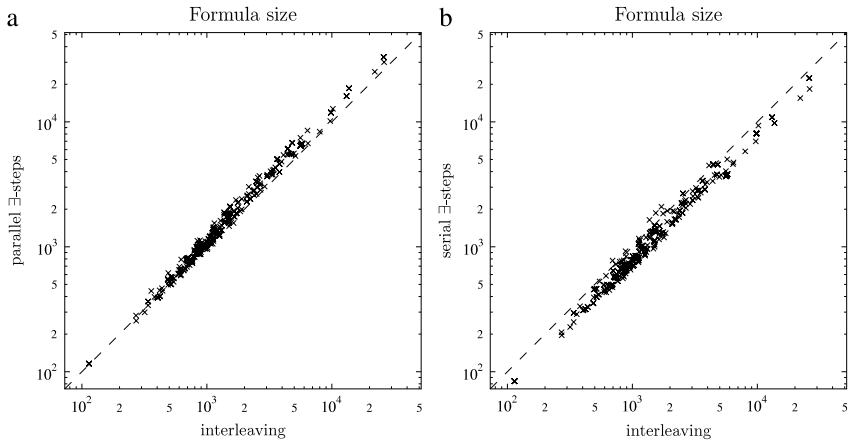


Fig. 7. ∃-step vs. interleaving transition formula sizes.

The benefit of employing step semantics is evident. The parallel ∃-steps usually outperforms and in the worst case matches the interleaving semantics in speed, with the relative speedup increasing with harder problems. The speed difference to the serial ∃-steps is even more prominent, with up to 10 000-fold speedup. In Figs. 4(b) and 5(b), we verify that the difference is explained by the decreased bounds at which counter-examples are found. Again, each marker represents a benchmark instance, the horizontal position denoting the smallest interleaving bound that witnesses the property, and the vertical position denoting the bound with step semantics. The range of the graphs is clipped for clarity—3 cases are left out. Markers on the diagonal in Fig. 4(b) are cases where the shortest witness execution cannot be compressed by fusing together independent unit steps, thus the parallel ∃-steps give no benefit. We point out that even a modest decrease in the bound helps, since in practice, the solver time grows super-linearly, even exponentially in the bound. Fig. 5(b) shows that with serial ∃-steps, the required bound almost always drops dramatically—in an extreme case, a witness of 22 unit steps is covered in 2 serial ∃-steps. Fig. 6 compares the parallel and the serial ∃-steps, indicating that serial ∃-steps quite consistently perform better.

To measure the size of a transition formula under sub-expression sharing, we count the total number of different sub-expressions in the formula. This is not an ideal measure, as it makes no distinction between simple Boolean operators and expressive bit-vector arithmetic operators, for example. The sizes computed for all benchmark instances are plotted in Fig. 7. On average, the parallel ∃-step transition formula is 13% larger than the interleaving transition formula, and the serial ∃-step transition formula is 21 % smaller. The small size of the latter is probably explained by the use of if-then-else formulas (16) instead of implications (8) for encoding write operations. To illustrate the difference, the formula $(\tilde{v}' = \textbf{if } c \textbf{ then } x \textbf{ else } \tilde{v})$ measures smaller than the equivalent pair of implications $\big(c \Rightarrow (\tilde{v}' = x)\big) \wedge \big(\neg c \Rightarrow (\tilde{v}' = \tilde{v})\big)$. Nevertheless, we see that

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

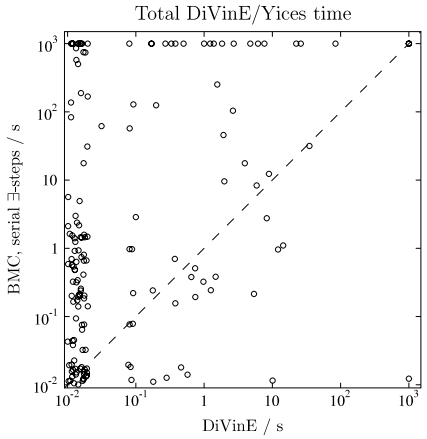


Fig. 8. Comparison of BMC vs. explicit-state model checking.

there are no essential differences in the sizes of the three representations. At least with these benchmarks, the potential swelling of the serial ∃-step transition formula due to the intermediate states is not significant.

5.2.2. Comparison with explicit-state model checking

In Fig. 8, we compare the run time of the DiVinE explicit-state model checker (x-axis) to the cumulative solver time with BMC using the serial ∃-step transition relation (y-axis). Each marker denotes a single benchmark instance, and only those instances are included where one of the methods can find a witness to the property. Markers at the 1000 s limit denote timeouts or memouts. Although DiVinE is designed for multi-core model checking, we run it single-threadedly to avoid biasing the comparison. Of the three model checking algorithms implemented in DiVinE, we use *nested-dfs*, which performs best for our benchmarks (technically, reachability properties have to be expressed in LTL for DiVinE).

The mass close to the left border of the plot is the set of cases where DiVinE finds a witness instantly, while the BMC run time varies over several orders of magnitude. In the remaining cases, BMC performs generally worse than DiVinE, but is sometimes significantly faster. There is almost no correlation between the performance of the two methods, suggesting that the two approaches complement each other. For some insight, the BEEM web site at http://anna.fi.muni.cz/models/ contains size estimates of the reachable state spaces of the benchmarks. Ignoring the cases where neither BMC or DiVinE finds a witness, as well as trivial cases where both approaches take less than 0.05 s, we have 69 small benchmark instances (less than 750 000 reachable states). BMC is faster than DiVinE on just 5 of them. Of the 59 large benchmark instances, 18 are such that BMC is faster. Apparently, the explicit method gets relatively less successful when the search space grows. The two clearest wins for BMC are benchmarks where a very small bound (2 serial ∃-steps) suffices to witness the property, but the explicit-state model checker gets lost in the large state space. Benchmarks that involve a clock variable seem to be especially challenging for BMC—even step semantics does not help if the clock has to be repeatedly incremented by an action that models time-elapse.

A factor in favor of DiVinE is that, as the name suggests, the BEEM benchmarks are designed for explicit-state model checkers. For example, a frequent pattern in BEEM is to pack tuples of values into integers of the form $a + bN + cN^2$, where N is some constant radix. Division and modulo operators are then used to unpack the original values a, b, and c from the sum. In the transition formula for BMC, these are instantiated as bit-precise operations with signed 32-bit arithmetic, which is potentially extremely cumbersome for the solver to handle. Also, symbolic model checking would naturally incorporate non-deterministic choice of data values from a domain (cf. Example 5), but the DVE language does not support this, and in BEEM, the only kind of nondeterminism is the choice of the next action to be executed. Considering these facts together with the fact that incremental SAT/SMT solving is not applied in the BMC experiments, BMC with the serial ∃-step transition formulas performs surprisingly well against DiVinE on its home field.

According to a comparison in 2007 (http://spinroot.com/spin/beem.html), the explicit-state model checker Spin covers the state space of most BEEM benchmarks faster than DiVinE, but reachability property checking is not included in the comparison for the lack of a translation of these properties to the language of Spin.

5.3. Action orderings

For each total order $\prec$ of the set of actions, we get a different parallel or serial ∃-step transition formula that may also allow a different set of steps. Hence, the ordering may affect the smallest bound required to find a witness. We measure this

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

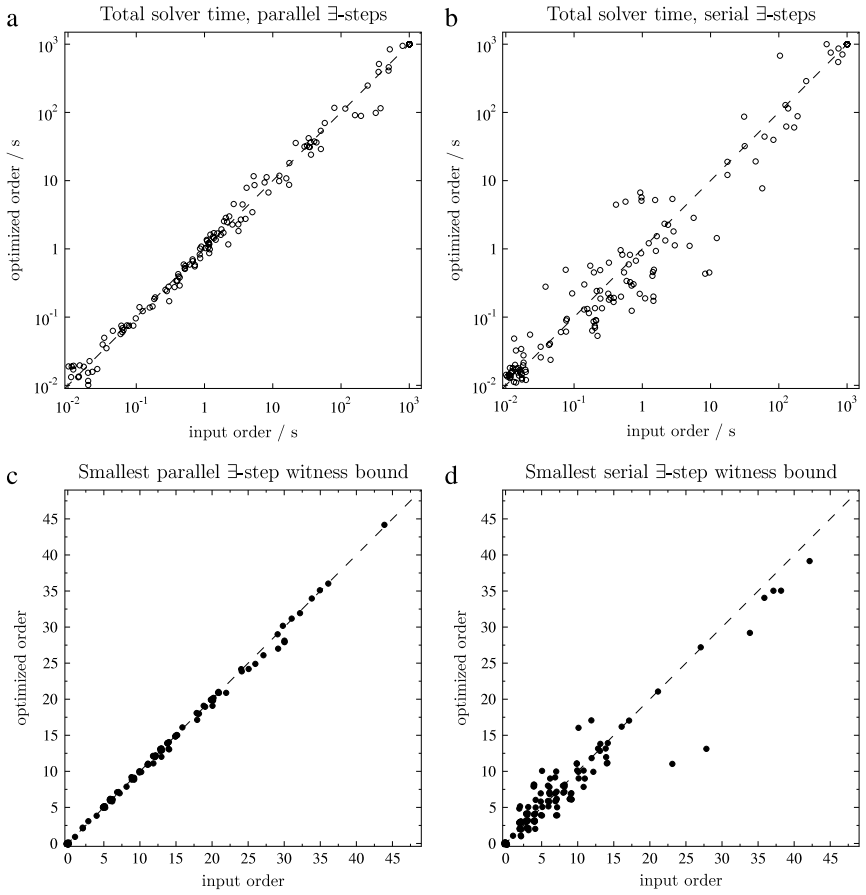


Fig. 9. BMC run times and required bounds with different orderings of actions.

by using heuristics to find good orderings. In all previous experiments, we have used the unmodified "input" order, which corresponds to the order in which transitions are specified in the DVE input.

With parallel ∃-steps, to forbid as few steps as possible and also to minimize the transition formula size, we want to avoid pairs of actions $\mathrm{act}^i \prec \mathrm{act}^j$ such that act^i potentially writes a variable that act^j reads. We form a graph consisting of the directed edges $\mathrm{act}^j \rightarrow \mathrm{act}^i$ such that a simple static analysis finds a variable that act^i can write and act^j can read. Then, we want to find a total order $\prec$ that minimizes the pairs of actions such that $\mathrm{act}^j \rightarrow \mathrm{act}^i$ and $\mathrm{act}^i \prec \mathrm{act}^j$. This is an instance of the NP-hard *minimum feedback arc set* problem [42]: find a minimum set of edges such that removing the set leaves an acyclic graph. For the experiments, a simulated annealing algorithm was implemented to find a heuristically optimized order. For another approach to finding action orders, see [13] that uses a data structure called disabling graphs to compute good orderings of actions.

Fig. 9(a) compares the cumulative solver time for finding a witness using the parallel ∃-step transition formula with the input order vs. the optimized order, and Fig. 9(c) shows the respective minimum bounds. The time spent by simulated annealing is not included. The optimization reduces the transition formula size by 5% on average, but there is hardly any effect on performance or the bound. This is not surprising, since the power of parallel ∃-steps mostly comes from the ability to select any set of independent enabled actions for execution in the same step, regardless of the total order. The order only affects the bound if it is critical that at some point in the shortest witness, a variable is read before it is overwritten by another action later in the same step.

With serial ∃-steps, the situation is different because not only can several processes execute one transition in the same step, but each process can potentially execute many transitions. For example, a path of n transitions in the control flow graph

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

of a process can be executed in one step with an optimal ordering, but requires n steps with the worst ordering. To promote executing such paths, we heuristically solve the minimum feedback arc set problem for the graph with the directed edges $\text{act}^j \to \text{act}^i$ such that there is a control location that is the target of act^j and the source of act^i. When control flow induces no ordering between act^i and act^j but there is potential data flow from act^i to act^j, it seems that $\text{act}^i \prec \text{act}^j$ is the appropriate order. We therefore add to the graph the same edges as in the parallel case, but with the opposite direction and only a small weight of 1/8.

The effects of the obtained order on the solver time and the bound with the serial ∃-step transition relation are shown in Fig. 9(b) and (d). The optimization is beneficial on average, but not consistently. From Fig. 9(d), we see that the optimization can improve the required bound, but in many cases increases it instead. Finding more robust ways to order the actions for efficient serial ∃-steps is left for future work.

Experimentation was also done with the reversed input order, a random order, and using the parallel-optimized order for the serial case and vice versa. As expected, BMC with the parallel ∃-step transition formula is indifferent to these choices, and BMC with the serial ∃-step transition formula shows slight performance degradation with respect to using the input order. The latter is because in the input order, usually the actions are already approximately in the order in which the system designer intended them to be executed.

6. Serial process semantics

A known drawback of step semantics is that by adding shortcut edges to the state space, the number of executions reaching a given state may increase significantly. This may have an adverse effect on bounded model checking because, roughly speaking, the satisfiability solver needs to explore a larger set of executions. The so-called *process* semantics [12] has been proposed as a way to limit the search to executions in a certain normal form. The principle is that actions can only be executed in the immediately following step after their preconditions become fulfilled. If an action is not executed at the earliest opportunity, it is not executed at all. Process semantics, like step semantics, preserves reachability properties. We now adapt the idea to serial ∃-steps. In Section 6.1 below, we introduce the serial process normal form of executions. In Section 6.2, we present a way to add extra constraints to the serial ∃-step formula to achieve the process normal form. Section 6.3 makes an experimental comparison between serial ∃-step and serial process semantics. A topic not included here is the possibility to adapt the process semantics optimization to parallel instead of serial ∃-steps.

6.1. Serial process normal form

Let us examine an execution with a BMC bound k under the serial ∃-step semantics. The execution consists of steps numbered from $t = 0$ to $k - 1$. With the action ordering $\text{act}^1 \prec \cdots \prec \text{act}^K$, a subsequence of $\text{act}^1, \ldots, \text{act}^K$ is executed at each step t. Correspondingly, we divide the step t into sub-steps denoted by $t{:}1, \ldots, t{:}K$. In total, we have a sequence of sub-steps $0{:}1, 0{:}2, \ldots, 0{:}K, 1{:}1, 1{:}2, \ldots, k{-}1{:}K$. At sub-step $t{:}i$, either act^i is executed or nothing happens. This is illustrated by an example in Fig. 10—the middle columns are explained later.

The principle of the serial process semantics is as follows. Assume that at a sub-step $t{:}i$, the action act^i is selected for execution (say, act^3 at 2:3 in Fig. 10), and it is independent of all actions at the $K - 1$ preceding sub-steps (in this case, the sub-steps 1:4, 2:1, and 2:2). Also assume that the sub-step $t{-}1{:}i$ is idle. Then, we can move act^i from sub-step $t{:}i$ to $t{-}1{:}i$ to get a modified execution. Because of the independence, this does not affect which state is reached at the end of the sub-step $t{:}i$. In this way, we move actions to earlier time steps across independent actions until no action can be moved further. We say that a serial ∃-step execution where the transformation is not possible is in the *serial process normal form*.

Observe that transforming an execution to the serial process normal form does not change the final state reached. The number of steps either stays the same or decreases because of the last step k becoming empty. The new execution contains the same actions as the original one, in a possibly different but Mazurkiewicz trace equivalent order [43,12]. In general, given a set of letters and an independence relation between them, two sequences of letters are in the same Mazurkiewicz trace if and only if one can be obtained from the other by repeatedly transposing subsequent independent letters. To the authors' knowledge, the serial process normal form does not correspond to any of the canonical normal forms of Mazurkiewicz traces defined in the literature. Consequently, it is not obvious whether two Mazurkiewicz trace equivalent executions always have the same serial process normal form for a given order $\prec$. This question is not yet pursued in this work.

To restrict the model checking search to executions in the serial process normal form, we need to characterize it symbolically. As with the parallel ∃-steps, the dependences between actions are computed (or over-approximated) using the dynamic information of read and written variables. In short, two actions might be dependent if they access the same variable, and at least one of the accesses is a write. In a serial ∃-step execution, we say that a variable is *write-fresh* (*read-fresh*) at a sub-step $t{:}i$ if it is written (read, resp.) at least once during the $K - 1$ sub-steps preceding $t{:}i$. In the example of Fig. 10, the action act^2 at 0:2 writes to the state variable x, and x stays write-fresh for the following 3 sub-steps.

Using these notions, a serial ∃-step execution in the serial process normal form can be characterized as follows. For each sub-step $t{:}i$ in which an action act^i is selected for execution, at least one of the following conditions must hold.

1. $t = 0$, that is, the action occurs in the first step, or otherwise,
2. act^i is also selected for execution at the sub-step $t{-}1{:}i$,

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

sub-step	read-fresh	write-fresh	executed	reads	writes
0:1					
0:2			act^2	x	x
0:3	x	x	act^3	x	z
0:4	x	x z			
1:1	x	x z	act^1	y	z
1:2	x y	z			
1:3	y	z			
1:4	y	z	act^4	x, y	y
2:1	x y	y			
2:2	x y	y			
2:3	x y	y	act^3	x, z	z
2:4	x z	z			

Fig. 10. An example of a serial ∃-step execution of a system with state variables $\{x, y, z\}$ and actions $\{\mathsf{act}^1, \ldots, \mathsf{act}^4\}$. The execution is not in the serial process normal form because the action act^3 at sub-step 2:3 violates the normal form property. The action could be moved to the earlier sub-step 1:3.

3. act^i at $t{:}i$ writes a variable that is read-fresh,
4. act^i at $t{:}i$ writes a variable that is write-fresh, or
5. act^i at $t{:}i$ reads a variable that is write-fresh.

If none of the conditions hold, then the action at $t{:}i$ can be pushed back K sub-steps into $t{-}1{:}i$. Otherwise, in cases 1 and 2, the action cannot be moved simply because there is no free space to move it to. In the remaining cases, the transformation might not be legal because there is an interfering dependent action instance. For example, the action at 1:4 in Fig. 10 cannot be moved to 0:4 because it might change the value of y read by act^1 at the sub-step 1:1 (case 3). In turn, act^1 at 1:1 cannot be moved to 0:1 because after the transformation, the value of z written by act^1 would be overwritten at the sub-step 0:3 (case 4). However, the execution is not in the serial process normal form because the sub-step 2:3 does not fulfill any of the conditions.

6.2. Serial process execution formula

We apply the serial process semantics to bounded model checking by taking the serial ∃-step transition formula, unrolling it to a bound, and adding extra constraints that enforce the serial process normal form. We call the resulting formula the *serial process execution formula*. The extra constraints are a direct formalization of the conditions stated above. For this, we assume that a coherent operational encoding is given, i.e. a correct encoding of the actions including the read^i_v formulas as defined in Sections 3.1 and 4.1.

As the conditions of the serial process normal form cannot be expressed in terms of a single time step, we cannot technically formulate them as part of the transition formula. Instead, we use the unrolled transition formula. With bound k, according to the BMC formula (4), the serial ∃-step transition formula $\tilde{\Delta}_{\mathrm{ser}}$ is unrolled k times to obtain

$$\bigwedge_{t=0}^{k-1} \tilde{\Delta}_{\mathrm{ser}}(\tilde{V}^t, \tilde{C}^t, \tilde{V}^{t+1}),$$

which we call the *k-unrolling of* $\tilde{\Delta}_{\mathrm{ser}}$. We denote the timed copies of the choice encoding variables $\tilde{f}^i$ by $\tilde{f}^{0:i} \in \tilde{C}^0, \tilde{f}^{1:i} \in \tilde{C}^1, \ldots, \tilde{f}^{k-1:i} \in \tilde{C}^{k-1}$. Thus, $\tilde{f}^{t:i}$ means that the action act^i is selected for execution at the sub-step $t{:}i$. Similarly, for all time steps $0 \le t \le k-1$, action indices $1 \le i \le K$, and state variables $v \in V$, the formulas $\mathit{write}^{t:i}_v$ and $\mathit{read}^{t:i}_v$ are true if the action at $t{:}i$ writes or reads the variable v. Technically, these formulas are obtained from the encoding formulas $\mathit{temp\text{-}write}^i_v$ and $\mathit{temp\text{-}read}^i_v$ (see Section 4.3), respectively, by replacing each encoding variable $\tilde{u}$ by its timed copy $\tilde{u}^t$. To formalize write- and read-freshness, we define for all $1 \le t \le k-1$, all $1 \le i \le K$, and all $v \in V$ the formulas

$$\mathit{writefresh}^{t:i}_v := \left(\bigvee_{j=i+1}^{K} \tilde{f}^{t-1:j} \wedge \mathit{write}^{t-1:j}_v\right) \vee \left(\bigvee_{j=1}^{i-1} \tilde{f}^{t:j} \wedge \mathit{write}^{t:j}_v\right), \text{ and} \tag{27}$$

$$\mathit{readfresh}^{t:i}_v := \left(\bigvee_{j=i+1}^{K} \tilde{f}^{t-1:j} \wedge \mathit{read}^{t-1:j}_v\right) \vee \left(\bigvee_{j=1}^{i-1} \tilde{f}^{t:j} \wedge \mathit{read}^{t:j}_v\right). \tag{28}$$

That is, $\mathit{writefresh}^{t:i}_v$ is true iff v is written to at one or more of the sub-steps $t{-}1{:}i{+}1, \ldots, t{-}1{:}K, t{:}1, \ldots, t{:}i{-}1$. Read-freshness is analogous.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

The following definition describes symbolically the serial process normal form.

Definition 9. For $k \geq 0$, the *serial process execution formula* with bound k is the conjunction of the k-unrolling of the serial $\exists$-step transition formula $\tilde{\Delta}_{\text{ser}}$ and the constraints

$$\tilde{f}^{t:i} \Rightarrow \tilde{f}^{t-1:i} \vee \bigvee_{v \in V} \begin{pmatrix} (write_v^{t:i} \wedge readfresh_v^{t:i}) \vee \\ (write_v^{t:i} \wedge writefresh_v^{t:i}) \vee \\ (read_v^{t:i} \wedge writefresh_v^{t:i}) \end{pmatrix} \tag{29}$$

for all $1 \leq t \leq k-1$ and $1 \leq i \leq K$.

For bounded model checking with the serial process semantics, we take a coherent operational encoding and construct the BMC formula (4) using the serial process execution formula instead of the k-unrolling of a transition formula. Concerning correctness, the approach is obviously sound: the serial $\exists$-step transition formula is sound with respect to the relaxed $\exists$-step transition relation, and because we only add constraints to it, no spurious transitions can be introduced. On the completeness side, the serial process semantics is no longer complete with respect to the interleaving semantics because all unit steps are generally not allowed. However, if a state s' is reachable from a state s, then there is an execution from s to s' in the serial process normal form. Moreover, among those serial $\exists$-step executions from s to s' that have the least number of steps, at least one is in the serial process normal form. Thus, the set of states covered in at most k steps using the serial process semantics and the serial $\exists$-step semantics coincide, and switching to serial process semantics in BMC does not increase the required bound to find a counter-example. These properties are a consequence of the following proposition together with the completeness of the serial $\exists$-step transition formula.

Proposition 9. *Given a coherent encoding, a bound $k' \geq 0$, and an interpretation $\mathcal{I}'$ over $\tilde{V}^0 \cup \tilde{C}^0 \cup \tilde{V}^1 \cup \tilde{C}^1 \cup \cdots \cup \tilde{V}^{k'}$ that satisfies the k'-unrolling of the serial $\exists$-step transition formula $\tilde{\Delta}_{\text{ser}}$, there is a bound $k \leq k'$ and an interpretation $\mathcal{I}$ over $\tilde{V}^0 \cup \tilde{C}^0 \cup \tilde{V}^1 \cup \tilde{C}^1 \cup \cdots \cup \tilde{V}^k$ such that $\mathcal{I}$ satisfies the serial process execution formula with bound k, and for all state variables $v \in V$, $\mathcal{I}(\tilde{v}^0) = \mathcal{I}'(\tilde{v}^0)$ and $\mathcal{I}(\tilde{v}^k) = \mathcal{I}'(\tilde{v}^{k'})$.*

Proof. Assume that $\mathcal{I}'$ is as given. For any $k \geq 0$, let us call an interpretation $\mathcal{I}$ over $\tilde{V}^0 \cup \tilde{C}^0 \cup \tilde{V}^1 \cup \tilde{C}^1 \cup \cdots \cup \tilde{V}^k$ a *k-step interpretation* iff $\mathcal{I}$ satisfies the k-unrolling of $\tilde{\Delta}_{\text{ser}}$ and for all $v \in V$, $\mathcal{I}(\tilde{v}^0) = \mathcal{I}'(\tilde{v}^0)$ and $\mathcal{I}(\tilde{v}^k) = \mathcal{I}'(\tilde{v}^{k'})$. In particular, $\mathcal{I}'$ is a k'-step interpretation. Define the *weight* of a k-step interpretation $\mathcal{I}$ as

$$w(\mathcal{I}) := \sum_{t=0}^{k-1} (K+1)^t |\mathrm{ex}_{\mathcal{I}}^t|,$$

where $\mathrm{ex}_{\mathcal{I}}^t := \left\{ \mathrm{act}^i \;\middle|\; 1 \leq i \leq K \text{ and } \mathcal{I} \text{ satisfies } \tilde{f}^{t:i} \right\}$ is the set of selected actions at step t. The weight function is chosen in such a way that moving action instances to be executed at earlier time steps (bringing the execution closer to serial process normal form) decreases weight.

The range of w consists of non-negative integers and it is nonempty because there is at least one k-step interpretation, namely $\mathcal{I}'$. Thus, there is a minimum value $w_{\min}$ in the range, and we can pick a bound $k \geq 0$ and a k-step interpretation $\mathcal{I}$ such that $w(\mathcal{I}) = w_{\min}$. By the constraint (17) of the serial $\exists$-step transition formula, the set $\mathrm{ex}_{\mathcal{I}}^t$ is never empty, so we see that this choice also minimizes k. In particular, $k \leq k'$.

It only remains to show that $\mathcal{I}$ satisfies the serial process execution formula with bound k. Assume that this is not the case. Thus, we can pick values t and i such that $1 \leq t \leq k-1$ and $1 \leq i \leq K$ and $\mathcal{I}$ satisfies the negation of (29), that is,

$$\tilde{f}^{t:i} \wedge \neg \tilde{f}^{t-1:i} \wedge \bigwedge_{v \in V} \begin{pmatrix} (write_v^{t:i} \Rightarrow \neg readfresh_v^{t:i}) \wedge \\ (write_v^{t:i} \Rightarrow \neg writefresh_v^{t:i}) \wedge \\ (read_v^{t:i} \Rightarrow \neg writefresh_v^{t:i}) \end{pmatrix}. \tag{30}$$

We will show that the execution of act^i at $t{:}i$ in $\mathcal{I}$ can be moved backwards to $t-1{:}i$ without affecting the end state, which reduces the weight and contradicts the choice of a minimum-weight interpretation.

Let us form a sequence of interpretations $\mathcal{I}^{0:0}, \mathcal{I}^{0:1}, \ldots, \mathcal{I}^{0:K}, \mathcal{I}^{1:0}, \mathcal{I}^{1:1}, \ldots, \mathcal{I}^{k-1:K}$ in such a way that for $0 \leq r \leq k-1$ and $1 \leq j \leq K$, the intermediate state at the beginning of sub-step $r{:}j$ is $\mathrm{state}_{\mathcal{I}^{r:j-1}}$, and the intermediate state right after $r{:}j$ is $\mathrm{state}_{\mathcal{I}^{r:j}}$. In particular, $\mathrm{state}_{\mathcal{I}^{r-1:K}}$ and $\mathrm{state}_{\mathcal{I}^{r:0}}$ coincide. First, we fix the values of choice encoding variables according to $\mathcal{I}$:

$$\mathcal{I}^{r:j}(\tilde{a}) := \mathcal{I}(\tilde{a}^r) \quad \text{for } 0 \leq r \leq k-1,\ 1 \leq j \leq K, \text{ and } \tilde{a} \in \tilde{C}.$$

The sequence of states then evolves in the same way as in the serial $\exists$-step transition formula. For $0 \leq r \leq k-1$ and $1 \leq j \leq K$ and $v \in V$,

$$\mathcal{I}^{r:j}(\tilde{v}) := \begin{cases} \mathcal{I}(\tilde{v}^0) & \text{if } t = 0 \text{ and } j = 0, \\ \mathcal{I}^{r-1:K}(\tilde{v}) & \text{if } t > 0 \text{ and } j = 0, \\ \mathcal{I}^{r:j-1}(newvalue_v^j) & \text{if } \mathcal{I}^{r:j-1} \text{ satisfies } \tilde{f}^j \wedge write_v^j, \\ \mathcal{I}^{r:j-1}(\tilde{v}) & \text{otherwise.} \end{cases} \tag{31}$$

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

This corresponds to the execution defined by interpretation $\mathcal{I}$. To capture the modified execution where the action at $t-1{:}i$ takes the role of $t{:}i$, we define a second sequence of interpretations $\mathcal{I}_*^{0:0}, \mathcal{I}_*^{0:1}, \ldots, \mathcal{I}_*^{0:K}, \mathcal{I}_*^{1:0}, \mathcal{I}_*^{1:1}, \ldots, \mathcal{I}_*^{k-1:K}$ as follows. For the choice encoding variables, we set

$$\mathcal{I}_*^{r:j}(\tilde{a}) := \begin{cases} \mathbf{T} & \text{if } r = t-1 \text{ and } \tilde{a} = \tilde{f}^i, \\ \mathcal{I}(\tilde{a}^t) & \text{if } r = t-1 \text{ and } \tilde{a} \in \tilde{A}^i, \\ \mathbf{F} & \text{if } r = t \text{ and } \tilde{a} = \tilde{f}^i, \\ \mathcal{I}(\tilde{a}^r) & \text{otherwise.} \end{cases} \tag{32}$$

Thus, we select act^i at $t-1{:}i$ for execution with the same values for the action-specific choice variables $\tilde{A}^i$ as in step t of the original execution, and deselect the action at $t{:}i$. The evolution is defined like before: for $0 \le r \le k-1$ and $1 \le j \le K$ and $v \in V$,

$$\mathcal{I}_*^{r:j}(\tilde{v}) := \begin{cases} \mathcal{I}(\tilde{v}^0) & \text{if } t = 0 \text{ and } j = 0, \\ \mathcal{I}_*^{r-1:K}(\tilde{v}) & \text{if } t > 0 \text{ and } j = 0, \\ \mathcal{I}_*^{r:j-1}(\mathit{newvalue}_v^j) & \text{if } \mathcal{I}_*^{r:j-1} \text{ satisfies } \tilde{f}^j \wedge \mathit{write}_v^j, \\ \mathcal{I}_*^{r:j-1}(\tilde{v}) & \text{otherwise.} \end{cases} \tag{33}$$

The states defined by the two sequences coincide from $\mathrm{state}_{\mathcal{I}^{0:0}} = \mathrm{state}_{\mathcal{I}_*^{0:0}}$ until $\mathrm{state}_{\mathcal{I}^{t-1:i-1}} = \mathrm{state}_{\mathcal{I}_*^{t-1:i-1}}$. By (30), for any variable $v \in V$ such that $\mathcal{I}$ satisfies $\mathit{read}_v^{t:i}$, or equivalently, $\mathcal{I}^{t:i-1}$ satisfies read_v^i, we have $\mathcal{I}(\mathit{writefresh}_v^{t:i}) = \mathbf{F}$, or equivalently, $\mathcal{I}^{t-1:j-1}(\tilde{f}^j \wedge \mathit{write}_v^j) = \mathbf{F}$ for $j = i+1, \ldots, K$ and $\mathcal{I}^{t:j-1}(\tilde{f}^j \wedge \mathit{write}_v^j) = \mathbf{F}$ for $j = 1, \ldots, i-1$. Since $\mathcal{I}^{t-1:i-1}(\tilde{f}^i)$ is known to be false from (30), we see from (31) that the value of v does not change when moving from $\mathrm{state}_{\mathcal{I}^{t-1:i-1}}$ to $\mathrm{state}_{\mathcal{I}^{t:i-1}}$. Thus, when v is such that $\mathcal{I}^{t:i-1}$ satisfies read_v^i, $\mathcal{I}^{t-1:i-1}(\tilde{v}) = \mathcal{I}_*^{t-1:i-1}(\tilde{v}) = \mathcal{I}^{t:i-1}(\tilde{v})$. Furthermore, $\mathcal{I}_*^{t-1:i-1}$ coincides with $\mathcal{I}^{t:i-1}$ on $\tilde{A}^i$ by (32). This means that $\mathcal{I}_*^{t-1:i-1}$ i-conforms to $\mathcal{I}^{t:i-1}$, and we can exploit the coherence of the encoding. Let W be the set of variables $v \in V$ such that $\mathcal{I}^{t:i-1}$ satisfies write_v^i. Definition 6 now tells us that $\mathcal{I}_*^{t-1:i-1}$ satisfies $\mathit{enabled}^i$, and $\mathcal{I}_*^{t-1:i-1}$ satisfies write_v^i iff $v \in W$, and for $v \in W$, $\mathcal{I}_*^{t-1:i-1}(\mathit{newvalue}_v^i) = \mathcal{I}^{t:i-1}(\mathit{newvalue}_v^i)$. Altogether, we get the following formulation for the intermediate state after the sub-step $t-1{:}i$. For all $v \in V$,

$$\mathcal{I}_*^{t-1:i}(\tilde{v}) = \begin{cases} \mathcal{I}^{t:i}(\tilde{v}) & \text{if } v \in W, \\ \mathcal{I}^{t-1:i}(\tilde{v}) & \text{otherwise.} \end{cases}$$

By (30), every $v \in W$ is both read-fresh and write-fresh at the sub-step $t{:}i$ is executed in $\mathcal{I}$. In particular, no variable $v \in W$ is read nor written at $t-1{:}i+1$. Therefore, $\mathcal{I}_*^{t-1:i}(v)$ $i+1$-conforms to $\mathcal{I}^{t-1:i}(v)$, and we can use the coherence of the encoding to conclude that $\mathcal{I}_*^{t-1:i+1}$ coincides with $\mathcal{I}^{t:i}$ on W and with $\mathcal{I}^{t-1:i+1}$ on $V \setminus W$. Repeating this reasoning for all $K-1$ sub-steps between $t-1{:}i$ and $t{:}i$, we get for all $v \in V$

$$\mathcal{I}_*^{t:i-1}(\tilde{v}) = \begin{cases} \mathcal{I}^{t:i}(\tilde{v}) & \text{if } v \in W, \\ \mathcal{I}^{t:i-1}(\tilde{v}) & \text{otherwise.} \end{cases}$$

The right-hand side is equal to $\mathcal{I}^{t:i}(\tilde{v})$ because of the way W is defined, and the left-hand side is equal to $\mathcal{I}_*^{t:i}(\tilde{v})$ by (33) as $\mathcal{I}_*^{t:i-1}$ does not satisfy $\tilde{f}^i$. In other words, $\mathrm{state}_{\mathcal{I}_*^{t:i}} = \mathrm{state}_{\mathcal{I}^{t:i}}$, and because $\mathcal{I}_*^{r:j}$ and $\mathcal{I}^{r:j}$ give the same interpretations to all choice encoding variables from this state onwards, also $\mathrm{state}_{\mathcal{I}_*^{k-1:K}} = \mathrm{state}_{\mathcal{I}^{k-1:K}}$.

We can now collect the information from $\mathcal{I}_*^{r:j}$ to an interpretation $\mathcal{I}_*$ over $\tilde{V}^0 \cup \tilde{C}^0 \cup \tilde{V}^1 \cup \tilde{C}^1 \cup \cdots \cup \tilde{V}^k$ that describes an execution which, according to the reasoning above, only differs from the execution described by $\mathcal{I}$ on the intermediate states between the sub-steps $t-1{:}i$ and $t{:}i$. Define

$$\begin{aligned} &\mathcal{I}_*(\tilde{v}^r) := \mathcal{I}_*^{r:0}(\tilde{v}) && \text{for } 0 \le r \le k-1 \text{ and } v \in V, \\ &\mathcal{I}_*(\tilde{v}^k) := \mathcal{I}_*^{k-1:K}(\tilde{v}) && \text{for } v \in V, \text{ and} \\ &\mathcal{I}_*(\tilde{a}^r) := \mathcal{I}_*^{r:0}(\tilde{a}) && \text{for } \tilde{a} \in \tilde{C}. \end{aligned}$$

It follows that $\mathcal{I}_*$ satisfies the k-unrolling of the serial $\exists$-step transition formula, except possibly the constraint (17) if the time step t was made empty by deselecting act^i at $t{:}i$. If this is not the case, then $\mathcal{I}_*$ is a k-step interpretation, and by (32), its weight is $w(\mathcal{I}_*) = w(\mathcal{I}) - (K+1)^t + (K+1)^{t-1} < w(\mathcal{I})$, a contradiction with the choice of $\mathcal{I}$ as the minimum-weight k-unrolling. If $\mathcal{I}_*$ selects no actions for the time step t, we can construct an interpretation $\mathcal{I}_{**}$ from $\mathcal{I}_*$ by dropping the empty step t from the middle. Then, $\mathcal{I}_{**}$ is a $k-1$-step interpretation with $w(\mathcal{I}_{**}) < w(\mathcal{I}_*) < w(\mathcal{I})$, again a contradiction.

Thus, we have shown that the k-step interpretation $\mathcal{I}$ satisfies the serial process execution formula with bound k, which concludes the proof. □

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

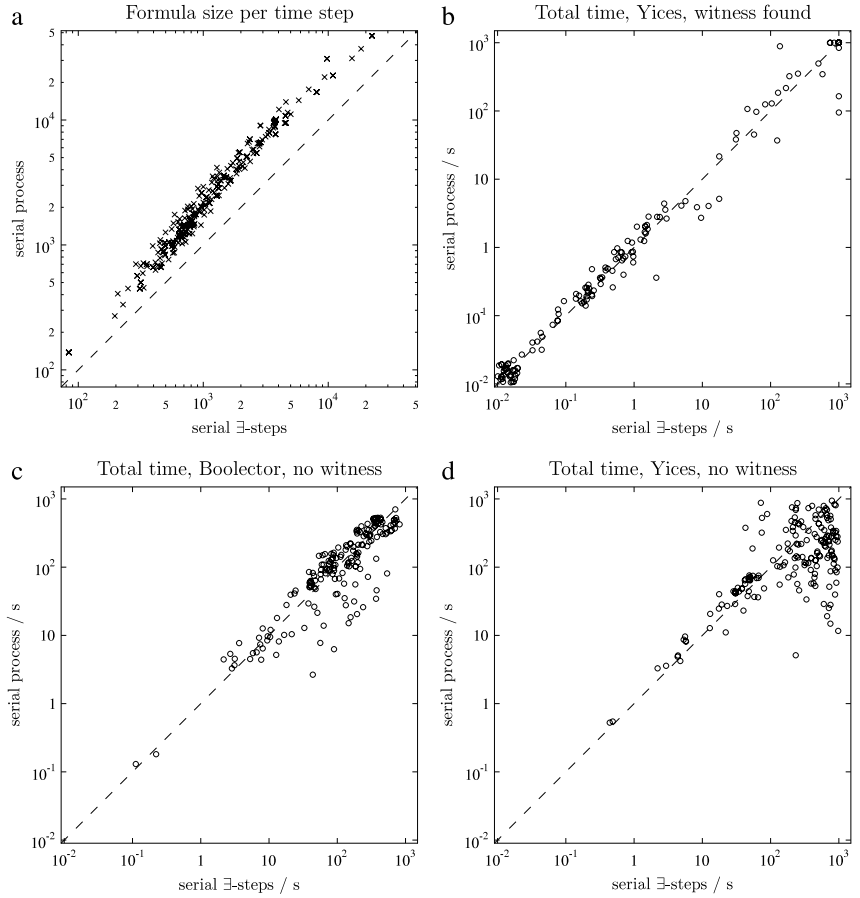


Fig. 11. Results with serial process vs. serial ∃-step semantics.

6.2.1. Size of the serial process execution formula

From the definition (27) of $writefresh_v^{t:i}$, we see that the left disjunct contains the left disjunct of $writefresh_v^{t:i+1}$ as a subformula, and the right disjunct contains the right disjunct of $writefresh_v^{t:i-1}$. We share these parts and use constant folding to eliminate all disjuncts $\tilde{f}^{t:j} \wedge write_v^{t:j}$ when the encoding formula $write_v^j$ is the constant **F**. For a fixed time step t and a variable v, the formulas $writefresh_v^{t:1}$ through $writefresh_v^{t:K}$ can thus be expressed in size proportional to the sum of the sizes of the encoding formulas $write_v^1, \ldots, write_v^K$. Using the same reasoning for $readfresh_v^{t:i}$, we see that the serial process constraints (29) for a given time step t are linear in the encoding size. For the first step $t = 0$, there are no constraints. Recalling the linear size of the serial ∃-step transition formula, we conclude that the size of the serial process execution formula is linear in the bound k times the total size of the encoding expressions $enabled^i$, $write_v^i$, $newvalue_v^i$, and $read_v^i$, excluding $write_v^i$ and $newvalue_v^i$ when $write_v^i$ is the constant **F**, and $read_v^i$ when it is **F**.

6.3. Experiments with serial process semantics

To assess the effect of process semantics in practice, we continue with the BEEM benchmark set and the experimental setup of Section 5. We measure BMC run time with the serial process execution formula, using the serial ∃-step transition formula as a baseline. For action ordering, we use the unmodified "input" order like in Section 5.2. First, the formula sizes of the two approaches are compared in Fig. 11(a). The horizontal axis shows the size of the serial ∃-step transition formula for each benchmark instance, and the vertical axis shows the size of the same formula conjuncted with the serial process constraints for a single time step. Serial process semantics clearly enlarges the formula, but only by a roughly constant

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

factor, which is 2.1 on average. For those benchmark instances for which a witness to the reachability property is found, the cumulative CPU times with the Yices solver are plotted in Fig. 11(b). There does not seem to be benefit from using the serial process execution formula in place of the unrolled serial ∃-step transition formula.

In the cases where a witness is not found and only unsatisfiable formulas are fed to the SMT solver until timeout, the situation is somewhat different. Because in these cases the two solvers Boolector and Yices exhibit noticeably different behavior, we show their run times in separate plots in Figs. 11(c) and (d), respectively. Each marker shows the cumulative solver CPU time for a single benchmark instance from bound 0 till the largest bound that was covered by all of the tested algorithms before the 1000 s timeout. This is an apples-to-apples comparison because with the same maximum bound k, the serial ∃-steps and the serial processes cover exactly the same states. The plots show a lot of variation in the relative speed difference, especially with the Yices solver, slightly to the favor of the serial process formula. This indicates that although the serial process semantics did not help finding witnesses in our benchmarks, the extra constraints in some cases help the solver with hard unsatisfiable BMC instances.

7. Conclusions

In this paper, we have shown how to substantially speed up bounded model checking of asynchronous systems by exploiting several partial order semantics, namely the parallel ∃-step, the serial ∃-step, and the serial process semantics. In the case of all these three semantics, the paper extends the previous state-of-the-art by allowing the asynchronous systems to perform non-trivial data manipulation. In order not to restrict ourselves to a single modeling language, we have devised novel, abstract concepts of operational and coherent encodings of actions. They capture the essential information of enabledness as well as which state variables are read and written when an action is executed. These encodings allow us to express the transition formulas for the interleaving, parallel ∃-step, serial ∃-step, and serial process semantics in a surprisingly concise and elegant manner. In particular, the sizes of the transition formulas with the ∃-step and process semantics are, by a careful linear construction, essentially the same as the size with the interleaving semantics; in this sense our step and process semantics are a "free lunch".

Although the encodings are expressed in an abstract level, they can be concretized with a reasonable effort for a real modeling language, as shown for the case of the DVE language in this paper. Furthermore, the extensive experimental results also conducted in the paper indicate that the step and process semantics do indeed very substantially speed up bounded model checking of asynchronous systems in the case the system violates an invariant property under verification. This speedup is explained by the need to unroll the transition formulas fewer times than with the interleaving semantics; as this unrolling constitutes the major computational challenge for the satisfiability solvers applied in bounded model checking, less unrolling (without increasing the transition formula size too much) usually leads to reduction in solving time. For systems where the invariant property holds, the semantics do not directly offer a better way of making bounded model checking a complete method (i.e., capable of also deciding that the property actually holds). However, running a bounded model checking tool for a fixed time with the step or process semantics instead of the interleaving semantics provides the user a higher confidence that the system respects the property, because the state space is in general covered faster.

Concerning future work, there are several interesting research directions.

First of all, in this work we have concentrated on verifying invariant properties. To verify more complex temporal logic properties, the approach here should be extended to model checking of stuttering invariant properties expressible in the temporal logic LTL-X. For the ∀-step semantics this has been done in [44] but the approach there needs to be extended to the ∃-step and process semantics used in this work.

Second, the properties of the serial process normal form introduced in Section 6 need additional formal investigation. In particular, the normal form should be analyzed to find out whether Mazurkiewicz trace equivalent executions always have the same normal form, and if not, how the normal form could be modified to ensure canonicity. In addition, one could study how the process execution constraints could be modified to work on top of the parallel ∃-step semantics; however, we conjecture that the resulting parallel process execution formulas do not perform as well as the serial process execution formulas simply because of the performance properties of the underlying ∃-step semantics formulas.

Making bounded model checking and other SAT-based model checking techniques complete, i.e. also capable of deciding that the desired property holds, is a challenging and ongoing research area. One such technique, proposed by McMillan [45], is to apply Craig interpolants to approximate reachable state sets. It would be interesting to see how using the step and process semantics (that allow larger sets of states to be reached within fewer steps) instead of the interleaving semantics would influence the performance of the interpolant technique. Such experimentation would call for better solver support for computing interpolants of bit-vector formulas.

Acknowledgements

The authors are grateful to the three reviewers of this paper for their helpful and well-thought remarks and suggestions.

References

[1] E.M. Clarke Jr., O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 1999.
[2] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (2) (1992) 142–170.
[4] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: Proc. TACAS 1999, in: LNCS, vol. 1579, Springer, 1999, pp. 193–207.
[5] J. Marques-Silva, I. Lynce, S. Malik, CDCL solvers, in: Handbook of Satisfiability, IOS Press, 2009, pp. 131–153.
[6] C. Barrett, R. Sebastiani, S.A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability, IOS Press, 2009, pp. 825–885.
[7] A. Biere, K. Heljanko, T. Junttila, T. Latvala, V. Schuppan, Linear encodings of bounded LTL model checking, Logical Methods in Computer Science 2 (5:5) (2006) 1–64.
[8] G.J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[9] J. Barnat, L. Brim, P. Ročkai, DiVinE 2.0: high-performance model checking, in: Proc. HiBi 2009, IEEE Computer Society Press, 2009, pp. 31–32.
[10] A. Valmari, The state explosion problem, in: Lectures on Petri Nets I: Basic Models, in: LNCS, vol. 1491, Springer, 1998, pp. 429–528.
[11] E. Best, R.R. Devillers, Sequential and concurrent behaviour in Petri net theory, Theoretical Computer Science 55 (1) (1987) 87–136.
[12] K. Heljanko, Bounded reachability checking with process semantics, in: Proc. CONCUR 2001, in: LNCS, vol. 2154, Springer, 2001, pp. 218–232.
[13] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artificial Intelligence 170 (12–13) (2006) 1031–1080.
[14] H.A. Kautz, B. Selman, Pushing the envelope: planning, propositional logic and stochastic search, in: Proc. AAAI'96/IAAI'96, vol. 2, AAAI Press, 1996, pp. 1194–1201.
[15] J. Malinowski, P. Niebert, SAT based bounded model checking with partial order semantics for timed automata, in: J. Esparza, R. Majumdar (Eds.), TACAS, in: Lecture Notes in Computer Science, vol. 6015, Springer, 2010, pp. 405–419.
[16] Y. Dimopoulos, B. Nebel, J. Koehler, Encoding planning problems in nonmonotonic logic programs, in: Proc. ECP 1997, in: LNCS, vol. 1348, Springer, 1997, pp. 169–181.
[17] M. Wehrle, J. Rintanen, Planning as satisfiability with relaxed ∃-step plans, in: AI 2007: Advances in Artificial Intelligence, in: LNCS, vol. 4830, Springer, 2007, pp. 244–253.
[18] S. Ogata, T. Tsuchiya, T. Kikuno, SAT-based verification of safe Petri nets, in: Proc. ATVA 2004, in: LNCS, vol. 3299, Springer, 2004, pp. 79–92.
[19] T. Jussila, BMC via dynamic atomicity analysis, in: Proc. ACSD 2004, IEEE Computer Society, 2004, pp. 197–206.
[20] T. Jussila, K. Heljanko, I. Niemelä, BMC via on-the-fly determinization, International Journal on Software Tools for Technology Transfer 7 (2) (2005) 89–101.
[21] T. Jussila, On bounded model checking of asynchronous systems, Research Report A97, Helsinki University of Technology, Laboratory for Theoretical Computer Science, doctoral dissertation, 2005.
[22] J. Dubrovin, T. Junttila, K. Heljanko, Symbolic step encodings for object based communicating state machines, in: Proc. FMOODS 2008, in: LNCS, vol. 5051, Springer, 2008, pp. 96–112.
[23] C. Wang, Z. Yang, V. Kahlon, A. Gupta, Peephole partial order reduction, in: Proc. TACAS 2008, in: LNCS, vol. 4963, Springer, 2008, pp. 382–396.
[24] V. Kahlon, C. Wang, A. Gupta, Monotonic partial order reduction: an optimal symbolic partial order reduction technique, in: Proc. CAV 2009, in: LNCS, vol. 5643, Springer, 2009, pp. 398–413.
[25] S. Burckhardt, R. Alur, M.M.K. Martin, CheckFence: checking consistency of concurrent data types on relaxed memory models, in: Proc. PLDI 2007, ACM, 2007, pp. 12–21.
[26] J. Dubrovin, Checking bounded reachability in asynchronous systems by symbolic event tracing, in: Proc. VMCAI 2010, in: LNCS, vol. 5944, Springer, 2010, pp. 146–162.
[27] K.L. McMillan, Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in: Proc. CAV 1992, in: LNCS, vol. 663, Springer, 1993, pp. 164–177.
[28] J. Esparza, K. Heljanko, Unfoldings — a partial-order approach to model checking, in: EATCS Monographs in Theoretical Computer Science, Springer-Verlag, 2008.
[29] S. Ranise, C. Tinelli, The SMT-LIB standard: Version 1.2, 2006.
[30] T. Bultan, R. Gerber, W. Pugh, Symbolic model checking of infinite state systems using Presburger arithmetic, in: Proc. CAV 1997, in: LNCS, vol. 1254, Springer, 1997, pp. 400–411.
[31] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: Proc. CAV 1997, in: LNCS, vol. 1254, Springer, 1997, pp. 72–83.
[32] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar, Symbolic model checking with rich assertional languages, Theoretical Computer Science 256 (1–2) (2001) 93–112.
[33] T. Rybina, A. Voronkov, A logical reconstruction of reachability, in: Proc. PSI 2003, in: LNCS, vol. 2890, Springer, 2003, pp. 222–237.
[34] P.A. Abdulla, G. Delzanno, A. Rezine, Parameterized verification of infinite-state processes with global conditions, in: Proc. CAV 2007, in: LNCS, vol. 4590, Springer, 2007, pp. 145–157.
[35] A. Bouajjani, P. Habermehl, Y. Jurski, M. Sighireanu, Rewriting systems with data, in: Proc. FCT 2007, in: LNCS, vol. 4639, Springer, 2007, pp. 1–22.
[36] S. Ghilardi, E. Nicolini, S. Ranise, D. Zucchelli, Towards SMT model checking of array-based systems, in: Proc. IJCAR 2008, in: LNCS, vol. 5195, Springer, 2008, pp. 67–82.
[37] R. Pelánek, BEEM: benchmarks for explicit model checkers, in: Proc. SPIN 2007, in: LNCS, vol. 4595, Springer, 2007, pp. 263–267.
[38] T. Junttila, J. Dubrovin, Encoding queues in satisfiability modulo theories based bounded model checking, in: Proc. LPAR 2008, in: LNCS, vol. 5330, Springer, 2008, pp. 290–304.
[39] SRI International, Yices 2.0 prototype, software, 2009. URL: http://yices.csl.sri.com/download-yices2.shtml.
[40] R. Brummayer, A. Biere, Boolector: an efficient SMT solver for bit-vectors and arrays, in: TACAS, in: LNCS, vol. 5505, Springer, 2009, pp. 174–177.
[41] N. Eén, N. Sörensson, Temporal induction by incremental SAT solving, Electronic Notes in Theoretical Computer Science 89 (4) (2003) 543–560.
[42] D.H. Younger, Minimum feedback arc sets for a directed graph, IEEE Transactions on Circuit Theory 10 (2) (1963) 238–245.
[43] V. Diekert, Y. Métivier, Partial Commutation and Traces, in: Handbook of Formal Languages, vol. 3, Springer, Berlin, 1997, pp. 457–534.
[44] K. Heljanko, I. Niemelä, Bounded LTL model checking with stable models, Theory and Practice of Logic Programming 3 (4–5) (2003) 519–550.
[45] K.L. McMillan, Interpolation and SAT-based model checking, in: Proc. CAV 2003, in: LNCS, vol. 2725, Springer, 2003, pp. 1–13.

Please cite this article in press as: J. Dubrovin, et al., Exploiting step semantics for efficient bounded model checking of asynchronous systems, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.07.005