

# Analysis of Recursive State Machines

RAJEEV ALUR

University of Pennsylvania

MICHAEL BENEDIKT

Bell Laboratories

KOUSHA ETESSAMI

University of Edinburgh

PATRICE GODEFROID

Bell Laboratories

THOMAS REPS

University of Wisconsin

and

MIHALIS YANNAKAKIS

Columbia University

---

Recursive state machines (RSMs) enhance the power of ordinary state machines by allowing vertices to correspond either to ordinary states or to potentially recursive invocations of other state machines. RSMs can model the control flow in sequential imperative programs containing recursive procedure calls. They can be viewed as a visual notation extending Statecharts-like hierarchical state machines, where concurrency is disallowed but recursion is allowed. They are also related to various models of pushdown systems studied in the verification and program analysis communities.

After introducing RSMs and comparing their expressiveness with other models, we focus on whether verification can be efficiently performed for RSMs. Our first goal is to examine the

---

This research was partially supported by NSF Grants CCR-9619219, CCR97-34115, CCR99-70925, CCR-0341658, CCR-9986308, ONR contract N00014-01-1-0796, the A. von Humboldt Foundation, SRC award 99-TJ-688, and a Sloan Faculty Fellowship.

Authors' addresses: R. Alur, Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104; email: alur@cis.upenn.edu; M. Benedikt, Bell Laboratories, Lucent Technologies, 2701 Lucent Lane, Lisle, IL 60532; email: benedikt@bell-labs.com; K. Etessami, Laboratory for Foundations of Computer Science, University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK; email: kousha@inf.ed.ac.uk; P. Godefroid, Bell Laboratories, Lucent Technologies, 2701 Lucent Lane, Lisle, IL 60532; email: god@bell-labs.com; T. Reps, Computer Science Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706-1685; email: reps@cs.wisc.edu; M. Yannakakis, Department of Computer Science, Columbia University, 455 Computer Science Building, 1214 Amsterdam Avenue, Mail Code 0401, New York, NY 10027; email: mihali@cs.columbia.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2005 ACM 0164-0925/05/0700-0786 \$5.00

verification of linear time properties of RSMs. We begin this study by dealing with two key components for algorithmic analysis and model checking, namely, reachability (Is a target state reachable from initial states?) and cycle detection (Is there a reachable cycle containing an accepting state?). We show that both these problems can be solved in time  $O(n\theta^2)$  and space  $O(n\theta)$ , where  $n$  is the size of the recursive machine and  $\theta$  is the maximum, over all component state machines, of the minimum of the number of entries and the number of exits of each component. From this, we easily derive algorithms for linear time temporal logic model checking with the same complexity in the model. We then turn to properties in the branching time logic CTL\*, and again demonstrate a bound linear in the size of the state machine, but only for the case of RSMs with a single exit node.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Software verification, recursive state machines, pushdown automata, context-free languages, model checking, temporal logic, program analysis

## 1. INTRODUCTION

In traditional model checking, the model is a finite state machine whose vertices correspond to system states and whose edges correspond to system transitions. In this article we consider the analysis of *recursive state machines* (RSMs), in which vertices can either be ordinary states or can correspond to invocations of other state machines in a potentially recursive manner. RSMs can model control flow in typical sequential imperative programming languages with recursive procedure calls. Alternatively, RSMs can be viewed as a variant of visual notations for hierarchical state machines, such as Statecharts [Harel 1987] and UML [Booch et al. 1997], where concurrency is disallowed but recursion is allowed.

More precisely, a recursive state machine consists of a set of component machines. Each component has a set of *nodes* (atomic states) and *boxes* (each of which is mapped to a component), a well-defined interface consisting of *entry* and *exit* nodes, and edges connecting nodes/boxes. An edge entering a box models the invocation of the component associated with the box, and an edge leaving a box corresponds to a return from that component. Due to recursion, the underlying global state-space is infinite and behaves like a *pushdown* system. While RSMs are closely related to pushdown systems, which are studied in verification and program analysis in many disguises [Reps et al. 1995; Bouajjani et al. 1997], RSMs appear to be the appropriate definition for visual modeling and allow tighter analysis. The relationship between RSMs and these other models is studied in detail in Section 3.

The goal of this article is to study verification problems for RSMs. The two most fundamental questions for model checking of safety and liveness properties, respectively are (1) *reachability*: Given sets of initial and target nodes, is some target node reachable from an initial one? and (2) *cycle detection*: Given sets of initial and target nodes, is there a cycle containing a target node reachable from an initial node? This analysis also forms the basis for model checking more complex properties in linear-time temporal logic. For cycle

detection, there are two natural variants depending on whether or not one requires the recursion depth to be bounded in infinite computations. We show that all these problems can be solved in time  $O(n\theta^2)$ , where  $n$  is the size of the RSM, and  $\theta$  is a parameter depending only on the number of entries and exits in each component. The number of entry points corresponds to the parameters passed to a component, while the number of exit points corresponds to the values returned. More precisely, for each component  $A_i$ , let  $d_i$  be the minimum of the number of entries and the number of exits of that component. Then  $\theta = \max_i(d_i)$ . Thus, if every component has either a “small” number of entry points, or a “small” number of exit points, then  $\theta$  will be “small.” The space complexity of the algorithms is  $O(n\theta)$ .

The first, and key, computational step in the analysis of RSMs involves determining reachability relationships among entry and exit points of each component. We show how the information required for this computation can be encoded as recursive Datalog-like rules of a special form. To enable efficient analysis, our rules will capture *forward* reachability from entry points for components with a small number of entries, and *backward* reachability from exit points for the other components. The solution to the rules can then be reduced to alternating reachability for AND-OR (game) graphs. In the second step of our algorithm, we reduce the problems of reachability and cycle detection with bounded/unbounded recursion depth to traditional graph-theoretic analysis on appropriately constructed graphs based on the information computed in the first step.

Our algorithms for cycle detection lead immediately to algorithms for model checking linear-time requirements expressed as LTL formulas or Büchi automata, via a product construction for Büchi automata with RSMs.

The above analysis, found in Section 4, completes the picture for verification of linear-time properties of RSMs. In Section 5 we extend this analysis to branching time, focusing on the standard branching-time logic  $CTL^*$ . Known bounds on the model checking of pushdown processes imply that we cannot get algorithms that are linear in the size of the machine for branching-time (see Section 3). However, we show that the data complexity is linear for single-exit RSMs. These results generalize bounds in Alur and Yannakakis [2001] for the restricted case of RSMs whose call graph is acyclic, while improving the bounds that can be derived from the literature on pushdown and context-free processes. In particular, our results show that  $CTL^*$  model checking for context-free processes can be done in linear time, in the size of the model.

**Organization.** In Section 2 we give the formal definition of RSMs. In Section 3 we compare their expressiveness to existing models. Section 4 investigates state-space analysis and the verification of linear time properties, while Section 5 deals with branching time. Section 6 summarizes and discusses open issues.

**Related work.** Our definition of recursive state machines naturally generalizes the definition of hierarchical state machines of Alur and Yannakakis [2001]. For hierarchical state machines, the underlying state-space is guaranteed to be finite, but can be exponential in the size of the original machine. Algorithms for

analysis of hierarchical state machines [Alur and Yannakakis 2001] are adaptations of traditional search algorithms to avoid searching the same component repeatedly, and have the same time complexity as the algorithms of this article. However, the “bottom-up” algorithms used in [Alur and Yannakakis 2001] for hierarchical machines cannot be applied to RSMs.

RSMs are closely related to *pushdown systems*. Model checking of pushdown systems has been studied extensively for both linear and branching time requirements [Bouajjani et al. 1997; Burkart and Steffen 1999; Esparza et al. 2000; Finkel et al. 1997]. These algorithms are based on an automata-theoretic approach. Each configuration is viewed as a string over stack symbols, and the reachable configurations are shown to be a regular set that can be computed by a fixpoint computation. Esparza et al [2000] do a careful analysis of the time and space requirements for various problems, including reachability and cycle detection. The resulting worst-case complexity is cubic, and thus, matches our worst case when  $\theta = O(n)$ . Their approach also leads, under more refined analysis, to the bound  $O(nk^2)$  Esparza et al. [2000], where  $n$  is the size of the pushdown system and  $k$  is its number of *control states*. We will see that the number of control states of a pushdown system is related to the number of exit nodes in RSMs, but that by working with RSMs directly we can achieve better bounds in terms of  $\theta$ .

Ball and Rajamani [2000] consider the model of Boolean programs, which can be viewed as RSMs extended with Boolean variables. They have implemented a BDD-based symbolic model checker that solves the reachability problem for Boolean programs. The main technique is to compute the *summary* of the input-output relations of a procedure. This in turn is based on algorithms for interprocedural dataflow analysis [Reps et al. 1995], which are generally cubic. As described in Section 6, when translating Boolean programs to RSMs, one must pay the standard exponential price to account for different combinations of values of the variables, but the price of analysis need not be cubic in the expanded state-space by making a careful distinction between local, read-global, and write-global variables.

Also worth mentioning is some early work in the 1970s within the natural language processing community [Woods 1970] where Woods and his associates considered “recursive transition networks” for modeling the grammatical structure of natural languages. These models are closely related to single-exit RSMs, although they were presented in a somewhat different way.

In the context of this rich history of research, the current article has five main contributions. First, while equivalent to pushdown systems and Boolean programs in theory, recursive state machines are a more direct, visual, state-based model of recursive control flow. Second, we give algorithms with time and space bounds of  $O(n\theta^2)$  and  $O(n\theta)$ , respectively, and thus our solution for analysis is more efficient than the generally cubic algorithms for related models, even when these were geared specifically to solve flow problems in control graphs of sequential programs. Third, our algorithmic technique for both reachability analysis and cycle detection, which combines a mutually dependent forward and backward reachability analyses using a natural Datalog formulation and AND-OR graph accessibility, along with the analysis of an augmented ordinary

graph, is new and potentially useful for solving related problems in program analysis to mitigate similar cubic bottlenecks. We also anticipate that it is more suitable for on-the-fly model checking and early error detection than the prior automata-theoretic solutions for analysis of pushdown systems. Fourth, we give results in the branching time setting that have no counterpart in the prior literature—indeed, using the translations in Section 3, a new bound on model checking of CTL\* formulas for context-free processes can be derived. Finally, using our RSM model, one is able, at no extra cost in complexity, to distinguish between infinite accepting executions that require a “bounded call stack” or “unbounded call stack.” This distinction had not been considered in all previous papers.

Preliminary versions of this material appeared independently in Alur et al. [2001] and Benedikt et al. [2001]. Since the publication of these conference papers a number of subsequent papers have built upon and extended our work, including Alur et al. [2003a,b], Etessami [2004], Alur et al. [2004].

## 2. RECURSIVE STATE MACHINES

**Syntax.** A *recursive state machine (RSM)*  $A$  over a finite alphabet  $\Sigma$  is given by a tuple  $\langle A_1, \dots, A_k \rangle$ , where each *component state machine*  $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$  consists of the following pieces:

- A set  $N_i$  of *nodes* and a (disjoint) set  $B_i$  of *boxes*.
- A labeling  $Y_i : B_i \mapsto \{1, \dots, k\}$  that assigns to every box an index of one of the component machines,  $A_1, \dots, A_k$ .
- A set of *entry nodes*  $En_i \subseteq N_i$ , and a set of *exit nodes*  $Ex_i \subseteq N_i$ .
- A transition relation  $\delta_i$ , where transitions are of the form  $(u, \sigma, v)$  where (1) the source  $u$  is either a node of  $N_i$ , or a pair  $(b, x)$ , where  $b$  is a box in  $B_i$  and  $x$  is an exit node in  $Ex_j$  for  $j = Y_i(b)$ ; (2) the label  $\sigma$  is in  $\Sigma$ ; and (3) the destination  $v$  is either a node in  $N_i$  or a pair  $(b, e)$ , where  $b$  is a box in  $B_i$  and  $e$  is an entry node in  $En_j$  for  $j = Y_i(b)$ .

We will often (outside of Section 3) assume further that for every  $u, v$  there is at most one  $\sigma$  such that  $\delta_i(u, \sigma, v)$ . This assumption makes for some simplifications in the notation, but is not critical to the results in the article.

We will use the term *ports* to refer collectively to pairs consisting of a box  $b$  of a machine  $A_i$  and corresponding entry and exit nodes of the machine  $A_j$  called by  $b$ . We will use the term *vertices* of  $A_i$  to refer to its nodes and the ports of its boxes that participate in some transition. That is, the transition relation  $\delta_i$  is a set of labelled directed edges on the set  $V_i$  of vertices of the machine  $A_i$ . We let  $E_i$  be the set of underlying edges of  $\delta_i$ , ignoring labels, and for  $(u, v) \in E_i$  we let  $\Sigma(u, v)$  be the unique  $\sigma$  such that  $\delta_i(u, \sigma, v)$ . We will often refer to a vertex  $(b, e)$  as a *call vertex* and  $(b, x)$  as a *return vertex*.

Figure 1 illustrates the definition. The sample RSM has three components. The component  $A_1$  has 4 nodes, of which  $u1$  and  $u2$  are entry nodes and  $u4$  is the exit node, and two boxes, of which  $b1$  is mapped to component  $A_2$  and  $b2$  is mapped to  $A_3$ . The entry and exit nodes are the control interface of a component by which it can communicate with the other components. Intuitively, think of

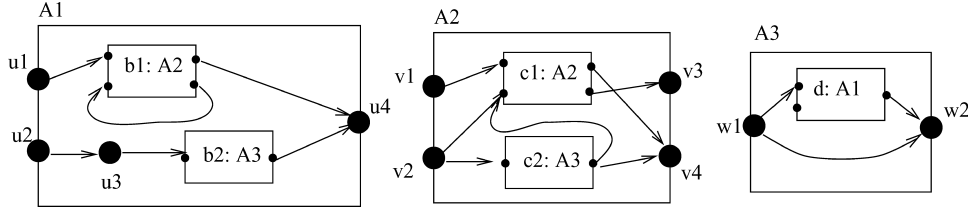


Fig. 1. A sample recursive state machine.

component state machines as procedures, and an edge entering a box at a given entry as invoking the procedure associated with the box with given argument values. Entry-nodes are analogous to arguments while exit-nodes model values returned.

**Semantics.** To define the executions of RSMs, we first define the global states and transitions associated with an RSM. A (global) *state* of an RSM  $A = \langle A_1, \dots, A_k \rangle$  is a tuple  $\langle b_1, \dots, b_r, u \rangle$  where  $b_1, \dots, b_r$  are boxes and  $u$  is a node. Equivalently, a state can be viewed as a string, and the set  $Q$  of global states of  $A$  is  $B^*N$ , where  $B = \cup_i B_i$  and  $N = \cup_i N_i$ . Consider a state  $\langle b_1, \dots, b_r, u \rangle$  such that  $b_i \in B_{j_i}$  for  $1 \leq i \leq r$  and  $u \in N_j$ . Such a state is *well-formed* if  $Y_{j_i}(b_i) = j_{i+1}$  for  $1 \leq i < r$  and  $Y_{j_r}(b_r) = j$ . A well-formed state of this form corresponds to the case when the control is inside the component  $A_j$ , which was entered via box  $b_r$  of component  $A_{j_r}$  (the box  $b_{r-1}$  gives the context in which  $A_{j_r}$  was entered, and so on). Henceforth, we assume states to be well-formed.

We define a (global) transition relation  $\delta$ . Let  $s = \langle b_1, \dots, b_r, u \rangle$  be a state with  $u \in N_j$  and  $b_r \in B_m$ . Then,  $(s, \sigma, s') \in \delta$  if and only if one of the following holds:

1.  $(u, \sigma, u') \in \delta_j$  for a node  $u'$  of  $A_j$ , and  $s' = \langle b_1, \dots, b_r, u' \rangle$ .
2.  $(u, \sigma, (b', e)) \in \delta_j$  for a box  $b'$  of  $A_j$ , and  $s' = \langle b_1, \dots, b_r, b', e \rangle$ .
3.  $u$  is an exit-node of  $A_j$ ,  $((b_r, u), \sigma, u') \in \delta_m$  for a node  $u'$  of  $A_m$ , and  $s' = \langle b_1, \dots, b_{r-1}, u' \rangle$ .
4.  $u$  is an exit-node of  $A_j$ ,  $((b_r, u), \sigma, (b', e)) \in \delta_m$  for a box  $b'$  of  $A_m$ , and  $s' = \langle b_1, \dots, b_{r-1}, b', e \rangle$ .

Case 1 is when the control stays within the component  $A_j$ , case 2 is when a new component is entered via a box of  $A_j$ , case 3 is when the control exits  $A_j$  and returns back to  $A_m$ , and case 4 is when the control exits  $A_j$  and enters a new component via a box of  $A_m$ . The *Labeled Transition System (LTS)*  $T_A = (Q, \Sigma, \delta)$  is called the “*unfolding*” of  $A$ .

When comparing expressiveness of labeled transition systems, we make use of the standard notion of *bisimulation*. A bisimulation between LTS  $T_1$  and  $T_2$  with the same alphabet  $\Sigma$  is a binary relation  $B$  relating states of  $T_1$  and states of  $T_2$ , with the property that  $B(x, y)$  implies that: 1) if  $(x, \sigma, x') \in T_1$ , then there is  $y' \in T_2$  with  $(y, \sigma, y')$  and  $B(x', y')$  and 2) if  $(y, \sigma, y') \in T_2$ , then there is  $x' \in T_1$  with  $(x, \sigma, x')$  and  $B(x', y')$ .  $T_1$  and  $T_2$  are said to be *bisimilar* if there is a bisimulation between them.

For two labeled transition systems  $T_1$  and  $T_2$  over alphabets consisting of  $\Sigma$  plus a distinguished letter  $\tau$ , we say they are *weakly bisimilar* if there is a  $B$  such that  $B(x, y)$  implies that 1) and 2) above hold for every  $\sigma \in \Sigma$ , and also that 3) if  $(x, \tau, x') \in T_1$ , then  $B(x', y)$  and 4) if  $(y, \tau, y') \in T_2$ , then  $B(x, y')$ .

We wish to consider recursive automata as generators of  $\omega$ -languages as well as labeled transition systems. For the *linear-time setting*, we augment RSMs with a designated set of initial nodes, and with Büchi acceptance conditions. A *recursive Büchi automaton* (RBA) over a finite alphabet  $\Sigma$  consists of an RSM  $A$  over  $\Sigma$ , a set  $Init \subseteq \cup_{i=1}^k En_i$  of initial nodes and a set  $F \subseteq \cup_{i=1}^k N_i$  of *repeating* (accepting) nodes. (If  $F$  is not given, by default we assume  $F = \cup_{i=1}^k N_i$  to associate a language  $L(A)$  with RSM  $A$  and its  $Init$  set). Given an RBA,  $(A, Init, F)$ , we obtain an (infinite) global Büchi automaton  $B_A = (T_A, Init^*, F^*)$ , where the initial states  $Init^*$  are states  $\langle e \rangle$  where  $e \in Init$ , and where a state  $\langle b_1, \dots, b_r, v \rangle$  is in  $F^*$  if  $v$  is in  $F$ . For an infinite word  $w = w_0 w_1 \dots$  over  $\Sigma$ , a *run*  $\pi$  of  $B_A$  over  $w$  is a sequence  $s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \dots$  of states  $s_i$  and symbols  $\Sigma$  such that (1)  $s_0 \in Init^*$ , (2)  $(s_i, \sigma_i, s_{i+1}) \in \delta$  for all  $i$ , and (3) the word  $w$  equals  $\sigma_0 \sigma_1 \sigma_2 \dots$ . A run  $\pi$  is *accepting* if for infinitely many  $i$ ,  $s_i \in F^*$ .

We call a run  $\pi$  *bounded* if there is an integer  $m$  such that for all  $i$ , the length of the tuple  $s_i$  is bounded by  $m$ . It is *unbounded* otherwise. In other words, in a bounded (infinite) run the stack-length (number of boxes in context) always stays bounded. A word  $w \in \Sigma^\omega$  is (*boundedly/unboundedly*) *accepted* by the RBA  $A$  if there is an accepting (bounded/unbounded) run of  $B_A$  on  $w$ . Note,  $w$  is boundedly accepted if and only if for some  $s \in F^*$  there is a run  $\pi$  on  $w$  for which  $s_i = s$  infinitely often. This is not so for unbounded accepting runs.

We let  $L(A)$ ,  $L_b(A)$  and  $L_u(A)$  denote the set of words accepted, boundedly accepted, and unboundedly accepted by  $A$ , respectively. Clearly,  $L_b(A) \cup L_u(A) = L(A)$ , but  $L_b(A)$  and  $L_u(A)$  need not be disjoint. Given RBA  $A$ , we will be interested in the following algorithmic problems:

1. *Reachability*: Given  $A$ , for nodes  $u$  and  $v$  of  $A$ , let  $u \Rightarrow v$  denote that some global state  $\langle b_1, \dots, b_r, v \rangle$ , whose node is  $v$ , is reachable from the global state  $\langle u \rangle$  in the global transition system  $T_A$ . Extending the notation, let  $Init \Rightarrow v$  denote that for some  $u \in Init$ ,  $u \Rightarrow v$ . Our goal in simple reachability analysis is to compute the set  $\{v \mid Init \Rightarrow v\}$  of reachable vertices.
2. *Language emptiness*: We want to determine if  $L(A)$ ,  $L_b(A)$  and  $L_u(A)$  are empty or not. We obtain thereby algorithms for model checking RSMs.
3. *LTL model-checking*: Given a formula  $\phi$  in linear time temporal logic, we want to check if  $\forall \omega \in L(A) \omega \models \phi$ , and similarly for  $L_b$  and  $L_u$ .

In the *branching-time* setting, we will consider the problem of checking a formula in the logic CTL\* over an RSM  $A$  with a given initial global state  $s$ . CTL\* uses the temporal operators  $U$  (until),  $X$  (nexttime) and the existential path quantifier  $E$ , in addition to the operators  $\neg$  (not) and  $\vee$  (or). We will give a slight variant of CTL\* appropriate for labeled transition systems. Two types of CTL\* formulas, path formulas and state formulas, are defined by mutual induction. In our case, a state formula is satisfied by a particular state in the LTS, while a path formula is satisfied by a sequence of labeled edges  $p = a_1 \dots a_n \dots$ .

An edge label  $a \in \Sigma$  is a basic path formula, and it is satisfied by  $p$  as above if and only if  $a_1 = a$ . Every state formula  $\phi$  is a path formula, and it is satisfied by  $p$  if and only if it is satisfied by the source of the edge  $a_1$ . If  $p, q$  are both state formulas (respectively, both path formulas) then  $p \vee q$  and  $\neg p$  are also state formulas (respectively, path formulas). If  $p$  and  $q$  are path formulas, then  $pUq$  and  $Xp$  are also path formulas while  $Ep$  is a state formula. We use the standard abbreviation  $Fp$  for  $trueUp$  and  $Gp$  for  $\neg F\neg p$ . We will use the fact that CTL\* state formula can be viewed as a Boolean combination of existential formulas, where an existential formula is either an atomic proposition or a CTL\* state formula of the form  $E \rho(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n)$ , where  $\rho$  is an LTL formula over propositions  $p(\gamma_1), \dots, p(\gamma_n)$  and the  $\gamma_i$  are CTL\* state formulas. More information on CTL\* can be found in Emerson [1990]. In particular, it is well-known that if  $B$  is a bisimulation of  $T_1$  and  $T_2$  and  $B(x, y)$  holds for some states  $x, y$  of  $T_1, T_2$  (respectively) then every CTL\* formula satisfied by  $T_1$  at  $x$  is also satisfied by  $T_2$  at  $y$ .

*Notation.* We use the following notation. Let  $v_i$  be the number of vertices and  $e_i$  the number of transitions (edges) of each component  $A_i$ , and let  $v = \sum_i v_i$ ,  $e = \sum_i e_i$  be the total number of vertices and edges. The *size*  $|A|$  of an RSM  $A$  is the space needed to write down its components. Assuming, without loss of generality, that each node and each box of each component is involved in at least one transition,  $v \leq 2e$ , and the size of  $A$  is proportional to its number of edges  $e$ . The other parameter that enters in the complexity is  $\theta$ , a bound on the number of entries or exits of the components. Let  $en_i = |En_i|$  and  $ex_i = |Ex_i|$ , be the number of entries and exits in the  $i$ 'th component,  $A_i$ . Then  $\theta = \max_{i \in \{1, \dots, k\}} \min(en_i, ex_i)$ . That is, every component has either no more than  $\theta$  entries or no more than  $\theta$  exits. There may be some components of each kind; we call components of the first kind *entry-bound* and the others *exit-bound*.

### 3. RELATION TO PUSHDOWN AND CONTEXT-FREE SYSTEMS

In this section we explore the relationship between recursive automata and a variant of pushdown automata. We then see what these results, together with the results of Alur and Yannakakis [2001] on RSMs without recursion, tell us about the complexity of model checking.

#### 3.1 Expressiveness of RSMs Versus other Models

For purposes of comparison with RSMs, we consider a *pushdown system* (PDS) given by  $P = (Q_P, \Gamma, \Delta)$  over an alphabet  $\Sigma$  consisting of a set of control states  $Q_P$ , a stack alphabet  $\Gamma$ , and a transition relation  $\Delta \subseteq (Q_P \times \Gamma) \times \Sigma \times (Q_P \times \{swap(\Gamma), swap - and - push(\Gamma \times \Gamma), pop\})$ . That is, based on the control state and the symbol on top of the stack, the machine can update the control state and either swap the top-of-the-stack with a new symbol, simultaneously swap the top-of-the-stack with a new symbol and then push a second new symbol, or pop the stack. A PDS is a *context-free system* if it has only one control state.

Note that the standard “push” operation can be simulated using swap-and-push. Also note that a push of several stack elements on the stack in one step can be simulated (for both PDS and context-free systems) by expanding the

stack alphabet (e.g., given a machine pushing in one step the sequence  $a_1a_2$  of two symbols with  $a_1, a_2 \in \Gamma$ , we use a machine whose stack is  $\Gamma \cup \Gamma^2$  and where pushing  $a_1a_2$  on the stack is done by pushing the single stack element  $(a_1, a_2)$  on the stack). A stack with letter  $\gamma$  at the top and remaining content  $\omega \in \Gamma^*$  (running from bottom to top) will be written  $\omega\gamma$ . The semantics of a PDS is given in terms of an LTS over  $\Sigma$  in the obvious way: a state is a pair  $(w, q)$  with  $q \in Q^P$  and  $w \in \Gamma^*$ , and, for example, PDS transition  $((q, \gamma), \sigma, (q', \text{swap}(\gamma')))$  leads to a transition  $((w\gamma, q), \sigma, (w\gamma', q'))$  in the LTS.

We now compare the expressiveness of PDS with RSM. In what follows, we say that a PDS is bisimilar to an RSM when the corresponding LTSs are bisimilar.

**THEOREM 1.** *Every PDS is bisimilar to a RSM, and vice versa. Moreover, every context-free system is bisimilar to a single-exit RSM, and vice versa.*

**PROOF.** Given a PDS  $P$ , we build a recursive automaton  $A(P)$  that is bisimilar to it. Our construction is done in two steps: first, we build an RSM  $A_s(P)$  with  $\tau$  transitions that is weakly bisimilar to  $P$ , and then we eliminate the  $\tau$  transitions in  $A_s(P)$  to obtain  $A(P)$  while preserving weak bisimulation.

$A_s(P)$  has only one component,  $A_1$ .  $A_1$  has an entry  $en_{(q,\gamma)}$  for every pair  $(q, \gamma)$  with  $q$  in  $Q_P$  and  $\gamma$  in  $\Gamma$ . It has one exit  $ex_q$  for every  $q \in Q_P$ . It also has one box  $b_\gamma$  associated with each stack symbol  $\gamma \in \Gamma$  that plays a role in some transition of  $P$ . All boxes are, obviously, mapped to  $A_1$ . The transitions of  $A_1$  are as follows:

1. For every transition  $((q, \gamma), \sigma, (q', \text{pop}))$  in  $\Delta$ , there is a transition  $(en_{(q,\gamma)}, \sigma, ex_{q'})$  in  $\delta_1$ .
2. For every transition  $((q, \gamma), \sigma, (q', \text{swap}(\gamma')))$  in  $\Delta$ , there is a transition  $(en_{(q,\gamma)}, \sigma, en_{(q',\gamma')})$  in  $\delta_1$ .
3. For every transition  $((q, \gamma), \sigma, (q', \text{swap} - \text{and} - \text{push}(\gamma_1, \gamma_2)))$  in  $\Delta$ , there is  $(en_{(q,\gamma)}, \sigma, (b_{\gamma_1}, en_{(q',\gamma_2)}))$  in  $\delta_1$ .
4. For every box exit  $(b_\gamma, ex_q)$  of each box  $b_\gamma$ , there is a  $\tau$ -transition  $((b_\gamma, ex_q), \tau, en_{(q,\gamma)})$  in  $\delta_1$ .

The intuition behind this construction should be clear: each entry node  $en_{(q,\gamma)}$  of  $A_1$  corresponds to the configuration of  $P$  with control state  $q$  and top-of-stack  $\gamma$ . The remainder of the content of the pushdown stack of  $P$  (with the top element excluded) is coded in the call stack of  $A_s(P)$ , with box  $b_\gamma$  on the call stack acting like  $\gamma$  on the pushdown stack. The size  $|A_s(P)|$  is  $O(|P|)$ , and the number of exits of  $A_1$  is  $|Q_P|$ . The translation preserves boundedness, and indeed stack/call depth, so that runs of  $P$  that require bounded (unbounded) stack depth correspond to runs of  $A_s(P)$  that are bounded (unbounded).

It is easy to prove that the unfolding of  $A_s(P)$  is weakly bisimilar to the unfolding of  $P$ . This is done by taking as bisimulation relation  $B$  the mapping that relates every global state  $\langle \gamma_1 \dots \gamma_n, q \rangle$  of  $P$  to the global states  $\langle b_{\gamma_1} \dots b_{\gamma_n}, ex_q \rangle$  and  $\langle b_{\gamma_1} \dots b_{\gamma_{n-1}}, en_{(q,\gamma_n)} \rangle$  of  $A_s(P)$ .

The  $\tau$  transitions in  $A_s(P)$  can be eliminated as follows: remove all the  $\tau$  transitions, and for each transition  $(en_{(q,\gamma)}, \sigma, x)$  out of  $en_{(q,\gamma)}$ , add a transition  $((b_\gamma, ex_q), \sigma, x)$ . Let  $A(P)$  be the resulting RSM. It is easy to check that the

unfolding of  $A(P)$  is weakly bisimilar to the unfolding of  $A_s(P)$ . This in turn implies that the unfolding of  $A(P)$  is bisimilar to  $P$ , since neither of them have  $\tau$ -transitions. Note that the size of  $A(P)$  is linear in the size of  $P$ . Finally, note that, given a PDS with a single state, our translation generates a single-exit RSM.

In the other direction, given an RSM  $A$ , we now describe how to build a PDS  $P(A)$  that is bisimilar to  $A$ . We proceed in two stages again: first, we build a PDS  $P_s(A)$  with  $\tau$  transitions that is weakly bisimilar to  $A$ , and then we eliminate the  $\tau$  transitions in  $P_s(A)$  to obtain  $P(A)$  while preserving weak bisimulation.

We begin with the construction of  $P_s(A)$ . The stack symbols  $\Gamma$  of  $P_s(A)$  correspond to all possible “locations”  $(j, v)$  in the recursive automaton:  $j$  gives the index of the component, and  $v$  is either a node of  $A_j$  or a box  $b$  of  $A_j$ . The control states  $Q$  of  $P_s(A)$  are  $\{1 \dots ex\}$ , where  $ex$  is the maximum number of exit nodes of any component. For every transition  $(v, \sigma, v')$  between vertices  $v$  and  $v'$  of a component  $A_j$  where  $v$  is not of the form  $(b, x)$  and  $v'$  is not of the form  $(b', e)$ , there is a transition  $((1, (j, v)), \sigma, (1, swap(j, v')))$  in  $P_s(A)$ . For every transition  $(v, \sigma, (b, e))$  within a component  $A_j$  where  $v$  is not of the form  $(b', x)$  and where  $b$  is a box of  $A_j$  mapped to component  $A_{j'}$  and  $e$  is an entry node of  $A_{j'}$ , there is a transition  $((1, (j, v)), \sigma, (1, swap-and-push((j, b), (j', e))))$  in  $P_s(A)$ . For every exit node  $x_i$  of component  $A_j$  (where  $x_i$  is the  $i$ th exit of  $A_j$ ), there is a transition  $((1, (j, x_i)), \tau, (i, pop))$ . For every box  $b$  of  $A_j$  mapped to component  $A_{j'}$ , for every exit  $x_i$  of  $A_{j'}$  and every transition  $((b, x_i), \sigma, v')$  with  $v'$  not of the form  $(b', e)$ , there is a transition  $((i, (j, b)), \sigma, (1, swap(j, v')))$  in  $P_s(A)$ . Finally, for every box  $b$  of  $A_j$  mapped to component  $A_{j'}$ , for every exit  $x_i$  of  $A_{j'}$  and every transition  $((b, x_i), \sigma, (b', e))$  with  $e$  an entry node of  $A_k$ , there is a transition  $((i, (j, b)), \sigma, (1, swap-and-push((j, b'), (k, e))))$  in  $P_s(A)$ .

The size of  $P_s(A)$  is linear in the size of recursive automaton  $A$ . Note that in the case where  $A$  is single-exit,  $P_s(A)$  is context-free. This translation also preserves call/stack depth, and thus boundedness.

We now define  $P(A)$  by eliminating the  $\tau$  transitions in  $P_s(A)$  as follows: remove all the  $\tau$  transitions; for every transition  $(n, \sigma, x_i)$  to an exit node  $x_i$  of a component  $A_j$  (where  $x_i$  is the  $i$ th exit of  $A_j$ ), add a transition  $((1, (j, n)), \sigma, (i, pop))$  (note that stack elements of the form  $(j, x_i)$  are never used in the definition of  $P(A)$ ); for every transition  $(x_i, \sigma, n)$  from an exit node  $x_i$  of a component  $A_j$  (where  $x_i$  is the  $i$ th exit of  $A_j$ ) to a node  $n \in N_j$ , add a transition  $((i, -), \sigma, (1, push(j, n)))$  (where  $-$  indicates any stack element); for every transition  $(x_i, \sigma, x_j)$  from an exit node  $x_i$  of a component  $A_j$  (where  $x_i$  is the  $i$ th exit of  $A_j$ ) to an exit node  $x_j$ , add a transition  $((i, -), \sigma, (j, -))$  (meaning the stack is left unchanged); for every transition  $(x_i, \sigma, (b, e))$  from an exit node  $x_i$  of a component  $A_j$  (where  $x_i$  is the  $i$ th exit of  $A_j$ ) to a call node  $(b, e)$  calling machine  $A_k$ , add a transition  $((i, -), \sigma, (1, double-push((j, b), (k, e))))$  (where  $-$  indicates any stack element and *double-push* is syntactic sugaring for pushing a pair of values on the stack as explained earlier).

It is easy to check that the unfolding of  $P(A)$  is bisimilar to the unfolding of  $A$ , by taking as bisimulation relation  $B$  the following mapping: for every global state  $g = \langle b_1 \dots b_n, v \rangle$  of  $A$  where  $v$  is not an exit node,  $g$  is mapped to the global state  $\langle \gamma_1 \dots \gamma_{n+1}, 1 \rangle$  of  $P(A)$  with for all  $i \leq n$ ,  $\gamma_i = (j_i, b_i)$  (where  $b_i$  is a box of component  $A_{j_i}$ ), and  $\gamma_{n+1} = (j_{n+1}, v)$  with  $j_{n+1}$  being the index of the machine

called by box  $b_n$ ; for every global state  $g = \langle b_1 \dots b_n, x_i \rangle$  of  $A$  where  $x_i$  is the  $i$ th exit node of a component  $A_j$  called by box  $b_n$ ,  $g$  is mapped to the global state  $\langle \gamma_1 \dots \gamma_n, i \rangle$  of  $P(A)$  with for all  $i \leq n$ ,  $\gamma_i = (j_i, b_i)$  (where  $b_i$  is a box of component  $A_{j_i}$ ). Finally, note that, given a single-exit RSM  $A$ , our translation generates a PDS  $P(A)$  with a single state. This concludes the proof of Theorem 1.  $\square$

Since it is known [Caucal and Monfort 1990] that there exist pushdown systems that are not bisimilar to any context-free systems, we obtain the following result from the previous theorem.

**COROLLARY 2.** *There exist multiple-exit RSMs  $R$  for which the unfolding  $T_R$  is not bisimilar to the unfolding of any single-exit RSM.*

We now find that the relationships are also tight if we consider the complexity of model checking, rather than expressiveness.

**THEOREM 3.**

- *The LTL model checking problem for RSMs and for pushdown systems are inter-reducible in linear time and logarithmic space, and similarly for CTL and CTL\*.*
- *The LTL model checking problem for single-exit RSMs and for context-free systems are inter-reducible in linear time and logarithmic space, and similarly for CTL and CTL\*.*

**PROOF.** The reduction from PDS model checking to RSM model checking follows from the linear translation from a PDS  $P$  to an RSM  $A(P)$  given in the proof of Theorem 1 since  $P$  and  $A(P)$  are bisimilar and since bisimulation preserves all the logics we consider here.

Conversely, we note that the translation from  $A$  to  $P(A)$  given in the proof of Theorem 1 is not linear, so the proof is not as direct. However, the translation from  $A$  to  $P_s(A)$  is linear, so we proceed by reducing a RSM model-checking problem  $(A, \phi)$  to a PDS model-checking problem  $(P_s(A), \phi')$  where  $P_s(A)$  is a PDS over alphabet  $\Sigma^+ = \Sigma \cup \{\tau\}$  and  $\phi'$  is a formula of the same logic as  $\phi$  but over  $\Sigma^+$ .

Formula  $\phi'$  is defined by translating formula  $\phi$  as follows. We start by defining how to translate LTL (i.e., path) formulas:

- $T(a) = a \vee (\tau \wedge Xa)$  for  $a \in \Sigma$ ,
- $T(X\rho) = (\neg\tau \wedge XT(\rho)) \vee (\tau \wedge XXT(\rho))$ ,
- $T(F\rho) = FT(\rho)$ ,
- $T(\neg\rho) = \neg T(\rho)$ ,
- $T(\tau U\rho) = T(\tau)UT(\rho)$ .

To see that  $T(\phi)$  is the desired formula, note that in the unfolding of  $P_s(A)$ , every  $\tau$ -move always leads to a vertex from which no  $\tau$  move is possible. Hence any infinite run through  $P_s(A)$  has no two consecutive  $\tau$  transitions in it. Given a word  $\omega$  with no consecutive occurrences of  $\tau$ , let  $C(\omega)$  be the word formed by

removing each  $\tau$ . The correctness of  $T(\phi)$  now follows from the following claim:

$$\omega \models T(\phi) \Leftrightarrow C(\omega) \models \phi.$$

The proof of the claim is by induction on the length of the formula. For a string  $\omega$  we let  $\omega(i)$  be the  $i$ th letter and  $\omega^i$  be the suffix of  $\omega$  starting after the  $i$ th letter. In the base case, the result follows since the initial letter of  $C(\omega)$  is either the same  $\omega(1)$  if  $\omega(1) \neq \tau$ , or is  $\omega(2)$  otherwise.

For the case of the next operator, note that  $C(\omega) \models X\rho$  holds if and only if  $(\omega(1) \neq \tau$  and  $C(\omega^1) \models \rho)$  or  $(\omega(1) = \tau$  and  $C(\omega^1) \models X\rho)$ . In the latter case, since  $\omega(2) \neq \tau$  we know that  $C(\omega^1) \models X\rho$  if and only if  $C(\omega^2) \models \rho$ . Hence  $C(\omega) \models X\rho$  is equivalent to  $\omega(1) \neq \tau$  and  $C(\omega^1) \models \rho$  or  $\omega(1) = \tau$  and  $C(\omega^2) \models \rho$ , and by induction this is the same as  $\omega(1) \neq \tau$  and  $\omega^1 \models T(\rho)$  or  $\omega(1) = \tau$  and  $\omega^2 \models T(\rho)$ .

For the eventually operator, we have  $C(\omega) \models F\rho$  if and only if  $\exists n C(\omega)^n \models \rho$  if and only if  $\exists i C(\omega^i) \models \rho$  if and only if  $\exists i \omega^i \models T(\rho)$  if and only if  $\omega \models FT(\rho)$ . The second equivalence holds since every suffix of  $C(\omega)$  is a suffix of  $C(\omega^i)$  for some  $i$ .

For the until case, suppose  $C(\omega) \models \tau U\rho$ . We show  $\omega \models T(\tau)UT(\rho)$ . We know that either  $C(\omega) \models G\tau \wedge \neg\rho$  or for some  $n C(\omega)^n \models \rho$  and  $\forall i < n C(\omega)^i \models \tau \wedge \neg\rho$ . If  $C(\omega) \models G\tau \wedge \neg\rho$  then by the definition of  $G$ ,  $\forall n C(\omega)^n \models \tau \wedge \neg\rho$ . Therefore  $\forall m C(\omega)^m \models \tau \wedge \neg\rho$ , since  $C(\omega)^m = C(\omega)^n$  for some  $n$ . Hence by induction we have  $\forall m \omega^m \models T(\tau \wedge \neg\rho) = T(\tau) \wedge \neg T(\rho)$ . So  $\omega \models GT(\tau) \wedge \neg T(\rho)$ , therefore  $\omega \models (T(\tau) \wedge \neg T(\rho))UT(\rho)$  as required. Now assume that we have  $n$  such that  $C(\omega)^n \models \rho$  and  $\forall i < n C(\omega)^i \models \tau \wedge \neg\rho$ . Arguing as before we see that there is  $m$  such that  $C(\omega)^m \models \rho$  and  $\forall i < m C(\omega)^i \models \tau \wedge \neg\rho$ . Then by induction we have  $\omega^m \models T(\rho)$  and  $\forall i < m \omega^i \models T(\tau) \wedge \neg T(\rho)$ , and from this we see that  $\omega \models T(\tau)UT(\rho)$ .

Now suppose  $\omega \models T(\tau)UT(\rho)$ . Suppose that there is  $m$  with  $\omega^m \models T(\rho)$  and  $\forall i < m \omega^i \models T(\tau) \wedge \neg T(\rho)$ . By induction,  $C(\omega)^m \models \rho$  and  $\forall i < m C(\omega)^i \models \tau \wedge \neg\rho$ . Choosing  $n$  such that  $C(\omega)^n = C(\omega)^m$  we see  $C(\omega)^n \models \rho$  and  $\forall i < n C(\omega)^i \models \tau \wedge \neg\rho$ , but then  $C(\omega) \models \tau U\rho$ . The case where  $\omega \models GT(\rho)$  is similar. This completes the proof of the claim.

To translate a CTL\* formula, we add the rule  $T(E\rho) = E T(\rho)$ . Since  $A$  and  $P_s(A)$  are weakly bisimilar and the above claim, the reduction from the RSM model-checking problem  $(A, \phi)$  to the PDS model-checking problem  $(P_s(A), T(\phi))$  suffices to prove the other half of Theorem 3.

Finally, we have shown above that a PDS with a single state can be translated to a single-exit RSM, and vice versa. Therefore, model checking for context-free systems and single-exit RSM are inter-reducible.  $\square$

### 3.2 Consequences for Model Checking Problems

We now review what the expressive results above tell us about model checking.

Since the hierarchical state machines of Alur and Yannakakis [2001] are special cases of RSMs, it is worth reviewing some of the results presented in Alur and Yannakakis [2001] for this restricted case. The following table summarizes the results of Alur and Yannakakis [2001] concerning the complexity in the

Table I. Known Results for RSMs Without Recursion

| Class of RSM             | Reachability | Cycle Detection | LTL    | CTL    |
|--------------------------|--------------|-----------------|--------|--------|
| Restricted Single-exit   | Linear       | Linear          | Linear | Linear |
| Restricted Multiple-exit | Linear       | Linear          | Linear | PSPACE |

size of the RSM for RSMs without recursion. It deals with each of the major verification problems we consider here, except  $CTL^*$  model checking, which was not discussed in Alur and Yannakakis [2001]. For the model checking problems, the complexity below means the size of the formula is fixed. Note that Alur and Yannakakis [2001] also showed that CTL model checking is also PSPACE-complete in the size of the formula for any fixed restricted RSM.

Thanks to the correspondence theorems established in the previous section, we can obtain algorithms and complexity bounds for the verification of RSMs from previously existing algorithms and bounds for the verification of context-free and pushdown systems.

Consider the case of single-exit RSMs. By Theorem 3, model checking for single-exit RSMs can be reduced to model checking for context-free systems. Since LTL model checking for context-free systems can be solved in time linear in the size of the pushdown automaton [Finkel et al. 1997], LTL model checking for single-exit RSMs can be solved in time linear in the size of the RSM. This also implies a linear-time algorithm for the reachability and cycle detection problems. A linear-time algorithm for CTL model checking for single-exit RSMs can be derived from the CTL model checking algorithm for context-free systems given in Burkart and Steffen [1992]. Finally, since the  $\mu$ -calculus model checking algorithm of Burkart and Steffen [1999] for context-free systems runs in quadratic time for formulae in the second level of the  $\mu$ -calculus alternation hierarchy, which is known to contain  $CTL^*$  [Emerson and Lei 1986],  $CTL^*$  model checking for single-exit RSMs can be solved in time quadratic in the size of the RSM.

Let us turn to the case of multiple-exit RSMs. By Theorem 3, we know that model checking for multiple-exit RSMs can be reduced to model checking for pushdown systems. Since LTL model checking for pushdown automata can be solved in time cubic in the size of the pushdown automaton [Finkel et al. 1997; Esparza et al. 2000], LTL model checking for multiple-exit RSMs can be solved in time cubic in the size of the RSM. Moreover, a cubic-time algorithm for the reachability and cycle detection problems can easily be derived from this LTL model checking algorithm. Since we know that the model checking problem for pushdown systems and the alternation-free  $\mu$ -calculus is EXPTIME-hard [Walukiewicz 2001] and that an exponential-time algorithm for solving this problem is presented in Bouajjani et al. [1997], and since CTL is contained in the alternation-free  $\mu$ -calculus, we can deduce that the CTL model checking problem for multiple-exit RSMs is EXPTIME-complete in the size of the RSM. Similarly, the exponential-time model-checking algorithm given in Burkart and Steffen [1999] for pushdown systems and the full  $\mu$ -calculus, which contains  $CTL^*$ , and the EXPTIME-hardness result of [Walukiewicz 2001] imply that the  $CTL^*$  model checking problem for multiple-exit RSMs is also EXPTIME-complete in the size of the RSM.

Table II. Algorithms and Complexity Bounds for RSMs

| Class of RSM  | Reachability | Cycle Detection | LTL    | CTL     | CTL*      |
|---------------|--------------|-----------------|--------|---------|-----------|
| Single-exit   | Linear       | Linear          | Linear | Linear  | Quadratic |
| Multiple-exit | Cubic        | Cubic           | Cubic  | EXPTIME | EXPTIME   |

Table III. Improved Algorithms for RSMs

| Class of RSM                 | Reachability  | Cycle Detection | LTL           | CTL     | CTL*          |
|------------------------------|---------------|-----------------|---------------|---------|---------------|
| Single-exit                  | Linear        | Linear          | Linear        | Linear  | <i>Linear</i> |
| Single-entry Multiple-exit   | <i>Linear</i> | <i>Linear</i>   | <i>Linear</i> | EXPTIME | EXPTIME       |
| Multiple-entry Multiple-Exit | Cubic         | Cubic           | Cubic         | EXPTIME | EXPTIME       |

The following table summarizes the results we have obtained in the previous discussion. Complexity bounds are given in the size of the RSM.

In the remainder of this article, we present two improvements to the results of Table II. First, in the next section, we present an LTL model checking algorithm for RSMs and define precisely the complexity of this algorithm. We then show that LTL model checking for single-entry multiple-exit RSMs can be solved with this algorithm in time *linear* in the size of the RSM, instead of *cubic* time (see Table II). This implies that the reachability and cycle detection problems can also be solved in linear-time for single-entry RSMs. Second, in Section 5, we present a new CTL\* algorithm for single-exit RSMs that runs in time *linear* in the size of the RSM, instead of *quadratic* time (see Table II).

After taking into account these two new results, the complexity of the five verification problems for RSMs is summarized in Table III. (Improvements introduced in the two next sections are highlighted in italic.)

#### 4. LINEAR-TIME PROPERTIES

Given a recursive automaton,  $A$ , our ultimate goal in this section is to show how one can verify any properties in linear time temporal logic. We begin by showing that two key analysis problems, reachability and language emptiness, can be solved in time  $O(|A|\theta^2)$ ; more precisely, in time  $O(e\theta + v\theta^2)$  and space  $O(e + v\theta)$ . For notational convenience, we will assume without loss of generality that all entry nodes of the machines have no incoming edges and all exit nodes have no outgoing edges.

We will then show how these results immediately imply bounds on model-checking of properties given as Büchi automata or LTL formulas.

##### 4.1 Reachability

Given  $A$ , we wish to compute the set  $\{v \mid \text{Init} \Rightarrow v\}$ . For clarity, we present our algorithm in two stages. First, we define a set of Datalog rules and construct an associated AND-OR graph  $G_A$ , which can be used to compute information about reachability within each component automaton. Next, we use this information to obtain an ordinary graph  $H_A$ , such that we can compute the set  $\{v \mid \text{Init} \Rightarrow v\}$  by simple reachability analysis on  $H_A$ .

**Step 1: The Rules and the AND-OR graph construction.** As a first step we will compute, for each component  $A_i$ , a predicate (relation)  $R_i(x, y)$ . If  $A_i$  is entry-bound, then the variable  $x$  ranges over all entry nodes of  $A_i$  and  $y$  ranges

over all vertices of  $A_i$ . If  $A_i$  is exit-bound, then  $x$  ranges over all vertices of  $A_i$  and  $y$  ranges over all exit nodes of  $A_i$ . The meaning of the predicate is defined as follows:  $R_i(x, y)$  holds if and only if there is a path in  $T_A$  from  $\langle x \rangle$  to  $\langle y \rangle$ .

The predicates  $R_i(x, y)$  are determined by a series of simple recursive relationships that we will write in the style of Datalog rules [Ullman 1988]. Recall some terminology. An *atom* is a term  $P(\tau)$  where  $P$  is a predicate (relation) symbol and  $\tau$  is a tuple of appropriate arity consisting of variables and constants from appropriate domains. A *ground atom* has only constants. A Datalog rule has the form  $head \leftarrow body$ , where  $head$  is an atom and  $body$  is a conjunction of atoms. The meaning of a rule is that if for some instantiation  $\sigma$ , mapping variables of a rule to constants, all the (instantiated) conjuncts in the body of the rule  $\sigma(body)$  are true, then the instantiated head  $\sigma(head)$  must also be true. For readability, we deviate slightly from this notation and write the rules as “ $head \leftarrow body$ , under constraint  $C$ ,” where  $body$  includes only recursive predicates, and nonrecursive constraints are in  $C$ . We now list the rules for the predicates  $R_i$ . We distinguish two cases, depending on whether the component  $A_i$  has more entries or exits. Suppose first that  $A_i$  is entry-bound. Then, we have the following three rules. (Technically, there is one instance of rule 3 for each box  $b$  of  $A_i$ .)

1.  $R_i(x, x)$  ,  $x \in En_i$
2.  $R_i(x, w) \leftarrow R_i(x, u)$  ,  $x \in En_i, (u, w) \in E_i$
3.  $R_i(x, (b, w)) \leftarrow R_i(x, (b, u)) \wedge R_j(u, w)$  ,  $x \in En_i, b \in B_i, Y_i(b)$   
 $= j, u \in En_j, w \in Ex_j$ .

Rule 1 says every entry node  $x$  can reach itself. Rule 2 says if an entry  $x$  can reach vertex  $u$ , which has an edge to vertex  $w$ , then  $x$  can reach  $w$ . Rule 3 says if entry  $x$  of  $A_i$  can reach an entry port  $(b, u)$  of a box  $b$ , mapped say to the  $j$ 'th component  $A_j$ , and the entry  $u$  of  $A_j$  can reach its exit  $w$ , then  $x$  can reach the exit port  $(b, w)$  of box  $b$ ; we further restrict the domain to only apply this rule for ports of  $b$  that are vertices (i.e.,  $(b, u), (b, w)$  are incident to some edges of  $A_i$ ). Rules for exit-bound component machines  $A_i$  are similar.

1.  $R_i(x, x)$  ,  $x \in Ex_i$
2.  $R_i(u, x) \leftarrow R_i(w, x)$  ,  $x \in Ex_i, (u, w) \in E_i$
3.  $R_i((b, u), x) \leftarrow R_i((b, w), x) \wedge R_j(u, w)$  ,  $x \in Ex_i, b \in B_i, Y_i(b)$   
 $= j, u \in En_j, w \in Ex_j$ .

The Datalog program can be evaluated incrementally by initializing the relations with all ground atoms corresponding to instantiations of heads of rules with empty body (i.e., the atoms  $R_i(x, x)$  for all entries/exits  $x$  of  $A_i$ ), and then using the rules repeatedly to derive new ground atoms that are heads of instantiations of rules whose bodies contain only atoms that have been already derived. As we'll see below, if implemented properly, the time complexity is bounded by the number of possible instantiated rules and the space is bounded by the number of possible ground atoms. The number of possible ground atoms of the form  $R_i(x, y)$  is at most  $v_i\theta$ , and thus the total number of ground atoms is at most  $v\theta$ . The number of instantiated rules of type 1 is bounded by the number of nodes, and the number of rules of type 2 is at most  $e\theta$ . The number of instantiated rules of type 3 is at most  $v\theta^2$ .

The evaluation of the Datalog program can be seen equivalently as the evaluation (reachability analysis) of a corresponding AND-OR graph  $G_A = (V, E, Start)$ . Recall that an AND-OR graph is a directed graph  $(V, E)$  whose vertices  $V = V_\vee \cup V_\wedge$  consist of a disjoint union of *and* vertices,  $V_\wedge$ , and *or* vertices,  $V_\vee$ , and a subset of vertices  $Start$  is given as the initial set. Reachability is defined inductively: a vertex  $p$  is *reachable* if: **(a)**  $p \in Start$ , or **(b)**  $p$  is a  $\vee$ -vertex and  $\exists p'$  such that  $(p', p) \in E$  and such that  $p'$  is reachable, or **(c)**  $p$  is a  $\wedge$ -vertex and for  $\forall p'$  such that  $(p', p) \in E$ ,  $p'$  is reachable. It is well-known that reachability in AND-OR graphs can be computed in linear time (see, e.g., Anderson [1994]).

We can define from the rules an AND-OR graph  $G_A$  with one  $\vee$ -vertex for each ground atom  $R_i(x, y)$  and one  $\wedge$ -vertex for each instantiated body of a rule with two conjuncts (rule of type 3). The set  $Start$  of initial vertices is the set of ground atoms resulting from the instantiations of rules 1 that have empty bodies. Each instantiated rule of type 2 and 3 introduces the following edges: For a rule of type 2 (one conjunct in the body) we have an edge from the ( $\vee$ -vertex corresponding to the ground) atom in the body of the rule to the atom in the head. For an instantiated rule of type 3, we have edges from the  $\vee$ -vertices corresponding to the ground atoms in the body to the  $\wedge$ -vertex corresponding to the body, and from the  $\wedge$ -vertex to the  $\vee$ -vertex corresponding to the head. It can be shown that the reachable  $\vee$ -vertices in the AND-OR graph correspond precisely to the ground atoms that are derived by the Datalog program.

The AND-OR graph has  $O(v\theta)$   $\vee$ -vertices,  $O(v\theta^2)$   $\wedge$ -vertices and  $O(e\theta + v\theta^2)$  edges and can be constructed in a straightforward way and evaluated in this amount of time. However, it is not necessary to construct the graph explicitly. Note that the  $\wedge$ -vertices have only one outgoing edge, so there is no reason to store them: once a  $\wedge$ -vertex is reached, it can be used to reach the successor  $\vee$ -vertex and there is no need to remember it any more. Indeed, evaluation methods for Datalog programs maintain only the relations of the program recording the tuples (ground atoms) that are derived. We describe now how to evaluate the program within the stated time and space bounds.

Process the edges of the components  $A_i$  to compute the set of vertices and record the following information: If  $A_i$  is entry-bound (respectively, exit-bound) create the successor list (respectively predecessor list) for each vertex. For each box, create a list of its entries and exits that are vertices (have some incident edges). For each component  $A_i$  and each of its ports  $u$  create a list of all boxes  $b$  in all the machines of the RSM  $A$  that are mapped to  $A_i$  in which the port  $u$  of  $b$  has an incident edge (is a vertex). The reason for the last two data structures is that it is possible that many of the ports of the boxes have no incident edges, and we do not want to waste time looking at them, since our claimed complexity bounds charge only for ports that have incident edges. It is straightforward to compute the above information from a linear scan of the edges of the RSM  $A$ .

Each predicate (relation)  $R_i$  can be stored using either a dense or a sparse representation. For example, a dense representation is a bit-array indexed by the domain (possible tuples) of the relation:  $En_i \times V_i$  or  $V_i \times Ex_i$ . Initially all the bits are 0, and they are turned to 1 as new tuples (ground atoms) are derived. We maintain a list  $S$  of tuples that have been derived but not processed. The

processing order (e.g., FIFO or LIFO or any other) is not important. Initially, we insert into  $S$  (and set their corresponding bits) all the ground atoms from rule 1: atoms of the form  $R_i(x, x)$  for all entries  $x$  of entry-bound machines  $A_i$  and exits of exit-bound machines  $A_i$ . In the iterative step, as long as  $S$  is not empty, we remove an atom  $R_i(x, y)$  from  $S$  and process it. Suppose that  $A_i$  is entry-bound (the exit-bound case is similar). Then we do the following. For every edge  $(y, z) \in E_i$  out of  $y$ , we check if  $R_i(x, z)$  has been already derived (its bit is 1) and if not, then we set its bit to 1 and insert  $R_i(x, z)$  into  $S$ . If  $y = (b, u)$  is an entry node of a box  $b$ , where say  $b$  is mapped to  $A_j$ , then for every exit vertex  $(b, w)$  of  $b$  we check if  $R_j(u, w)$  holds; if it does and if  $R_i(x, (b, w))$  has not been derived, we set its bit and insert  $R_i(x, (b, w))$  into  $S$ . If  $y$  is an exit of  $R_i$ , then for every box  $b$  that is mapped to  $A_i$  and in which the corresponding port  $(b, y)$  is a vertex we do the following. Let  $A_k$  be the machine that contains the box  $b$ . If  $(b, x)$  is not a vertex of  $A_k$  nothing needs to be done. Otherwise, if  $A_k$  is entry-bound (respectively, exit-bound), we check for every entry (respectively, exit)  $z$  of  $A_k$  whether the corresponding rule 3 can be fired, that is, whether  $R_k(z, (b, x))$  holds but  $R_k(z, (b, y))$  does not (respectively,  $R_k((b, y), z)$  holds but  $R_k((b, x), z)$  does not). If so, we set the bit of  $R_k(z, (b, y))$  (respectively,  $R_k((b, x), z)$ ) and insert the atom into  $S$ . Correctness follows from the following lemma.

**LEMMA 4.** *Let  $(x, y)$  be a tuple such that  $x$  and  $y$  are vertices of  $A_i$ , and if  $A_i$  is entry-bound then  $x \in En_i$ , and if  $A_i$  is exit-bound then  $y \in Ex_i$ . Then  $(x, y)$  is added to the predicate  $R_i$  if and only if  $\langle x \rangle$  can reach  $\langle y \rangle$  in the transition system  $T_A$ .*

**PROOF.** An observation we use is that  $\langle x \rangle$  can reach  $\langle y \rangle$  if and only if for any global state  $\langle \bar{b}, x \rangle$ , there is a path from  $\langle \bar{b}, x \rangle$  to  $\langle \bar{b}, y \rangle$  such that every state along this path has a prefix  $\bar{b}$ . This follows easily from the structure of RSMs.

For the forward direction, if  $(x, y)$  is added to  $R_i$ , all that needs to be observed is that all the “rules” encoded in  $G_A$  are sound, and if they can be used to derive  $R_i(x, y)$ , then  $\langle y \rangle$  must be reachable from  $\langle x \rangle$  in  $T_A$ .

For the other direction, suppose there is a path  $\pi$  from  $\langle x \rangle$  to  $\langle y \rangle$ . We prove by induction on the length of  $\pi$  that tuple  $(x, y)$  is added to  $R_i$  by the rules. If  $|\pi| = 1$ , then  $x = y$ , and since  $R_i(x, x) \leftarrow true$ , is a rule,  $(x, x)$  is added to  $R_i$ . Suppose  $|\pi| = n + 1$ , with the last transition being  $\sigma_n \rightarrow \langle y \rangle$ . Either  $\sigma_n$  has the form  $\langle x' \rangle$ , or the form  $\langle b, x \rangle$ , in which case we know there is a rule  $R_i(x) \leftarrow R_i(x')$ . Thus, there is a corresponding edge  $(p_{x'}, p_x)$  in  $G_A$ , and since by the induction hypothesis  $x'$  is reachable, so is  $x$ . Otherwise,  $\sigma_n$  has the form  $\langle b, x \rangle$ , and by the fact that  $\langle b, x \rangle \rightarrow \langle x \rangle$ , it must be the case that  $x = ex_c$  for some exit  $ex_c$  of the component  $A_j$ , where  $Y_i(b) = j$ . In this case, consider the *longest suffix*  $\pi'$  of the path  $\pi$ , such that each state in  $\pi'$ , except the last state  $\langle ex_c \rangle$ , has a prefix  $b$ . Clearly, such a suffix  $\pi'$  must begin with a state  $\langle b, en_k \rangle$ . Then by the alternative characterization of  $R_i$ , and by the induction hypothesis, it must be the case that there is a path from  $\langle en_k \rangle$  to  $\langle ex_c \rangle$ , and hence (assuming, w.l.o.g., that  $A_j$  is single-exit) that  $R_j(en_k)$  holds. Thus, by the rule  $R_i(b, ex_c) \leftarrow R_i(b, en_k) \wedge R_j(en_k)$ , (and the corresponding edge in  $G_A$ ),  $p_{(b, ex_c)}$  is reachable in  $G_A$ .  $\square$

**THEOREM 5.** *Given RSM  $A$ , all predicates  $R_i$  can be computed in time  $O(|A|\theta^2)$  and space  $O(|A|\theta)$ . (More precisely, time  $O(e\theta + v\theta^2)$  and space  $O(e + v\theta)$ .)*

**Step 2: The augmented graph  $H_A$ .** Having computed the predicates  $R_i$ , for each component, we know the reachability among its entry and exit nodes. We need to determine the set of nodes reachable from the initial set  $Init$  in a global manner. For this, we build an ordinary graph  $H_A$  as follows. The set of vertices of  $H_A$  is  $V = \cup V_i$ , the set of vertices of all the components. The set of edges of  $H_A$  consists of all the edges of the components, and the following additional edges. For every box  $b$  of  $A_i$ , say  $b$  is mapped to  $A_j$ , include edges from the entry vertices  $(b, u)$  of  $b$  to the exit vertices  $(b, w)$  such that  $R_j(u, w)$  holds. Last, add an edge from each entry vertex  $(b, u)$  of a box to the corresponding entry node  $u$  of the component  $A_j$  to which  $b$  is mapped. The main claim about  $H_A$  is:

**LEMMA 6.**  *$u \Rightarrow v$  in RSM  $A$  if and only if  $v$  is reachable from  $u$  in  $H_A$ .*

Thus, to compute  $\{v \mid Init \Rightarrow v\}$ , all we need to do is a linear-time depth first search in  $H_A$ . Clearly,  $H_A$  has  $v$  vertices and  $e + v\theta$  edges. Thus we have:

**THEOREM 7.** *Given an RSM  $A$ , the set  $\{v \mid Init \Rightarrow v\}$  of reachable nodes can be computed in time  $O(|A|\theta^2)$  and space  $O(|A|\theta)$ .*

In invariant verification we are given RSM  $A$ , and a set  $T$  of target nodes, and want to determine if  $Init \Rightarrow v$  for some  $v \in T$ . This problem can be solved as above in the given complexity. Note that, unlike reachability in FSMs, this problem is PTIME-complete even for single-entry, non-recursive, RSMs [Alur and Yannakakis 2001].

For conceptual clarity, we have presented the reachability algorithm as a two-stage process. However, the two stages can be combined and carried out simultaneously, and this is what one would do in practice. In fact, we can do this *on-the-fly*, and have the reachability process drive the computation and trigger the rules; that is, we derive tuples involving vertices only when they are reached by the search procedure. This is especially important if the RSM  $A$  is not given explicitly but has to be generated from an implicit description dynamically on-the-fly. We defer further elaboration to the full article.

## 4.2 Checking Language Emptiness

Given an RBA  $A = (\langle A_1, \dots, A_k \rangle, Init, F)$ , we wish to determine whether  $L(A)$ ,  $L_b(A)$ , and  $L_u(A)$  are empty. Since  $L_b(A) \cup L_u(A) = L(A)$ , it suffices to determine emptiness for  $L_b(A)$  and  $L_u(A)$ . We need to check whether there are any bounded or unbounded accepting runs in  $B_A = (T_A, Init^*, F^*)$ . Our algorithm proceeds in the same two stage fashion as our algorithm for reachability. Instead of computing predicates  $R_i(x, y)$ , we compute a different predicate  $Z_i(x, y)$  with the same domain  $En_i \times V_i$  or  $V_i \times Ex_i$ , depending on whether  $A_i$  is entry- or exit-bound.  $Z_i$  is defined as follows:  $Z_i(x, y)$  holds if and only if there is a path in  $B_A$  from  $\langle x \rangle$  to  $\langle y \rangle$  that passes through an accept state in  $F^*$ . We can compute  $Z_i$ s by rules analogous to those for  $R_i$ s. In fact, having previously computed the  $R_i$ s, we can use that information to greatly simplify the rules governing

$Z_i$ s, so that the corresponding rules are linear and can be evaluated by doing reachability in an ordinary graph (instead of an AND-OR graph). The rules for an entry-bound machine  $A_i$  are as follows.

1.  $Z_i(x, y)$  , if  $R_i(x, y)$ , and  $x$  or  $y \in F^*$ ,  $x \in En_i$ ,  $y \in V_i$
2.  $Z_i(x, w) \leftarrow Z_i(x, u)$  , for  $x \in En_i$ ,  $(u, w) \in E_i$
- 3a.  $Z_i(x, (b, w)) \leftarrow Z_i(x, (b, u))$  , if  $R_j(u, w)$ ,  $x \in En_i$ ,  $b \in B_i$ ,  $Y_i(b)$   
 $= j$ ,  $u \in En_j$ ,  $w \in Ex_j$
- 3b.  $Z_i(x, (b, w)) \leftarrow Z_j(u, w)$  , if  $R_i(x, (b, u))$ ,  $x \in En_i$ ,  $b \in B_i$ ,  $Y_i(b)$   
 $= j$ ,  $u \in En_j$ ,  $w \in Ex_j$

The rules for exit-bound components  $A_i$  are similar. Let  $G'_A$  be an ordinary graph whose vertices are the possible ground atoms  $Z_i(x, y)$  and with edges  $(t_1, t_2)$  for each instantiated rule  $t_2 \leftarrow t_1$ . The set *Start* of initial vertices is the ground atoms from rule 1. Then the reachable vertices are precisely the set of true ground atoms  $Z_i(x, y)$ .  $G'_A$  has  $O(v\theta)$  vertices and  $O(e\theta + v\theta^2)$  edges. Again we do not need to construct it explicitly, but can store only its vertices and generate its edges as needed from the rules.

LEMMA 8. *All predicates  $Z_i$  can be computed in time  $O(|A|\theta^2)$  and space  $O(|A|\theta)$ .*

Having computed  $Z_i$ s, we can analyze the graph  $H_A$  for cycle detection. Let  $F_a$  be the set of edges of  $H_A$  of the form  $((b, x), (b, y))$ , connecting an entry vertex to an exit vertex of a box  $b$ , mapped say to  $A_j$ , and for which  $Z_j(x, y)$  holds. Let  $F_u$  be the set of edges of the form  $((b, x), x)$  where  $(b, x)$  is a vertex and  $x$  is an entry of the component to which box  $b$  is mapped (i.e., the edges that correspond to recursive invocations).

LEMMA 9. *The language  $L_u(A)$  is nonempty if and only if there is a cycle in  $H_A$  reachable from some vertex in *Init*, such that the cycle contains both: (1) an edge in  $F_a$  or a vertex in  $F$ , and (2) an edge in  $F_u$ .*

We need a modified version  $H'_A$  of  $H_A$  in order to determine emptiness of  $L_b(A)$  efficiently. The graph  $H'_A$  is the same as  $H_A$  except the invocation edges in  $F_u$  are removed. Also, the set of initial vertices need to be modified: let  $En'$  denote the vertices  $en$  of  $H_A$ , where  $en$  is an entry node of some component in  $A$ , and  $en$  is reachable from some vertex in *Init* in  $H_A$ .

LEMMA 10.  *$L_b(A)$  is nonempty if and only if there is a cycle in  $H'_A$  reachable from some vertex in  $En'$ , such that the cycle contains an edge in  $F_a$  or a vertex in  $F$ .*

Both  $H_A$  and  $H'_A$  have  $v$  vertices and  $O(e + v\theta)$  edges. We can check the conditions in the two lemmas in linear time in the graph size, using standard cycle detection algorithms.

THEOREM 11. *Given RBA  $A$ , we can check emptiness of  $L(A)$ ,  $L_b(A)$  and  $L_u(A)$  in time  $O(|A|\theta^2)$  and space  $O(|A|\theta)$ .*

### 4.3 Model Checking of LTL Properties and Büchi Automata

Following the automata-theoretic approach to model-checking [Vardi and Wolper 1986], a model-checking procedure for a formula  $\phi$  of linear-time temporal logic can be obtained by (1) building a finite-state Büchi automaton  $A_{\neg\phi}$  that accepts exactly all the infinite words satisfying the formula  $\neg\phi$ , (2) computing the product of  $A_{\neg\phi}$  with the system to be verified, and (3) checking if this product is nonempty. Hence to be able to check LTL properties, it suffices to be able to check properties given as a Büchi automaton, and any bound on the model checking of the latter in terms of the size of the RSM will apply to the former.

Thus we focus on the *automata-based model checking* problem whose input consists of an RSM  $A$  over  $\Sigma$  and an ordinary Büchi automaton  $B$  over  $\Sigma$ . The model checking problem is to determine, whether the intersection  $L(A) \cap L(B)$  is empty, or whether  $L_b(A) \cap L(B)$  ( $L_u(A) \cap L(B)$ ) is empty if we wish to restrict to bounded (unbounded) runs. Having given algorithms for determining emptiness of  $L(A)$ ,  $L_b(A)$ , and  $L_u(A)$  for RBAs, what model checking requires is a product construction, which given a Büchi automaton  $B$  and RSM  $A$ , constructs an RBA that accepts the intersection of the languages of the two.

The product RBA  $A' = A \otimes B$  of  $A$  and  $B$  is defined as follows.  $A'$  has the same number of components as  $A$ . For every component  $A_i$  of  $A$ , the entry-nodes  $En'_i$  of  $A'_i$  are  $En_i \times Q_B$ , and the exit-nodes  $Ex'_i$  of  $A'_i$  are  $Ex_i \times Q_B$ . The nodes  $N'_i$  of  $A'_i$  are  $N_i \times Q_B$  while the boxes  $B'_i$  are the same as  $B_i$  with the same label (that is, a box mapped to  $A_j$  is mapped to  $A'_j$ ). Transitions  $\delta'_i$  within each  $A'_i$  are defined as follows. Consider a transition  $(v, \sigma, v')$  in  $\delta_i$ . Suppose  $v \in N_i$ . Then for every transition  $(q, \sigma, q')$  of  $B$ ,  $A'_i$  has a transition  $((v, q), \sigma, (v', q'))$  if  $v'$  is a node, and a transition  $((v, q), \sigma, (b, (e, q')))$  if  $v' = (b, e)$ . The case when  $v = (b, x)$  is handled similarly. Repeating nodes of  $A'$  are nodes of the form  $(v, q)$  with  $q \in F_B$ . The construction guarantees that  $L(A \otimes B) = L(A) \cap L(B)$  (and  $L_b(A \otimes B) = L_b(A) \cap L(B)$  and  $L_u(A \otimes B) = L_u(A) \cap L(B)$ ). Analyzing the cost of the product, we get the following theorem:

**THEOREM 12.** *Let  $A$  be an RSM of size  $n$  with  $\theta$  as the maximum of minimum of entry-nodes and exit-nodes per component, and let  $B$  be a Büchi automaton of size  $m$  with  $a$  states. Then, checking emptiness of  $L(A) \cap L(B)$  and of  $L_{b,u}(A) \cap L(B)$ , can be solved in time  $O(n \cdot m \cdot a^2 \cdot \theta^2)$  and space  $O(n \cdot m \cdot a \cdot \theta)$ . For an LTL formula  $\phi$ , checking whether all paths through  $A$  satisfy  $\phi$  can be solved in time and space given the above, where now  $m = 2^{|\phi|}$ .*

## 5. BRANCHING TIME MODEL CHECKING FOR SINGLE-EXIT RSMs

In this section, we extend the results on linear-time model-checking to the branching time logic CTL\*, by presenting an algorithm for single-exit RSMs that runs in time linear in the size of the RSM. The algorithm is presented in the next subsection, while the remainder of the section is devoted to its correctness proof.

```

function DECOMP( $\phi$ : LTL formula ) returns Set of pairs ( $\beta \in \text{LTL}^+, \delta \in \text{LTL}$ )
[1] if ( $\phi = P$ ) then return ( $\{(P, true)\}$ )
[2] if ( $\phi = \phi_1 \vee \phi_2$ ) then return ( $\text{DECOMP}(\phi_1) \cup \text{DECOMP}(\phi_2)$ )
[3] if ( $\phi = \neg\phi_1$ ) then return ( $\bigcup_{A \subseteq \text{DECOMP}(\phi_1)} (\bigwedge_{(\beta, \delta) \in A} \neg\beta, \bigwedge_{(\beta, \delta) \in \text{DECOMP}(\phi_1) - A} \neg\delta)$ )
[4] if ( $\phi = X\phi_1$ ) then return ( $\bigcup_{(\beta, \delta) \in \text{DECOMP}(\phi_1)} (X\beta, \delta) \cup \{\{exit, \phi_1\}\}$ )
[5] if ( $\phi = \phi_1 U \phi_2$ ) then return ( $\bigcup_{\theta \neq A \subseteq \text{DECOMP}(\phi_1)} (G \bigvee_{(\beta, \delta) \in A} \beta, \bigwedge_{(\beta, \delta) \in A} \delta \wedge \phi_1 U \phi_2)$ 
 $\cup \bigcup_{\theta \neq A \subseteq \text{DECOMP}(\phi_1)} \bigcup_{(\beta', \delta') \in \text{DECOMP}(\phi_2)} (\bigvee_{(\beta, \delta) \in A} \beta U \beta', \delta' \wedge \bigwedge_{(\beta, \delta) \in A} \delta)$ )

```

Fig. 2. The function DECOMP.

### 5.1 The Branching-Time Algorithm

A key technical challenge is that the truth value of a temporal-logic formula in any state  $(\bar{b}, u)$  of the unfolding  $T_{\underline{A}}$  of an RSM may not only depend on the node  $u$  but also on the stack contents  $\bar{b}$ . Fortunately, it is sufficient to consider only finitely many equivalence classes of possible stack contents, each equivalence class being represented by a *context*, as already observed in Burkart and Steffen [1992, 1999]; Alur and Yannakakis [2001]. A context is a set of (here CTL\*) formulas whose truth value at the exit node of a machine  $A_i$  determine the truth value of a formula  $\phi$  at the root. The notion of context makes it possible to reason compositionally about RSMs.

Our algorithm exploits this idea and reduces the evaluation of a path formula  $\phi$  on a sequence  $w;w'$  of states, where  $w$  is finite while  $w'$  is infinite, to the evaluation of some formulas  $\beta$  and  $\delta$  on the sequences  $w$  and  $w'$ , respectively. We introduce a special atomic proposition *exit*, which holds only at the final state of a finite sequence  $w$ , and denote by  $\text{LTL}^+$  the set of LTL formulas that can be expressed using this extended set of atomic propositions. The function DECOMP given in Figure 2 specifies how the evaluation of an LTL formula  $\phi$  can be decomposed as described above. (A conjunction over an empty set of formulas is defined to have the value true.) For instance,  $w;w' \models Xp$  can be decomposed either into  $w \models Xp$  and  $w' \models true$  (for the case where  $|w| > 1$ ), or into  $w \models exit$  and  $w' \models p$  (for the case where  $|w| = 1$ ).

Given a set  $F$  of CTL\* state formulas, let  $\text{exists}(F)$  denote the set of existential formulas that are elements or subformulas of elements of  $F$ . A set  $F$  of existential CTL\* formulas is *closed* if, for every  $\gamma = E\rho(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n) \in \text{exists}(F)$ , for every  $\delta$  such that  $(\beta, \delta) \in \text{DECOMP}(\rho)$ ,  $E\delta(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n)$  is also in  $F$ . The *closure*  $\text{cl}(\phi)$  of a CTL\* formula  $\phi$  is the smallest closed set containing  $\text{exists}(\{\phi\})$ . One can show, using properties of DECOMP, that  $\text{cl}(\phi)$  is always finite for any CTL\* formula  $\phi$ . Let  $\text{pd}(\phi)$  be the maximal nesting of path quantifiers ( $E$ ) in a CTL\* formula  $\phi$ . Given a set  $F$  of CTL\* formulas, let  $\text{pd}(F) = \max_{\gamma \in F} (\text{pd}(\gamma))$ . For  $\phi$  with  $\text{pd}(\phi) \geq j$ , let  $\text{cl}^{\leq j}(\phi)$  be the elements of  $\text{cl}(\phi)$  with at most  $j$  nested path quantifiers. Clearly,  $\text{cl}^{\leq j}(\phi)$  is a closed set and  $\text{pd}(\text{cl}^{\leq j}(\phi)) = j$ .

For any closed set  $F$ , an  $F$ -*context* is any assignment of truth values to all elements of  $F$ . We say that a labeled transition system  $K$  with a fixed initial state  $s_0$  *satisfies an  $F$ -context  $C$* , written  $K \models C$ , if, for all  $\gamma \in F$ ,  $(K, s_0) \models \gamma$  if and only if  $C(\gamma) = true$ . An  $F$ -context is *consistent* if it is satisfied by some structure. All the  $F$ -contexts generated by our model-checking algorithm will be consistent by construction. We often identify an  $F$ -context with the elements

```

function CONT( $F$ : closed set of CTL*existential formulas,  $A$ : RSM over  $\{\gamma \in F : \text{pd}(\gamma) < \text{pd}(F)\}$ ,
 $C$ :  $F$ -CONTEXT) returns RSM over  $F$ .
/* We assume  $A = \{A_1, \dots, A_n\}$  with  $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$  */
1 for each  $\gamma \in F$  with  $\gamma = E \rho(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n)$  do /* Precompute LTL results needed */
2    $N(\gamma) = \text{LTLALG}(E \rho, A)$ 
3   for each  $(\beta, \delta) \in \text{DECOMP}(\rho)$ 
4      $N(\beta) = \text{LTLALG}(E (\beta \wedge F \text{exit}), A)$ 
5   for each  $A_i \in A$  do
6      $\text{Nodes}_1(A_i, \gamma) = N_i \cap N(\gamma)$ 
7     for each  $(\beta, \delta) \in \text{DECOMP}(\rho)$  do
8        $\text{Nodes}_2(A_i, \beta) = N_i \cap N(\beta)$ 
9     od
10  od
11   $\text{OLDCON} = \emptyset$  /* Find the pairs  $(A_i, c)$  reachable from  $(A_1, C)$  */
12   $\text{CON} = \bigcup_i \{(A_i, C)\}$ 
13  while ( $\text{CON} \neq \text{OLDCON}$ ) do
14     $\text{OLDCON} = \text{CON}$ 
15    for each  $(A_i, c) \in \text{OLDCON}$  do
16      for each  $\gamma \in F$  with  $\gamma = E \rho(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n)$ 
17         $\text{Sat}(A_i, c, \gamma) = \text{Nodes}_1(A_i, \gamma) \cup \bigcup_{(\beta, \delta) \in \text{DECOMP}(\gamma), c(E\delta(p(\gamma_1) \leftarrow \gamma_1, \dots, p(\gamma_n) \leftarrow \gamma_n)) = \text{true})} \text{Nodes}_2(A_i, \beta)$ 
18      for each  $b \in \text{Boxes}(A_i)$  do
19         $\text{Prop}(b, (A_i, c)) = \{Y_i(b), c'\}$  such that  $\forall \gamma \in F : c'(\gamma) = \text{true}$  iff  $(b, x) \in \text{Sat}(A_i, c, \gamma)$ 
20       $\text{CON} = \text{OLDCON} \cup \{\text{Prop}(b, (A_i, c))\}$ 
21      od
22    od
23  od
24  /* Now build the output RSM  $A^*$ 
25   $A^* = \{A_{i,c} \mid 1 \leq i \leq n \text{ and } c \in \text{CON}\}$ 
26  forall  $1 \leq i \leq n$ , for all  $c \in \text{CON}$ ,
27     $A_{i,c} = (N_i \times \{c\}, B_i \times \{c\}, Y'_{i,c}, En_i \times \{c\}, Ex_i \times \{c\} \delta'_i)$  where
28    for all  $b \in B_i$ ,  $Y'_{i,c}((b, c)) = (Y_i(b), c')$  with  $(A_k, c') = \text{Prop}(b, (A_i, c))$ 
29     $E'_i = \{((u, c), (v, c)), (u, v) \in E_i\}$ 
30    for all  $(u, v) \in E'_i$ ,  $\Sigma'_i((u, c), (v, c)) = \{\gamma \in F \mid v \in \text{Sat}(A_i, c, \gamma)\}$ 
31  return( $A^*$ )

```

Fig. 3. Construction of the context-dependent RSM.

set to true by it. For an RSM  $A$ , a node  $v \in A$ , an  $F$ -context  $C$ , and a formula  $\gamma \in F$ , we say  $(A, v)$  *satisfies  $\gamma$  in context  $C$* , written  $(A, v) \models_C \gamma$ , if, for all  $K$ ,  $K \models C \Rightarrow ((T_A; K), v) \models \gamma$ , where  $T_A; K$  is the labeled transition system obtained from  $T_A$  by identifying the top-level exit node of  $T_A$  with the initial state of  $K$ .

Given a closed set  $F$  of existential formulas, an RSM  $A$  whose nodes are labeled with formulas in  $\{\gamma \in F \mid \text{pd}(\gamma) < \text{pd}(F)\}$ , and an  $F$ -context  $C$ , the function CONT presented in Figure 3 constructs a new RSM  $A^*$  from multiple copies of  $A$ , each of which is indexed by an  $F$ -context  $c$ . The edges of  $A^*$  in copy  $(A_j, c)$  are supplementarily labeled by state formulas  $\gamma \in F$  representing the truth value of  $\gamma$  in the corresponding node of  $A$  in the context  $c$ . It will be shown below that an edge of the form  $((u, c), (v, c))$  in  $A^*$  is labeled with  $\gamma \in F$  if and only if  $(M, u) \models_c \gamma$  (hence, in particular, for labels other than the atomic action symbols, the label of an edge will depend only on the source of the edge).

In algorithm CONT, we will abuse notation somewhat by having the function  $Y_i$  in input and output map a box to a component, rather than its index.

CONT uses a variant of the LTL model-checking algorithm from Section 4, called LTLALG. Given a formula of the form  $E \rho(p(\gamma_1), \dots, p(\gamma_n))$  where  $\rho$  is an LTL<sup>+</sup> formula over atomic propositions including  $p(\gamma_1), \dots, p(\gamma_n)$ , and an RSM  $A$  whose edges are also labeled with propositions in  $p(\gamma_1), \dots, p(\gamma_n)$ , LTLALG( $E \rho, A$ ) returns the set of nodes  $v$  of  $A$  such that  $(v, \epsilon) \models E \rho$ . This

```

function CHECK( $\phi$ : existential CTL* formula,  $A$ : single-exit RSM,  $C$  :  $\text{cl}(\phi)$ -CONTEXT)
  returns set of nodes in  $A_1$ 
[1] begin
[2]    $A^0 = A$ 
[3]   for  $\{j = 0; j < \text{pd}(\phi); j++\}$ 
[4]      $A^{j+1} = \text{CONT}(\text{cl}^{\leq j+1}(\phi), A^j, C \cap \text{cl}^{\leq j+1}(\phi))$ 
[5]   return  $\{v \in (A^{\text{pd}(\phi)}) \mid \text{Label}(v) \text{ includes } \phi\}$ 
end

```

Fig. 4. CTL\* model-checking algorithm.

is done exactly as described in Section 4, except for the following three modifications. First, LTLALG evaluates formulas of the form  $E\rho$  instead of  $A\rho$ . Second, we still need to define how formulas of LTL<sup>+</sup> are evaluated on  $A$ : we say that a formula  $E\rho$  where  $\rho$  is in LTL<sup>+</sup> is satisfied in a node  $v$  of a machine  $A_i$  if there is a path  $w$  from  $(v, \epsilon)$  that satisfies  $\rho$ , such that either  $w$  is infinite or  $w$  terminates at  $(x, \epsilon)$ , where  $x$  is the exit node of  $A_i$ . Third, we also extend the evaluation of formulas to include “return vertices” (pairs  $(b, x)$ ). We say that the return vertex  $(b, x)$  satisfies a formula  $E\rho$  if and only if the exit node  $x$  of box  $b$  satisfies  $E\rho$  when  $b$  is the only element of the stack; in other words, we define  $\langle \epsilon, (b, x) \rangle \models E\rho$  if and only if  $\langle b, x \rangle \models E\rho$ . It is easy to extend the LTL model-checking algorithm of Section 4 to meet these additional requirements.

By repeatedly invoking CONT with  $\text{cl}^{\leq j}(\phi)$  for increasing values of  $j$ ,  $1 \leq j \leq \text{pd}(\phi)$ —larger and larger subsets of  $\text{cl}(\phi)$ —one can thus evaluate CTL\* formulas in a bottom-up manner. This is what is done in function CHECK presented in Figure 4. Since any CTL\* formula  $\phi$  is a Boolean combination of existential formulas  $\phi_i$ , finding the vertices of a component  $A_j$  of an RSM  $A$  satisfying  $\phi$  can be reduced to finding the vertices of  $A_j$  satisfying each  $\phi_i$ . This is done by computing CHECK( $\phi_i, A, C_\emptyset$ ) where  $C_\emptyset$  is the set of formulas  $\gamma$  in  $\text{cl}(\phi_i)$  that evaluate to true at a single node with a self-loop. Since  $C_\emptyset$  is consistent, all subcontexts derived from it during the execution of the algorithm are also consistent. The correctness of the algorithm is established by the following theorem, proved later on in this section.

**THEOREM 13.** *Given a single-exit RSM  $A$ , a node  $v$  of some machine  $A_j$ , and an existential CTL\* formula  $\phi_i$ ,  $(v, \epsilon)$  satisfies  $\phi_i$  if and only if  $v$  is included in the set of nodes returned by CHECK( $\phi_i, A, C_\emptyset$ ).*

An analysis of the overall complexity of CHECK reveals that the number of contexts over  $F = \text{cl}(\phi)$  and the number of pairs of formulas returned by DECOMP on these formulas depends only on  $\phi$ . This implies that the size of each  $A^j$  in Figure 4 is linear in  $A$  for any fixed  $\phi$ . Moreover, the number of formulas on which the LTL algorithm is invoked in CONT is bounded independently of the size of  $A$ . Hence, the run-time complexity of the function CONT and the size of the returned RSM  $A^*$  are linear in the input RSM  $A$  for any fixed formula  $\phi$  and closed set  $F$ . Therefore, *the CTL\* model-checking problem for a single-exit RSM  $A$  can be solved in time linear in the size of  $A$ .*

## 5.2 Correctness Proofs

**5.2.1 Proof of Correctness of Algorithm DECOMP.** The following theorem summarizes the property we need of algorithm DECOMP:

**THEOREM 14.** *For any  $\phi \in LTL$  for any  $\omega$ -string  $\rho$ , for any finite nonempty path  $\pi$  and infinite path  $\gamma$ ,  $\pi.\gamma \models \phi$  if and only if there are  $(\beta, \delta) \in DECOMP(\phi)$ , such that  $\pi \models \beta$  and  $\gamma \models \delta$ .*

**PROOF.** By induction on the structure of  $\phi$ . For atomic  $P$  it is clear that  $\pi.\gamma \models P$  if and only if  $\pi \models P$  for  $\pi$  nonempty. The disjunction case is routine. For the negation case, we have  $\pi.\gamma \models \neg\phi$  if and only if there is no  $(\beta, \delta) \in DECOMP(\phi)$  such that  $\pi \models \beta$  and  $\gamma \models \delta$ . This is a Boolean combination of statements about  $\pi$  and  $\gamma$ , and putting this into disjunctive normal form, one arrives at the expression in line [3] of *DECOMP*. For the  $X$  case, we have  $\pi.\gamma \models X\phi_1$  if and only if one of the following holds: either  $\pi = a.\pi'$  with  $|dom(a)| = 1$  and  $\pi'.\gamma \models \phi_1$ , or  $|\pi| = 1$  and  $\gamma \models \phi_1$ . From this and induction, we get equality with line [4] of *DECOMP*.

For the until case, suppose we have  $\pi.\gamma \models \phi_1 U \phi_2$ . We first show that this implies that  $\pi$  and  $\gamma$  satisfy one of the “disjuncts” (the two sets being unioned) of line [5] of *DECOMP*. For every  $n < |\pi|$ , let  $\pi_n$  be the suffix of  $\pi$  starting at  $n$ .

Case 1: Suppose that there is  $n$  such that  $\forall i < n \pi_i.\gamma \models \phi_1$  and  $\pi_n.\gamma \models \phi_2$ . By induction, for each  $i$  there is  $(\beta_i, \delta_i) \in DECOMP(\phi_1)$  such that  $\pi_i \models \beta_i$  and  $\gamma \models \delta_i$ . Likewise, there is  $(\beta', \delta') \in DECOMP(\phi_2)$  such that  $\pi_n \models \beta'$  and  $\gamma \models \delta'$ . Let  $A = \{(\beta_i, \delta_i) : i < n\}$ , we see that the second disjunct in line [5] is satisfied.

Case 2: If not case 1, then it must be that  $\forall i \pi_i.\gamma \models \phi_1$  and  $\gamma \models \phi_1 U \phi_2$ . By induction, choose  $(\beta_i, \delta_i) \in DECOMP(\phi_1)$  such that  $\pi_i \models \beta_i$  and  $\gamma \models \delta_i$ . Setting  $A = \{(\beta_i, \delta_i) : i < n\}$  we have a witness for the first disjunct in line [5].

Conversely, we prove that if  $\pi.\gamma$  satisfies either of the disjuncts in line [5] then it satisfies  $\phi_1 U \phi_2$ . If it satisfies the first disjunct, let nonempty  $A \subset DECOMP(\phi_1)$  witness this. Clearly, we have  $\gamma \models \phi_1 U \phi_2$ . Let  $\pi_n$  be as above. We claim that  $\forall n \pi_n.\gamma \models \phi_1$ , which would suffice to prove the conclusion, since this says  $\pi.\gamma$  satisfies  $\phi_1$  at all times up to  $|\pi|$  and afterwards  $\pi.\gamma$  satisfies  $\phi_1 U \phi_2$ . But this claim follows by induction, because for some  $(\beta, \delta) \in DECOMP(\phi_1)$   $\pi_n \models \beta$  and  $\gamma \models \delta$ . Now suppose  $\pi.\gamma$  satisfies the second disjunct, and let  $\emptyset \neq A \subset DECOMP(\phi_1)$  and  $(\beta', \delta') \in DECOMP(\phi_2)$  witness this. Choose  $(\beta_1, \delta_1) \in A$  such that  $\pi \models \beta_1 U \beta'$ . Let  $\pi_n$  be as above, and consider any  $n$  such that  $\pi_n$  satisfies  $\beta_1$ . Since  $\gamma \models \delta_1$  (by induction, since it satisfies every  $(\beta, \delta) \in A$ ), we have  $\pi_n.\gamma \models \phi_1$ . For any  $\pi_n$  satisfying  $\beta'$ , we have  $\pi_n.\gamma \models \phi_2$ , since  $\gamma \models \delta'$ . So using  $\pi \models \beta_1 U \beta'$  we conclude  $\pi.\gamma \models \phi_2$ . This completes the proof of Theorem 14.  $\square$

**5.2.2 Correctness Statement for CHECK and Preliminaries.** Recall that in the machine  $A^k$  edges are labeled with collections of state formula (in addition to their action labels). For these new labels, the value of the label depends only on the source of the edge, hence we will talk equivalently about the label of a vertex in  $A^k$ . The main correctness result for branching time is the following:

**THEOREM 15.**  $\forall (n, C) \in A^k, \forall \phi \in F : (n, C)$  is labeled with  $\phi$  if and only if  $(A, v) \models_C \phi$ .

This theorem says that we can read off the result of any context-dependent model-checking problem by just looking at labels in  $A^k$ . Before we start the proof, we will need to build up some facts about the construction in algorithms

CONT and CHECK used in the proof. We will first get a nice representation of vertices and component machines in the context-dependent RSM  $A^k$  produced by CHECK. We will then get information on the labeling of edges in  $A^k$ .

First notice (line [27] of CONT) that the RSM  $A^* = CONT(A)$  has component machines of the form  $(A_i, C)$  where  $C$  is a  $k$ -context and  $A_i$  is a component of the initial machine  $A$ , with each  $(A_i, C)$  being a copy of  $A_i$ , with vertices of the form  $(n, C)$  for  $n \in A_i$ . When we iterate this process to get  $A^k$ , this will result in components of the form  $((\dots((A_i, C_1), C_2), C_3)\dots, C_k))$  and vertices of the form  $n^k = (((\dots((n, C_1), C_2), C_3)\dots, C_k))$  for  $n \in A$ . Given  $j < k$  we let  $\pi_{kj}$  be the projection map taking  $n^k$  of the above form to  $((\dots((n, C_1), C_2), C_3)\dots, C_j)$ . The following observation will show that we do not actually have to deal with such a messy representation:

**CLAIM 1.** *For any component of the above form occurring in  $A^k$ , the  $C_i$  must be consistent, in the sense that  $C_j | \text{dom}(C_i) = C_i$ . Furthermore, the map  $\pi_{kj}$  preserves labels, in the sense that if a node  $n^k$  is labeled with  $\phi$  in  $A^k$  and  $pd(\phi) < k$ , then  $\pi_{kj}(n^k)$  is labeled with  $\phi$  as well.*

This claim implies that the components of lower index than  $k$  are determined by the component  $C_k$ , hence we can without loss of generality treat a component machine in  $A^k$  as being of the form  $(A_i, C_k)$  with  $C_k$  a  $k$ -context and  $A_i$  a component of the original machine. Recalling that nodes  $v^k$  of  $A^k$  have the form  $((\dots((n, C_1), C_2), C_3)\dots, C_k))$ , for  $n \in A_i$  where  $v^k \in ((\dots((A_i, C_1), C_2), C_3)\dots, C_k))$  it also follows from the claim that we can simplify the representation of nodes in  $A^k$ : they can be taken to be of the form  $(n, C_k)$  where  $n \in A$ .

Claim 1 is in turn implied by applying inductively the following two facts about the CONT algorithm:

- For  $F$  and  $A$  as in CONT, for any node  $(m, C) \in M^*$  and  $\phi$  with  $pd(\phi) < pd(F)$ , if  $m$  is labeled with  $\phi$  in  $A$  then  $(m, C)$  is labeled with  $\phi$  in  $A^*$  (assignments are consistent).
- Suppose we have a call vertex  $(c, C)$  on box  $(b, C)$  in component machine  $(A_i, C) \in M^*$ , calling machine  $(A_j, D) \in M^*$ . Then  $c$  calls  $A_j$  from  $A_i$  in  $A$ , and  $D$  is exactly the label of  $((b, x), C)$  in  $A^*$ .

Note that the first item shows that the labeling is consistent, and the second shows that every context used in  $A^*$  corresponds to some labeling in  $A^*$ , hence the claim follows from these two items. The first item follows by tracing back what happens to a simple formula  $\gamma = E P(\gamma_1)$  with  $pd(\gamma_1) < pd(F)$  in lines [2]–[8] (in CONT, as always when we give line numbers in this subsection): we have  $N(\gamma)$  is just  $\{n \in M : n \text{ is labeled with } P(\gamma_1)\}$ , and  $Nodes_1(A_i, \gamma)$  is just the above set intersected with  $N_i$ . From this it follows that at line [17],  $Sat(A_i, c, \gamma)$  will again be just the nodes labeled with  $P(\gamma_1)$ . Now at line [30], we see that a node  $(m, c)$  is labeled with  $\gamma$  of the form above if and only if  $m$  is labeled with  $P(\gamma_1)$ .

The second item above follows from comparing line [30] (referring back to [19]) and line [28] for the case where  $v$  is an exit node. Line [30] says that

$((b, x), C)$  is labeled with  $\{\gamma : (b, x) \in \text{Sat}(A_i, C, \gamma)\}$ , and line [28] combined with [19] says that  $(b, C)$  calls  $(A_k, C')$  with  $C' = \{\gamma : (b, x) \in \text{Sat}(A_i, C, \gamma)\}$ .

This completes the proof of Claim 1.

So given the now simplified form of the elements of  $A^k$  as  $(m, C)$ , we let  $\eta$  be the projection map taking a node  $(m, C)$  in  $A^k$  to the node  $m \in A$ . The mapping  $\eta$  maps nodes in  $A^k$  back to nodes in  $A$ , while conversely if we are given a path  $\pi$  starting at some node  $n$  in  $A$  and an initial context  $C$ , we can get a path through  $A^k$  that starts at  $(n, C)$ . Formally, for every  $k$ -context  $C$  there is a unique mapping taking a finite path  $\pi$  in  $T_A$  to a path  $\pi^{k,C}$  in  $A^k$  such that:  $\pi(0) \in A_i \leftrightarrow \pi^{k,C}(0) \in (A_i, C)$  and  $\eta(\pi^{k,C}(n)) = \pi(n)$ .

Recall that nodes in  $A^k$  are labeled with formulas  $\phi$ . We give two conditions **(C1)** and **(C2)** below that characterize how the labeling of formulae  $E \rho(p(\lambda_1) \leftarrow \lambda_1, \dots, p(\lambda_q) \leftarrow \lambda_q)$  relates to the labeling by  $\lambda_i$  in  $A^k$ .

**LEMMA 16.** *Let  $\phi = E \rho(p(\lambda_1) \leftarrow \lambda_1, \dots, p(\lambda_q) \leftarrow \lambda_q)$  where  $\rho$  is in LTL with  $q$  propositional variables and  $\text{pd}(\lambda_i) \leq k - 1$ , and let  $(n, C)$  be a node in machine  $(M, C) \in M^k$ . Then  $(n, C)$  is labeled by  $\phi$  in  $A^k$  if and only if one of the following holds:*

- *There is an infinite path  $\pi$  from  $(n, C)$  through  $A^k$  whose labels satisfy  $\rho$  **(C1)***
- *There is a finite path  $\pi_0$  from  $(n, C)$  leading to the exit node of  $(M, C)$  with empty stack, and formulae  $(\beta, \delta) \in \text{DECOMP}(\rho)$  such that  $\pi_0$  satisfies  $\beta$  and  $E \delta(p(\lambda_1) \leftarrow \lambda_1, \dots, p(\lambda_q) \leftarrow \lambda_q) \in C$  **(C2)***

**PROOF.** To see that nodes satisfying condition **(C1)** are labeled with  $\phi$ , observe that the construction lines [2],[6],[17] includes in  $\text{Sat}(A_i, c, \gamma)$  all nodes that satisfy  $\text{LTLALG}(\rho)$ . So by correctness of the LTL algorithm, and the way  $\text{Sat}$  determines the labeling in [30], we see that nodes with paths satisfying the first item above are included in the label. Similarly, lines [3]-[4],[7]-[8] and [17] guarantee that  $\text{Sat}(A_i, C, \gamma)$  includes all nodes that satisfy  $\text{LTLALG}(E(\beta \wedge \text{Exit}))$  for  $(\beta, \delta) \in \text{DECOMP}(\rho)$  with  $E \delta$  in  $C$ . The fact that the nodes satisfying condition **(C2)** are labeled with  $\phi$  now follows from the correctness of the LTL algorithm. It is also easy to check that every node labeled with  $\phi$  satisfies **(C1)** or **(C2)**.

We give one final fact about the  $A^k$  which follows from the argument in Claim 1, which we will use implicitly in the sequel:  $\square$

**PROPOSITION 17.** *A call vertex  $(A_i, c)$  from machine  $(A_i, C)$  in  $A^k$  goes to  $(A_j, D)$  if and only if  $c$  goes from  $A_i$  to  $A_j$  in  $A$  and the return vertex  $(r, C)$  of  $(A_i, C)$  is labeled with (exactly) the formulas in  $D$  in  $(A_i, C)$  **(P3)***

**5.2.3 Proof of Theorem 15.** We will now prove the following result  $\pi(k)$ , by induction on  $k$ :

**CLAIM 2.**  $\forall C \in \text{Context}(k) \forall n \forall \phi$  with  $\text{pd}(\phi) \leq k \forall$  path  $\pi$  with  $\pi(0) \in A_i$ , and  $\forall K$  with  $K \models C$ , suppose that  $D$  and  $j$  are such that  $\pi^{k,C}(n) \in (A_j, D)$ , and let  $s$  be the stack at  $\pi^{k,C}(n)$ . Then:

*For all  $m \in A_j (T_A; K, \langle s, m \rangle) \models \phi$  if and only if  $(m, D)$  is labeled with  $\phi$  in  $A^k$ .*

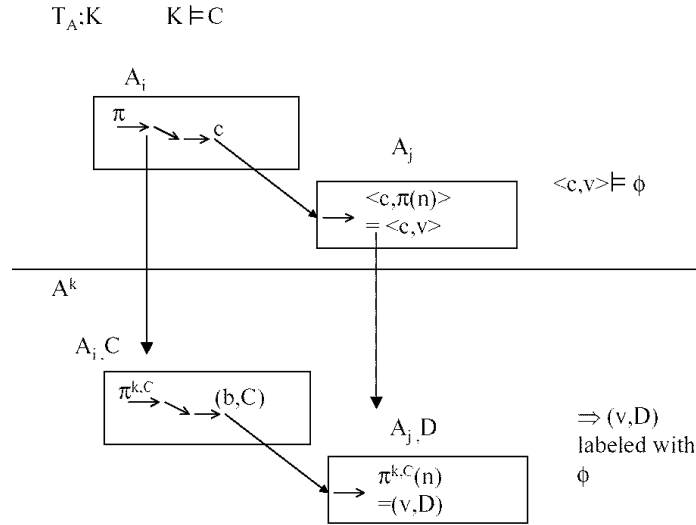


Fig. 5. The situation of Claim 2.

Figure 5 shows a simplified version of the situation we are in here: we have a path  $\pi$  going through the infinite LTS  $T_A; K$ , where  $K$  satisfies the context  $C$ . This path may go through a call vertex  $c$  (or several) and  $\pi(n)$  may arrive in another machine  $A_j$  with  $c$  on the stack. The original path in  $T_A; K$  is shown above the line in the picture, while the corresponding path through  $A^k$  is shown below the line. The lifted path  $\pi^{k,C}$  must go through calls every time  $\pi$  does, going through  $(c, C)$  to get to machine  $(A_j, D)$  and arriving at time  $n$  at  $(v, D)$  for some other context  $D$ . At this point we compare the formulae satisfied by the node  $\langle c, v \rangle$  (in the general setting  $\langle c_1 \dots c_n, v \rangle$ ) in the unfolding  $T_A; K$  with the labels on  $(v, D)$  in  $A^k$ . The Claim asserts that these two sets of formulae are the same.

To see how this claim proves Theorem 15, fix  $\phi$  and  $C$ , and for any node  $m$  consider a path with  $\pi(0) = m$ . Then we have  $(T_A; K, \langle \emptyset, m \rangle) \models \phi$  if and only if  $(m, C)$  is labeled with  $\phi$  in  $A^k$ . Hence  $(m, C)$  is labeled with  $\phi$  in  $A^k$  if and only if  $A, m \models_C \phi$ .

To start the inductive proof, assume that the above holds for  $k$ . We prove the same result for  $k + 1$ . This we prove by a nested induction, on the *stack depth*  $l$  of  $\pi(n)$  (the height of the stack at  $\pi(n)$ ). Suppose we have this up to  $l$  and let  $n$  be minimal such that  $\pi(n)$  has nesting depth  $l + 1$  but the result doesn't hold. Fix  $C, K, \phi$  and  $\pi$  that witness this. Let  $s$  be the stack at  $\pi(n)$ . So we have  $\pi(n) \in A_j$  with  $(T_A; K, \langle s, \pi(n) \rangle) \models \phi$  but  $\pi^{k,C}(n)$  is not labeled with  $\phi$  in  $A^k$ .  $\pi(n)$  must be the start node  $s_0$  of machine  $A_j$  called from a call vertex  $c$  in a machine  $A_i$  (since otherwise  $\pi(n - 1)$  would be the minimal counterexample, since the statement refers to all nodes in the same machine). Hence  $s = s'.c$ , and  $\pi(n - 1)$  must be the predecessor of a call  $c$  to  $A_j$ , with return vertex  $r$ . So  $\pi^{k,C}(n)$  is in some machine  $(A_j, D) \in M^k$  while  $\pi^{k,C}(n - 1)$  is in  $(A_i, E)$ . Since  $\pi(n - 1)$  has stack depth  $k$ , we know by the inner induction that for all  $\rho$  with  $pd(\rho) \leq k \forall r \in A_i (r, E)$  is labeled with  $\rho$  if and only if  $(T_A; K, \langle s', r \rangle) \models \phi$ .

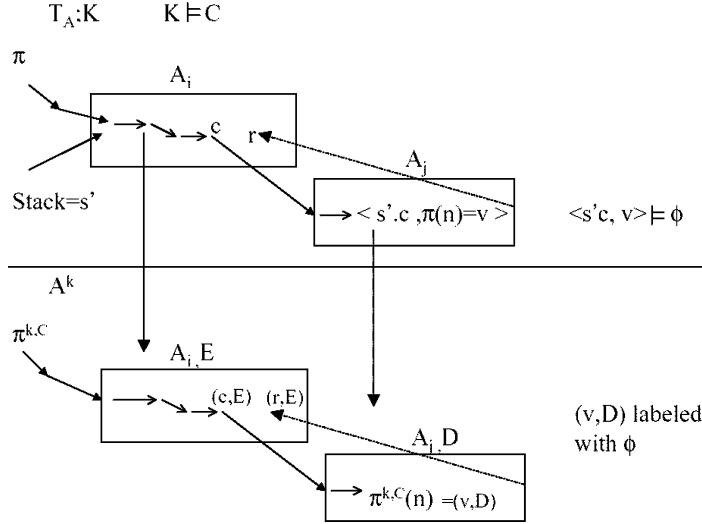


Fig. 6. The induction step of Claim 2.

Applying this to the return vertex  $r$  and noting that  $(r, E)$  is the return vertex for  $(c, D)$  in  $A^k$ , the node linked to by  $\pi^{k,C}(n-1)$ , we see by the construction of  $A^k$  (see comments at beginning of previous subsection) that  $(r, E)$  is the return vertex for  $\pi^{k,C}(n)$ . This in turn implies by **(P3)** that  $D = \{\rho : (r, E) \text{ is labeled with } \rho\}$ . Putting these facts together we have  $D = \{\rho : (T_A; K, \langle s', r \rangle) \models \phi\}$ . The situation is now illustrated by Figure 6.

Checking back what it means for  $\pi(n)$  to be a counterexample, we know that there is a node  $m \in A_j$ , such that  $(T_A; K, \langle s, m \rangle) \models \phi$  but  $(m, D)$  is not labeled with  $\phi$ . Note that  $(T_A; K, \langle s, m \rangle) \models \phi$  if and only if  $(T_A; K', \langle \emptyset, m \rangle) \models \phi$ , where  $K'$  is the machine  $T_A; K$  initialized at  $\langle s', r \rangle$ . By the last paragraph we know that the formulae of  $pd(k+1)$  satisfied by  $K'$  are exactly  $D$ .

Write  $\phi$  as  $E \rho(p(\lambda_1) \leftarrow \lambda_1, \dots, p(\lambda_q) \leftarrow \lambda_q)$  where  $\rho$  is in LTL and  $pd(\lambda_i) \leq k$ . By assumption, we know  $(T_A; K, \langle s, m \rangle) \models \phi$ , so there is a path  $\pi$  that witnesses this. We now divide up into cases depending on whether  $\pi$  leaves  $T_A$  or not.

**Case 1:** The stack on  $\pi$  never shrinks below  $s$  (i.e.  $\pi$  never emerges back through  $r$ ).

For all  $v \in N(T_A; K', \langle \emptyset, \pi(v) \rangle) \models \lambda_i$  if and only if in  $A^k$   $\pi^{D,k}(v)$  is labeled with  $\lambda_i$ , by the outer induction, the fact that all the nodes lie in  $T_A$ , the fact that  $K' \models D$ , and the equivalence of  $(T_A; K, \langle s, m \rangle)$  and  $(T_A; K', \langle \emptyset, m \rangle)$ . Hence if we consider  $\pi^{D,k}$ , it is a path starting at  $(m, D)$  whose labels satisfy  $\rho$  in  $A^k$ . But then, using the condition (C1) in Lemma 16 above,  $m$  is labeled with  $E \rho(p(\lambda_1) \leftarrow \lambda_1, \dots, p(\lambda_q) \leftarrow \lambda_q)$  in  $A^k$ , which is what we wanted.

**Case 2:**  $\pi$  exits  $(A_j, D)$  through  $r$ .

Let  $\pi_0$  be the part of  $\pi$  before the exit, and  $\gamma_0$  be the (infinite) suffix of  $\pi_0$  in  $\pi$ , considered as a path. By the definition of DECOMP and Theorem 14, there is  $(\beta, \delta) \in DECOMP(\rho)$  such that  $\pi_0$  witnesses  $(T_A; K, \langle s, m \rangle) \models E \beta(p(\lambda_1) \leftarrow \lambda_1, \dots)$  and  $\gamma_0$  witnesses  $(T_A; K, \langle s', r \rangle) \models E \delta(p(\lambda_1) \leftarrow \lambda_1, \dots)$ .

We now know from the above that  $D$  contains  $E \delta(p(\lambda_1) \leftarrow \lambda_1, \dots)$ .

As above, we can argue that for all  $v < |\pi_0|$   $(T_A; K', (\emptyset, \pi(v))) \models \lambda_i$  if and only if in  $A^k$   $\pi_0^{D,k}(v)$  is labeled with  $\lambda_i$ . Hence if we consider  $\pi_0^{D,k}$ , it is a path starting at  $(m, D)$  whose labels satisfy  $\beta(p(\lambda_1) \leftarrow \lambda_1 \dots)$  in  $A^k$ . But then, by the property **(C1)** of the construction listed above,  $(m, D)$  is labeled with  $E \beta(p(\lambda_1) \leftarrow \lambda_1, \dots p(\lambda_q) \leftarrow \lambda_q)$  in  $A^k$ . Using the fact that  $(\beta, \delta) \in \text{DECOMP}(\rho)$  and condition **(C2)** from Lemma 16, we have that  $(m, D)$  is labeled with  $E \rho(p(\lambda_1) \leftarrow \lambda_1, \dots p(\lambda_q) \leftarrow \lambda_q)$ , which is what we needed.

This completes the proof of Theorem 15.

## 6. DISCUSSION

**Efficiency and Context-Free Reachability:** Given a recursive state machine of size  $n$  with  $\theta$  maximum entry/exit-nodes per component, our reachability algorithm takes time  $O(n \cdot \theta^2)$  and space  $O(n\theta)$ . It is unlikely that our complexity can be substantially improved. Consider the standard parsing problem of testing CFL-membership: for a fixed context-free grammar  $G$ , and given a string  $w$  of length  $n$ , we wish to determine if  $G$  can generate  $w$ . The classic C-K-Y algorithm for this problem requires  $O(n^3)$  time. Using fast matrix multiplication, Valiant [1975] was able to slightly improve the asymptotic bound, but his algorithm is highly impractical. A related problem is CFL-reachability [Yannakakis 1990; Reps 1998], where for a fixed grammar  $G$ , we are given a directed, edge-labeled, graph  $H$ , having size  $n$ , with designated source and target nodes  $s$  and  $t$ , and we wish to determine whether  $s$  can reach  $t$  in  $H$  via a path labeled by a string in  $L(G)$ . CFL-membership is the special case of this problem where  $H$  is just a simple chain graph whose edges are labeled by the symbols of  $w$ . Unlike CFL-membership, CFL-reachability is  $P$ -complete, and the best known algorithms require  $\Omega(n^3)$  time [Yannakakis 1990]. Using the close correspondence between recursive machines and context-free grammars, a grammar  $G$  can be translated to a recursive state machine  $A_G$ . To test CFL-reachability, we can take the product of  $A_G$  with  $H$ , and check for reachability. The product has size  $O(n)$ , with  $O(n)$  entry-nodes per component, and  $O(n)$  exit-nodes per component. Thus, since our reachability algorithm runs in time  $O(n^3)$  in this case, better bounds on reachability for recursive state machines would lead to better-than-cubic bounds for parsing a string and for CFL-reachability, as well as for a variety of other automaton problems and program-analysis problems [Heintze and McAllester 1987; Melski and Reps 2000; Reps 1998].

Section 4 presented a method based on AND-OR graphs to check properties of recursive state machines when properties are expressed in linear-time temporal logic. An alternative method, based directly on CFL-reachability, was given by Benedikt, Godefroid, and Reps [Benedikt et al. 2001] their LTL model-checking algorithm makes use of a method for detecting cyclic paths that meet a CFL condition.

**Extended Recursive State Machines:** In the presence of variables, our algorithms can be adopted in a natural way by augmenting nodes with the values of the variables. Suppose that we have an extended recursive state machine  $A$  with Boolean variables (similar observations apply to more general finite domains), and the edges have guards and assignments that read/write these variables.

Suppose each component refers to at most  $k$  variables (local or global), and that it has at most either  $d$  input variables or  $d$  output variables (i.e., global variables that it reads or writes, or parameters passed to and from it). Then, we can construct a recursive state machine of size at most  $2^k \cdot |A|$  with the same number of components. The derived state machine has  $\theta = 2^d$ . Thus, reachability problems for such an extended recursive state machine can be solved in time  $O(2^{k+2d} \cdot |A|)$ . Note that such extended recursive state machines are basically the same as the Boolean programs of Ball and Rajamani [2000].

**Concurrency:** We have considered only sequential recursive state machines. Recursive state machines define context-free languages. Consequently, it is easy to establish that typical reachability analysis problems for a system of communicating recursive state machines are undecidable. Our algorithms can however be used in the case when only one of the processes is a recursive state machine and the rest are ordinary state machines. To analyze a system with two recursive processes  $M_1$  and  $M_2$ , one can possibly use abstraction and assume-guarantee reasoning: the user constructs finite-state abstractions  $P_1$  and  $P_2$  of  $M_1$  and  $M_2$ , respectively, and we algorithmically verify that (1) the system with  $P_1$  and  $P_2$  satisfies the correctness requirement, (2) the system with  $M_1$  and  $P_2$  is a refinement of  $P_1$ , and (3) the system with  $P_1$  and  $M_2$  is a refinement of  $P_2$ .

**Other Related Work:** Systems of hierarchically structured equations were used by Cousot and Cousot [1978] and Sharir and Pnueli [1981] to specify and solve interprocedural dataflow-analysis problems. Although these algorithms are not based on graph reachability, Sharir and Pnueli used an explicit, hierarchically structured, single-entry/single-exit graph to represent the program, and introduced a context-free language constraint to specify that the contributions of certain kinds of infeasible execution paths were to be filtered out.

CFL-reachability in multi-entry/multi-exit graphs has been used to give algorithms for context-sensitive interprocedural slicing [Horwitz et al. 1990, 1994] as well as for certain kinds of context-sensitive interprocedural dataflow-analysis problems [Reps et al. 1995]. Similar techniques—but ones that go beyond pure CFL-reachability—have been used to give cubic-time algorithms for a larger class of context-sensitive interprocedural dataflow-analysis problems [Sagiv et al. 1996]. The latter work deals with multi-entry/multi-exit graphs labeled with (function-valued) weights on the edges.

Methods for counting various quantities in hierarchically structured graphs are used in Melski and Reps [2000] and Chatterjee et al. [2003]. In this work, it is possible to perform arithmetic operations because the graphs are restricted so that the number of paths that meet a certain CFL condition is finite. Techniques given in Chatterjee et al. [2003] provide an alternative approach to the problem of testing a bounded-acceptance condition, and have been applied to stack-size analysis in embedded-systems code.

PDSs have also been extended with weights: weighted PDSs have weighted transition rules, where the weights are either drawn from a closed semiring [Bouajjani et al. 2003] or a meet semi-lattice [Schwoon et al. 2003; Reps et al. 2003].

**Systems:** Several systems exist that implement ideas related to the ones discussed in this article:

- The Wisconsin Program-Slicing Tool [Horwitz et al. 1997] supported context-sensitive interprocedural program slicing of ANSI C programs. In this case, the hierarchically structured graphs of interest were “system dependence graphs” [Horwitz et al. 1990, 1994], which capture a program’s control and data dependences.

This system became the basis for a commercial product, CodeSurfer<sup>®</sup> [Gamma Tech, Inc. 2000], a tool for code understanding and code inspection that supports browsing (“surfing”) of an ANSI C program’s system dependence graph, as well as a variety of operations for making queries about the dependence graph, such as slicing and chopping [Reps and Rosay 1995]. Recently, an experimental version of CodeSurfer has been created that builds system dependence graphs for x86 executables [Balakrishnan and Reps 2004].

- Slam [Ball and Rajamani 2000] performs reachability in a Boolean program (which models a C program of interest) to identify violations of temporal safety properties. Slam’s intended domain of application is the verification of properties of device drivers. In addition to reachability, Slam uses decision procedures to determine whether potential error paths are actually infeasible—and hence represent false alarms—in the program being modeled. Information gathered during this process is used to iteratively refine the Boolean program that models the program’s semantics.
- Moped [Schwoon 2002] is a model checker for pushdown systems. It supports pre\* and post\* queries, LTL model checking, and reachability analysis in Boolean programs.
- MOPS is a tool based on the theory of pushdown systems that performs context-sensitive exploration of interprocedural control-flow graphs to identify security vulnerabilities in C programs [Chen and Wagnet 2002].
- The Weighted PDS Library [Schwoon et al. 2003] provides a C library that allows client applications to instantiate weighted PDSs and stack-configuration automata, to invoke pre\* and post\* queries, and to read out answers from the weighted automata returned as results. WPDS++ [2004] is a similar library implemented in C++.

#### ACKNOWLEDGMENTS

We thank Tom Ball, Glenn Bruns, Javier Esparza, and Sriram Rajamani for useful discussions.

#### REFERENCES

- ALUR, R., ETESSAMI, K., AND MADHUSUDAN, P. 2004. A temporal logic of nested calls and returns. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’04)*, volume 2988 of *LNCS*, pages 467–481. Springer.
- ALUR, R., ETESSAMI, K., AND YANNAKAKIS, M. 2001. Analysis of recursive state machines. In *CAV 2001*, pages 207–220.

- ALUR, R., TORRE, S. L., AND MADHUSUDAN, P. 2003a. Modular strategies for recursive game graphs. In *Proceedings of TACAS*, volume 2619 of *LNCS*, pages 363–378.
- ALUR, R., TORRE, S. L., AND MADHUSUDAN, P. 2003b. Modular strategies for infinite games on recursive graphs. In *Proceedings of CAV'03*, volume 2725 of *LNCS*, pages 67–79.
- ALUR, R. AND YANNAKAKIS, M. 2001. Model checking of hierarchical state machines. *ACM Trans. Prog. Lang. Syst.* 23, 3, pages 273–303.
- ANDERSEN, H. 1994. Model checking and boolean graphs. *Theoret. Comput. Sci.* 126, 1, 3–30.
- BALL, T. AND RAJAMANI, S. 2000. Bebop: A symbolic model checker for boolean programs. In *SPIN '2000*, volume 1885 of *LNCS*, pages 113–130.
- BENEDIKT, M., GODEFROID, P., AND REPS, T. 2001. Model checking of unrestricted hierarchical state machines. In *ICALP'2001*, pages 652–666.
- BOOCH, G., JACOBSON, J., AND RUMBAUGH, J. 1997. *The Unified Modeling Language User Guide*. Addison Wesley.
- BOUAJJANI, A., ESPARZA, J., AND MALER, O. 1997. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97*, pages 135–150.
- BOUAJJANI, A., ESPARZA, J., AND TOUILI, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *POPL '03*, pages 62–73.
- BALAKRISHNAN, G. AND REPS, T. 2004. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC'04)*, volume 2985 of *LNCS*, pages 5–23. Springer.
- BURKART, O. AND STEFFEN, B. 1992. Model checking and context-free processes. In *CONCUR '92*, pages 122–137.
- BURKART, O. AND STEFFEN, B. 1999. Model checking the full modal mu-calculus for infinite sequential processes. *Theoret. Comput. Sci.* 221, 251–270.
- CHATTERJEE, K., MA, D., MAJUMDAR, R., ZHAO, T., HENZINGER, T. A., AND PALSBERG, J. 2003. Stack size analysis for interrupt-driven programs. In *Proceedings of the 10th Static Analysis Symposium*, pages 109–126.
- CHEN, H. AND WAGNER, D. 2002. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the Conference on Computer and Communication Security*.
- COUSOT, P. AND COUSOT, R. 1977. Static determination of dynamic properties of recursive procedures. In *IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, E.J. Neuhold (Ed.), pages 237–277, St-Andrews, N.B., Canada.
- CAUCAL, B. AND MONFORT, R. 1990. On the transition graphs of automata and grammars. In *Graph Theoretic Concepts in Computer Science*, Springer LNCS 484, pages 311–337.
- EMERSON, A. 1990. Modal and temporal logic. In *Handbook of Theoretical Computer Science, Volume B*, pages 995–1072, MIT Press.
- EMERSON, A. AND LEI, C. 1986. Efficient model-checking in fragments of the propositional mu-calculus. In *LICS 98*, pages 267–278.
- ESPARZA, J., HANSEL, D., ROSSMANITH, P., AND SCHWOON, S. 2000. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th International Conference*, volume 1855 of *LNCS*, pages 232–247. Springer.
- ETESSAMI, K. 2004. Analysis of recursive game graphs using data flow equations. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 282–296. Springer.
- FINKEL, A., WILLEMS, B., AND WOLPER, P. 1997. A direct symbolic approach to model checking pushdown systems. In *Infinity'97 Workshop*, volume 9 of *Electronic Notes in Theoretical Computer Science*.
- GRAMMATECH, INC. 2000. CodeSurfer System. “<http://www.grammatech.com/products/codesurfer/>”.
- HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Prog.* 8, 231–274.
- HEINTZE, N. AND MCALLESTER, D. A. 1997. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of Logic in Computer Science*, pages 342–351.
- HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. In *Trans. Prog. Lang. Syst.* 12, 1, 26–60.

- HORWITZ, S., REPS, T., BRICKER, T., AND ROSAY, G. 1997. Wisconsin Program-Slicing Tool. "[http://www.cs.wisc.edu/wpis/slicing\\_tool/](http://www.cs.wisc.edu/wpis/slicing_tool/)".
- HORWITZ, S., REPS, T., SAGIV, M., AND ROSAY, G. 1994. Speeding up slicing. In *Proceedings of the 2nd ACM Symposium on Foundation of Software Engineering*, pages 11–20.
- MELSKI, D. AND REPS, T. 1999. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 47–62.
- MELSKI, D. AND REPS, T. 2000. Interconvertibility of a class of set constraints and context-free language reachability. *Theoret. Comput. Sci.*, 248(1–2), 29–98.
- REPS, T. 1998. Program analysis via graph reachability. *Info. Soft. Tech.* 40(11–12), 701–726.
- REPS, T., HORWITZ, S., AND SAGIV, S. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61.
- REPS, T. AND ROSAY, G. 1995. Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on Foundation of Software Engineering*, pages 41–52.
- REPS, T., SCHWOON, S., AND JHA, S. 2003. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of the 10th Static Analysis Symposium*, pages 189–213.
- SAGIV, M., REPS, T., AND HORWITZ, S. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoret. Comput. Sci.* 167(1–2), 131–170.
- SCHWOON, S. 2002. Moped System. "<http://www.fmi.uni-stuttgart.de/szs/tools/moped/>".
- SCHWOON, S., REPS, T., AND JHA, S. 2003. Weighted PDS Library. "<http://www.fmi.uni-stuttgart.de/szs/tools/wpds/>".
- SCHWOON, S., JHA, S., REPS, T., AND STUBBLEBINE, S. 2003. On generalized authorization problems. In *Proceedings of the 16th Computer Section Foundations Workshop*, pages 202–218.
- BALL, T. AND RAJAMANI, S. 2000. SLAM Toolkit. "<http://research.microsoft.com/slam/>".
- SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (eds.), Prentice-Hall, Englewood Cliffs, NJ, pages 189–234.
- ULLMAN, J. D. 1988. *Principles of Database and Knowledge-base systems*. Computer Science Press.
- VALIANT, L. G. 1975. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.* 10, 308–315.
- VARDI, M. AND WOLPER, P. 1986. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Softw.* 32, 2, 183–221.
- WALUKIEWICZ, I. 2001. Pushdown processes: Games and model-checking. *Information and Computation* 164, 2, 234–263.
- YANNAKAKIS, M. 1990. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 230–242.
- WOODS, W. A. 1970. Transition network grammars for natural language analysis. *Commun. ACM* 13, 10, 591–606.
- WPDS++: 2004. A C++ Library for Weighted Pushdown Systems, University of Wisconsin.

Received September 2003; accepted February 2004