Analysis of Recursive State Machines

Rajeev Alur[1,2], Kousha Etessami[1], and Mihalis Yannakakis[1]

[1] Bell Labs, Murray Hill, NJ
[2] Dept. of Comp. and Inf. Science, U. of Pennsylvania
email: {alur,kousha,mihalis}@research.bell-labs.com

Abstract. 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, we focus on whether state-space analysis can be performed efficiently for RSMs. We consider the two central problems 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 θ is the maximum, over all component state machines, of the minimum of the number of entries and the number of exits of each component. We also study the precise relationship between RSMs and closely related models.

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 paper 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 [10] and UML [5], 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 [12, 6], RSMs appear to be the appropriate definition for visual modeling and allow tigher analysis.

We study the two most fundamental questions for model checking of safety and liveness properties, respectively: (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. 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 θ is a parameter depending only on the number of entries and exits in each component. The number of entry points correspond to the parameters passed to a component, while the number of exit points correspond 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 θ 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 for linear-time requirements expressed as LTL formulas or Büchi automata, via a product construction for Büchi automata with RSMs.

Related work. Our definition of recursive state machines naturally generalizes the definition of hierarchical state machines of [1]. 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 [1] are adaptations of traditional search algorithms to avoid searching the same component repeatedly, and have the same time complexity as the algorithms of this paper. However, the "bottom-up" algorithms used in [1] for hierarchical machines can not 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 [6, 7, 9, 8]. 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 [8] 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 $O(nk^2)$ [8], 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 θ.

Ball and Rajamani consider the model of Boolean programs, which can be viewed as RSMs extended with boolean variables [3]. 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 relation of a procedure. This in turn is based on algorithms for interprocedural dataflow analysis [12], which are generally cubic. As described in Section 5, 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.

In the context of this rich history of research, the current paper has four 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. Finally, using our RSM model one is able to, at no extra cost in complexity, 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.
Note: Results similar to ours have been obtained independently, and submitted concurrently, by [4] on a model identical to RSMs.

2 Recursive State Machines

Syntax. A *recursive state machine (RSM)* A over a finite alphabet Σ is given by a tuple $\langle A_1, \ldots, A_k \rangle$, 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*.

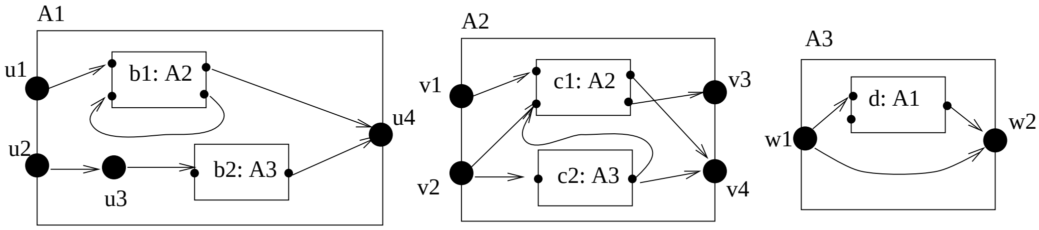

Fig. 1. A sample recursive state machine

– A labeling $Y_i : B_i \mapsto \{1, \ldots, k\}$ that assigns to every box an index of one of the component machines, $A_1, \ldots, A_k$.
– A set of *entry* nodes $En_i \subseteq N_i$, and a set of *exit* nodes $Ex_i \subseteq N_i$.
– A transition relation δ_i, where transitions are of the form (u, σ, 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 σ is either ε, a silent transition, or in Σ; 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 use the term *ports* to refer collectively to the entry and exit nodes of a machine A_i, and 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 δ_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 δ_i, ignoring labels. 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 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 $A = \langle A_1, \ldots A_k \rangle$ is a tuple $\langle b_1, \ldots, b_r, u \rangle$ where $b_1, \ldots, b_r$ 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 $\langle b_1, \ldots, b_r, u \rangle$ 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 δ. Let $s = \langle b_1, \ldots, b_r, u \rangle$ be a state with $u \in N_j$ and $b_r \in B_m$. Then, $(s, \sigma, s') \in \delta$ iff one of the following holds:

1. $(u, \sigma, u') \in \delta_j$ for a node u' of A_j, and $s' = \langle b_1, \ldots, b_r, u' \rangle$.
2. $(u, \sigma, (b', e)) \in \delta_j$ for a box b' of A_j, and $s' = \langle b_1, \ldots, b_r, b', e \rangle$.
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 $s' = \langle b_1, \ldots, b_{r-1}, u' \rangle$.
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 $s' = \langle b_1, \ldots, b_{r-1}, b', e \rangle$.

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 global states Q along with the transition relation δ define an ordinary transition system, denoted T_A.

We wish to consider recursive automata as generators of ω-languages. For this, we augment RSMs with a designated set of initial nodes, and with Büchi acceptance conditions. A *recursive Büchi automaton* (RBA) over an alphabet Σ consists of an RSM A over Σ, together with 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 $B_A = (T_A, Init^\star, F^\star)$, where the initial states $Init^\star$ are states $\langle e \rangle$ where $e \in Init$, and where a state $\langle b_1, \ldots b_r, v \rangle$ is in $F^\star$ if v is in F. For an infinite word $w = w_0 w_1 \ldots$ over Σ, a *run* π of B_A over w is a sequence $s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \cdots$ of states s_i and symbols $\sigma_i \in \Sigma \cup \{\varepsilon\}$ such that (1) $s_0 \in Init^\star$, (2) $(s_i, \sigma_i, s_{i+1}) \in \delta$ for all i, and (3) the word w equals $\sigma_0 \sigma_1 \sigma_2 \cdots$ with all the ε symbols removed. A run π is *accepting* if for infinitely many i, $s_i \in F^\star$.

We call a run π *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 iff for some $s \in F^\star$ there is a run π 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 two central algorithmic problems:

1. *Reachability:* Given A, for nodes u and v of A, let $u \Rightarrow v$ denote that some global state $\langle b_1, \ldots b_r, v \rangle$, 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.

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 $v = \Sigma_i v_i$,

$e = \Sigma_i e_i$ be the total number of vertices and edges. The *size* $|A|$ of a RSM A is the space needed to write down its components. Assuming, w.l.o.g., 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 θ, 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 $\theta = \max_{i \in \{1,\ldots,k\}} \min(en_i, ex_i)$. That is, every component has either no more than θ entries or no more than θ exits. There may be some components of each kind; we call components of the first kind *entry-bound* and the others *exit-bound*.

3 Algorithms for State-space Analysis

Given a recursive automaton, A, in this section we show how problems such as 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.

3.1 Reachability

Given A, we wish to compute the set $\{v \mid Init \Rightarrow v\}$. 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 $\{v \mid Init \Rightarrow v\}$ 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 iff 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 which we will write in the style of Datalog rules [13]. Recall some terminology. An *atom* is a term $P(\tau)$ where P is a predicate (relation) symbol and τ 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 σ, 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., [2]).

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, i.e., 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 (resp. predecessor list) for each vertex. For each box, create a list of its entries and exits that are vertices, i.e., 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, i.e., $En_i \times V_i$ 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, i.e., 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 is an entry node of a box b, i.e. $y=(b,u)$, 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 $R_k(z,(b,y))$ (resp., $R_k((b,x),z)$) and insert the atom into S. Correctness follows from the following lemma.

Lemma 1. *A tuple (x, y) is added to the predicate R_i iff $R_i(x, y)$ is true in the given RSM A, i.e., $\langle x \rangle$ can reach $\langle y \rangle$ in the transition system T_A.*

Theorem 1. *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. Lastly, 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 2. *$u \Rightarrow v$ in RSM A iff 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 2. *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 [1].

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 only 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 paper.

3.2 Checking Language Emptiness

Given an RBA $A = (\langle A_1, \ldots, A_k \rangle, Init, F)$, 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 $B_A = (T_A, Init^\star, F^\star)$. Our algorithm below for emptiness testing treats edges labeled by ε no differently than ordinary edges. This makes the algorithm report that $L(A)$ is non-empty even when the only infinite accepting runs in A include an infinite suffix of ε-labeled edges. We show in the next section how to overcome this. 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 iff there is a path in B_A from $\langle x \rangle$ to $\langle y \rangle$ which passes through an accept state in $F^\star$. 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 $y \in F^\star$, $x \in En_i, y \in V_i$
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 3. *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 4. *The language $L_u(A)$ is nonempty iff 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 5. *$L_b(A)$ is nonempty iff 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 3. *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 $O(|A|\theta)$.*

3.3 Model Checking with Büchi Automata

The input to the *automata-based model checking* problem consists of a RSM A over Σ and an ordinary Büchi automaton B over Σ. 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. Also, in the last section we ignored ε's, now we show how to disallow an infinite suffix of ε's in an accepting run. We modify the given Büchi automaton B to obtain $B' = \langle Q'_B, \delta'_B, I_B, F_B \rangle$ as follows. For every state q, we add an extra state q'. We add a transition from q to q' labeled with ε, from q' to itself labeled with ε, and for every transition (q, σ, q'') of B, add a transition (q', σ, q''). B' accepts exactly the same words as B, except it allows a finite number of ε's to be interspersed between consecutive characters in a word from $L(B)$.

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 δ'_i within each A'_i are defined as follows. Consider a transition (v, σ, v') in δ_i. Suppose $v \in N_i$. Then for every transition (q, σ, 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 4. *Let A be an RSM of size n with θ 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 $O(n \cdot m \cdot a \cdot \theta)$.*

4 Relation to Pushdown Systems

The relation between recursive automata and pushdown automata is fairly tight. Consider a *pushdown system* (PDS) given by $P = (Q_P, \Gamma, \Delta)$ over an alphabet Σ consisting of a set of control states Q_P, a stack alphabet Γ, and a transition relation $\Delta \subseteq (Q_P \times \Gamma) \times \Sigma \times (Q_P \times \{push(\Gamma), swap(\Gamma), pop\})$. That is, based on the control state and the symbol on top of the stack, the machine can update the control state, and either push a new symbol, swap the top-of-the-stack with a new symbol, or pop the stack. When P is augmented with a Büchi acceptance condition, the ω-language of the pushdown machine, $L(P)$, can then be defined in a natural way. Given a PDS P (or RBA A), we can build a recursive automaton A (PDS P, respectively) such that $L(P) = L(A)$, and $|A| \in O(|P|)$ ($|P| \in O(|A|)$, respectively). Translating from P to A, A has one component with number of exits (and hence θ) bounded by $|Q_P|$. Translating from A to P, the number of control states of P is bounded by the maximum number of exit points in A, not by θ. Both translations preserve boundedness. We have to omit details. For the detailed construction, please see the full paper.

5 Discussion

Efficiency and Context-Free Reachability: Given a recursive automaton of size n with θ 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 [14] was able to slightly improve the asymptotic bound, but his algorithm is highly impractical. A related problem is CFL-reachability ([15, 11]), 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 path labeled by w. Unlike CFL-membership, CFL-reachability is P-complete, and the best known algorithms require $\Omega(n^3)$ time ([15]). Using the close correspondence between recursive automata and context-free grammars, a grammar G can be translated to a recursive automaton 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 automata would lead to better than cubic bounds for parsing a string and for CFL-reachability.

Extended Recursive Automata: In presence of variables, our algorithms can be adopted in a natural way by augmenting nodes with the values of the variables. Suppose we have an extended recursive automaton 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 automaton of size at most $2^k \cdot |A|$ with the same number of components. The derived automaton has $\theta = 2^d$. Thus, reachability problems for such an extended recursive automaton can be solved in time $O(2^{k+2d} \cdot |A|)$. Note that such extended recursive automata are basically the same as the boolean programs of [3].

Concurrency: We have considered only sequential recursive automata. Recursive automata define context-free languages. Consequently, it is easy to establish that typical reachability analysis problems for a system of communicating recursive automata are undecidable. Our algorithms can however be used in the case when only one of the processes is a recursive automaton 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.

Acknowledgements: We thank Tom Ball, Javier Esparza, and Sriram Rajamani for useful discussions. Research partially supported by NSF Grants CCR97-34115, CCR99-70925, SRC award 99-TJ-688, and a Sloan Faculty Fellowship.

References

1. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proc. 6th ACM Symp. on Found. of Software Engineering*, pages 175–188, 1998.
2. H. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
3. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'2000*, volume 1885 of *LNCS*, pages 113–130, 2000.
4. M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. To appear in ICALP'2001.
5. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1997.
6. A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97*, pages 135–150, 1997.
7. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.
8. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th Int. Conference*, volume 1855 of *LNCS*, pages 232–247. Springer, 2000.
9. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Infinity'97 Workshop*, volume 9 of *Electronic Notes in Theoretical Computer Science*, 1997.
10. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
11. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.
12. T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
13. J. D. Ullman. *Principles of Database and Knowledge-base systems.* Computer Science Press, 1988.
14. L. G. Valiant. General context-free recognition in less than cubic time. *J. of Computer and System Sc.*, 10:308–315, 1975.
15. M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 230–242, 1990.