Model Checking of Unrestricted Hierarchical State Machines (Extended Abstract)

Michael Benedikt
Patrice Godefroid
Thomas Reps

Technical Report #1425

March 2001

1425

Model Checking of Unrestricted Hierarchical State Machines (Extended Abstract)

Michael Benedikt
Bell Laboratories
benedikt@bell-labs.com

Patrice Godefroid
Bell Laboratories
god@bell-labs.com

Thomas Reps
University of Wisconsin
reps@cs.wisc.edu

Abstract

Hierarchical State Machines (HSMs) are a natural model for representing the reactive behavior of complex software systems. We investigate in this paper an extension of the HSM model where state machines are allowed to call each other recursively. Such "unrestricted" HSMs can model classes of infinite state systems such as arbitrary combinations of control-flow graphs of procedures in programming languages such as C. We precisely compare the expressiveness of unrestricted HSMs with known classes of infinite state systems, namely context-free and pushdown processes. We then discuss several verification problems on HSMs, and present original model-checking algorithms for unrestricted HSMs.

1 Introduction

Hierarchical State Machines (HSMs) are finite state machines whose states themselves can be other machines. HSMs are a popular model for specifying complex software systems, which is why they form the basis of commercial modeling languages such as StateCharts, ObjecTime, and UML. Various verification problems for HSMs without recursion were recently studied in [AY98, AKY99, AG00].

In this paper, we investigate an extension of the HSM model where state machines are allowed to call each other recursively. Such "unrestricted" HSMs are strictly more expressive than the previously-studied HSM model since HSMs with recursion can model classes of infinite state systems. For instance, unrestricted HSMs can be used to model arbitrary combinations of control-flow graphs of procedures in programming languages such as C. Unrestricted HSMs are therefore a natural model for reasoning about the abstract behavior of reactive software programs.

We study in this paper several verification problems defined on unrestricted HSMs. First, we define several classes of unrestricted HSMs (or HSMs for short), and establish correspondence theorems with previously-existing classes of infinite state systems. Specifically, we show that "single-exit" HSMs, i.e., HSMs composed exclusively of machines each with a single exit state, have the same expressiveness as context-free processes, while general "multiple-exit" HSMs have the same expressiveness as pushdown processes. From these correspondence theorems and known verification results for context-free and pushdown systems, we immediately obtain algorithms and complexity bounds for various verification problems on HSMs.

We then show how some of the above results can be improved using new verification algorithms. We present a linear-time temporal-logic (LTL) model-checking algorithm for unrestricted HSMs. With this algorithm, we show that LTL model checking for single-entry multiple-exit HSMs (i.e., HSMs composed of machines each with a single entry state, but possibly multiple exit states) can be solved in time linear in the size of the HSM, instead of cubic time as previously known. This implies that the reachability and cycle-detection problems can also be solved in linear time for single-entry HSMs.

We also present a new model-checking algorithm for the logic CTL* and single-exit HSMs. Our algorithm runs in time linear in the size of the HSM, instead of quadratic time, the best previously-known upper bound.

Due to the correspondence results mentioned above, this algorithm also provides a new improved upper bound for CTL* model checking of context-free processes.

2 Unrestricted Hierarchical State Machines

Various definitions of state machines have been proposed for representing the behavior of reactive systems. A popular model used in the model-checking literature [CE81] is the Kripke structure. Given a set P of atomic propositions, a (flat) *Kripke structure* K is a tuple (S, R, L) where S is a (possibly infinite) set of states, $R \subseteq S \times S$ is a transition relation, and $L : S \mapsto 2^P$ is a labeling function that associates with each state the set of atomic propositions that are true in that state.

In this paper, we consider an alternative representation for reactive systems that supports a notion of hierarchy. In particular, we define an *unrestricted hierarchical state machine* (HSM) M over a set P of atomic propositions to be a set of *component structures* $\{M_1, \ldots, M_n\}$, where each of the M_i has

- A nonempty finite set N_i of *nodes*.
- A finite set B_i of *boxes* (or *supernodes*).
- A nonempty subset I_i of N_i, called the *entry-nodes* of N_i.
- A nonempty subset O_i of N_i, called the *exit-nodes* of N_i.
- A labeling function $X_i : N_i \mapsto 2^P$ that labels each node in N_i with a subset of P.
- An indexing function $Y_i : B_i \mapsto \{1, \ldots, n\}$ that maps each box of M_i to the index j of some structure M_j.
- A set C_i of pairs of the form (b, e), where b is a box in B_i and e is an entry-node of M_j with $j = Y_i(b)$, called the *call-nodes* of B_i.
- A set R_i of pairs of the form (b, x), where b is a box in B_i and x is an exit-node of M_j with $j = Y_i(b)$, called the *return-nodes* of B_i.
- An edge relation E_i. Each edge in E_i is a pair (u, v) such that (1) u is either a node in N_i or a return node in R_i, and (2) v is either a node in N_i or a call-node in C_i.

M_1 is called the top-level structure of M. The above definition is essentially that of Alur and Yannakakis [AY98]; however, we permit component structures to call each other recursively. An example of an unrestricted HSM is shown in Figure 1.

To simplify notation in what follows, we assume that the sets I_i and O_i are all pairwise disjoint, as are the sets C_i and R_i. Note that C_i and R_i are technically not part of N_i. An HSM M is called *single-entry* if every structure M_i in M has exactly one entry-node (i.e., $\forall 1 \leq i \leq n : |I_i| = 1$). An HSM M is called *single-exit* if every structure M_i in M has exactly one exit-node (i.e., $\forall 1 \leq i \leq n : |O_i| = 1$).

Each structure M_i can be associated with an ordinary Kripke structure, denoted $K(M_i)$, by recursively substituting each box $b \in B_i$ by the structure M_j with $j = Y_i(b)$. Since we allow state machines to call each other recursively, the expanded structure $K(M_i)$ can be infinite. A state of the expanded Kripke structure $K(M)$ is defined by a node and a finite sequence of boxes that specify the context. Formally, the *expansion* $K(M)$ *of an HSM* M is the Kripke structure (S, R, L) defined as follows:

- $S \subseteq \bigcup_{i=1}^{n} N_i \times (\bigcup_{i=1}^{n} B_i)^*$.
- The transition relation R is the set of transitions $((v, w), (v', w'))$ that satisfy any of the following conditions:
  - $(v, v') \in E_i$, $v, v' \in N_i$ and $w = w'$.
  - $(v, (b', e')) \in E_i$, $v \in N_i$, $v' = e'$, and $w' = wb'$.

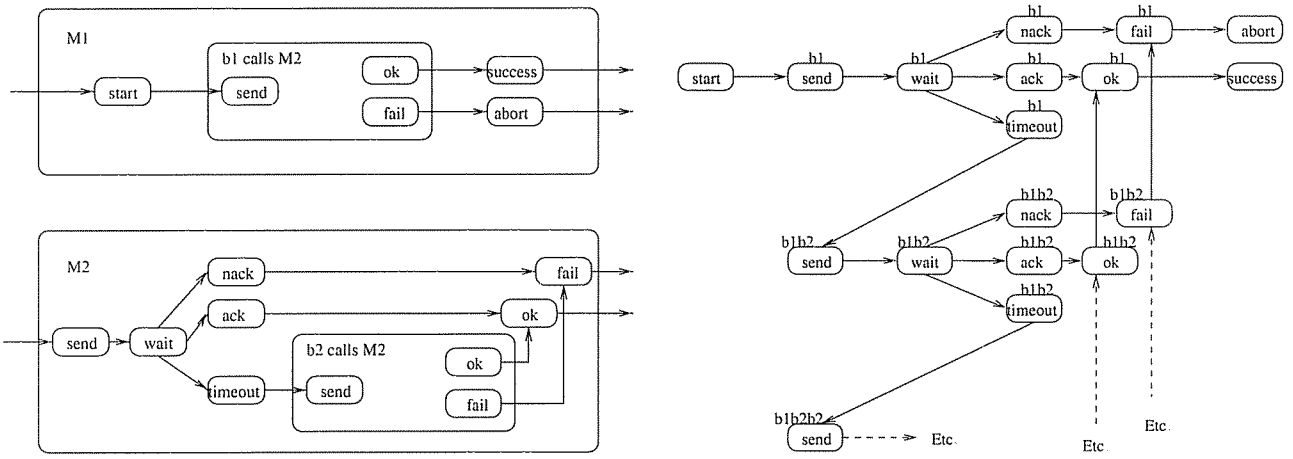

Figure 1: An example of an unrestricted HSM (left) and its expansion (right). The top-level structure M_1 has one box, which calls structure M_2. M_2 models an attempt to send a message; if no positive or negative acknowledgment is received, a timeout occurs and a recursive call to M_2 is performed.

- $((b,x),v') \in E_i$, $v = x$, $v' \in N_i$, and $w = w'b$.
- $((b,x),(b',e')) \in E_i$, $v = x$, $v' = e'$, and $w' = w''b'$ with $w = w''b$.

- The labeling function $L : S \mapsto 2^P$ is defined by $L((v,w)) = X_i(v)$ with $v \in N_i$.

The (infinite) expansion $K(M_1)$ of the HSM of Figure 1 is shown on the right of the figure, where the finite sequence of boxes corresponding to each state is indicated on top of the state when it is nonempty (e.g., the state "(send,b1b2)" is depicted as the state "send" labeled with "b1b2"). In what follows, we will write $K(M)$ to denote the expansion of the top-level structure M_1 of an HSM M.

3 Expressiveness of Unrestricted HSMs

As mentioned in the introduction, the unrestricted HSM model is closely related to several existing models for infinite-state systems, namely context-free grammars and pushdown automata. In this section, we compare the expressiveness and conciseness of these models. We also compare the expressiveness of the four classes of unrestricted HSMs defined in the previous section, namely single-entry single-exit, single-entry multiple-exit, multiple-entry single-exit, and multiple-entry multiple-exit HSMs.

Since we are interested in the temporal behavior of systems, our comparison of expressiveness is based on the existence of bisimulation relations between the Kripke structures corresponding to the expansions of these different classes of models. We now recall the definition of a bisimulation relation on Kripke structures.

Definition 1 Let $M_1 = (S_1, R_1, L_1)$ and $M_2 = (S_2, R_2, L_2)$ be Kripke structures. A binary relation $\mathcal{B} \subseteq S_1 \times S_2$ is a *bisimulation relation* if $(s_1, s_2) \in \mathcal{B}$ implies:

- $L_1(s_1) = L_2(s_2)$,
- if $(s_1, s_1') \in R_1$, then there is some $s_2' \in S_2$ such that $(s_2, s_2') \in R_2$ and $(s_1', s_2') \in \mathcal{B}$, and
- if $(s_2, s_2') \in R_2$, then there is some $s_1' \in S_1$ such that $(s_1, s_1') \in R_1$ and $(s_1', s_2') \in \mathcal{B}$.

Two states s_1 and s_2 are *bisimilar*, denoted $s_1 \sim s_2$, if they are related by some bisimulation relation. ■

By extension, we say that two Kripke structures M_1 and M_2 are bisimilar if $\forall s_1 \in S_1 : \exists s_2 \in S_2 : s_1 \sim s_2$ and $\forall s_2 \in S_2 : \exists s_1 \in S_1 : s_1 \sim s_2$.

Obviously, any multiple-entry machine with k entry-nodes can be replaced by k machines, each with a single entry-node. Therefore, the expressiveness of single-entry and multiple-entry HSMs is the same, although multiple-entry HSMs can be more concise than their equivalent single-entry HSM. In contrast, we show in the remainder of this section that single-exit and multiple-exit HSMs have different expressivenesses. Indeed, single-exit HSMs have the same expressiveness as context-free processes while multiple-exit HSMs have the same expressiveness as pushdown processes.

An *alphabetic labeled rewrite system* [CM90] is a triple $\mathcal{R} = (V, Act, R)$ where V is an alphabet, Act is a set of labels, and $R \subset V \times Act \times V^*$ is a finite set of rewrite rules. The *prefix rewriting relation* of $\mathcal{R}$ is defined by $\mapsto_{\mathcal{R}} = \{(uw, a, vw) | (u, a, v) \in R, w \in V^*\}$. The labeled transition graph $T_{\mathcal{R}} = (V^*, Act, \mapsto_{\mathcal{R}})$ is called the *prefix transition graph* of $\mathcal{R}$. Since the leftmost derivation graph of any context-free grammar [HU79] is the prefix transition graph of an alphabetic rewrite system [CM90], such prefix transition graphs are sometimes called *context-free processes*. For purposes of comparison with HSMs, we define the *expansion* of $\mathcal{R}$ as the (possibly infinite) Kripke structure $K(\mathcal{R})$ defined as follows: a state of $K(\mathcal{R})$ is a pair $(a, w) \in Act \times V^*$ such that $(v, a, w) \in \mapsto_{\mathcal{R}}$ for some $v \in V^*$; a transition of $K(\mathcal{R})$ is a pair $((a, w), (a', w'))$ such that (w, a', w') is in $\mapsto_{\mathcal{R}}$; the label of state (a, w) is a. We can now prove the following theorem:

Theorem 2 *For any alphabetic labeled rewrite system $\mathcal{R}$, one can construct in linear time a single-exit HSM M such that $K(\mathcal{R})$ and $K(M)$ are bisimilar.*

Proof: All proofs are omitted in this extended abstract. ■
The converse of the previous theorem also holds:

Theorem 3 *For any multiple-entry single-exit HSM M, one can construct in linear time an alphabetic labeled rewrite system $\mathcal{R}$ such that $K(M)$ and $K(\mathcal{R})$ are bisimilar.*

We now establish a similar correspondence between multiple-exit HSMs and pushdown processes. A *pushdown automaton* (e.g., [HU79]) is a tuple $A = (Q, Act, \Gamma, \delta, q_0)$ where Q is a finite set of states, Act is an alphabet called the input alphabet, Γ is a set of stack symbols, $q_0 \in Q$ is the initial state, and δ is a mapping from $Q \times Act \times \Gamma$ to finite subsets of $Q \times \Gamma^*$. The initial configuration of the system is (q_0, ϵ). The *expansion* of A is the (possibly infinite) Kripke structure $K(A)$ defined by the expansion of the prefix rewriting relation $\mapsto_\delta \subseteq (Q \times \Gamma^*) \times Act \times (Q \times \Gamma^*)$ itself defined by $\mapsto_\delta = \{((q, Z\gamma), a, (q', \beta\gamma)) | (q', \beta) \in \delta(q, a, Z), \gamma \in \Gamma^*\}$. We call such a Kripke structure a *pushdown process*. We have the following:

Theorem 4 *For any pushdown automaton A, one can construct in linear time a multiple-exit HSM M such that $K(A)$ and $K(M)$ are bisimilar.*

Conversely, the following theorem also holds:

Theorem 5 *For any multiple-entry multiple-exit HSM M, one can construct in linear time a pushdown automaton A such that $K(M)$ and $K(A)$ are bisimilar.*

Since it is known [CM90] that there exist pushdown processes that are not bisimilar to any context-free processes, we obtain the following result:

Theorem 6 *There exist multiple-exit HSMs whose expansion is not bisimilar to the expansion of any single-exit HSM.*

4 Complexity of Verification Problems for Unrestricted HSMs

In this section, we discuss the complexity of five verification problems for unrestricted HSMs: the reachability problem, the cycle-detection problem, and the model-checking problems for the logics LTL, CTL, and CTL*. Given an unrestricted HSM M and a set $T \subseteq \bigcup_{i=1}^{n} N_i$ of distinguished nodes, the *reachability problem* is

Class of HSM	Reachability	Cycle Detection	LTL	CTL	CTL*
Restricted Single-exit	Linear	Linear	Linear	Linear	
Restricted Multiple-exit	Linear	Linear	Linear	PSPACE	
Unrestricted Single-exit	Linear	Linear	Linear	Linear	Quadratic
Unrestricted Multiple-exit	Cubic	Cubic	Cubic	EXPTIME	EXPTIME

Figure 2: Known complexity results. Those for restricted HSMs are from [AY98]; those for unrestricted HSMs follow from [BS92,Wal96,FWW97,BS97,BEM97,EHRS00] and the theorems of Section 3. (Complexity bounds are given in terms of the size of the HSM.)

the problem of determining whether some state (v, w) of $K(M)$, with $v \in T$, is reachable from some initial state (v_0, ϵ), with $v_0 \in I_1$. Given M and T, the *cycle-detection problem* is to determine whether there exists some state $(v, w)$of $K(M)$, with $v \in T$, such that (i) (v, w) is reachable from some initial state (v_0, ϵ), with $v_0 \in I_1$, and (ii) (v, w) is reachable from itself. We refer the reader to [Eme90] for precise definitions of the model-checking problems for the logics LTL, CTL, and CTL*.

Since restricted HSMs are special cases of unrestricted HSMs, it is worth reviewing some of the results presented in [AY98] for the restricted case. Lines 2 and 3 of Figure 2 summarize the results of [AY98] concerning the complexity of the verification problems considered here, except for CTL* model checking, which was not discussed in [AY98]. Complexity bounds are given in terms of the size of the restricted HSM; in the case of LTL and CTL model checking, this means the size of the formula is fixed. (It is also shown in [AY98] that, for any fixed restricted HSM, CTL model checking is PSPACE-complete in the size of the formula.)

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

Consider the case of single-exit unrestricted HSMs. By Theorem 3, model checking for single-exit HSMs can be reduced to model checking for context-free processes. Since context-free processes can be viewed as pushdown processes defined by pushdown automata with only one state [BS92, Wal96], and since LTL model checking for one-state pushdown automata can be solved in time linear in the size of the pushdown automaton [EHRS00, FBW97], LTL model checking for single-exit HSMs can be solved in time linear in the size of the HSM. 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 HSMs can be derived from the CTL model-checking algorithm for context-free processes given in [BS92]. Finally, since the μ-calculus model-checking algorithm of [BS97] for context-free processes runs in quadratic time for formulae in the second level of the μ-calculus alternation hierarchy, which is known to contain CTL* [EL86], CTL* model checking for single-exit HSMs can be solved in time quadratic in the size of the HSM.

Let us turn to the case of multiple-exit unrestricted HSMs. By Theorem 5, we know that model checking for multiple-exit HSMs can be reduced to model checking for pushdown processes. Since LTL model checking for pushdown automata can be solved in time cubic in the size of the pushdown automaton [FBW97, EHRS00], LTL model checking for multiple-exit HSMs can be solved in time cubic in the size of the HSM. Moreover, a cubic-time algorithm for the reachability and cycle-detection problems can easily be derived from this LTL model-checking algorithm. Since CTL model checking for pushdown processes is EXPTIME-hard [Wal96] and since CTL is contained in the alternation-free μ-calculus for which the model-checking problem can be solved with the exponential-time algorithm presented in [BEM97], we can deduce from Theorems 4 and 5 that the CTL model-checking problem for multiple-exit HSMs is EXPTIME-complete in the size of the HSM. Similarly, the exponential-time model-checking algorithm given in [BS97] for pushdown

Class of HSM	Reachability	Cycle detection	LTL	CTL	CTL*
Unrestricted Multiple-entry Single-exit	Linear	Linear	Linear	Linear	*Linear*
Unrestricted Single-entry Multiple-exit	*Linear*	*Linear*	*Linear*	EXPTIME	EXPTIME
Unrestricted Multiple-entry Multiple-exit	Cubic	Cubic	Cubic	EXPTIME	EXPTIME

Figure 3: Improved complexity bounds for unrestricted HSMs. The improved bounds obtained in Sections 5 and 6 are highlighted in italic.

processes and the full μ-calculus, which contains CTL*, and the EXPTIME-hardness result of [Wal96] imply that the CTL* model-checking problem for multiple-exit HSMs is also EXPTIME-complete in the size of the HSM. The bottom two lines of Figure 2 summarize the results obtained from the foregoing discussion.

In the remainder of this paper, we present two improvements to the results listed in Figure 2. First, in Section 5, we present an LTL model-checking algorithm for unrestricted HSMs, and analyze the complexity of this algorithm. We then show that LTL model checking for single-entry multiple-exit HSMs can be solved with this algorithm in time *linear* in the size of the HSM, instead of *cubic* time. This implies that the reachability and cycle-detection problems can also be solved in linear time for single-entry HSMs. Second, in Section 6, we present a new CTL* algorithm for single-exit HSMs that runs in time *linear* in the size of the HSM, instead of *quadratic* time. Improved complexity bounds that take into account these two new results are listed in Figure 3.

5 LTL Model Checking

Following the automata-theoretic approach to model checking [VW86], a model-checking procedure for a formula ϕ 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) creating a product automaton for $A_{\neg\phi}$ and the system to be verified, and (3) checking if the language accepted by the product automaton is empty. To apply this procedure in our context, we define the product of a Büchi automaton $A_{\neg\phi}$ with an HSM[1] $M = \{M_1, \ldots, M_n\}$ to be a *Büchi-constrained HSM* $M' = \{M'_1, \ldots, M'_n\}$: M' is an HSM as defined earlier, where the labeling function encodes a Büchi acceptance condition. In particular, the nodes in node set N'_i of component structure M'_i are pairs (v, s), where $v \in N_i$ and s is a state of $A_{\neg\phi}$. Each box in B'_i is also a pair (b, s), where $b \in B_i$ and s is a state of $A_{\neg\phi}$, and such that $Y'_i((b, s)) = Y_i(b)$. Moreover, we have $C'_i = \{((b, s), (e, s)) | (b, s) \in B'_i \text{ and } (b, e) \in C_i\}$ and $R'_i = \{((b, s), (x, s)) | (b, s) \in B'_i \text{ and } (b, x) \in R_i\}$. Edges in the edge sets E'_i are of the form $(v, s) \rightarrow (v', s')$, such that there is an edge $v \rightarrow v'$ in E_i and a transition (s, ℓ, s') in $A_{\neg\phi}$, where $\ell \in 2^P$ agrees with the set of propositions true at v if $v \in N_i$, or else ℓ agrees with the set of propositions true at x if v is a return-node $(b, x) \in R_i$.

We define the labeling function X' on nodes (v, s) of M' such that $X'((v, s))$ equals *true* if s is an accepting state of $A_{\neg\phi}$, and equals *false* otherwise. Let T denote the set of nodes of M' where X' equals *true*. The LTL model-checking problem for an HSM M and a formula ϕ is thus reduced to checking whether there exists an infinite sequence w of states in $K(M')$ such that w passes through a node in T infinitely often. (Note that we have $K(M') = K(M) \times A_{\neg\phi}$, where $\times$ denotes the traditional definition of the product of a Kripke structure with a Büchi automaton.)

The latter problem can in turn be reduced to a graph-theoretic problem expressed in terms of the finite graph $G(M')$ whose nodes are the nodes of M' and whose edges are the edges of M' plus the set $CallEdges(M') \cup ReturnEdges(M')$, where $CallEdges(M') = \{((b, e), e) \mid e \in I'_i, b \in B'_j, Y'_j(b) = i\}$ and $ReturnEdges(M') = \{(x, (b, x)) \mid x \in O'_i, b \in B'_j, Y'_j(b) = i\}$. This graph finitely and completely represents

[1] As usual in this context, we assume for technical convenience that every node in N_i has an E_i successor.

$K(M')$, while making explicit how behaviors of component structures M_i' can be combined with calls and returns between component structures: every possible execution sequence in $K(M')$ is represented by a path in $G(M')$. However, not all paths in $G(M')$ represent execution paths of $K(M')$: a path in $G(M')$ corresponds to a path in $K(M')$ if, when a call finishes, the path in $G(M')$ returns to a return-node of the invoking box. The following definitions characterize the paths of $G(M')$ that correspond to executions of $K(M')$.

Definition 7 Let each box in M' be given a unique index in the range $1 \ldots |B|$, where $|B|$ is the total number of boxes in M'. For each box b in M', label the associated call-edges $((b,e),e)$ and return-edges $(x,(b,x))$ with the symbols "$(_b$" and "$)_b$", respectively; label all other edges with "e". A path in $G(M')$ is called a *Matched-path* (resp. *UnbalLeft-path*) iff the word formed by concatenating, in order, the symbols on the path's edges is in the language $L(Matched)$ (resp. $L(UnbalLeft)$), defined as follows:

$Matched \rightarrow Matched \;\; Matched$ $\mid (_j \;\; Matched \;\;)_j \qquad 1 \leq j \leq \lvert B \rvert$ $\mid e$ $\mid \epsilon$	$UnbalLeft \rightarrow UnbalLeft \;\; (_j \;\; Matched \qquad 1 \leq j \leq \lvert B \rvert$ $\mid Matched$

■

$L(Matched)$ is a language of balanced-parenthesis strings (interspersed with strings of zero or more e's). In general, a *Matched*-path from u to v, where u and v are in the same component structure, represents a sequence of transition steps from u to v during which the call stack may temporarily grow deeper—because of calls—but may never be shallower than its original depth, before eventually returning to its original depth. $L(UnbalLeft)$ is a language of *partially* balanced parentheses (again, interspersed with strings of zero or more e's): every right parenthesis "$)_j$" is balanced by a preceding left parenthesis "$(_j$", but the converse need not hold. An *UnbalLeft*-path from u to v represents a sequence of transition steps from u to v during which the call stack may never be shallower than its original depth, but may end with the call stack deeper than it was originally. The (additional) pending calls left on the stack by such a transition sequence correspond to the unmatched $(_j$'s in the path's word. Every infinite path in $K(M')$ either corresponds to a *Matched*-path (a repeated computation pattern after which the stack is preserved) or a *UnbalLeft*-path (a repeated computation pattern after which the stack grows) of $G(M')$.

LTL model checking is carried out directly on the Büchi-constrained product-HSM by means of the two-phase algorithm presented in Figures 4 and 5. The first phase consists of applying function ComputeSummaryEdges of Figure 4 to an HSM[2] M with Büchi acceptance condition T to create a set of *summary-edges*. Each summary-edge represents a *Matched*-path between a call-node and a return-node, where the two nodes are associated with the same box. More precisely, ComputeSummaryEdges creates the set *SummaryEdges*, which consists of pairs of the form $((b,e) \rightarrow (b,x),B)$. Summary-edge $((b,e) \rightarrow (b,x),B)$ indicates that (i) there exists a *Matched*-path from e to x, and (ii) if Boolean value B is *true*, then there exists such a path that passes through at least one node in T.

ComputeSummaryEdges is a dynamic-programming algorithm. In addition to tabulating summary-edges, ComputeSummaryEdges builds up the set *PathEdges*: a *path-edge* $(e \rightarrow v,B)$ in *PathEdges* indicates the existence of a *Matched*-path from an entry-node $e \in I_i$ of component structure M_i to v, where $v \in N_i \cup C_i \cup R_i$. As with summary-edges, the Boolean value B records whether the *Matched*-path summarized by the edge traverses at least one node in T. ComputeSummaryEdges starts by asserting that there is a zero-length *Matched*-path from every entry-node to itself (lines [10]–[12]); the corresponding path-edges are inserted into *PathEdges*, and also placed into the set *WorkList*. (The condition "$e \in T$" on line [11] sets the Boolean value of the path-edge according to whether e itself is a T node.) Path-edges are always tabulated by invoking procedure *Propagate*, which maintains *PathEdges* and *WorkList*, and hence only edges whose source is an

[2]Henceforth, we will drop prime symbols (') on the components of Büchi-constrained HSMs.

```
function ComputeSummaryEdges(M: HSM, T ⊆ ⋃_{i=1}^{n} N_i) returns set of pairs (edge,Boolean)
    declare
[1]     PathEdges, SummaryEdges, WorkList: set of pairs (edge,Boolean)

    procedure Propagate(e → v: edge, B: Boolean)
[2]   if there is no pair of the form (e → v, B') in PathEdges then
[3]     Insert (e → v, B) into PathEdges
[4]     Insert (e → v, B) into WorkList
[5]   else if (e → v, B') ∈ PathEdges ∧ B = true ∧ B' = false then /* B subsumes B' */
[6]     PathEdges := (PathEdges − {(e → v, B')}) ∪ {(e → v, B)}
[7]     WorkList := (WorkList − {(e → v, B')}) ∪ {(e → v, B)}
[8]   fi
    end

    begin
[9]   PathEdges := ∅; SummaryEdges := ∅; WorkList := ∅
[10]  for each entry-node e in some set I_i, for 1 ≤ i ≤ n do
[11]    Propagate(e → e, e ∈ T)
[12]  od
[13]  while WorkList ≠ ∅ do
[14]    Select and remove a pair (e → v, B) from WorkList
[15]    switch v
[16]      case v = (b, e') ∈ C_i: /* v is a call-node */
[17]        for each (b, x) such that ((b, e') → (b, x), B') ∈ SummaryEdges do
[18]          Propagate(e → (b, x), B ∨ B')
[19]        od
[20]      end case
[21]      case v = x ∈ O_i: /* v is an exit-node */
[22]        for each box b in some component structure M_j such that Y_j(b) = i do
[23]          if there is no pair of the form ((b, e) → (b, x), B') in SummaryEdges then
[24]            Insert ((b, e) → (b, x), B) into SummaryEdges
[25]          else if ((b, e) → (b, x), B') ∈ SummaryEdges ∧ B = true ∧ B' = false then /* B subsumes B' */
[26]            SummaryEdges := (SummaryEdges − {((b, e) → (b, x), B')}) ∪ {((b, e) → (b, x), B)}
[27]          fi
[28]          for each e' ∈ I_j such that (e' → (b, e), B'') ∈ PathEdges do
[29]            Propagate(e' → (b, x), B ∨ B'')
[30]          od
[31]        od
[32]      end case
[33]      default : /* v ∈ (N_i − O_i) ∪ R_i, i.e., v is other than a call-node or an exit-node */
[34]        for each v' such that v → v' ∈ E_i do
[35]          Propagate(e → v', B ∨ (v' ∈ T))
[36]        od
[37]      end case
[38]    end switch
[39]  od
[40]  return(SummaryEdges)
    end
```

Figure 4: An algorithm for computing summary-edges for a Büchi-constrained HSM M with Büchi acceptance condition T.

entry node are included in *PathEdges* and in *WorkList*. *Propagate* also enforces the property that a path-edge associated with the Boolean value *false* is "overwritten" if the same pair of endpoints is later discovered to be associated with the Boolean value *true* (lines [1]–[8]). Function ComputeSummaryEdges then finds new path-edges by repeatedly choosing an edge $e \to v$ from *WorkList* and extending the path that it represents as appropriate, depending on the type of target node v (lines [13]–[15]).

If the target node v is a call-node (b, e') in component structure M_i, previously computed summary-edges of the form $(b, e') \to (b, x)$ are used to create new path-edges in M_i (lines [16]–[20]). When v is an exit-node of component structure M_i, ComputeSummaryEdges tabulates a summary-edge $((b, e) \to (b, x), B)$ (with an

```
function ContainsTCycle(M: HSM, T ⊆ ⋃_{i=1}^{n} N_i) returns a pair (set of nodes, set of edges)
    begin
[1]   SummaryEdges = ComputeSummaryEdges(M, T)
[2]   G = (⋃_{i=1}^{n} N_i ∪ C_i ∪ R_i, ⋃_{i=1}^{n} E_i ⋃ CallEdges(M) ⋃ SummaryEdges)
[3]   SCCSet = FindSCCs(G, I_1) /* I_1 is the set of roots of the depth-first search */
[4]   for each non-trivial SCC (Nodes, Edges) ∈ SCCSet do
[5]     if (Nodes ∩ T ≠ ∅) or (∃((b, e) → (b, x), B) ∈ Edges : B = true) then
[6]       return(Nodes)
[7]     fi
[8]   od
[9]   return(∅)
    end
```

Figure 5: An algorithm for detecting T-cycles.

appropriate value of B) between the corresponding call-nodes and return-nodes at all boxes b that can invoke M_i (lines [21]–[27])—possibly "overwriting" a previously tabulated summary-edge in cases where B is *true*; each new summary-edge in a component structure M_j may in turn induce new path-edges in M_j: if the new summary-edge is $(b, e) \rightarrow (b, x)$, then there is a path-edge $e' \rightarrow (b, x)$ inserted for every entry-node e' of M_j such that there is already a path-edge $e' \rightarrow (b, e) \in PathEdges$ (lines [28]–[30]). Otherwise, by default, every edge $v \rightarrow v'$ from node v in M_i is used to attempt to generate new path-edges that extend $e \rightarrow v$ to $e \rightarrow v'$ (lines [33]–[37]).

The second phase of the model-checking algorithm consists of lines [2]–[8] of function ContainsTCycle of Figure 5. The goal of ContainsTCycle is to determine whether any component structure M_i contains a node n such that (i) n is reachable from some entry-node of I_1 along an *UnbalLeft*-path, and (ii) there is a non-empty cyclic *UnbalLeft*-path (which might merely be a cyclic *Matched*-path) that starts at n and contains a member of T.

ContainsTCycle checks this condition by searching for (nontrivial) strongly connected components that are reachable from an entry-node of I_1 (line [3]) in a directed graph G that consists of the nodes and edges of all component structures of M, together with all M's call-edges, plus the set of summary-edges computed by the function ComputeSummaryEdges (line [2]). The presence of call-edges and summary-edges is what allows information to be recovered from G about *UnbalLeft*-paths in M. The summary-edges permit ContainsTCycle to avoid having to explore *Matched*-paths between call-nodes and return-nodes of the same box, and, in particular, whether such nodes are connected by a *Matched*-path that contains a T node.

Graph G is searched for nontrivial strongly connected components that are reachable from the initial node of M (line [3]). A *strongly connected component* (SCC) of a graph G is a subgraph $(Nodes, Edges)$ of G such that (1) for every pair of vertices $v, w \in Nodes$, there is a path in G from v to w and a path in G from w to v, and (2) $Edges$ consists of the edges of G that connect the vertices in $Nodes$; an SCC is *nontrivial* if $Edges$ is nonempty. If there exists a nontrivial SCC $(Nodes, Edges)$ in G such that $Nodes$ contains a member of T or $Edges$ contains a summary-edge with a Boolean value *true*, the algorithm returns the set of nodes of this SCC (lines [4]–[8]), which represents an infinite computation of the Büchi-constrained HSM M that passes through a T node infinitely often. Otherwise, the empty set $\emptyset$ is returned (line [9]), which means that the language accepted by M is empty, and hence that the original HSM satisfies the LTL formula. The correctness of the algorithm is established by the following theorem.

Theorem 8 *Given an HSM M and an LTL formula ϕ, $K(M)$ satisfies ϕ iff the algorithm of Figure 5 applied to the Büchi-constrained HSM $M \times A_{\neg\phi}$ and its corresponding set T returns $\emptyset$.*

We now turn to the question of the runtime complexity of Figures 4 and 5. Function ComputeSummaryEdges discovers the need for a new path-edge $e \rightarrow v'$ by extending some previously discovered path-edge $e \rightarrow v$ (taken from *WorkList*) with an edge $v \rightarrow v'$, which can be either an E_i edge (lines [34]–[36]) or a

summary-edge (lines [17]–[19] and [28]–[30]). Because node v' can have in-degree greater than one, a path-edge $e \rightarrow v'$ can be "discovered" by function ComputeSummaryEdges more than once (as much as once for each in-edge to v'), but it will only be inserted into *PathEdges* and *WorkList* at most twice, once when associated with the Boolean value *false*, followed by once when associated with the Boolean value *true*, due to the guards in lines [2] and [5] in procedure *Propagate*.

Thus, for any given component structure M_i, the worst-case time complexity of the function ComputeSummaryEdges is equal to the number $|I_i|$ of entry-nodes of M_i, multiplied by the number of E_i edges plus summary-edges in M_i. In the worst case, each box $b \in B_i$ can have a summary-edge from every call-node (b, e) to every return-node (b, x) in box b in M_i. If C_b denotes the number of call-nodes (b, e) in box b, and R_b denotes the number of return-nodes (b, x) in box b, the number of summary-edges in component structure M_i is bounded by $O(\Sigma_{b \in B_i} C_b R_b)$. Therefore, the worst-case time complexity of the function ComputeSummaryEdges that can be attributed to M_i is bounded by $O(|I_i| \ (|E_i| + \Sigma_{b \in B_i} C_b R_b))$. Summing this expression over all component structures, the overall worst-case time complexity of function ComputeSummaryEdges is bounded by $O(\Sigma_{i=1}^{n} \ |I_i| \ (|E_i| + \Sigma_{b \in B_i} C_b R_b))$.

It is possible to make two improvements to ComputeSummaryEdges, which we now sketch. First, notice that path-edges in each component structure could be "anchored" at exit-nodes rather than at entry-nodes, and path-edges could be "grown" backwards rather than forwards (a technique also used in [HRS95]). The cost of this first variant algorithm is thus bounded by $O(\Sigma_{i=1}^{n} \ |O_i| \ (|E_i| + \Sigma_{b \in B_i} C_b R_b))$. Second, path-edges in component structures M_i where $|O_i| < |I_i|$ could be anchored at exit-nodes (and path-edges grown backwards), whereas in other component structures the path-edges could be anchored at entry-nodes (and path-edges grown forwards). The cost of this second variant algorithm is thus bounded by $O(\Sigma_{i=1}^{n} \ \min(|I_i|, |O_i|) \ (|E_i| + \Sigma_{b \in B_i} C_b R_b))$ (1).

Since the size of the graph G computed by function ContainsTCycle is $O(\Sigma_{i=1}^{n} \ (|E_i| + \Sigma_{b \in B_i} C_b R_b + \Sigma_{b \in B_i} C_b)$ (2) and since finding all strongly connected components in a directed graph can be carried out in time linear in the size of this graph (e.g., see [AHU74]), we can conclude that the overall worst-case time complexity of our algorithm is given by (1) + (2), i.e.,

$$O(\Sigma_{i=1}^{n} \ [\min(|I_i|, |O_i|) \ (|E_i| + \Sigma_{b \in B_i} C_b R_b) + |E_i| + \Sigma_{b \in B_i} C_b R_b + \Sigma_{b \in B_i} C_b]) \ (3).$$

In the case of single-entry, single-exit, and single-entry single-exit HSMs, these bounds simplify to

Single-entry HSM	Single-exit HSM	Single-entry single-exit HSM
$O(\|E\| + \|R\|)$	$O(\|E\| + \|C\|)$	$O(\|E\| + \|B\|)$

We can thus conclude that, *for any fixed LTL formula ϕ, the LTL model-checking problem for an unrestricted HSM M that is single-entry or single-exit can be solved in time linear in the size of M.*

Note that the Büchi-constrained HSM $M' = M \times A_{\neg\phi}$ obtained by combining a single-entry (or single-exit) HSM M with the Büchi automaton $A_{\neg\phi}$ for an LTL formula ϕ will typically be multiple-entry (resp. multiple-exit). However, each component structure M'_i of M' will have at most $|S_{\neg\phi}|$ entry-nodes (resp. exit-nodes), where $|S_{\neg\phi}|$ is the number of states of the automaton $A_{\neg\phi}$. Therefore, for a fixed LTL formula ϕ, the term $min(|I'_i|, |O'_i|)$ for component structure M'_i is bounded by the fixed constant $|S_{\neg\phi}|$ in Expression (3). Hence our claim that the LTL model-checking problem can be solved in linear-time when the HSM M is single-entry (resp. single-exit) does hold.

6 CTL* Model Checking for Single-Exit HSMs

In this section, we present a CTL* model-checking algorithm for single-exit HSMs which runs in time linear in the size of the HSM. The logic CTL* uses the temporal operators U (until), X (nexttime) and the existential path quantifier E in addition to the traditional operators $\neg$ (not) and $\vee$ (or) of propositional logic. Two types of CTL^* formulas, path formulas and state formulas, are defined by mutual induction. Every atomic

function DECOMP(ϕ: LTL formula) **returns** Set of pairs ($\beta \in$LTL$^+, \delta \in$LTL)
[1] **begin**
[2] **if** ($\phi = P$) **then return**($\{(P, true)\}$)
[3] **if** ($\phi = \phi_1 \vee \phi_2$) **then return**(DECOMP(ϕ_1) $\cup$ DECOMP(ϕ_2))
[4] **if** ($\phi = \neg\phi_1$) **then return**($\bigcup_{A \subseteq \text{DECOMP}(\phi_1)}(\bigwedge_{(\beta,\delta)\in A} \neg\beta, \bigwedge_{(\beta,\delta)\in \text{DECOMP}(\phi_1)-A} \neg\delta)$)
[5] **if** ($\phi = Xp$) **then return**($\bigcup_{(\beta,\delta)\in\text{DECOMP}(p)}(X\beta, \delta) \cup \{(exit, p)\}$)
[6] **if** ($\phi = pUq$) **then return**($\bigcup_{\emptyset \neq A \subseteq \text{DECOMP}(p)}(G \bigvee_{(\beta,\delta)\in A} \beta, \bigwedge_{(\beta,\delta)\in A} \delta \wedge pUq)$
$\cup \bigcup_{\emptyset \neq A \subseteq \text{DECOMP}(p)} \bigcup_{(\beta',\delta')\in\text{DECOMP}(q)}(\bigvee_{(\beta,\delta)\in A} \beta U \beta', \delta' \wedge \bigwedge_{(\beta,\delta)\in A} \delta)$)$\}$
end

Figure 6: The function DECOMP

proposition is a state formula as well as a path formula. If p, q are both state formulas (resp., both path formulas) then $p \vee q$ and $\neg p$ are also state formulas (resp., 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 abbreviation Fp for $trueUp$ and Gp for $\neg F \neg p$. Any CTL* state formula can thus 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 $E\ \rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$ with ρ an LTL formula over propositions $p(\gamma_1), \ldots, p(\gamma_n)$ that are each substituted by the corresponding CTL* state formula γ_i. We refer the reader to [Eme90] for a detailed presentation of the semantics of CTL*.

A key technical challenge is that the truth value of a temporal-logic formula in any state (v, w) of $K(M)$ may not only depend on the node v but also on the stack content w. 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 [BS92, BS97, AY98]. A context is a set of (here CTL*) formulas whose truth value at the exit node of a machine M_i determine the truth value of a formula ϕ at the root. The notion of context makes it possible to reason compositionally about HSMs.

Our algorithm exploits this idea and reduces the evaluation of a path formula ϕ on a sequence $w; w'$ of states, where w is finite while w' is infinite, to the evaluation of some formulas β and δ on the sequences w and w' respectively. We introduce a special atomic proposition *exit* that holds only at the final state of a finite sequence w, and denote by LTL$^+$ the set of LTL formulas that can be expressed using this extended set of atomic propositions. The function DECOMP given in Figure 6 specifies how the evaluation of an LTL formula ϕ 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 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 $\gamma = E\rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n) \in$ exists(F), for every δ such that $(\beta, \delta) \in$DECOMP(ρ), $E\delta(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$ is also in F. The *closure* cl(ϕ) of a CTL* formula ϕ is the smallest closed set containing exists($\{\phi\}$). One can show, using properties of DECOMP, that cl(ϕ) is always finite for any CTL* formula ϕ. Let pd(ϕ) be the maximal nesting of path quantifiers (E) in a CTL* formula ϕ. Given a set F of CTL* formulas, let pd(F) =max$_{\gamma\in F}$(pd(γ)). For ϕ with pd(ϕ) $\geq j$, let cl$^{\leq j}(\phi)$ be the elements of cl(ϕ) with at most j nested path quantifiers. Clearly, cl$^{\leq j}(\phi)$ is a closed set and pd(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 Kripke structure K with a single initial state s_0 *satisfies an F-context C*, written $K \models C$, if, for all $\gamma \in F$, $(K, s_0) \models \gamma$ iff $C(\gamma) = true$. 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 a F-context with the elements set to true by it. For an HSM M, a node $v \in M$, an F-context C, and a formula $\gamma \in F$, we say (M, v) *satisfies γ in context C*, written $(M, v) \models_C \gamma$, if, for all K', $K' \models C \Rightarrow ((K(M); K'), v) \models \gamma$, where $K(M); K'$ is the Kripke structure obtained from $K(M)$ by identifying the top-level exit node of $K(M)$

function CONTEXTUALIZE(F: closed set of CTL*existential formulas, M: HSM over $\{\gamma \in F : \mathrm{pd}(\gamma) < \mathrm{pd}(F)\}$, C: F-CONTEXT) **returns** HSM over F.
/* We assume $M = \{M_1, \ldots, M_n\}$ with $M_i = (N_i, B_i, I_i, O_i, X_i, Y_i, C_i, R_i, E_i)$ */
[1] **begin**
[2] M_1=TopLevelMachine(M)
[3] **for** each $\gamma \in F$ with $\gamma = E\ \rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$ **do** /* Precompute all the LTL results needed */
[4] N(γ)=LTLALG($E\ \rho, M$)
[5] **for** each $(\beta, \delta) \in$DECOMP(ρ)
[6] N(β)=LTLALG(E $(\beta \wedge F\ exit), M$)
[7] **for** each $M_i \in M$ **do**
[8] Nodes$_1$(M_i,γ)= $N_i \cap$N(γ)
[9] **for** each $(\beta, \delta) \in$DECOMP(ρ)
[10] Nodes$_2$(M_i,β)= $N_i \cap$N(β)
[11] **od**
[12] **od**
[13] OLDCON = $\emptyset$ /* Find the pairs (M_i, c) reachable from (M_1, C) */
[14] CON= $\{(M_1, C)\}$
[15] **while** (CON $\neq$ OLDCON) **do**
[16] OLDCON =CON
[17] **for** each $(M_i, c) \in$OLDCON **do**
[18] **for** each $\gamma \in F$ with γ=E$\rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$
[19] Sat(M_i, c, γ)=Nodes$_1$(M_i,γ) $\cup \bigcup_{(\beta,\delta) \in \mathrm{DECOMP}(\gamma), \mathrm{c}(\mathrm{E}_{\delta(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)) = true}$ Nodes$_2$(M_i,β)
[20] **for** each b $\in$Boxes(M_i) **do**
[21] Prop(b, (M_i, c))= ($Y_i(b), c'$) such that $\forall \gamma \in F : c'(\gamma) = true$ iff $(b, x) \in$Sat(M_i, c, γ)
[22] CON=OLDCON $\cup$\{Prop(b, (M_i, c))\}
[23] **od**
[24] **od**
[25] **od**
[26] /* Now build the output HSM M^*
[27] $M^* = \{M_{i,c} | 1 \leq i \leq n$ and $c \in$ CON $\}$
[28] Forall $1 \leq i \leq n$, for all $c \in$ CON,
[29] $M_{i,c} = (N_i \times \{c\}, B_i \times \{c\}, I_i \times \{c\}, O_i \times \{c\}, X'_{i,c}, Y'_{i,c}, C'_{i,c}, R_i \times \{c\}, E_i \times \{c\})$ where
[30] $C'_{i,c} = \{((b, c), (e, c')) | (b, e) \in C_i$ and $(M_k, c') =$Prop(b, (M_i, c))$\}$
[31] For all $b \in B_i$, $Y'_{i,c}((b, c)) = (Y_i(b), c')$ with $(M_k, c') =$Prop(b, (M_i, c))
[32] For all $v \in N_i$, $X'_{i,c}((v, c)) = \{\gamma \in F | v \in$Sat($M_i, c, \gamma$)$\}$
[33] TopLevelMachine(M^*)=$M_{1,C}$
[34] **return**(M*)
end

Figure 7: Construction of the context-dependent HSM

with the initial state of K'.

Given a closed set F of existential formulas, an HSM M whose nodes are labeled with formulas in $\{\gamma \in F | \mathrm{pd}(\gamma) < \mathrm{pd}(F)\}$, and an F-context C, the function CONTEXTUALIZE presented in Figure 7 constructs a new HSM M* from multiple copies of M, each of which is indexed by an F-context c. The nodes of M* in copy (M$_j$,c) are labeled by formulas $\gamma \in F$ representing the truth value of γ in the corresponding node of M in the context c. It can be shown that any node (v, c) in M^* is labeled with $\gamma \in F$ iff $(M, v) \models_c \gamma$.

The function CONTEXTUALIZE uses a variant, denoted LTLALG, of the LTL model-checking algorithm presented in the previous section. Given a formula of the form $E\rho(p(\gamma_1,) \ldots, p(\gamma_n))$ where ρ is an LTL$^+$ formula over atomic propositions including $p(\gamma_1,) \ldots, p(\gamma_n)$, and an HSM M whose nodes are also labeled with propositions in $p(\gamma_1,) \ldots, p(\gamma_n)$, LTLALG($E\rho, M$) returns the set of nodes v of M such that $(v, \epsilon) \models E\rho$. This is done exactly as described in the previous section, except for the three following modifications. First, LTLALG evaluates formulas of the form $E\rho$ instead of $A\rho$. Second, we still need to define how formulas of LTL$^+$ are evaluated on M: we say that a formula $E\rho$ where ρ is in LTL$^+$ is satisfied in a node v of a machine M_i if there is a path w from (v, ϵ) that satisfies ρ, such that either w is infinite or w terminates at (x, ϵ), where x is the exit node of M_i. Third, we also extend the evaluation of formulas to include return nodes: we say that the return node (b, x) of a box b satisfies a formula $E\rho$ iff the corresponding exit node x

function CHECK(ϕ: existential CTL* formula, M: single-exit HSM, C : cl(ϕ)-CONTEXT) **returns** set of nodes in M_1
[1] **begin**
[2] $M^0 = M$
[3] **for**$\{j = 0; j <$pd$(\phi); j{+}{+}\}$
[4] M^{j+1}=CONTEXTUALIZE$(cl^{\leq j+1}(\phi), M^j, C \cap cl^{\leq j+1}(\phi)))$
[5] **return** $\{v \in$ TopLevelMachine$(M^{\mathrm{pd}(\phi)})|$Label(v) includes $\phi\}$
end

Figure 8: CTL* model-checking algorithm

satisfies $E\rho$ when b is the only element of the stack; in other words, we define $((b,x),\epsilon) \models E\rho$ iff $(x,b) \models E\rho$. It is easy to extend the LTL model-checking algorithm of the previous section to meet these three additional requirements.

By repeatedly invoking CONTEXTUALIZE with $\mathrm{cl}^{\leq j}(\phi)$ with increasing values of j, $1 \leq j \leq \mathrm{pd}(\phi)$, i.e., larger and larger subsets of cl(ϕ), one can thus evaluate CTL* formulas in a bottom-up manner. This is what is done in function CHECK presented in Figure 8. Since any CTL* formula ϕ is a boolean combination of existential formulas ϕ_i, finding the nodes of the top-level machine M_1 of an HSM M satisfying ϕ can be reduced to finding the nodes of M_1 satisfying each ϕ_i. This is done by computing CHECK$(\phi_i, M, C_\emptyset)$ where $C_\emptyset$ is the set of formulas γ in cl(ϕ_i) that evaluate to true at a single node labeled as the exit node of M_1 and 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.

Theorem 9 *Given a single-exit HSM M, a node v of M_1, and an existential CTL* formula ϕ_i, (v,ϵ) satisfies ϕ_i iff v is included in the set of nodes returned by $CHECK(\phi_i, M, C_\emptyset)$.*

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

7 Concluding Remarks

The LTL model-checking algorithm presented in Section 5 is closely related to algorithms for solving so-called "Context-Free-Language" reachability problems [Yan90, Rep98], as well as to CFL-reachability-based algorithms for such program-analysis problems as interprocedural slicing [HRSR94] and interprocedural dataflow analysis [RHS95, HRS95]. In particular, the notions of path-edges and summary-edges, and the dynamic-programming technique used to compute such edges in function ComputeSummaryEdges already appeared in this earlier work, although the cycle-detection and LTL model-checking problems considered in Section 5 have not been previously explored in the literature on CFL-reachability. The "transfer functions" used in [BS92] are also similar to the "summary-edges" used here.

Thanks to Theorem 2, which provides a linear-time translation from context-free processes to single-exit HSMs, the linear-time CTL^* model-checking algorithm of Section 6 can also be used for CTL^* model-checking of context-free processes, and hence provides a new improved upper bound for this problem: CTL^* model checking of context-free processes can now be solved in linear-time, instead of quadratic-time.

Our other results, however, cannot even be stated in the context of context-free or pushdown processes. For example, the distinction between single-entry and multiple-entry HSMs has no obvious counterpart in

the literature on pushdown automata, and the linear bounds for single-entry multiple-exit HSMs presented here could not be derived from such previous work.

References

[AG00] R. Alur and R. Grosu. Modular Refinement of Hierarchic State Machines. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 390–402, January 2000.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKY99] R. Alur, S. Kannan, and M. Yannakakis. Communicating Hierarchical State Machines. In *Proceedings of the 26th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer-Verlag, 1999.

[AY98] R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. In *Proceedings of the Sixth ACM Symposium on the Foundations of Software Engineering (FSE'98)*, pages 175–188, 1998.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. of CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1997.

[BS92] O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In *Proc. of CONCUR'92*, 1992.

[BS97] O. Burkart and B. Steffen. Model Checking the Full Modal Mu-Calculus for Infinite Sequential Processes. In *Proc. of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429, Bologna, 1997. Springer-Verlag.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CM90] B. Caucal and R. Monfort. On the Transition Graphs of Automata and Grammars. In *Graph Theoretic Concepts in Computer Science*, volume 484 of *Lecture Notes in Computer Science*, pages 311–337. Springer-Verlag, 1990.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proceedings of the 12th Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247, Chicago, July 2000. Springer-Verlag.

[EL86] E. A. Emerson and C. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990.

[FBW97] A. Finkel, B.Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, New York, NY, October 1995. ACM Press.

[HRSR94] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New York, NY, December 1994. ACM Press.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Rep98] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November 1998. Special issue on program slicing.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM Press.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[Wal96] I. Walukiewicz. Pushdown Processes: Games and Model-Checking. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74, New Brunswick, August 1996. Springer-Verlag.

[Yan90] M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. of the Symp. on Princ. of Database Syst.*, pages 230–242, 1990.