Model Checking of Unrestricted Hierarchical State Machines

Michael Benedikt[1] Patrice Godefroid[1] Thomas Reps[2]

[1] Bell Laboratories, Lucent Technologies, {benedikt,god}@bell-labs.com
[2] University of Wisconsin, reps@cs.wisc.edu

Abstract. Hierarchical State Machines (HSMs) are a natural model for representing the behavior of software systems. In this paper, we investigate a variety of model-checking problems for an extension of HSMs in which state machines are allowed to call each other recursively.

1 Introduction

Hierarchical State Machines (HSMs) are finite-state machines whose states themselves can be other machines. HSMs form the basis of several commercial modeling languages, such as StateCharts, ObjecTime, and UML. Various verification problems for HSMs without recursion have been studied in [5, 4, 3].

In this paper, we investigate an extension of HSMs in which 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 the 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 several verification problems for 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 via new verification algorithms. We present an LTL model-checking algorithm for unrestricted HSMs. This algorithm shows 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 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. The 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 an improved upper bound for CTL* model checking of context-free processes.

2 Unrestricted Hierarchical State Machines

A (flat) *Kripke structure* K over a set of atomic propositions P 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 *unrestricted hierarchical state machines* (HSMs) M over a set P of atomic propositions; these consist of 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*.
- 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 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 [5]; however, we permit component structures to call each other recursively. An example of an unrestricted HSM is shown in Fig. 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.) To be able to find all of the boxes that call a given component machine j, we define $callers(j) = \{(b, i) \mid Y_i(b) = j\}$.

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:

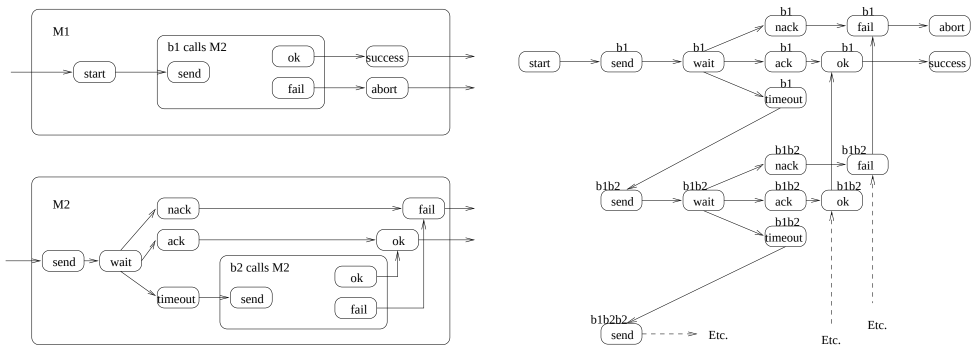

Fig. 1. An example of an unrestricted HSM (left) and its expansion (right). The top-level structure M_1 has one box, which calls structure M_2. Structure M_2 models an attempt to send a message; if no positive or negative acknowledgment is received, a timeout occurs and a recursive call to M_2 is performed.

- $S \subseteq \bigcup_{i=1}^{n} N_i \times (\bigcup_{i=1}^{n} B_i)^*$.
- R is the set of transitions $((v,w),(v',w'))$ that satisfy any of the following:
  - $(v,v') \in E_i$, $v,v' \in N_i$ and $w = w'$.
  - $(v,(b',e')) \in E_i$, $v \in N_i$, $v' = e'$, and $w' = wb'$.
  - $((b,x),v') \in E_i$, $v = x$, $v' \in N_i$, and $w = w'b$.
  - $((b,x),(b',e')) \in E_i$, $v = x$, $v' = e'$, and $w' = w''b'$ with $w = w''b$.
- $L : S \mapsto 2^P$ is defined by $L((v,w)) = X_i(v)$ with $v \in N_i$.

The (infinite) expansion $K(M_1)$ of the HSM of Fig. 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"). 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

Unrestricted HSMs are 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 concision 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. Given two Kripke structures $M_1 = (S_1, R_1, L_1)$ and $M_2 = (S_2, R_2, L_2)$, a binary relation $\mathcal{B} \subseteq S_1 \times S_2$ is a *bisimulation relation* if $(s_1, s_2) \in \mathcal{B}$ implies: (1) $L_1(s_1) = L_2(s_2)$, (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 (3) 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* [9] 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 [14] is the prefix transition graph of an alphabetic rewrite system [9], 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 1. *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.*

The converse of the previous theorem also holds:

Theorem 2. *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., [14]) 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 3. *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:

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

Fig. 2. Complexity bounds derived from Sect. 3 and previously known results. (Complexity bounds are given in terms of the size of the HSM.)

Theorem 4. *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 [9] that there exist pushdown processes that are not bisimilar to any context-free processes, we obtain the following result:

Theorem 5. *There exist multiple-exit HSMs whose expansion is not bisimilar to the expansion of any single-exit HSM.*

4 Complexity of Verification Problems for 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* [10]. Given an unrestricted HSM M and a set $T \subseteq \bigcup_{i=1}^{n} N_i$ of distinguished nodes, the *reachability problem* is 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.

Since restricted HSMs are special cases of unrestricted HSMs, it is worth reviewing some of the results presented in [5] for the restricted case. Lines 2 and 3 of Fig. 2 summarize the results of [5] concerning the complexity of the verification problems considered here, except for CTL* model checking, which was not discussed in [5]. 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 [5] 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.

For single-exit unrestricted HSMs, Theorem 2 implies that 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 [7, 20], and since LTL model checking

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

Fig. 3. Improved complexity bounds for unrestricted HSMs. The improved bounds obtained in Sects. 5 and 6 are highlighted in italic.

for one-state pushdown automata can be solved in time linear in the size of the pushdown automaton [12, 13], 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 [7]. Finally, since the μ-calculus model-checking algorithm of [8] 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* [11], CTL* model checking for single-exit HSMs can be solved in time quadratic in the size of the HSM.

In the case of multiple-exit unrestricted HSMs, Theorem 4 implies 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 [13, 12], 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 [20] 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 [6], we can deduce from Theorems 3 and 4 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 [8] for pushdown processes and the full μ-calculus, which contains CTL*, and the EXPTIME-hardness result of [20] 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 Fig. 2 summarize the results obtained from the foregoing discussion.

In the remainder of this paper, we present two improvements to the results listed in Fig. 2. First, in Sect. 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 Sect. 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 Fig. 3.

5 LTL Model Checking

Following the automata-theoretic approach to model checking [19], 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 $X'((v, s))$ is *true* if s is an accepting state of $A_{\neg\phi}$, and *false* otherwise. Let T denote the set of nodes of M' where X' is *true*. The LTL model-checking problem for an HSM M and 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 $K(M') = K(M) \times A_{\neg\phi}$, 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 $\mathit{CallEdges}(M') \cup \mathit{ReturnEdges}(M')$, where $\mathit{CallEdges}(M') = \{((b, e), e) \mid e \in I'_i, b \in B'_j, Y'_j(b) = i\}$ and $\mathit{ReturnEdges}(M') = \{(x, (b, x)) \mid x \in O'_i, b \in B'_j, Y'_j(b) = i\}$. This graph finitely and completely represents $K(M')$, 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 definition characterizes the paths of $G(M')$ that correspond to executions of $K(M')$.

Definition 6. Give each box in M' a unique index in the range $1 \ldots |B|$, where $|B|$ is the total number of boxes in M'. For each box b, label the associated call-edges and return-edges with the symbols "$(_b$" and "$)_b$", respectively; label all other edges with "e". A path in $G(M')$ is called a *Bal-path* (resp. *UnbalLeft-path*) iff the word formed by concatenating, in order, the symbols on the path's edges is in the language $L(\mathit{Bal})$ (resp. $L(\mathit{UnbalLeft})$), defined as follows:

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

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

procedure Propagate(e → v: edge, B: Bool)
  if there is no pair of the form (e → v, B′) in PathEdges then
   Insert (e → v, B) into PathEdges
   Insert (e → v, B) into WorkList
  else if (e → v, B′) ∈ PathEdges ∧ B = true ∧ B′ = false then
    PathEdges := (PathEdges − {(e → v, B′)}) ∪ {(e → v, B)}
    WorkList := (WorkList − {(e → v, B′)}) ∪ {(e → v, B)}
  fi
end

PathEdges := ∅; SummaryEdges := ∅; WorkList := ∅
for each entry-node e of some I_i, for 1 ≤ i ≤ n do Propagate(e → e, e ∈ T) od
while WorkList ≠ ∅ do
  Select and remove a pair (e → v, B) from WorkList
  switch v
   case v = (b, e′) ∈ C_i: /* v is a call-node */
     for each (b, x) such that ((b, e′) → (b, x), B′) ∈ SummaryEdges do
      Propagate(e → (b, x), B ∨ B′)
     od
   end case
   case v = x ∈ O_i: /* v is an exit-node */
     for each pair (b, j) ∈ callers(i) do /* b ∈ B_j and Y_j(b) = i */
      if there is no pair of the form ((b, e) → (b, x), B′) in SummaryEdges then
       Insert ((b, e) → (b, x), B) into SummaryEdges
      else if ((b, e) → (b, x), B′) ∈ SummaryEdges ∧ B = true ∧ B′ = false then
        SummaryEdges := (SummaryEdges − {((b, e) → (b, x), B′)}) ∪ {((b, e) → (b, x), B)}
      fi
      for each e′ ∈ I_j such that (e′ → (b, e), B″) ∈ PathEdges do
       Propagate(e′ → (b, x), B ∨ B″)
      od
     od
   end case
   default : /* v ∈ (N_i − O_i) ∪ R_i, i.e., v is not a call-node or an exit-node */
     for each v′ such that v → v′ ∈ E_i do Propagate(e → v′, B ∨ (v′ ∈ T)) od
   end case
  end switch
od
return(SummaryEdges)
```

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

$Bal \rightarrow Bal \;\; Bal$		$UnbalLeft \rightarrow UnbalLeft \;\; (_j \;\; Bal$	$1 \leq j \leq \vert B \vert$
$\quad \vert \;\; (_j \;\; Bal \;\;)_j$	$1 \leq j \leq \vert B \vert$	$\quad \vert \;\; Bal$	
$\quad \vert \;\; e$			
$\quad \vert \;\; \epsilon$			

LTL model checking is carried out directly on the Büchi-constrained product-HSM by means of the two-phase algorithm presented in Figs. 4 and 5. In the first phase, the dynamic-programming algorithm CompSummaryEdges, shown

function ContainsTCycle(M: HSM, $T \subseteq \bigcup_{i=1}^{n} N_i$) **returns** a set of nodes
[1] $\quad$ $\mathit{SummaryEdges}$ = CompSummaryEdges(M, T)
[2] $\quad$ $G = (\bigcup_{i=1}^{n} N_i \cup C_i \cup R_i, \bigcup_{i=1}^{n} E_i \bigcup \mathit{CallEdges}(M) \bigcup \mathit{SummaryEdges})$
[3] $\quad$ SCCSet = FindSCCs(G, I_1) /* I_1 is the set of roots of the depth-first search */
[4] $\quad$ **for** each non-trivial SCC $(\mathit{Nodes}, \mathit{Edges}) \in \mathit{SCCSet}$ **do**
[5] $\quad\quad$ **if** $(\mathit{Nodes} \cap T \neq \emptyset)$ or $(\exists ((b,e) \to (b,x), B) \in \mathit{Edges} : B = \mathit{true})$ **then**
[6] $\quad\quad\quad$ **return**(Nodes)
[7] $\quad\quad$ **fi**
[8] $\quad$ **od**
[9] $\quad$ **return**($\emptyset$)

Fig. 5. An algorithm for detecting T-cycles.

in Fig. 4 is applied to an HSM[2] M with Büchi acceptance condition T to create a set of *summary-edges*. Each summary-edge represents a *Bal*-path between a call-node and a return-node, where the two nodes are associated with the same box. More precisely, CompSummaryEdges creates the set *SummaryEdges*, which consists of pairs of the form $((b,e) \to (b,x), B)$. Summary-edge $((b,e) \to (b,x), B)$ indicates that (i) there exists a *Bal*-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. In addition to tabulating summary-edges, CompSummaryEdges builds up the set *PathEdges*: a *path-edge* $(e \to v, B)$ in *PathEdges* indicates the existence of a *Bal*-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 *Bal*-path summarized by the edge traverses at least one node in T.

It is possible to make two improvements to CompSummaryEdges: first, path-edges in each component structure can be "anchored" at exit-nodes rather than at entry-nodes, and path-edges can be "grown" backwards rather than forwards (a technique also used in [15]); second, path-edges in component structures M_i where $|O_i| < |I_i|$ can be anchored at exit-nodes (and path-edges grown backwards), whereas in other component structures the path-edges can be anchored at entry-nodes (and path-edges grown forwards). Henceforth, we mean the latter version whenever we refer to CompSummaryEdges in what follows.

The second phase of the model-checking algorithm consists of lines [2]–[8] of function ContainsTCycle of Fig. 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 *Bal*-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 of M's call-edges, plus the set of summary-edges computed by CompSummaryEdges (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 *Bal*-paths between call-nodes

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

and return-nodes of the same box, and, in particular, whether such nodes are connected by a Bal-path that contains a T node.

Theorem 7. *Given an HSM M and an LTL formula ϕ, $K(M)$ satisfies ϕ iff the algorithm of Fig. 5 applied to the Büchi-constrained HSM $M \times A_{\neg\phi}$ and its corresponding set T returns $\emptyset$.*

For any component structure M_i, the worst-case time complexity of CompSummaryEdges is equal to I_i, the number of entry-nodes of M_i (or O_i, if the number of exit-nodes is smaller), 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). Therefore, the contribution of M_i to the time complexity of CompSummaryEdges is bounded by $O(\min(I_i, O_i)\ (E_i + \Sigma_{b \in B_i} C_b R_b))$.

The size of the graph G computed by function ContainsTCycle is bounded by $O(\Sigma_{i=1}^{n}\ (E_i + \Sigma_{b \in B_i} C_b R_b + \Sigma_{b \in B_i} C_b)$, and finding the strongly connected components of G can be carried out in time linear in the size of G (e.g., see [1]). Thus, the total worst-case cost of ContainsTCycle is bounded by $O(\Sigma_{i=1}^{n}\ [\min(I_i, O_i)\ (E_i + \Sigma_{b \in B_i} C_b R_b)])$. In the case of single-entry, single-exit, and single-entry single-exit HSMs, this bound simplifies as follows:

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

where E, R, C, and B are the total numbers of ordinary edges, return-edges, call-edges, and boxes, respectively.

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 $\min(I'_i, O'_i)$ is bounded by the fixed constant $|S_{\neg\phi}|$. Thus, *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.*

6 CTL* Model Checking for Single-Exit HSMs

In this section, we present a CTL* model-checking algorithm for single-exit HSMs that 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 operators $\neg$ (not) and $\vee$ (or). Two types of CTL^* formulas, path formulas and state formulas, are defined by mutual induction. Every atomic 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

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

Fig. 6. The function SPLIT.

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 be viewed as a boolean combination of existential formulas. An existential formula is either an atomic proposition or a CTL* state formula of the form $E\ \rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$, where ρ is an LTL formula over propositions $p(\gamma_1), \ldots, p(\gamma_n)$ in which each proposition $p(\gamma_i)$ is substituted by the corresponding CTL* state formula γ_i. (For a description of the semantics of CTL*, see [10].)

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 contents 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 [7, 8, 5]. 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 $exit$, which 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 SPLIT given in Fig. 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 $(\beta, \delta) \in$SPLIT(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 SPLIT, 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$(\phi) \geq j$, 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

function MAKE_CONT(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 $M_i = (N_i, B_i, I_i, O_i, X_i, Y_i, C_i, R_i, E_i)$ */
M_1 = TopLevelMachine(M)
for each $\gamma \in F$ with $\gamma = E\ \rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$ **do**
N(γ) = LTLALG($E\ \rho, M$) /* Precompute all the LTL results needed */
for each $(\beta, \delta) \in$ SPLIT(ρ)
N(β) = LTLALG(E $(\beta \wedge F\ exit), M$)
for each $M_i \in M$ **do**
Nodes$_1$(M_i,γ) = $N_i \cap$N(γ)
for each $(\beta, \delta) \in$ SPLIT(ρ) **do**
Nodes$_2$(M_i,β) = $N_i \cap$N(β)
od
od
OLDCONT = $\emptyset$ /* Find the pairs (M_i, c) reachable from (M_1, C) */
CONT = $\{(M_1, C)\}$
while (CONT $\neq$ OLDCONT) **do**
OLDCONT = CONT
for each $(M_i, c) \in$ OLDCONT **do**
for each $\gamma \in F$ with $\gamma = \mathrm{E}\rho(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)$
Sat(M_i, c, γ) =Nodes$_1$(M_i,γ) $\cup$
$\bigcup_{(\beta,\delta) \in \mathrm{SPLIT}(\gamma), c(\mathrm{E}\delta(p(\gamma_1) \leftarrow \gamma_1, \ldots, p(\gamma_n) \leftarrow \gamma_n)) = true}$ Nodes$_2$(M_i,β)
for each b $\in$ Boxes(M_i) **do**
$P_{(b,i,c)} = (Y_i(b), c')$ such that $\forall \gamma \in F : c'(\gamma) = true$ iff $(b, x) \in$ Sat(M_i, c, γ)
CONT = OLDCONT $\cup \{P_{(b,i,c)}\}$
od
od
od
/* Now build the output HSM M^*
$M^* = \{M_{i,c} | (M_i, c) \in$ CONT $\}$
Forall $1 \leq i \leq n$, for all $c \in$ CONT,
$M_{i,c} = (N_i \times \{c\}, B_i \times \{c\}, I_i \times \{c\}, O_i \times \{c\}, X'_{i,c}, Y'_{i,c}, C'_{i,c}, R_i \times \{c\}, E_i \times \{c\})$
where
$C'_{i,c} = \{((b, c), (e, c')) | (b, e) \in C_i$ and $(M_k, c') = P_{(b,i,c)}\}$
For all $b \in B_i$, $Y'_{i,c}((b, c)) = (Y_i(b), c')$ with $(M_k, c') = P_{(b,i,c)}$
For all $v \in N_i$, $X'_{i,c}((v, c)) = \{\gamma \in F | v \in$ Sat$(M_i, c, \gamma)\}$
TopLevelMachine(M^*) = $M_{1,C}$
return(M*)

Fig. 7. Construction of the context-dependent HSM.

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 an 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 $K(M)$ with the initial state of K'.

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

Fig. 8. CTL* model-checking algorithm.

Given a closed set F of existential formulas, an HSM M whose nodes are labeled with formulas in $\{\gamma \in F|\text{pd}(\gamma) < \text{pd}(F)\}$, and an F-context C, the function MAKE_CONT presented in Fig. 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 $(M, v) \models_c \gamma$.

MAKE_CONT uses a variant of the LTL model-checking algorithm from Sect. 5, called LTLALG. 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 Sect. 5, except for the following three modifications. First, LTLALG evaluates formulas of the form $E\rho$ instead of $A\rho$. Second, we still need to define how formulas of LTL$^+$ 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 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 Sect. 5 to meet these additional requirements.

By repeatedly invoking MAKE_CONT with cl$^{\leq j}(\phi)$ for increasing values of j, $1 \leq j \leq$pd(ϕ), 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 Fig. 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 8. *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 $CHECK(\phi_i, M, C_\emptyset)$.*

An analysis of the overall complexity of CHECK reveals that the number of contexts over $F = \mathrm{cl}(\phi)$ and the number of pairs of formulas returned by SPLIT 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 MAKE_CONT is bounded independently of the size of M. Hence, the run-time complexity of the function MAKE_CONT 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

Function CompSummaryEdges from Sect. 5 is closely related to algorithms for solving so-called "context-free-language" reachability problems [21, 17], as well as to CFL-reachability-based algorithms for such program-analysis problems as interprocedural slicing [16] and interprocedural dataflow analysis [18, 15]. In particular, the notions of path-edges and summary-edges, and the dynamic-programming technique used to compute such edges in CompSummaryEdges already appeared in this earlier work, although the cycle-detection and LTL model-checking problems considered in Sect. 5 have not been previously explored in the literature on CFL-reachability. The "transfer functions" used in [7] are also similar to the "summary-edges" used here. Results similar to those of Sect. 5 (obtained independently and contemporaneously) are reported in [2].

Thanks to Theorem 1, which provides a linear-time translation from context-free processes to single-exit HSMs, the linear-time CTL^* model-checking algorithm of Sect. 6 can also be used for CTL^* model-checking of context-free processes, and hence provides an improved upper bound for this problem: the problem 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

1. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. R. Alur, K. Etessami, and M. Yannakakis. Analysis of Recursive State Machines. In *To appear in Proceedings of CAV 2001*, Paris, July 2001.
3. 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.

4. 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.
5. 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.
6. 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.
7. O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In *Proc. of CONCUR'92*, 1992.
8. 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.
9. 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.
10. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990.
11. 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.
12. 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.
13. A. Finkel, B.Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. *Electronic Notes in Theoretical Comp. Sc.*, 9, 1997.
14. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
15. 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.
16. 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.
17. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November 1998. Special issue on program slicing.
18. 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.
19. 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.
20. 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.
21. M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. of the Symp. on Princ. of Database Syst.*, pages 230–242, 1990.