Model Checking Procedural Programs

Rajeev Alur, Ahmed Bouajjani, and Javier Esparza

Abstract We consider the model-checking problem for sequential programs with procedure calls. We first present basic algorithms for solving the reachability problem and the fair computation problem. The algorithms are based on two techniques: *summarization*, that computes reachability information by solving a set of fixpoint equations, and *saturation*, that computes the set of all reachable program states (including call stacks) using automata. Then, we study formalisms to specify requirements of programs with procedure calls. We present an extension of the linear temporal logic allowing propagation of information across the hierarchical structure induced by procedure calls and matching returns. Finally, we show how model-checking can be extended to this class of programs and properties.

1 Introduction

We consider the model-checking problem for sequential programs consisting of procedures that call one another, possibly in a recursive manner. We assume that all program variables have a finite range. These programs, called procedural programs or Boolean programs [9], are used as abstractions of C programs in highly influential software verification tools like SLAM [8]. The state of a procedural program has three parts: the current value of the program counter, the current values of the pro-

Rajeev Alur
University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104, USA.
e-mail: alur@cis.upenn.edu

Ahmed Bouajjani
Université Paris Diderot (Paris 7) and Institut Universitaire de France,
LIAFA, Case 7014, 75205 Paris cedex 13, France. e-mail: abou@liafa.univ-paris-diderot.fr

Javier Esparza
Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching bei München, Germany. e-mail: esparza@in.tum.de

gram variables, and the current stack of procedure calls whose execution has not yet finished. Since procedures may be recursive, and the recursion depth in not bounded a priori, the state space of a procedural program may be infinite, and so procedural programs cannot be verified using standard finite-state model-checking algorithms.

We model procedural programs as recursive state machines (RSM) [2]. Each procedure of the program is modeled by a different machine. A machine has a finite number of control states with some distinguished entry and exit points. Control states are connected by edges that correspond to either a local change in the control state, or to a full execution (from an entry to an exit point) of another state machine. The latter case models a procedure call. Recursion is allowed since the dependencies among state machines can be cyclic in general. RSMs with acyclic dependencies among state machines are called hierarchical state machines [7].

The operational semantics of RSMs can be defined in terms of pushdown systems (PDS), state machines whose transitions are labeled with stack operations [13, 30]. Push and pop operations correspond to procedure calls and returns, respectively. The PDS corresponding to a given RSM can be easily computed and has roughly the same size as the RSM itself, and either representation can serve as an input to a verification algorithm, depending on the computational task at hand.

We present two basic techniques for the reachability analysis of RSMs and PDSs. The first one, called summarization, computes reachability information by solving a set of fixpoint equations, and is closely related to inter-procedural data-flow analysis [25, 49]. Roughly speaking, summarization computes the pairs of program counter and program variable values that can be reached from the initial state of the program, but not the stack contents with which they can be reached. For example, after applying summarization we may know that program location 13 can be reached with $x = 3$ and $y = 5$, but not that this can only happen when the current procedure is called from a procedure P. The complete set of reachable program states (i.e., the set of all reachable triples consisting of the current value of the program counter, the current values of the program variables, and the current stack of procedure calls) can be obtained by employing the second technique, called saturation. Saturation takes as input the PDS associated with the RSM, and computes its set of reachable configurations. Since this set may be infinite, saturation does not enumerate its elements, but computes a finite symbolic representation in the shape of a finite-automaton (compare with the BDD-based techniques of Chap. 9 for compactly representing a large but finite set of states). Summarization and saturation can also be applied to the fair computation problem, a core computational problem underlying the analysis of infinite program executions. They have been implemented and extensively applied in tools like Bebop (the model checker inside SLAM [9], MOPED and jMOPED [29, 47, 52], and WALi [38].

In the second part of the chapter we discuss extensions of automata and logics suitable for specifying properties of procedural programs. We show that many natural specifications require relating the truth of propositions at a procedure call with the matching return position. A typical example is the property "if the status of a global variable x is *locked* when a procedure P is called, then its status is guaranteed to be *locked* when the procedure P returns". Asserting such properties is

not possible using formalisms defining regular languages of computations, such as finite-state automata and Linear Temporal Logic (see Chap. 2). Aimed at specifying such properties, the notion of nested words is introduced to represent behaviors of RSMs. They correspond to (finite/infinite) words with additional hierarchical edges that expose the matching between call and return positions. We define automata and logics on nested words, and show that model-checking algorithms for RSMs naturally extend to these formalisms.

In the last section of the paper, we survey some further existing results concerning RSMs/PDSs and their extensions that are relevant to the domain of verification.

2 Models of Procedural Programs

While programs, consisting of assignment statements, if-then-else statements and while loops, can be modeled as *extended state machines*: state machines whose transitions are guarded by and operate on variables. The states or *nodes* of an extended state machine correspond to the control points of the program. The transitions of the machine are labeled either with assignments, or with the Boolean conditions appearing in the conditional statements and the while loops. Figure 1 shows an example of a while program with two boolean variables x, y (top left), and its corresponding extended state machine (top right). The nodes l_1 and l_5 are called the *entry* and *exit* nodes, respectively.

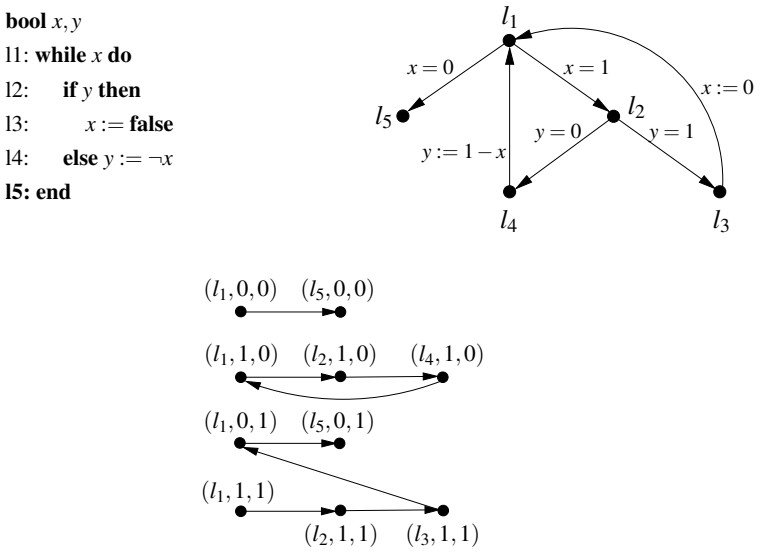


Fig. 1 A program, its extended state machine, and its state machine.

An extended state machine with a set V of variables can be *flattened* into a *state machine*. A node of the state machine is a pair $\langle \ell, v \rangle$, where ℓ is a node of the extended state machine, and v is a valuation of the variables of V. Figure 1 shows at the bottom the state machine obtained by flattening the extended state machine. For instance, the entry node l_1 is split into 4 entry nodes, one for each possible valuation of the variables x and y. Only the nodes of the state machine reachable from the entry nodes are shown.

Procedural programs extend while programs with (possibly recursive) procedures. They can no longer be faithfully modeled by state machines. For this reason we introduce two abstract models of computation, *recursive state machines*, and *pushdown systems*, which play for procedural programs the same role that state machines play for while programs.

2.1 (Extended) Recursive State Machines

Figure 2 shows a procedural program and its corresponding extended recursive state machine (ERSM). The program has two global Boolean variables x and y, and con-

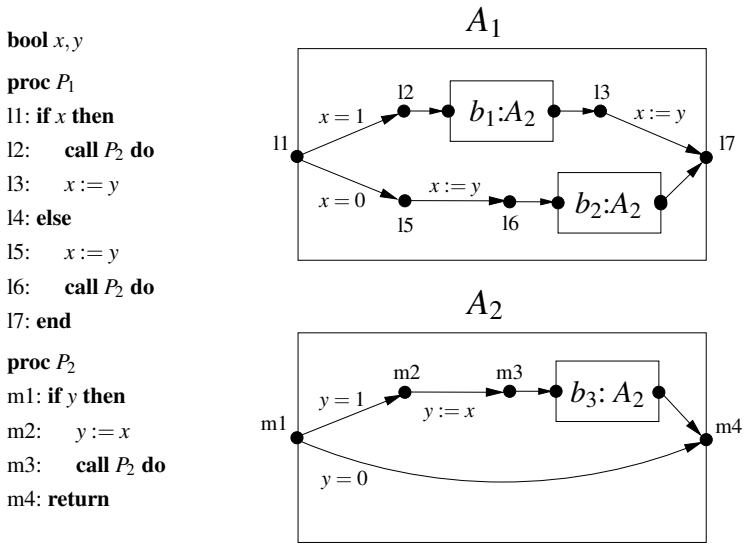


Fig. 2 A procedural program and its extended recursive state machine.

sists of two procedures P_1 and P_2. The ERSM reflects the structure of the program. It consists of two *components* A_1 and A_2, modeling the procedures P_1 and P_2, respectively. The nodes of A_1 and A_2 correspond to the control points of P_1 and P_2; assignments, conditionals, etc. are modeled as for while programs. Moreover, for

each call in procedure P_i to the procedure P_j, the component A_i contains a *box* labeled by A_j. In our example, component A_1 contains two boxes, and component A_2 contains one box. Each box has an *entry port* and an *exit port*. Ports are pairs (n,b), where b is a box, n is an entry or exit node of $A_{Y(b)}$, and $Y(b)$ denotes the component called by the box b. A transition leads from the control point at which the call is made to the entry port, and a second transition leads from the exit port to the return address (the control point at which the computation of the caller continues after the execution of the callee returns).

As in the case of extended state machines, ERSMs can be flattened into *recursive state machines*. Flattening preserves the number of components and boxes, but multiplies the number of nodes and ports. As for state machines, a node or a port of a recursive state machine is a pair $\langle \ell, v \rangle$, where ℓ is a state of the extended machine, and v is a valuation of the variables. Figure 3 shows the result of flattening component A_2 of Fig. 2.

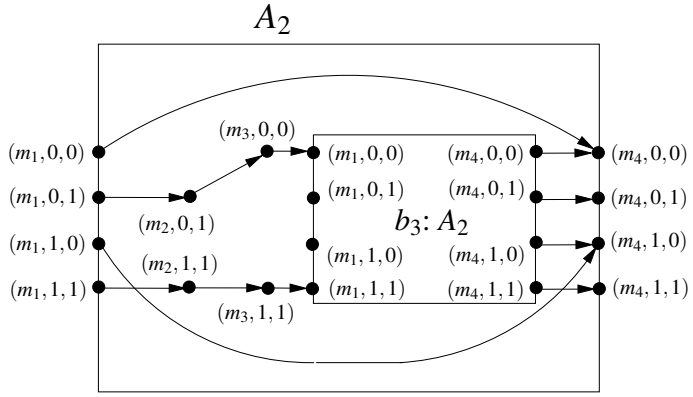


Fig. 3 Result of flattening component A_2 of Fig. 2.

Definition 1. A *recursive state machine* (RSM) is a tuple $\mathcal{M} = (A_1, \ldots, A_k)$ of *components* $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$, where:

- N_i is a finite set of *nodes*, with two distinguished subsets En_i and Ex_i of *entry* and *exit* nodes.
- B_i is a finite set of *boxes*. A box b is labeled with an integer $Y(b) \in \{1, \ldots, k\}$, and has a *call port* (en,b) for each entry node en of $A_{Y(b)}$, and a *return port* (ex,b) for each exit node ex of $A_{Y(b)}$.
- δ_i is a set of *transitions* $u \to v$ where u is either a non-exit node or a return port, and v is either a non-entry node, or a call port.

We denote by

$$En = \bigcup_{i=1}^{k} En_i \qquad Ex = \bigcup_{i=1}^{k} En_i \qquad N = \bigcup_{i=1}^{k} N_i \qquad B = \bigcup_{i=1}^{k} B_i$$

the set of all entry nodes, exit nodes, nodes, and boxes of $\mathcal{M}$, respectively. The set of all ports is $\Pi = (En \cup Ex) \times B$.

Observe that there are four kinds of transitions: $n \to m$ (node-to-node), $n \to (en, b)$ (node-to-call-port), $(ex, b) \to m$ (return-port-to-node), and $(ex, b) \to (en, b')$ (return-port-to-call-port).

In the component of Fig. 3, the entry and exit nodes are the triples with m_1 and m_4 as first element, respectively. The only box is b_3, and $Y(b_3) = 2$. The call ports are $((m_1,0,0),b3),\ldots,((m_1,1,1),b_3)$. In the figure the ports are labeled just by $(m_1,0,0),\ldots,(m_1,1,1)$, and the return ports are $((m_4,0,0),b_3),\ldots,((m_4,1,1),b_3)$.

2.2 Pushdown Systems

Intuitively, an RSM can be executed using a *stack* that at every point in the computation contains the sequence of boxes that have been entered but not yet exited. If a component enters a box b (which corresponds to calling the procedure modeled by the component $A_{Y(b)}$), then b is pushed into the stack; if the component exits the box b (which corresponds to a return from the called procedure), then b is popped. This suggests to give RSMs an operational semantics in terms of pushdown systems.

Definition 2. A *pushdown system* (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of *control states*, Γ is a finite *stack alphabet*, and Δ is a finite set of *rules* of the form $pX \hookrightarrow q\alpha$ with $p,q \in P$, $X \in \Gamma \cup \{\varepsilon\}$, and $\alpha \in \Gamma^*$.

A *configuration* of a PDS is a string of the form $p\sigma$, where $p \in P$ and $\sigma \in \Gamma^*$. The transition system associated to a PDS is the graph having the configurations as vertices, and an edge $c \to c'$ between two configurations c and c' if there is a rule $pX \hookrightarrow q\alpha$ and a word $\sigma \in \Gamma^*$ such that $c = pX\sigma$ and $c' = q\alpha\sigma$. We then say that c is an *immediate predecessor* of c' and c' an *immediate successor* of c.

Figure 4 shows a PDS and a fragment of its transition system. Notice that the transition system of a PDS may be infinite, even if we only consider the configurations reachable from some initial configuration.

2.3 From RSMs to PDSs

Loosely speaking, the PDS associated with an RSM is the pushdown machine that executes the RSM. In programming terms, an RSM is a formal model of a procedural program, and its corresponding PDS is a formal model of the executable code of the program.

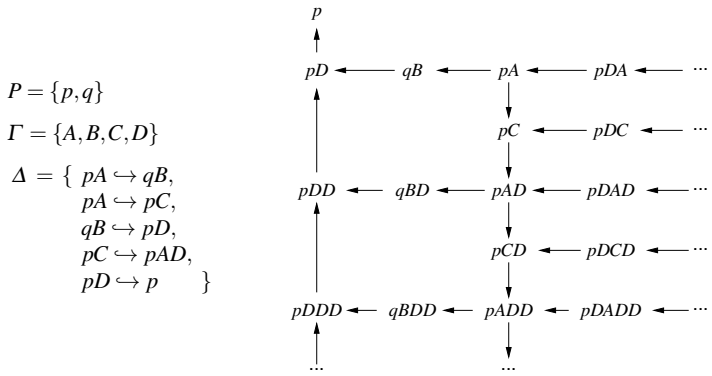


Fig. 4 A PDS and its transition system.

Formally, the PDS $\mathcal{P}_{\mathcal{M}} = (P_{\mathcal{M}}, \Gamma_{\mathcal{M}}, \Delta_{\mathcal{M}})$ corresponding to an RSM $\mathcal{M} = (A_1, \ldots, A_k)$ is defined as follows:

- $P_{\mathcal{M}} = N$ is the set of all nodes of $\mathcal{M}$;
- $\Gamma_{\mathcal{M}} = B$ is the set of all boxes of $\mathcal{M}$; and
- $\Delta_{\mathcal{M}}$ is the set containing
  - a rule $n \hookrightarrow m$ for each transition $n \to m$;
  - a rule $n \hookrightarrow en\,b$ for every transition $n \to (en, b)$;
  - a rule $ex\,b \hookrightarrow m$ for every transition $(ex, b) \to m$; and
  - a rule $ex\,b \hookrightarrow en\,b'$ for every transition $(ex, b) \to (en, b')$.

Observe that the PDS has exactly one rule for each transition of the RSM.

As an example, for the RSM obtained by flattening the ERSM of Fig. 2, we get $P_{\mathcal{M}} = \{l_1, \ldots, l_7, m_1, \ldots, m_4\} \times \{0,1\} \times \{0,1\}$ and $\Gamma_{\mathcal{M}} = \{b_1, b_2, b_3\}$. Examples of rules of $\Delta_{\mathcal{M}}$ are

$$\begin{array}{rlll} (m_1,0,0) \hookrightarrow (m_4,0,0) & \text{derived from} & (m_1,0,0) \to (m_4,0,0) \\ (m_3,0,0) \hookrightarrow (m_1,0,0)\,b_3 & \text{derived from} & (m_3,0,0) \to ((m_1,0,0),b_3)) \\ (m_4,0,1)\,b_3 \hookrightarrow (m_4,0,1) & \text{derived from} & ((m_4,0,1),b_3) \to (m_4,0,1) \end{array}$$

Observe that every rule $pX \hookrightarrow q\alpha$ of a PDS associated with an RSM satisfies $|\alpha| \leq 2$.

3 Basic Verification Algorithms

We proceed to define basic computational problems that are useful for checking safety and liveness properties of RSMs.

Definition 3. Let $\mathcal{M} = (A_1, \ldots, A_k)$ be an RSM, where $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$ for each $i \in \{1, \ldots, k\}$. Let $\xrightarrow{*}$ be the reflexive-transitive closure of the relation $\rightarrow$ between configurations, i.e., $\xrightarrow{*} = \bigcup_{n=0}^{\infty} (\rightarrow)^n$ and let $\xrightarrow{+} = \bigcup_{n=1}^{\infty} (\rightarrow)^n$.

The *state reachability problem* is to determine, given an entry node $p \in En$ and a node $q \in P_{\mathcal{M}}$, if $p \xrightarrow{*} q\sigma$ for some $\sigma \in \Gamma_{\mathcal{M}}^*$.

The *configuration reachability problem* is to determine, given two configurations $p\sigma$ and $p'\sigma'$, where $p, p' \in P_{\mathcal{M}}$ and $\sigma, \sigma' \in \Gamma_{\mathcal{M}}^*$, if $p\sigma \xrightarrow{*} p'\sigma'$.

The *fair computation problem* is to determine, given an entry node $p \in En$ and a finite set of *repeat* entry nodes $F \subseteq En$, whether p has an F-fair computation, i.e., an infinite sequence of configurations $p_0\sigma_0, p_1\sigma_1, p_2\sigma_2, \cdots$ such that (1) $p_0 = p$ and $\sigma_0 = \varepsilon$, (2) $p_i\sigma_i \xrightarrow{+} p_{i+1}\sigma_{i+1}$ for every $i \geq 0$, and (3) $p_j \in F$ for infinitely many $j \geq 0$.

Consider the RSM of Fig. 3 on its own (not as part of the larger RSM obtained by flattening the extended RSM of Fig. 2). Choose p as the entry node $(m_1, 0, 1)$, and q as the node $(m_4, 1, 0)$. The state reachability problem for this choice of p and q formalizes the question whether some computation of procedure P_2 of Fig. 2 with $x = 0$ and $y = 1$, can reach the point m_4 with $x = 1$ and $y = 0$. However, since the procedure P_2 is recursive, m_4 can be visited several times along a computation, and so the question is whether at one of these visits x and y are equal to 1 and 0, respectively, not whether these are the values *after termination*. To check this we can use the configuration reachability problem: Procedure P_2 terminates with $x = 1$ and $y = 0$ if and only if the RSM can reach the configuration with $(m_4, 1, 0)$ as control state and empty stack. Notice that, in general, we cannot reduce termination (a liveness property) to reachability (a safety property), but inspection of this program shows that it terminates if and only if it reaches m_4 with no pending procedure calls.

The problem of checking liveness properties can be easily reduced to the *fair computation problem* by means of the automata-theoretic techniques introduced in Chap. 7.

3.1 The State Reachability Problem: Computing Summaries

In this section we show how to solve the state reachability problem using the *summarization technique*. We present the technique for RSMs.

Let $\mathcal{M} = (A_1, \ldots, A_k)$ be an RSM, and let $\Theta_{\mathcal{M}} = N \cup \Pi$ be the set containing all nodes and all ports in $\mathcal{M}$. For every $i \in \{1, \ldots, k\}$, consider the relation $R_i \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ given by

$$(p, q) \in R_i \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_i, \text{ and } p \xrightarrow{*} q.$$

Further, for every $i, j \in \{1, \ldots, k\}$ consider the relation $R_{(i,j)} \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ given by

$$(p, q) \in R_{(i,j)} \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_j, \text{ and } p \xrightarrow{*} q\sigma \text{ for some } \sigma \in \Gamma_{\mathcal{M}}^*.$$

We call these relations *summaries*, since they can be seen as the result of summarizing executions by their initial and final states. Now, let $R = \bigcup_{i,j=1}^{k} R_{(i,j)}$. Clearly, given $p \in En$ and $q \in N$, solving the state reachability problem consists of checking whether $(p,q) \in R$.

It is easy to see that for every $i, j \in \{1,\ldots,k\}$ the relations R_i and $R_{(i,j)}$ are the smallest relations satisfying the following conditions (where we write $R_i(p,q)$ and $R_{(i,j)}(p,q)$ instead of $(p,q) \in R_i$ and $(p,q) \in R_{(i,j)}$):

S1: $R_i(e,e)$ for every $e \in En_i$.

S2: If $R_i(e,p)$ and $(p,q) \in \delta_i$, where $e \in En_i$,
then $R_i(e,q)$.

S3: If $R_i(e,(p,b))$ and $R_\ell(p,q)$,
where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in Ex_\ell$,
then $R_i(e,(q,b))$.

S4: If $R_i(e,q)$
then $R_{(i,i)}(e,q)$.

S5: If $R_i(e,(p,b))$ and $R_{(\ell,j)}(p,q)$,
where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in N_j$,
then $R_{(i,j)}(e,q)$.

The relations R_i and $R_{(i,j)}$ can be simultaneously computed by, starting from the empty relations, iteratively applying the rules **S1**-**S5** until stabilization. Since the set $\Theta_{\mathcal{M}}$ is finite, the computation necessarily terminates. This yields a decision procedure for the state reachability problem of polynomial complexity. More precisely, as shown in [2], reachability can be solved in time $O(|\mathcal{M}|\theta_e^2)$ and space $O(|\mathcal{M}|\theta_e)$, where $|\mathcal{M}|$ is the total number of nodes and transitions in the RSM, and θ_e is the maximum number of entry nodes of a component, i.e., $\theta_e = \max_{i=1}^{k} |En_i|$.

It is straightforward to define a dual algorithm that starts at the exit nodes and computes the summaries *backwards*. For instance, in the dual algorithm the rule **S2** is replaced by the dual rule

D2 : If $R_i(p,x)$ and $(q,p) \in \delta_i$, where $x \in Ex_i$,
then $R_i(q,x)$.

The dual algorithm runs in $O(|\mathcal{M}|\theta_x^2)$ time, where θ_x is the maximum number of exit nodes of a component, i.e., $\theta_x = \max_{i=1}^{k} |Ex_i|$. The primal and dual rules can be combined component-wise: if the number of entry nodes of component A_i is smaller than its number of exit nodes, then we compute R_i from the entry nodes using the primal rules, otherwise from the exit nodes using the dual rules. The complexity of this algorithm is $O(|\mathcal{M}|\theta^2)$, where $\theta = \max_{i=1}^{k} \min(|En_i|,|Ex_i|)$. Since $\theta \in O(|\mathcal{M}|)$, the combined algorithm has cubic complexity in $|\mathcal{M}|$. For the class of RSMs in which θ is bounded by a constant (which contains in particular procedural programs whose procedures can only return a fixed number of values, say a Boolean), reachability can be decided in linear time.

As an example, we compute part of the relations for the RSMs obtained by flattening the extended machines of Fig. 2. In particular, we show that

$$R_1((\ell_1,1,0),(\ell_7,0,0))$$

holds, i.e., if we start at location ℓ_1 with $x=1$ and $y=0$, we may reach location ℓ_7 with $x=0=y$.

We first apply rule (**S1**) twice and obtain

$$R_1((\ell_1,1,0),(\ell_1,1,0)) \tag{1}$$
$$R_2((m_1,1,0),(m_1,1,0)) \tag{2}$$

Now we use rule (**S2**) to establish relations corresponding to single edges in the graphs of the RSMs. From (1) and $((\ell_2,1,0),((m_1,1,0),b_1)) \in \delta_1$, and from (2) and $((m_1,1,0),(m_4,1,0)) \in \delta_2$, respectively, we obtain

$$R_1((\ell_1,1,0),((m_1,1,0),b_1)) \tag{3}$$
$$R_2((m_1,1,0),(m_4,1,0)) \tag{4}$$

Next we apply rule (**S3**) to (3) and (4). Together with $Y_1(b_1)=2$ we get

$$R_1((\ell_1,1,0),((m_4,1,0),b_1)) \tag{5}$$

Then we apply rule (**S2**) to (5) using the transition $(((m_4,1,0),b_1),(\ell_3,1,0)) \in \delta_1$, to obtain

$$R_1((\ell_1,1,0),(\ell_3,1,0)) \tag{6}$$

Finally, applying rule (**S2**) to (6) and $((\ell_3,1,0),(\ell_7,0,0)) \in \delta_1$ yields

$$R_1((\ell_1,1,0),(\ell_7,0,0)) \tag{7}$$

and we are done.

Let us now show

$$R_{(1,2)}((\ell_1,0,1),(m_3,1,1))$$

Applying rule (**S1**) and then (**S2**) to the entry nodes $(\ell_1,0,1)$ and $(m_1,1,1)$ we obtain

$$R_1((\ell_1,0,1),((m_1,1,1),b_2)) \tag{8}$$
$$R_2((m_1,1,1),(m_3,1,1)) \tag{9}$$

Applying rule (**S4**) to (9) yields

$$R_{(2,2)}((m_1,1,1),(m_3,1,1)) \tag{10}$$

Finally, applying rule (**S5**) to (8) and (10) we get

$$R_{(1,2)}((\ell_1,0,1),(m_3,1,1)) \tag{11}$$

Next we show that calls to A_2 starting from $(m_1,1,1)$ never return, i.e., that $R_2((m_1,1,1),n)$ does not hold for any exit node n of A_2. Since every path from the entry node $(m_1,1,1)$ leads to $(m_2,1,1)$ and $(m_3,1,1)$, rule (**S2**) only allows to derive $R_2((m_1,1,1),(m_2,1,1))$ and $R_2((m_1,1,1),((m_1,1,1),b_3))$. Since no other rule can be applied, we are done.

Finally, similar reasoning shows that no exit node of A_1 is reachable from $(\ell_1,0,1)$. Indeed, this follows easily from the fact that rule (**S3**) cannot be applied to (8).

3.2 The Fair Computation Problem

It is shown in [13] that the fair computation problem can be reduced to the state reachability problem. The key observation, not difficult to prove, is that, given $p \in En$ and $F \subseteq En$, the node p has an F-fair computation if and only if there exists $p' \in F$ such that

$$p \xrightarrow{*} p'\sigma \text{ for some } \sigma \in \Gamma_{\mathcal{M}}^* \quad \text{and} \quad p' \xrightarrow{+} p'\sigma' \text{ for some } \sigma' \in \Gamma_{\mathcal{M}}^*.$$

This reduction allows us to solve the fair computation problem using summarization. We define a new reachability relation $R' \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ (in addition to the relation R defined in Sect. 3.1) as follows:

$$R'(p,q) \text{ holds if and only if } p \xrightarrow{+} q\sigma \text{ for some } \sigma \in \Gamma_{\mathcal{M}}^*.$$

Then, by the observation above, p has a F-fair computation if and only if there exists $p' \in F$ such that $R(p,p')$ and $R'(p',p')$.

The relation R' can be computed similarly to the relation R in Sect. 3.1. For every $i \in \{1,\ldots,k\}$, define $R'_i \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ by

$$R'_i(p,q) \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_i, \text{ and } p \xrightarrow{+} q.$$

Further, for every $i,j \in \{1,\ldots,k\}$, define $R'_{(i,j)} \subseteq \Theta_{\mathcal{M}} \times \Theta_{\mathcal{M}}$ by

$$R'_{(i,j)}(p,q) \text{ iff } p \in En_i,\ q \text{ is a node or port of } A_j, \text{ and } p \xrightarrow{+} q\sigma \text{ for some } \sigma \in \Gamma_{\mathcal{M}}^*.$$

We clearly have $R' = \bigcup_{i,j=1}^{k} R'_{(i,j)}$.

For all $i,j \in \{1,\ldots,k\}$, the relations R'_i and $R'_{(i,j)}$ are the smallest relations such that:

S2': If $R_i(e,p)$ or $R'_i(e,p)$, and $(p,q) \in \delta_i$, where $e \in En_i$,
then $R'_i(e,q)$.

S3': If $R'_i(e,(p,b))$ and $R_\ell(p,q)$,
where $e \in En_i$, $Y_j(b) = \ell$, $p \in En_\ell$, and $q \in Ex_\ell$,
then $R'_i(e,(q,b))$.

S4': If $R'_i(e,q)$, where $e \in En_i$,
then $R'_{(i,i)}(e,q)$.

S5': If $R'_i(e,(p,b))$ and $R_{(\ell,j)}(p,q)$,
where $e \in En_i$, $Y_i(b) = \ell$, $p \in En_\ell$, and $q \in N_j$,
then $R'_{(i,j)}(e,q)$.

The relations can again be computed by applying the rules until stabilization. The time complexity is again cubic in $|\mathcal{M}|$, and linear if each component has a small number of either enter or exit nodes [2].

The model-checking problem for Linear Temporal Logic can be reduced to the fair computation problem using the automata-theoretic techniques of Chap. 7 and Sec. 4.

3.3 The Configuration Reachability Problem: Saturating Automata

In this section we solve the configuration reachability problem for RSMs and PDSs. We present two decision procedures for PDSs. The procedures for RSMs are obtained by applying the translation from RSMs to PDSs shown in Sect. 2.3.

Given two configurations $p\sigma$ and $p'\sigma'$ of a PDS, we can decide whether $p\sigma \xrightarrow{*} p'\sigma'$ holds by computing the set of all configurations reachable from $p\sigma$ and checking whether $p'\sigma'$ belongs to it, or by computing all configurations from which $p'\sigma'$ can be reached and checking whether $p\sigma$ belongs to it. Since these sets may be infinite, we have to explain the meaning of "compute". A configuration $p\sigma$ of a PDS can be seen as a word over the union of the set of control states and stack symbols, and so a set of configurations is a language over the same alphabet. Recall that a language is regular if it is recognized by a finite automaton. It turns out that, given a regular set C of configurations, the set of configurations reachable from C and the set of configurations from which C can be reached are again regular. This theorem, which can be traced back to Büchi (see Chap. 5 of [16]), allows us to define "computing the set" as "computing a finite automaton recognizing the set".

We fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ for the rest of the section, and let $\mathcal{C}$ denote the set of all configurations of $\mathcal{P}$. The successor function $post\colon 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ of $\mathcal{P}$ is defined as follows: c belongs to $post(C)$ if some immediate predecessor of c belongs to C. The reflexive and transitive closure of $post$ is denoted by $post^*$ and so, given a set C of configurations, $post^*(C)$ denotes the set of configurations reachable from C. Similarly, we define $pre(C)$ as the set of immediate predecessors of elements in C and pre^* as the reflexive and transitive closure of pre.

It is convenient to define a variant of finite automata tailored for the task of representing sets of configurations of $\mathcal{P}$. A $\mathcal{P}$-*automaton* is an automaton with Γ as its alphabet, and P as the set of initial states. Formally, a $\mathcal{P}$-automaton is an automaton $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ where Q is the finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is the set of *transitions*, $P \subseteq Q$ is the set of *initial states* and $F \subseteq Q$ the set of *final states*.

All the automata used in this section are $\mathcal{P}$-automata, and so we drop the $\mathcal{P}$ from now on. An automaton *accepts* or *recognizes* a configuration pw if $p \xrightarrow{\sigma} q$ for some $q \in F$, where $p \xrightarrow{\sigma} q$ denotes that there is a path from state p to state q labeled by σ. A set of configurations of $\mathcal{P}$ is *regular* if it is recognized by some automaton.

In the next sections we present algorithms that given an automaton recognizing a set C of configurations compute automata recognizing $post^*(C)$ and $pre^*(C)$. We start with $pre^*(C)$, since in this case the algorithm is a bit simpler.

3.3.1 Computing $pre^*(C)$ for a Regular Set C by Saturation

The input to our algorithm is an automaton $\mathcal{A}$ accepting C. Without loss of generality, we assume that $\mathcal{A}$ has no transitions leading to an initial state (by adding new initial states if necessary, every automaton can be easily transformed into another one satisfying this condition and recognizing the same language). We compute $pre^*(C)$ as the language accepted by an automaton $\mathcal{A}_{pre^*}$ obtained from $\mathcal{A}$ by means of a *saturation procedure*. The procedure adds new transitions to $\mathcal{A}$, but no new states. New transitions are added according to the following *saturation rule*:

> If $p\gamma \hookrightarrow p'\sigma$ and $p' \xrightarrow{\sigma} q$ in the current automaton, then add a transition (p, γ, q).

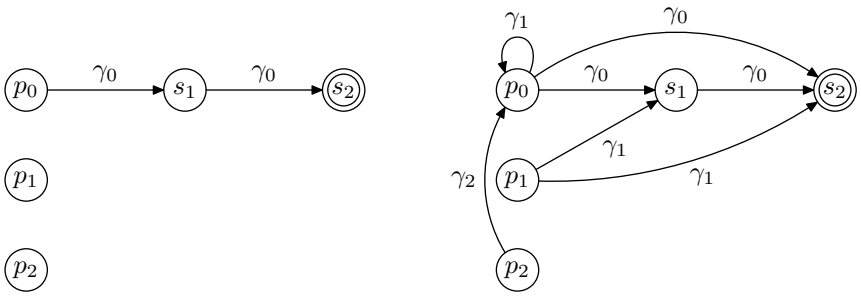
Notice that we can have $\sigma = \varepsilon$, in which case $p' = q$, and that all new transitions start at initial states.

Before explaining the intuition for the rule, let us illustrate the procedure by means of an example. Let $\mathcal{P}$ be the pushdown shown at the top of Fig. 5, and let $\mathcal{A}$ be the automaton recognizing the singleton set $C = \{p_0\gamma_0\gamma_0\}$, shown on the left. The automaton $\mathcal{A}_{pre^*}$ is shown on the right. The saturation procedure adds five additional transitions. The table at the bottom of the figure gives for each new transition of the automaton the transition rule $p\gamma \hookrightarrow p'\sigma$ of the PDS and the path $p' \xrightarrow{\sigma} q$ of the current automaton used to apply the saturation rule. The procedure eventually terminates because the number of possible new transitions is finite.

The intuition for the saturation rule is as follows. Imagine that before adding the transition (p, γ, q) as indicated in the rule, the automaton accepts a configuration $p'\sigma\tau$ by means of a run $p' \xrightarrow{\sigma} q \xrightarrow{\tau} q'$ leading to a final state q'. This means that $p'\sigma\tau \in pre^*(C)$. Since $p\gamma \hookrightarrow p'\sigma$, we have $p\gamma\tau \in pre^*(C)$, and so the automaton should also accept $p\gamma\tau$. This is precisely what the saturation rule achieves: after adding the transition (p, γ, q) the automaton has the run $p \xrightarrow{\gamma} q \xrightarrow{\tau} q'$, and so it accepts $p\gamma\tau$.

$$P = \{\ p_0, p_1, p_2\ \} \quad \Gamma = \{\ \gamma_0, \gamma_1, \gamma_2\ \}$$

$$\Delta = \{\ p_0\gamma_0 \hookrightarrow\ p_1\gamma_1\gamma_0\ ,\ \ p_2\gamma_2 \hookrightarrow\ p_0\gamma_1,\ \ p_1\gamma_1 \hookrightarrow\ p_2\gamma_2\gamma_0\ ,\ \ p_0\gamma_1 \hookrightarrow p_0\ \ \}$$



$p\gamma \hookrightarrow p'\sigma$	$p' \xrightarrow{\sigma} q$	New transition
$p_0\gamma_1 \hookrightarrow p_0$	$p_0 \xrightarrow{\varepsilon} p_0$	(p_0, γ_1, p_0)
$p_2\gamma_2 \hookrightarrow p_0\gamma_1$	$p_0 \xrightarrow{\gamma_1} p_0$	(p_2, γ_2, p_0)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_2 \xrightarrow{\gamma_2\gamma_0} s_1$	(p_1, γ_1, s_1)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_1 \xrightarrow{\gamma_1\gamma_0} s_2$	(p_0, γ_0, s_2)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_2 \xrightarrow{\gamma_2\gamma_0} s_2$	(p_1, γ_1, s_2)

Fig. 5 The automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$ (right).

This argument shows that $pre^*(L(\mathcal{A})) \subseteq L(\mathcal{A}_{pre^*})$ holds. Proving the other inclusion requires some more care, and it is outside the scope of this article. The proof can be found in [13]. This direction relies on the assumption that $\mathcal{A}$ has no transitions leading to an initial state. Notice that without this assumption the algorithm is incorrect.

It is clear that the saturation procedure runs in polynomial time in the size of the PDS $\mathcal{P}$ and the automaton $\mathcal{A}$. An efficient implementation and a more careful complexity analysis can be found in [26]:

Theorem 1. *[26] Given $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{A} = (\Gamma, Q, \delta, P, F)$, the automaton $\mathcal{A}_{pre^*}$ can be computed in $O(n_Q^2 n_\Delta)$ time and $O(n_Q n_\Delta + n_\delta)$ space, where $n_Q = |Q|$, $n_\delta = |\delta|$, and $n_\Delta = |\Delta|$.*

3.3.2 Computing $post^*(C)$ for a Regular Set C by Saturation

We provide an algorithm for the case in which each transition rule $p\gamma \hookrightarrow p'\sigma$ of Δ satisfies $|\sigma| \leq 2$. This restriction is not essential, but leads to a simpler solution. Moreover, any PDS can be transformed into an equivalent one in this form, and the PDSs derived from RSMs directly satisfy this condition.

Our input is an automaton $\mathcal{A}$ accepting C. Again, we assume that $\mathcal{A}$ has no transitions leading to an initial state. We compute $post^*(C)$ as the language accepted by an automaton $\mathcal{A}_{post^*}$ with ε-moves. We denote the relation $(\xrightarrow{\varepsilon})^*\xrightarrow{\gamma}(\xrightarrow{\varepsilon})^*$ by $\stackrel{\gamma}{\Longrightarrow}$. $\mathcal{A}_{post^*}$ is obtained from $\mathcal{A}$ in two stages:

- Add to $\mathcal{A}$ a new state r for each transition rule $r \in \Delta$ of the form $p\gamma \hookrightarrow p'\gamma'\gamma''$, and a transition (p', γ', r).
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

> If $p\gamma \hookrightarrow p'\varepsilon \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, then add a transition (p', ε, q).
>
> If $p\gamma \hookrightarrow p'\gamma' \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, then add a transition (p', γ', q).
>
> If $r = p\gamma \hookrightarrow p'\gamma'\gamma'' \in \Delta$ and $p \stackrel{\gamma}{\Longrightarrow} q$ in the current automaton, then add a transition (r, γ'', q).

Figure 6 shows again the PDS and the automaton from Fig. 5, and, on the right, the automaton $\mathcal{A}_{post^*}$ obtained by applying the algorithm. Since the PDS has two rules of the form $p\gamma \hookrightarrow p'\gamma'\gamma''$, namely $r_1 = p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$, and $r_2 = p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$, the first stage of the algorithm adds to $\mathcal{A}_{post^*}$ two new states r_1, r_2, and two new transitions (p_1, γ_1, r_1) and (p_2, γ_2, r_2). In the second stage the algorithm adds another five transitions. The table at the bottom of the figure gives for each new transition the transition rule $p\gamma \hookrightarrow p'w$ of the PDS, the path $p' \stackrel{\gamma}{\Longrightarrow} q$ of the current automaton, and the saturation rule used to produce it. Again, an efficient implementation and a more careful complexity analysis can be found in [26].

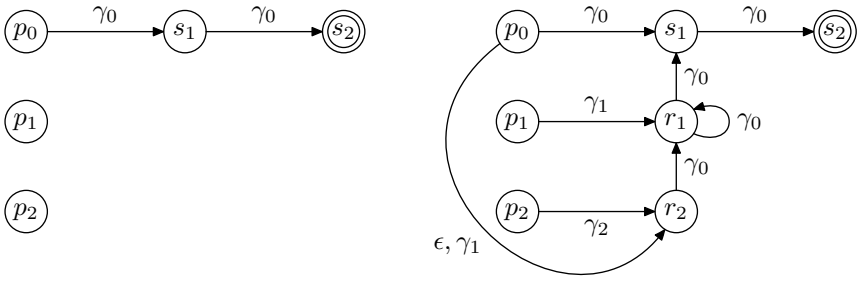
Theorem 2. *[26] Given $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{A} = (\Gamma, Q, \delta, P, F)$, the automaton $\mathcal{A}_{post^*}$ can be computed in $O(n_P n_\Delta (n_Q + n_\Delta) + n_P n_\delta)$ time and space, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, and $n_\delta = |\delta|$.*

3.4 The Generalized Fair Computation Problem

Section 3.2 presents a summarization algorithm for the fair computation problem: given a node p and a set $F \subseteq En$ of repeat nodes, decide if p has an F-fair computation. We now use saturation to solve a generalized version of the problem: compute the set of *all* configurations of $\mathcal{M}$ having an F-fair computation, i.e., an infinite computation that visits infinitely often nodes in F.

Let $\mathcal{P}_\mathcal{M} = (P_\mathcal{M}, \Gamma_\mathcal{M}, \Delta_\mathcal{M})$ be the PDS associated with $\mathcal{M}$. It is easy to see that $p\sigma$ has an F-fair computation if and only if there exists $p' \in F$ such that $p\sigma \xrightarrow{*} p'\sigma'$ for some $\sigma' \in \Gamma_\mathcal{M}^*$ and $p' \xrightarrow{+} p'\tau$ for some $\tau \in \Gamma_\mathcal{M}^*$. We first compute the set *Rep* of states $q \in F$ such that $q \xrightarrow{+} q\sigma$ for some $\sigma \in \Gamma_\mathcal{M}^*$. The set of configurations that

$$P = \{\ p_0, p_1, p_2\ \} \quad \Delta = \{\ p_0\gamma_0 \hookrightarrow\ p_1\gamma_1\gamma_0\ ,\ p_2\gamma_2\ \hookrightarrow\ p_0\gamma_1,$$
$$\Gamma = \{\ \gamma_0, \gamma_1, \gamma_2\ \} \qquad\qquad p_1\gamma_1\ \hookrightarrow\ p_2\gamma_2\gamma_0\ ,\ p_0, \gamma_1 \hookrightarrow p_0 \qquad \}$$



$p\gamma \hookrightarrow p'\sigma$	$p \stackrel{\gamma}{\Longrightarrow} q$	Saturation rule	New transition
$p_2\gamma_2 \hookrightarrow p_0\gamma_1$	$p_2 \stackrel{\gamma_2}{\Longrightarrow} r_2$	second	(p_0, γ_1, r_2)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_0 \stackrel{\gamma_0}{\Longrightarrow} s_1$	third	(r_1, γ_0, s_1)
$p_0\gamma_1 \hookrightarrow p_0$	$p_0 \stackrel{\gamma_1}{\Longrightarrow} r_2$	first	(p_0, ε, r_2)
$p_1\gamma_1 \hookrightarrow p_2\gamma_2\gamma_0$	$p_0 \stackrel{\gamma_0}{\Longrightarrow} r_1$ $(p_0 \stackrel{\varepsilon}{\longrightarrow} r_2 \stackrel{\gamma_0}{\longrightarrow} r_1)$	third	(r_2, γ_0, r_1)
$p_0\gamma_0 \hookrightarrow p_1\gamma_1\gamma_0$	$p_0 \stackrel{\gamma_0}{\Longrightarrow} r_1$	third	(r_1, γ_0, r_1)

Fig. 6 The automata $\mathcal{A}$ (left) and $\mathcal{A}_{post^*}$ (right).

have an infinite fair computation is then equal to $pre^*(Rep\Gamma^*_{\mathcal{M}})$, which is regular and computable using the construction of Sect. 3.3.1.

To compute *Rep* we observe that for every state q we have $q \in Rep$ if and only if $q \in pre^+(q\Gamma^*_{\mathcal{M}})$, where $pre^+(C) = pre(pre^*(C))$. We construct a finite automaton $\mathcal{A}_{pre^+}$ recognizing $pre^+(q\Gamma^*_{\mathcal{M}})$ from an automaton $\mathcal{A}$ recognizing C. Since in Sect. 3.3.1 we already constructed an automaton $\mathcal{A}_{pre^*}$ recognizing $pre^*(C)$, it suffices to provide another construction doing the same for *pre* (instead of pre^*). The construction for pre^+ is the result of concatenating the two, i.e., of applying the construction for *pre* to the result of applying the construction for pre^*.

The construction for *pre* is, not surprisingly, simpler than the one for pre^*. It starts with some preprocessing. Given an input automaton $\mathcal{A} = (\gamma, Q, \delta, P, F)$, the preprocessing adds to it a fresh set $\hat{P} = \{\hat{p} \mid p \in P_{\mathcal{M}}\}$ of states, and changes the set of initial states to $\hat{P}$. Formally, the preprocessing returns the automaton $\hat{\mathcal{A}} = (\gamma, Q \cup \hat{P}, \delta, \hat{P}, F)$. After preprocessing, the construction exhaustively applies the following modification of the saturation rule:

> If $p\gamma \hookrightarrow p'\sigma$ and $p' \stackrel{\sigma}{\longrightarrow} q$ in the current automaton, then add a transition $(\hat{p}, \gamma, q)$.

(The only change is the substitution of $\hat{p}$ for p in the last line.) With this rule all new transitions start from states in $\hat{P}$, and so new transitions cannot generate further transitions. The correctness of the construction is easy to prove.

This algorithm for computing *Rep*, presented in [13], has polynomial complexity, but can be improved. A more efficient procedure involving Tarjan's algorithm for computing strongly connected components is presented in [26].

Theorem 3. *[26] Given $\mathcal{P} = (P, \Gamma, \Delta)$ and a set $F \subseteq P$ of repeat states, the set Rep can be computed in $O(n_P^2 n_\delta)$ time and $O(n_P n_\delta)$ space.*

Recall that the algorithm for the generalized fair computation problem first computes *Rep* and then $pre^*(Rep\Gamma_{\mathcal{M}}^*)$. By Theorem 1, $pre^*(Rep\Gamma_{\mathcal{M}}^*)$ can be computed in $O(|P_{\mathcal{M}}|^2|\Delta_{\mathcal{M}}|)$ time and $O(|P_{\mathcal{M}}||\Delta_{\mathcal{M}}|)$ space, and so the generalized fair computation problem can also be solved within the same time and space bounds.

4 Specifying Requirements

In order to specify requirements of programs modeled by RSMs, we first choose a set Σ of *observables*. Each program statement, or a transition of the RSM, is labeled with an observation $\sigma \in \Sigma$. A (possibly infinite) execution of the RSM then produces a sequence of observations. In this manner, we can associate a language $L(\mathcal{M})$ with the RSM $\mathcal{M}$ as its observational (linear) semantics. Requirements can be written using linear-time specification formalisms such as Linear Temporal Logic (LTL) (see Chap. 2). Given an LTL specification φ over the observables Σ, and an RSM model $\mathcal{M}$, the model checking question is to check if every sequence in $L(\mathcal{M})$ satisfies the formula φ. To solve this problem, we can compile the negation of the specification into a Büchi automaton $A_{\neg\varphi}$ that accepts all computations that violate φ (see Chap. 7) and check that the intersection of the languages of $\mathcal{M}$ and $A_{\neg\varphi}$ is empty. This can be algorithmically solved using the analysis algorithms discussed in Sect. 3. In this setup, even though the language $L(\mathcal{M})$ is context-free (since the underlying model is a pushdown system), the requirement is given as an ω-regular language.

While many analysis problems such as identifying dead code and accesses to uninitialized variables can be captured as regular requirements, many others require inspection of the stack or matching of calls and returns, and are context-free. These include access control requirements such as "a procedure P should be invoked only if the procedure P' belongs to the call-stack," bounds on stack size such as "if the number of interrupt-handling procedures in the call-stack currently is less than 5, then a property p holds," and correctness specifications using pre and post conditions such as "if the property p holds when a procedure P is invoked, the procedure P must return, and the property q holds upon return." When viewed in isolation, each of these requirements is a context-free language, and checking context-free requirements of RSMs (or pushdown systems) is undecidable in general. However, the key feature of these example requirements is that the stacks in the model and the

requirement are correlated: while the stacks are not identical, the two synchronize on when to push and when to pop, and are always of the same depth. To formalize this, we view an execution of the program as a *nested word*, which consists of a linear sequence of states (or observations), augmented with nesting edges connecting calls with matching returns, that impart a tree-like hierarchical structure to the execution. Automata and logics over nested words can be used to express a variety of requirements such as stack-inspection properties, pre-post conditions, and interprocedural data-flow properties. Closure properties and decision problems of these automata can then be used for algorithmic verification of procedural programs.

4.1 Nested Words

Nested words model data with both linear and hierarchical structure. Here we consider only *infinite* nested words (which can model nonterminating executions of programs).

Given a linear sequence, the hierarchical structure is added using edges that are well nested (that is, they do not cross). We will use edges starting at $-\infty$ and edges ending at $+\infty$ to model "pending" edges. Assume that $-\infty < i < +\infty$ for every integer i. A *matching relation* $\rightsquigarrow$ is a subset of $\{-\infty, 1, 2, \ldots\} \times \{1, 2, \ldots + \infty\}$ such that (1) nesting edges go only forward: if $i \rightsquigarrow j$ then $i < j$; (2) no two nesting edges share a position: for each natural number i, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$; and (3) nesting edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$ then it is not the case that $i < i' \leq j < j'$.

When $i \rightsquigarrow j$ holds, the position i is called a *call position*. For a call position i, if $i \rightsquigarrow +\infty$, then i is called a *pending call*, otherwise i is called a *matched call*, and the unique position j such that $i \rightsquigarrow j$ is called its *return-successor*. Similarly, when $i \rightsquigarrow j$ holds, the position j is called a *return position*. For a return position j, if $-\infty \rightsquigarrow j$, then j is called a *pending return*, otherwise j is called a *matched return*, and the unique position i such that $i \rightsquigarrow j$ is called its *call-predecessor*. A position i that is neither a call nor a return is called *internal*.

A *nested word* w over an alphabet Σ is a pair $(a_1 a_2 \cdots, \rightsquigarrow)$ such that each a_i is a symbol in Σ, and $\rightsquigarrow$ is a matching relation. Let us denote the set of all nested words over Σ as $NW(\Sigma)$. A *language* of nested words over Σ is a subset of $NW(\Sigma)$.

As an example, consider the program of Fig. 2 again. Suppose we are interested in tracking read/write accesses to the global program variable x. Then, we can choose the following set of symbols for the observables Σ: *rd* to denote a read access to x, *wr* to denote a write access to x, *cl* to denote beginning of a new scope (such as a call to the procedure P_2), *rt* to denote the ending of the current scope, and *sk* to denote all other actions of the program. Note that in any structured programming language, in a given execution, there is a natural nested matching of the symbols *cl* and *rt*. Figure 7 shows a sample execution of the program modeled as a nested word (this execution corresponds to the initial state in which x is 0 and y is 1). For example, the second symbol (labeled *rd*) corresponds to the execution of the test "**if** x",

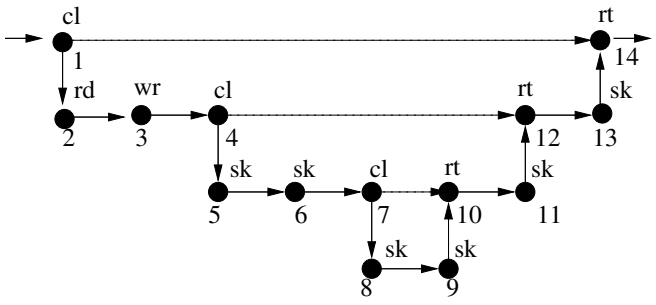


Fig. 7 Sample execution as a nested word.

and the next corresponds to the assignment $x := y$. Both these steps do not involve a change of context, and are internal positions. The procedure P_2 is called at position 4, and this call has a nesting edge to the matching position 12 (labeled *rt*). The subword from position 5 to position 11 encodes the execution of the called procedure. The main benefit of explicitly augmenting the linear structure with the nesting edges, is that using nesting edges one can skip call to a procedure entirely, and continue to trace a local path through the calling procedure. Consider the property that "if a procedure writes to x then it later reads x." This requires keeping track of the context. If we were to model executions as words, the set of executions satisfying this property would be a context-free language of words, and hence, not specifiable in classical temporal logics. Soon we will see that when we model executions as nested words, the set of executions satisfying this property is a regular language of nested words, and is amenable to algorithmic verification.

4.2 Nested Word Automata

We define and study finite-state automata as acceptors of nested words. A *nested word automaton* (NWA) is similar to a classical finite-state word automaton, and reads the input from left to right according to the linear sequence. At a call, it can propagate states along both linear and nesting outgoing edges, and at a return, the new state is determined based on states labeling both the linear and nesting incoming edges. Thus, NWA combines the features of top-down and bottom-up tree automata. It can also be viewed as a restricted form of a pushdown automaton: at a call position, it pushes a symbol on the stack; at a return position, it pops a symbol from the stack; and at an internal position, it does not update or examine the stack. Thus, the updates to the stack are determined by the call/return structure of the input word, and that's why a nested word automaton is also called a *visibly pushdown* automaton.

In the context of program verification, we are interested in *nondeterministic* NWAs: nondeterminism can arise due to inputs, due to abstraction, or when mul-

tiple states/transitions are associated with the same observation. We focus only on automata over infinite words using the Büchi acceptance condition.

A *nondeterministic Büchi nested word automaton* (BNWA) A over an alphabet Σ consists of

- a finite set of states Q,
- a set of initial states $Q_0 \subseteq Q$,
- a set of Büchi states $Q_f \subseteq Q$,
- a finite set of hierarchical states P,
- a set of initial hierarchical states $P_0 \subseteq P$,
- a call transition relation $\delta_c \subseteq Q \times \Sigma \times Q \times P$,
- an internal transition relation $\delta_l \subseteq Q \times \Sigma \times Q$, and
- a return transition relation $\delta_r \subseteq Q \times P \times \Sigma \times Q$.

Given a nested word w, the automaton A starts in an initial state, and reads the nested word from left to right according to the linear order. The state is propagated along the linear edges as in case of a standard word automaton. However, at a call, the nested word automaton can propagate a hierarchical state along the outgoing nesting edge also. At a return, the new state is determined based on the states propagated along the linear edge as well as along the incoming nesting edge. A pending nesting edge incident upon a pending return is labeled with an initial hierarchical state. The run is accepting if one of the Büchi states repeats infinitely often.

Formally, a run r of the BNWA A over a nested word $w = (a_1 a_2 \cdots, \rightsquigarrow)$ is an infinite sequence $q_i \in Q$, for $i \geq 0$, of states corresponding to linear edges, and a sequence $p_i \in P$, for call positions i, of hierarchical states corresponding to nesting edges, such that $q_0 \in Q_0$, and for each position $i \geq 1$, if i is a call position then $(q_{i-1}, a_i, q_i, p_i) \in \delta_c$; if i is an internal position then $(q_{i-1}, a_i, q_i) \in \delta_l$; if i is a matched return with call-predecessor j then $(q_{i-1}, p_j, a_i, q_i) \in \delta_r$, and if i is a pending return then $(q_{i-1}, p_0, a_i, q_i) \in \delta_r$ for some $p_0 \in P_0$. The run is accepting if $q_i \in Q_f$ for infinitely many indices $i \geq 0$. The automaton A accepts the nested word w if A has some accepting run over w. The language $L(A)$ is the set of nested words it accepts. A set L of nested words is *ω-regular* iff there is a BNWA A such that $L(A) = L$.

4.2.1 RSMs as NWAs

An RSM can be interpreted as a nested word automaton. Consider an RSM $\mathcal{M} = (A_1, \ldots, A_k)$ with components $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$. For the corresponding NWA $A_{\mathcal{M}}$, for each component A_i, for every node, call-port, and return port of A_i, there is a corresponding linear state in $A_{\mathcal{M}}$. The set of hierarchical states is the set of boxes of all the components. The entry nodes of the main component are the initial states, and the NWA does not rely on initial hierarchical states (since there will be no pending returns in the nested words it generates). For every transition $u \to v$ of each component A_i, there is a corresponding internal transition in $A_{\mathcal{M}}$. For every call port (en, b) of A_i, the NWA has a call transition from the state (en, b) to the state en

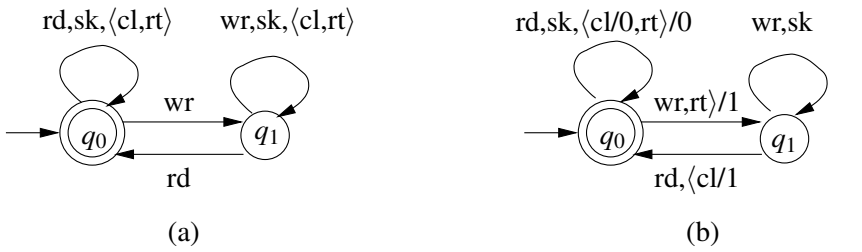


Fig. 8 Using NWA to specify program requirements.

(corresponding to the entry node of the component $A_{Y(b)}$) propagating the hierarchical state b along the nesting edge. For every return port (ex, b) of A_i, the NWA has a return transition to the state (ex, b) from the state ex (corresponding to the exit node of the component $A_{Y(b)}$) provided the hierarchical state along the incoming nesting edge is b. The labels on the transitions correspond to observations suitable for the analysis problem. In the example corresponding to Fig. 7, each call transition is labeled with the symbol *cl*, each return transition is labeled with the symbol *rt*, and each internal transition is labeled with either *rd*, *wr*, or *sk*, depending on the type of the statement executed. The NWA is augmented with Büchi acceptance condition if needed (for instance, to ensure fair resolution of choice when nondeterminism is used for abstraction).

4.2.2 NWAs for Requirements

The requirements of a program can also be described as an ω-regular languages of nested words. Let us revisit the example used in Fig. 7. Suppose we want to specify that each write to x is followed by some read of x. We will consider two variations of this requirement.

First, suppose we want to specify that a symbol *wr* is followed by *rd*, without any reference to the procedural context. This can be captured by standard word automata, and also by NWAs. Figure 8 (a) shows the 2-state (deterministic) NWA for the requirement. We use the prefix $\langle$ with a symbol to indicate a call transition, and the suffix $\rangle$ with a symbol to indicate a return transition. Call and return transitions also have associated hierarchical states. In this example, hierarchical states are not needed.

Now suppose, we want to specify that if a procedure writes to x, then the same invocation should read it before it returns. That is, between every pair of matching call and return, along the local path obtained deleting every enclosed well-matched subword between a call and its matching return, every *wr* is followed by *rd*. Viewed as a property of words, this is not a regular language, and thus, not expressible in the classical specification languages. However, over nested words, this can easily be specified using an NWA, see Fig. 8 (b). The initial state is q_0, and has no pending

obligations, and is the only final state. The hierarchical states are $\{0,1\}$, where 0 is the initial state. The state q_1 means that along the local path of the current scope, a write-access has been encountered with no following read access. While processing the call, the automaton remembers the current state by propagating 0 or 1 along the nesting edge, and starts checking the requirement for the called procedure by transitioning to the initial state q_0. While processing internal read/write symbols, it updates the state as in the finite-state word automaton of case (a). At a return, if the current state is q_0 (meaning the current context satisfies the desired requirement), it restores the state of the calling context. Note that there are no return transitions from the state q_1, and this means that if a return position is encountered while in state q_1, the automaton rejects the input word.

We now review some key properties of nested word automata that are useful in their application to model checking

4.2.3 Closure Properties

The class of ω-regular (and regular) languages of nested words is closed under a variety of operations including union, intersection, complementation, prefixes, suffixes, concatenation, Kleene-*, and language homomorphisms. For verification, the most relevant operation is *language intersection*: given two BNWAs A_1 and A_2, one can construct a product BNWA A such that $L(A) = L(A_1) \cap L(A_2)$. If A_1 captures the set of nested words generated by an RSM, and A_2 captures the set of nested words that violate a desired correctness requirement, then verification corresponds to checking non-emptiness of the language of A. The product construction for NWAs is a simple extension of the product construction for finite (word) automata. A linear state of A is a pair of linear states of A_1 and A_2, and a hierarchical state of A is a pair of hierarchical states of A_1 and A_2. The call/internal/return transitions synchronize the transitions of A_1 and A_2 on a common input symbol, and update the two state components. Ensuring that Büchi acceptance conditions of both are satisfied can be done the same way as in the product construction for Büchi automata (see Chap. 7). It is worth noting that nested word automata can also be complemented and determinized. Determinization requires maintaining a set of "summaries" that capture executions of the nondeterministic automaton on the subword between a call and its matching return, and the acceptance condition needed is a parity condition over states that repeat infinitely often at the "top-level" of the input word (see [6] for details). The complexity of determinization as well as complementation is exponential.

4.2.4 Decision Problems

The *emptiness* problem for NWAs (given a BNWA A, is $L(A) = \emptyset$?) is solvable in polynomial-time (in time cubic in the size of the automaton). The technique is the same as the one used in solving the fair computation problem for pushdown

systems discussed in Section 3.2. Problems such as *universality* (given a BNWA A, is $L(A) = \Sigma^\omega$?), language *inclusion* (given BNWAs A_1 and A_2, is $L(A_1) \subseteq L(A_2)$?), and language *equivalence* (given BNWAs A_1 and A_2, is $L(A_1) = L(A_2)$?) can all be solved in EXPTIME by employing the complementation construction. Note that these problems are undecidable for pushdown automata (or context-free languages). Thus, given two RSMs, checking whether they generate the same sets of words is undecidable, while checking whether they generate the same sets of nested words is decidable. The latter is a stronger requirement which considers two executions equivalent when the two produce the same sequences of observations, and also agree on entries to and exits from procedural contexts.

4.2.5 MSO Equivalence

For word languages, the notion of regularity has many equivalent characterizations using finite automata, monadic second-order logic, and regular expressions. The notion of regularity for nested words also turns out to be robust. In particular, the monadic second order logic (MSO) of nested words has the same expressiveness as nested word automata. The vocabulary of nested sequences includes the linear successor and the matching relation $\rightsquigarrow$. In order to model pending edges, we will use two unary predicates `call` and `ret` corresponding to call and return positions. The *monadic second-order logic of nested words* is given by the syntax:

$$\phi := a(x) \mid X(x) \mid \texttt{call}(x) \mid \texttt{ret}(x) \mid x = y+1 \mid x \rightsquigarrow y \mid \phi \vee \phi \mid \neg\phi \mid \exists x.\phi \mid \exists X.\phi,$$

where $a \in \Sigma$, x, y are first-order variables, and X is a second-order variable. The semantics is defined over nested words in a natural way. The first-order variables are interpreted over positions of the nested word, while set variables are interpreted over sets of positions. The formula $a(x)$ holds if the symbol at the position interpreted for x is a, $\texttt{call}(x)$ holds if the position interpreted for x is a call, $x = y+1$ holds if the position interpreted for y is (linear) next to the position interpreted for x, and $x \rightsquigarrow y$ holds if the positions x and y are related by a nesting edge. For example,

$$\forall x.(\ \texttt{call}(x)\ \rightarrow \exists y.\ x \rightsquigarrow y\ )$$

holds in a nested word iff it has no pending calls;

$$\forall x.\forall y.\ (a(x) \wedge x \rightsquigarrow y) \Rightarrow b(y)$$

holds in a nested word iff for every matched call labeled a, the corresponding return-successor is labeled b.

For a sentence ϕ (a formula with no free variables), the language it defines is the set of all nested words that satisfy ϕ. It turns out that: a language L of nested words over Σ is ω-regular iff there is an MSO sentence ϕ over Σ that defines L.

4.3 Temporal Logics

Over infinite words, Linear Temporal Logic (LTL) has long been considered the temporal logic of choice for program verification, not only because its temporal operators offer the right abstraction for reasoning about events over time, but also because it provides a good balance between expressiveness (first-order complete), conciseness (can be exponentially more succinct compared to automata), and the complexity of model-checking (linear time in the size of the finite transition system, and PSPACE in the size of the temporal formula). This has motivated the study of temporal logics over nested words such as CARET [4] and NWTL [1]. We briefly review these logics in this section.

Let us first recall the syntax and semantics of LTL (see Chap. 2). Given a set AP of atomic propositions, a formula of propositional LTL is built from atomic propositions, logical connectives (such as conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\rightarrow$), and temporal operators (such as next $\bigcirc$, always $\Box$, eventually $\Diamond$, and until $\mathcal{U}$). An LTL formula is evaluated with respect to an infinite sequence $w = a_1 a_2 \cdots$ over $\Sigma = 2^{AP}$, that is, each observation a_j is an assignment of truth values to the propositions in AP. The semantics of LTL is defined using the satisfaction relation $(w, j) \models \phi$, which means that the formula ϕ is satisfied at position j in the model w. Example rules for evaluation are: $(w, j) \models p$, for an atomic proposition p, if the observation w_j assigns the value 1 to p; $(w, j) \models \bigcirc\phi$ if $(w, j+1) \models \phi$; $(w, j) \models \Box\phi$ if $(w, k) \models \phi$ for every position $k \geq j$; and $(w, j) \models \phi_1 \mathcal{U} \phi_2$ if there exists a position $k \geq j$ such that $(w, k) \models \phi_2$ and $(w, l) \models \phi_1$ for all positions $j \leq l < k$.

In the revised setting of nested words, a formula is interpreted over a nested word w over the set $\Sigma = 2^{AP}$ of observations. To motivate the definition of new temporal operators, let us examine the nested word shown in Fig. 7. Notice that unlike a linear sequence, the graph-like structure of a nested word means that one can define different kinds of paths. If we ignore the nesting edges, and focus on the linear sequence of positions, we obtain the *linear* path, and we can continue to interpret LTL operators over this linear path. In this example, the sequence $1, 2, 3, 4, \ldots, 13, 14$ of positions forms the linear path. Suppose we want to express the requirement that, along a global program execution, every write to a variable is followed by a read (see the automaton in Fig. 8 (a)). If *wr* and *rd* denote the atomic propositions that capture write and read operations, respectively, then the requirement is expressed by the LTL formula:

$$\Box\, [\, wr \rightarrow \Diamond\, rd\,]$$

4.3.1 Abstract Next

In a nested word, a call position has two successors: a linear edge to the next position, and a nesting edge to the matching return. This motivates adding, besides the original LTL operator $\bigcirc$ corresponding to the linear successor, another next operator, called *abstract-next*, denoted $\bigcirc^a$. Its semantics is defined by the rule:

$(w,j) \models \bigcirc^a \phi$ holds if the position j is a call position, has a matching return position l (that is, $j \rightsquigarrow l$), and $(w,l) \models \phi$.

It is easy to establish that the abstract-next operator is not definable in LTL. In the classical verification formalisms such as Hoare logic, correctness of procedures is expressed using pre and post conditions. Partial correctness of a procedure P specifies that if the pre-condition p holds when the procedure P is invoked, if the procedure terminates, the post-condition q is satisfied upon return. Total correctness, in addition, requires the procedure to terminate. Assume that all calls to the procedure P are characterized by the proposition cl_P. Then, the requirement

$$\Box\,[\,(cl_P \wedge p) \rightarrow \bigcirc^a q\,]$$

expresses the total correctness, while

$$\Box\,[\,(cl_P \wedge p \wedge \bigcirc^a \texttt{True}) \rightarrow \bigcirc^a q\,]$$

expresses the partial correctness.

4.3.2 Abstract Paths

An *abstract* path in a nested word w is a sequence of positions $i_1, i_2, \ldots i_j$ such that, for each $1 \leq l < j$, either i_l is a call position with matching return position i_{l+1}, or i_l is an internal or a return position and i_{l+1} equals $i_l + 1$ and is not a return position. For a nested word that models an execution of a procedural program, the abstract path starting at a position inside a procedure P is obtained by successive applications of internal and nesting edges, and skips over invocations of other procedures called from P. In the nested word of Fig. 7, examples of abstract paths are $1, 14$, and $2, 3, 4, 12, 13$, and $5, 6, 7, 10, 11$, and $8, 9$. We can now define the abstract versions of temporal operators such as *abstract-always* $\Box^a$, *abstract-eventually* $\Diamond^a$, and *abstract-until* $\mathcal{U}^a$. The semantics of these operators is defined by interpreting them over abstract paths. For example,

$(w,j) \models \phi_1 \mathcal{U}^a \phi_2$ if there exists an abstract path $j = i_1, i_2, \ldots i_k$ such that $(w,i_k) \models \phi_2$ and $(w,i_l) \models \phi_1$ for all $1 \leq l < k$.

That is, $\phi_1 \mathcal{U}^a \phi_2$ holds if there is abstract path leading to a position satisfying ϕ_2 such that at all preceding positions along this abstract path ϕ_1 holds. We can use these abstract modalities to specify context-bounded requirements. Let us revisit the requirement that if a procedure writes to a variable, then it (that is, the same invocation of the same procedure) will later read it (see the NWA of Fig. 8 (b)). The requirement is expressed by the following formula over abstract paths:

$$\Box\,[\,wr \rightarrow \Diamond^a rd\,]$$

4.3.3 Summary Paths

A *summary path* between positions i and j, with $i < j$, of a nested word w is a sequence $i = i_1, i_2 \ldots i_k = j$ of positions such that for $1 \leq l < k$, if i_l is a matched call with a matching return position $r \leq j$ then $i_{l+1} = r$, else $i_{l+1} = i_l + 1$. Intuitively, a summary path between i and j is the "shortest" path from i to j that one can construct using linear and nesting edges. For example, in the nested word of Figure 7, the summary path between positions 2 and 14 is the sequence $2, 3, 4, 12, 13, 14$, while the summary path between positions 2 and 11 is the sequence $2, 3, 4, 5, 6, 7, 10, 11$. The summary-versions of temporal operators such as summary-until $\mathcal{U}^{\sigma}$, are defined by interpreting the temporal modalities over the summary paths. While not particularly natural for specifying program requirements, the interest in the summary paths stems from their theoretical expressiveness: the expressiveness of the logic with abstract-next, and its past dual, abstract-previous, and summary-until, and its past dual, summary-since, coincides exactly with the first-order logic over nested words (that is, logic with first-order variables, quantification over first-order variables, logical connectives, binary predicates $x = y + 1$, $x < y$, $x \rightsquigarrow y$, and unary predicates corresponding to `call`, `ret`, and atomic propositions) [1]. This result is the analog of the result that the expressiveness of LTL coincides with the first-order logic over words. Global, abstract, and other versions of temporal modalities are definable using first-order logic over nested words, and this implies that requirements about abstract paths can be defined using modalities over summary paths. It seems unlikely that a similar completeness result holds for abstract modalities (more specifically, it is conjectured, but not proved, that the logic CaRet [4] is not first-order complete).

4.3.4 Model Checking

Chapter 7 discusses the tableau-based approach to checking satisfiability and model checking of LTL. This approach can be extended to temporal logics over nested words. In the sequel, we use NWTL to denote the logic with all the connectives we have discussed so far, and also their past duals. Given an NWTL formula φ, we can construct a BNWA A_{φ} such that (1) $L(A_{\varphi})$ contains exactly those nested words that satisfy φ, and (2) the size of A_{φ} is $2^{O(|\varphi|)}$. To check whether φ is satisfiable, we can test whether the language of A_{φ} is nonempty, and to check whether all executions of an RSM $\mathcal{M}$ satisfy the NWTL specification φ, we can test language-emptiness of the product of the automata $A_{\mathcal{M}}$ and $A_{\neg\varphi}$. Both satisfiability and model checking problems for NWTL are EXPTIME-complete.

The construction of the BNWA A_{φ} corresponding to the NWTL formula φ follows the same recipe as the tableau construction for LTL discussed in Chap. 7. We first define the set $Closure(\varphi)$ of formulas; the linear and hierarchical states of A_{φ} are subsets of $Closure(\varphi)$ that satisfy local consistency requirements; the transitions of A_{φ} are defined so that next-time requirements are correctly propagated along the linear edges, and the abstract-next-time requirements are correctly propagated along

the nesting edges; and each until-formula in the closure gives a Büchi acceptance condition that ensures eventual fulfillment of the until obligations (this results in a *generalized* Büchi acceptance condition, which can be translated to Büchi acceptance condition by introducing a counter as described in Chap. 7). We refer the reader to [1] for details, but illustrate the essence of the construction by focusing on the *abstract-until* formulas of the form $\phi_1 \mathcal{U}^a \phi_2$.

The closure contains propositions `call`, `ret`, and `int`, that indicate the position types. Additionally, a proposition `top` is used to indicate whether the current position is "top-level": a position i of a nested word w is top-level if it is not within a pair of matching call-return positions, that is, there are no positions j and k such that $j < i < k$ and $j \rightsquigarrow k$.

The closure rule for the abstract-until formula says that if $\phi_1 \mathcal{U}^a \phi_2$ is in $Closure(\varphi)$ then so are the formulas ϕ_1, ϕ_2, $\bigcirc(\phi_1 \mathcal{U}^a \phi_2)$ and $\bigcirc^a(\phi_1 \mathcal{U}^a \phi_2)$. The size of the closure is linear in $|\varphi|$.

States correspond to subsets of the closure that satisfy consistency requirements. Sample consistency requirements on a state $\Phi \subseteq Closure(\varphi)$ are: exactly one of `call`, `ret`, and `int`, belongs to Φ, and $\phi_1 \mathcal{U}^a \phi_2 \in \Phi$ iff either $\phi_2 \in \Phi$, or ($\phi_1 \in \Phi$ and $\texttt{call} \in \Phi$ and $\bigcirc^a(\phi_1 \mathcal{U}^a \phi_2) \in \Phi$) or ($\phi_1 \in \Phi$ and $\texttt{call} \notin \Phi$ and $\bigcirc\texttt{ret} \notin \Phi$ and $\bigcirc(\phi_1 \mathcal{U}^a \phi_2) \in \Phi$). Note this rule for the abstract-until formula captures its semantics inductively: to satisfy the formula $\phi_1 \mathcal{U}^a \phi_2$ at a position either ϕ_2 is satisfied in that position, or at a call position, ϕ_1 is satisfied and the formula is propagated along the nesting edge, or at a return/internal position, ϕ_1 is satisfied and the formula is propagated along the linear edge, provided the linear successor is not a return.

The transitions of the automaton ensure that the desired propagation expressed by next and abstract-next formulas in a state is enforced. If there is an internal transition from state Φ to state Ψ, then it must be the case that $\texttt{top} \in \Phi$ iff $\texttt{top} \in \Psi$ and for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi$ iff $\bigcirc\psi \in \Phi$. If there is a call transition from state Φ to state Φ_l while propagating state Φ_h on the nesting edge, then it must be the case that either none of Φ, Ψ_l and Ψ_h contain `top`, or $\texttt{top} \in \Phi$ and exactly one of Ψ_l and Ψ_h contains `top`; and for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi_l$ iff $\bigcirc\psi \in \Phi$; and for each $\bigcirc^a\psi \in Closure(\varphi)$, $\psi \in \Psi_h$ iff $\bigcirc^a\psi \in \Phi$. Finally, if there is a return transition to state Ψ from state Φ_l using the incoming hierarchical state Φ_h, then it must be the case that $\texttt{top} \notin \Phi_l$, and $\texttt{top} \in \Phi_h$ iff $\texttt{top} \in \Psi$; for each $\bigcirc\psi \in Closure(\varphi)$, $\psi \in \Psi$ iff $\bigcirc\psi \in \Phi_l$; and for each $\bigcirc^a\psi \in Closure(\varphi)$, $\psi \in \Phi_h$ iff $\psi \in \Phi_l$.

The Büchi acceptance condition to ensure the eventual fulfillment of the abstract-until formula $\phi_1 \mathcal{U}^a \phi_2$ demands that some state Φ such that $\texttt{top} \in \Phi$ and either $\phi_2 \in \Phi$ or $\phi_1 \mathcal{U}^a \phi_2 \notin \Phi$ repeats infinitely often. This is based on the fact that the fulfillment of an abstract-until can be delayed forever by the propagation rules only along an abstract path that contains only top-level positions.

5 Bibliographical Remarks

5.1 Summarization

Two early papers proposing general frameworks for computing procedure summaries in the context of inter-procedural program analysis are [25] by Cousot and Cousot and [49] by Sharir and Pnueli. There is a lot of subsequent work aimed at investigating efficient techniques for various kinds of abstract domains to account for data manipulated by the program, and designing efficient and precise algorithmic techniques for special classes of properties (c.f. [44, 46, 40, 31]). In particular, Reps et al. propose in [44] efficient algorithms for inter-procedural data-flow analysis based on graph reachability that is similar to checking reachability in pushdown systems. The tool Bebop by Ball and Rajamani [10] allows to verify sequential Boolean programs with procedure calls using basically the reachability analysis algorithm of [44]. The model of recursive state machines was defined in [2] as a generalization of the model of hierarchical state machines [7], and this work gives a detailed analysis of the complexity of solving reachability, fair computation problem, and model-checking problems for temporal logics such as LTL and CTL*, based on summarization. Working directly with RSMs allows an understanding of the dependence of the computational complexity on the number of entry/exit nodes per component.

5.2 Saturation

The regularity of $pre^*(L)$ for a regular language L seems to have been first observed by Büchi in his work on regular canonical systems (see Chap. 5 of [16]), and has been rediscovered many times in slightly different contexts, for instance by Caucal in [22] and by Book and Otto in [11]. Book and Otto also present the saturation algorithms for monadic string-rewriting systems, a model closely related to PDSs.

Saturation algorithms for computing sets of forward- and backward-reachable configurations of PDSs were presented by Bouajjani et al. and Finkel et al. [13, 30]. Efficient versions with a detailed complexity analysis were obtained by Esparza et al. [26] (see also [47]). Symbolic versions of the algorithms were implemented in the MOPED tool by Schwoon and applied to verification problems of Linux drivers [29, 47]. The jMOPED tool adds to MOPED a front-end that transforms Java programs into extended pushdown systems and allows to apply MOPED [52].

The saturation technique has been extended in a number of ways. We briefly summarize some of the contributions.

Bouajjani et al. extend the technique to *alternating pushdown systems*, and apply the algorithms to the global[1] model-checking problem of CTL [13]. They show

[1] Here *global model-checking* means computing the set of all states in a given model that satisfy some given formula.

how to compute for a given CTL formula ϕ and a PDS $\mathcal{P}$ the set of all configurations of $\mathcal{P}$ satisfying ϕ. A different extension leading to a similar algorithm for CTL* is described by Esparza et al. in [27]. An efficient algorithm for CTL model-checking based on solving emptiness of alternating Büchi pushdown automata has been defined in [50].

Reps et al. show how to apply saturation to *weighted pushdown systems*, in which transition rules are labeled with elements of an idempotent semiring [45]. The saturation algorithm is extended so that it returns not only the sets $pre^*(C)$ and $post^*(C)$, but for each configuration c in them the total weight of the paths leading from c to C or from C to c, respectively. The extensions are implemented in the Weighted Automata Library WALi [38]. While the original motivation of this work was to obtain a general framework for inter-procedural data-flow analysis, the developed framework and algorithms were shown to be also useful for other applications, like modeling and verifying trust-management systems [37].

Cachat describes a saturation algorithm for computing the *attractor* of a regular set C of configurations of a *pushdown game system* [21]. A pushdown game system is a PDS in whose states are partitioned into two sets under the control of two different players. A play is a sequence of configurations, where the successor of the current configuration is decided by the player owning its control state. The attractor is the set of configurations such that the first player can force the play to visit C. Hague and Ong extend Cachat's ideas to algorithms for computing the winning regions of a given parity game [33], and for a given PDS $\mathcal{P}$ and a given formula ϕ of the μ-calculus the set of all configurations of $\mathcal{P}$ satisfying ϕ [34].

Higher-order pushdown systems (HPDSs) generalize PDSs by allowing nested stacks, i.e., stacks whose elements can be stacks themselves. Bouajjani and Meyer extend the saturation algorithm to HPDSs with one control state, also called higher-order context-free processes [15]. Hague and Ong extend the results to general HPDSs [32]. Seth gives an alternative construction for order 2 [48].

5.3 Temporal Logic Model Checking

Model checking of pushdown systems has been studied extensively for both linear- and branching-time requirements (see e.g. [20, 55, 13, 30, 26, 27, 42, 2]). The decidability of the model-checking problem of pushdown systems for the propositional μ-calculus (which subsumes in expressiveness regular propositional temporal logics such as LTL and CTL*) is a consequence of [39]. However, the model-checking algorithm derived from this result, that is based on a reduction to satisfiability problem of the monadic second order logic of two successors, has a non-elementary complexity. In [17], an elementary algorithm is provided for the class of context-free processes (equivalent to pushdown systems with a single control state) and the alternation-free (branching-time) propositional μ-calculus. Basically, this algorithm generalizes the summarization construction as it is based on computing pairs of pre/post conditions of process. The algorithm has been extended to the full class of

pushdown systems, but still for alternation-free μ-calculus) in [18], and then later to the full propositional μ-calculus, but only for context-free processes, in [19]. The algorithms defined in this series of work have been implemented in a tool called "The Fixpoint-Analysis Machine" [51] that has been used in practice for tackling various problems such as intra/inter-procedural data flow analysis, model-checking, and behavioral equivalence checking. The first elementary model-checking algorithm for the full class of pushdown systems and the full propositional μ-calculus has been defined in [53].The algorithm is based on solving pushdown parity games. A global model-checking algorithm for this general case has been provided first in [43]. In [53], the model-checking problem of pushdown systems for the full μ-calculus is shown to be EXPTIME-complete. In [54], the problem is shown to be EXPTIME-complete even for CTL, and that it is PSPACE-complete for the EF fragment. In [13], the problem is shown to be EXPTIME-complete for LTL and the linear-time propositional μ-calculus.

Even though the general problem of checking context-free properties of pushdown automata is undecidable, algorithmic solutions have been proposed for checking many different kinds of non-regular properties. For example, numerical properties have been considered in [12, 14] where model-checking algorithms are defined for extension of temporal logics with constraints on the number of occurrences of events/states along computations. These logics allow for instance to express properties such as "*between every two pairs of events a and b, there is the same number of c's and d's*". The model-checking algorithms proposed for these logics are based on reductions to the satisfiability of Presburger arithmetics, using the fact that Parikh-images of context-free languages are semi-linear sets [41].

Non-numerical properties have also been considered in several works. For instance, access control requirements such as "*a module A should be invoked only if the module B belongs to the call-stack*", and bounds on stack size such as "*if the number of interrupt-handlers in the call-stack currently is less than 5, then a property p holds*" require inspection of the stack, and decision procedures for certain classes of stack properties have been proposed [36, 28, 23].

The idea of explicit modalities that can reference to the matching structure of calls and returns first appears in the temporal logic CARET [4]. Subsequently, the model of visibly pushdown automata [5] and the theory of regular languages of nested words [6] were proposed as a unifying basis to explain which class of properties are algorithmically checkable against pushdown models. [1] defines the temporal logic NWTL, and presents a systematic study of linear temporal logics over nested words. [24] describes a specification language called PAL that extends the query language of the software model checker BLAST [35] for writing nested word monitors, along with a tool to instrument C code.

The nested structure on words can be extended to trees, and automata on nested trees are studied in [3]. A version of the μ-calculus on nested structures has been defined in [3], and is shown to be more powerful than the standard μ-calculus, while at the same time remaining robust and tractable.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science*, pages 151–160, 2007.
2. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.
3. R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*, pages 153–165, 2006.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04: Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2988, pages 467–481. Springer, 2004.
5. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 202–211, 2004.
6. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3), 2009.
7. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):1–31, 2001.
8. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.
9. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer, 2000.
10. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.
11. R. Book and F. Otto. *String-Rewriting Systems*. Springer, 1993.
12. A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *10th Annual IEEE Symposium on Logic in Computer Science (LICS), San Diego, CA, USA*, pages 123–133. IEEE Computer Society, 1995.
13. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR'97*, LNCS 1243, pages 135–150, 1997.
14. A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and petri nets. In *7th International Conference on Concurrency Theory (CONCUR), Pisa, Italy*, volume 1119 of *Lecture Notes in Computer Science*, pages 481–497. Springer, 1996.
15. A. Bouajjani and A. Meyer. Symbolic reachability analysis of higher-order context-free processes. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2004.
16. J. R. Büchi. *Finite Automata, Their Algebras and Grammars*. Springer, 1988. D. Siefkes (ed.).
17. O. Burkart and B. Steffen. Model checking for context-free processes. In *3rd International Conference on Concurrency Theory (CONCUR), Stony Brook, NY, USA*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1992.
18. O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *5th International Conference on Concurrency Theory (CONCUR), Uppsala, Sweden*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 1994.
19. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *24th International Colloquium on Automata, Languages and Programming (ICALP), Bologna, Italy*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429. Springer, 1997.
20. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theor. Comput. Sci.*, 221(1-2):251–270, 1999.
21. T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 704–715. Springer, 2002.

22. D. Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1):61–86, 1992.
23. K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. *Information and Computation*, 194(2):144–174, 2004.
24. S. Chaudhuri and R. Alur. Instrumenting C programs with nested word monitors. In *SPIN2007: 14th Internation Workshop on Model Checking Software*, 2007.
25. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP WG2.2 Conference on Formal Description of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1978.
26. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV 2000*, LNCS 1855, pages 232–247. Springer, 2000.
27. J. Esparza, A. Kucera, and S. Schwoon. Model checking ltl with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.
28. J. Esparza, A. Kucera, and S. S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.
29. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. CAV'01*, LNCS 2102, pages 324–336. Springer, 2001.
30. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electr. Notes Theor. Comput. Sci.*, 9:27–37, 1997.
31. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2007.
32. M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In H. Seidl, editor, *FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2007.
33. M. Hague and C.-H. L. Ong. Winning regions of pushdown parity games: A saturation method. In M. Bravetti and G. Zavattaro, editors, *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2009.
34. M. Hague and C.-H. L. Ong. A saturation method for the modal -calculus over pushdown systems. *Inf. Comput.*, 209(5):799–821, 2011.
35. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.
36. T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 89–103, 1999.
37. S. Jha, S. Schwoon, H. Wang, and T. W. Reps. Weighted pushdown systems and trust-management systems. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2006.
38. N. Kidd, A. Lal, and T. Reps. WALi: The weighted automata library. See http://www.cs.wisc.edu/wpis/wpds/.
39. D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theor. Comput. Sci.*, 37:51–75, 1985.
40. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341. ACM, 2004.
41. R. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.
42. N. Piterman and M. Y. Vardi. Global model-checking of infinite-state systems. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–400. Springer, 2004.
43. N. Piterman and M. Y. Vardi. Global model-checking of infinite-state systems. In *16th International Conference on Computer Aided Verification (CAV), Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 387–400. Springer, 2004.
44. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of POPL*, pages 49–61. ACM, 1995.

45. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2):206–263, October 2005. Special Issue on the Static Analysis Symposium 2003.
46. S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1&2):131–170, 1996.
47. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, TU München, 2002.
48. A. Seth. An alternative construction in symbolic reachability analysis of second order pushdown systems. *Int. J. Found. Comput. Sci.*, 19(4):983–998, 2008.
49. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program flow analysis: Theory and applications*, pages 189–233. Prentice-Hall, 1981.
50. F. Song and T. Touili. Efficient CTL model checking for pushdown systems. In *CONCUR'11*, volume 6901 of *Lecture Notes in Computer Science*, pages 434–449. Springer, 2011.
51. B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *6th International Conference on Concurrency Theory (CONCUR), Philadelphia, PA, USA*, volume 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer, 1995.
52. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Proceedings of TACAS 2005*, LNCS 3440, pages 541–545. Springer, 2005.
53. I. Walukiewicz. Pushdown processes: Games and model checking. In *8th International Conference on Computer Aided Verification (CAV), New Brunswick, NJ, USA*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1996.
54. I. Walukiewicz. Model checking ctl properties of pushdown systems. In *20th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), New Delhi, India*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2000.
55. I. Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001.