

Resource-Constrained Model Checking of Recursive Programs ^{*}

Samik Basu¹, K. Narayan Kumar^{1,2}, L. Robert Pokorny¹, and
C.R. Ramakrishnan¹

¹ Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, U.S.A.

E-mail: {`bsamik,kumar,pokorny,cram`}@`cs.sunysb.edu`

² Chennai Mathematical Institute, Chennai, India.

E-mail: `kumar@smi.ernet.in`

Abstract. A number of recent papers present efficient algorithms for LTL model checking for recursive programs with finite data structures. A common feature in all these works is that they consider infinitely long runs of the program without regard to the size of the program stack. Runs requiring unbounded stack are often a result of abstractions done to obtain a finite-data recursive program. In this paper, we introduce the notion of resource-constrained model checking where we distinguish between stack-diverging runs and finite-stack runs. It should be noted that finiteness of stack-like resources cannot be expressed in LTL.

We develop resource-constrained model checking in terms of good cycle detection in a finite graph called R-graph, which is constructed from a given push-down system (PDS) and a Büchi automaton. We make the formulation of the model checker “executable” by encoding it directly as Horn clauses. We present a local algorithm to detect a good cycle in an R-graph. Furthermore, by describing the construction of R-graph as a logic program and evaluating it using tabled resolution, we do model checking without materializing the push-down system or the induced R-graph. Preliminary experiments indicate that the local model checker is at least as efficient as existing model checkers for push-down systems.

1 Introduction

Model checking is a widely used technique for verifying whether a system specification possesses a property expressed as a temporal logic formula [7, 8, 14]. Most early works on model checking have restricted system specifications to be finite state. A number of recent works have addressed the problem of model checking push-down processes with finite alphabets, which are natural models for recursive programs operating on finite data structures (e.g. [12, 4, 10, 5, 3]).

In this paper, we consider the problem of LTL model checking of recursive programs. Models of LTL formulas are usually described in terms of infinite runs

^{*} This work was supported in part by NSF grants EIA-9705998, CCR-9876242, EIA-9805735, N000140110967, and IIS-0072927.

```

bool g;
procedure main() {
  g = false;
  while (true) {
    flip();
    flip();
    if (!g)
      reach: skip
  }
}
(a)

procedure flip() {
  if (g) {
    if (*) {
      flip();
      flip();
    }
    g = !g;
    return;
  }
}
(b)

void flip(N) {
  int (0..7) i;
  if (g) {
    i = 0;
    while (i < 7) i++;
  } else if (N > 0) {
    flip(N - 1);
    flip(N - 1);
  }
  g = !g;
  return;
}
(c)

```

Fig. 1. Recursive programs with finite-domain variables

of a system. For push-down systems, the stacks may diverge on some infinite runs, indicating runs not realizable in any implementation of the system. In fact, stack-diverging runs may be an artifact of abstractions performed to obtain finite-data recursive programs. Such abstractions are often performed to obtain a single program that represents the behaviors of an infinite family of programs. For instance, consider the finite-domain program shown in Fig. 1(a) and (b). The example was derived from ones used in [2] and [11]. The procedure `flip()` in Fig. 1(b) is an abstraction of procedure `flip(N)` in Fig. 1(c) (from [11]). In the program, the statement `if (*) . . .` indicates a non-deterministic choice, the result of abstracting away the conditional expression.

The need for resource-constrained model checking. For the program in Fig. 1(a,b) consider the verification of the LTL property `AGF reach` starting from a state representing the first statement in procedure `main`. This property does not hold since the program has a run where it keeps recursively invoking `flip`. However, such a run is clearly unfeasible in any concrete implementation of the program, since the program stack grows without bound.

It is hence natural to restrict our attention to runs where the stacks remain finite. However, traditional mechanisms to restrict the runs under consideration such as adding fairness constraints cannot be used to capture stack-finiteness: separating a run that involves infinite number of unmatched pushes from the rest cannot be done using a regular language.

Returning to the example in Fig. 1(a,b), observe that the property `AGF reach` holds for every run that consumes only a finite stack. It is easy to see that `flip`, whenever it terminates, negates the global variable `g`. Hence two consecutive calls to `flip` leave `g` unchanged, making `reach` true in every iteration of the loop in `main`. Since `flip` terminates if and only if the program stack remains finite, `AGF reach` holds on all finite-stack runs.

Our approach. In this paper, we describe a model checker, called *resource-constrained model checker*, that separates the finite-stack runs from stack-diverging runs. Our technique can determine that `AGF reach` holds for all finite-stack runs of the program in Fig. 1(a,b), while there are stack-diverging runs

that violate the property. We give a brief overview of our technique below. For simplicity we assume in the following that the push-down system has a single control state: i.e., a context-free process. Our formal development in the later sections considers general push-down systems.

Given a push-down system \mathcal{P} and a Büchi automaton \mathcal{B} (corresponding to the given LTL property), we develop a model checker as follows. We first build a *finite* graph \mathcal{R} , called the *R-graph*, that abstracts the product of \mathcal{P} and \mathcal{B} . The nodes of \mathcal{R} are labeled with pairs (b, γ) , where γ is a stack alphabet of \mathcal{P} and b is a state in \mathcal{B} . Edges in \mathcal{R} are labeled with a *goodness* label (*true* or *false*) and a *resource* label (0 or 1).

Intuitively, an edge in \mathcal{R} , say from (b, γ) to (b', γ') corresponds to a finite sequence of moves that take \mathcal{P} from a configuration with γ on top of stack to one with γ' on top of stack, and correspondingly moves \mathcal{B} from state b to state b' . The edge is *good* (i.e. its goodness label is *true*) if and only if there is some good state in \mathcal{B} that is visited in that corresponding run in \mathcal{B} from b to b' . The resource label on the edge is 0 if the corresponding run in \mathcal{P} leaves the size of the stack unchanged; the resource label is 1 if the stack size increases by 1.

An accepting path in \mathcal{R} is an infinite path where good edges appear infinitely often and only finitely many edges have resource label 1. We show that there is an accepting path in \mathcal{R} if, and only if, there is a finite-stack run of \mathcal{P} accepted by \mathcal{B} . The R-graph is analogous to the automaton A_{br} described in [12]. However, the resource labels of \mathcal{R} distinguish between finite-stack and stack-diverging runs of \mathcal{P} . Thus, ignoring the resource labels in the acceptance criterion of \mathcal{R} , we obtain a model checker that is, in concept, equivalent to the ones previously defined in the literature [4, 10, 12, 11]. Although R-graph has much in common with techniques described in these works in terms of formulation, our implementation strategy is substantially different, as described below.

Contributions.

- We introduce the notion of resource-constrained model checking of push-down systems. We formulate this problem in terms of good cycle detection in R-graph, a finite graph.
- We develop the R-graph \mathcal{R} so that the equations defining the edge relation can be readily specified as a Horn-clause logic program (Section 3). The transition relation of \mathcal{R} can be computed on the fly based on the transition relations of \mathcal{P} and \mathcal{B} , which may themselves be derived from more basic procedures (such as LTL tableaux for Büchi automata construction).
- We present a local good-cycle detection algorithm based on Tarjan’s algorithm [18] along the lines of [9], to handle the unique acceptance criteria of R-graph. The local algorithm detects good cycles as early as possible, ensuring that we explore only those transitions in \mathcal{P} and \mathcal{B} needed to complete the proof (Section 4). By evaluating these programs using the XSB logic programming system [19], we get a local, on-the-fly model checker.
- We show that, using tabled resolution, the model checker runs in $O(c \times b^3 \times 2^{g+l})$ time (where c is the size of program’s control flow graph, b the size of

Büchi automaton and g and l are the maximum number of global and local variables) and $O(c \times b^2 \times 2^{g+l})$ space. Our experiments show that our model checker is at least as efficient in practice as described in earlier literature, including the symbolic model checkers (Section 4).

We begin with a review of LTL model checking for push-down systems ignoring resource constraints.

2 Model Checking Push-Down Systems

In this section we give an overview of model checking push-down systems (PDS). PDSs can be used to model programs with procedures and can be extracted from the control flow graphs of programs. For details refer to [11].

Preliminaries. A PDS is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of *control locations*, Γ is a finite set of *stack alphabets* and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*. We shall use γ, γ' etc. to denote elements of Γ and use u, v, w etc. to denote elements of Γ^* . We write $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ to mean that $((p, \gamma), (p', w)) \in \Delta$.

We restrict ourselves to PDSs such that for every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, $|w| \leq 2$; any PDS can be put into such a form with linear size increase.

A *configuration* or *state* of \mathcal{P} is a pair $\langle p, w \rangle$ where $p \in P$ is a control location and $w \in \Gamma^*$ is a stack content. If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, then $\forall v \in \Gamma^*$ the configuration $\langle p', wv \rangle$ is an immediate successor of $\langle p, \gamma v \rangle$. Then we say $\langle p, \gamma v \rangle$ has a transition to $\langle p', wv \rangle$ and denote it by $\langle p, \gamma v \rangle \rightarrow \langle p', wv \rangle$. A *run* of \mathcal{P} is a sequence of the form $\langle p_0, w_0 \rangle, \langle p_1, w_1 \rangle, \dots, \langle p_n, w_n \rangle, \dots$ where $\langle p_i, w_i \rangle \rightarrow \langle p_{i+1}, w_{i+1} \rangle$ for all $i \geq 0$. A run denotes a finite or an infinite run.

A *Büchi automaton* is defined as $(Q, \rightarrow, \Sigma, Q_0, F)$ where Q is the finite set of states, $\rightarrow \subseteq (Q \times \Sigma \times Q)$, Σ is the set of edge labels, $Q_0 \subseteq Q$ is the set of start states and $F \subseteq Q$ is the set of final states. An accepting run in the automaton is defined to be a sequence $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_{k-1}} q_k \dots$ with $q_0 \in Q_0$ where $q_k \in F$ appears infinitely many times.

A *Büchi PDS* is defined as $(P_{bp}, P_0, \Gamma_{bp}, \Delta_{bp}, G_{bp})$ where P_{bp} is the finite set of control locations, $P_0 \in P_{bp}$ is the set of starting control location, Γ_{bp} is the set of stack alphabets, $\Delta_{bp} \subseteq (P_{bp} \times \Gamma_{bp}) \times (P_{bp} \times \Gamma_{bp}^*)$ and $G_{bp} \subseteq P_{bp}$ is the set of good control locations. The subscript bp may be dropped whenever it is obvious from the context. An accepting run in the Büchi PDS is defined to be an infinite sequence where configurations with control locations $\in G_{bp}$ appear infinitely many times.

Let $Prop$ be a finite set of propositions. Given a linear time temporal logic (LTL) formula ϕ over $Prop$, as is well known, one can construct a Büchi automaton with $\Sigma = 2^{Prop}$ that accepts the models of the formula ϕ .

Our aim is to verify PDSs against properties expressed as LTL formulas. Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, and let $\lambda : (P \times \Gamma) \rightarrow \Sigma$ be a labeling function. The truth of a proposition at a configuration is determined by the control location

and the symbol at the top of the stack in that configuration. Thus, any (infinite) run of \mathcal{P} defines a model for LTL over *Prop*.

In order to solve the model checking problem for PDSs, i.e. determine whether (the model defined by) every run of \mathcal{P} satisfies ϕ , it is sufficient to construct the Büchi automaton \mathcal{B} corresponding to $\neg\phi$ and verify that no run of \mathcal{P} is accepted by that Büchi automaton. This is done by constructing the product of \mathcal{P} and \mathcal{B} resulting in a Büchi PDS \mathcal{BP} and verifying that it accepts the empty language. The definition of the system \mathcal{BP} is as follows:

1. $P_{bp} = (P \times Q)$
2. $P_0 = \{(p, q) \in P_{bp} \mid q \in Q_0\}$
3. $\Gamma_{bp} = \Gamma$
4. $\Delta_{bp} = \{\langle (p, q), \gamma \rangle, \langle (p', q'), w \rangle \mid \langle p, \gamma \rangle \hookrightarrow \langle p, w \rangle, q \xrightarrow{\alpha} q', \text{ and } \alpha \subseteq \lambda(p, \gamma)\}$
5. $G_{bp} = \{(p, q) \mid q \in F\}$

In what follows we use \hookrightarrow to denote a transition rule in Δ_{bp} of \mathcal{BP} .

Definition 1 *Given a Büchi PDS \mathcal{BP} , we say that p_1 can weakly erase γ and get to p_2 (written $(p_1, \gamma, p_2) \in \text{Werase}$) if there is a run starting at the configuration $\langle p_1, \gamma \rangle$ and ending at $\langle p_2, \epsilon \rangle$. We say that p_1 can strongly erase γ and get to p_2 (written $(p_1, \gamma, p_2) \in \text{Serase}$) if there is a run starting at the configuration $\langle p_1, \gamma \rangle$ and ending at $\langle p_2, \epsilon \rangle$ in which at least one of the intermediate control states belongs to G .*

Proposition 1. *Let Erase be the least relation satisfying:*

1. $(p_1, \gamma_1, g, p') \in \text{Erase}$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \epsilon \rangle, g \equiv p_1 \in G$
2. $(p_1, \gamma_1, (g \vee g'), p') \in \text{Erase}$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma \rangle$ and $(p, \gamma, g', p') \in \text{Erase}, g \equiv p_1 \in G$
3. $(p_1, \gamma_1, (g \vee g' \vee g''), p'') \in \text{Erase}$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma \gamma_2 \rangle, (p, \gamma, g', p') \in \text{Erase}$ and $(p', \gamma_2, g'', p'') \in \text{Erase}, g \equiv p_1 \in G$

Then, $(p, \gamma, p') \in \text{Werase}$ iff $(p, \gamma, g, p') \in \text{Erase}$ for some g and $(p, \gamma, p') \in \text{Serase}$ iff $(p, \gamma, \text{true}, p') \in \text{Erase}$. Thus, Serase and Werase are computable.

In what follows, we shall often write $\text{Erase}(x, y, z)$ instead of $(x, y, z) \in \text{Erase}$.

Erase corresponds to $\text{pre}^*(P)$ in [10]. Since Erase is the least fixed point of its defining equations, the following corollary is immediate.

Corollary 1. *There is an integer k such that, for any pair of control locations p, p' and any stack symbol γ whenever $(p, \gamma, p') \in \text{Werase}(\text{Serase})$, there is witnessing run from (p, γ) to (p', ϵ) in which the size of the stack is bounded by k .*

The Erase relation for the Büchi PDS in Fig. 2(a) is given in Fig. 2(b).

Definition 2 *Given a Büchi PDS \mathcal{BP} , we associate with it two binary relations $\circ \xrightarrow{W}$ and $\circ \xrightarrow{S}$, over the set $P \times \Gamma$, as follows: $(p, \gamma) \circ \xrightarrow{W} (p', \gamma')$ iff there is a run from $\langle p, \gamma \rangle$ to $\langle p', \gamma' w \rangle$ for some $w \in \Gamma^*$. $(p, \gamma) \circ \xrightarrow{S} (p', \gamma')$ iff there is a run from $\langle p, \gamma \rangle$ to $\langle p', \gamma' w \rangle$, for some $w \in \Gamma^*$, that visits at least one configuration whose control location belongs to G .*

Proposition 2. Let the relation $\Longrightarrow \subseteq P \times \Gamma \times \{\text{false}, \text{true}\} \times P \times \Gamma$ be the least relation satisfying:

1. $(p_1, \gamma_1) \xrightarrow{p_1 \in G} (p', \gamma')$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma' \rangle$
2. $(p_1, \gamma_1) \xrightarrow{p_1 \in G} (p', \gamma')$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$
3. $(p_1, \gamma_1) \xrightarrow{p_1 \in G \vee g} (p', \gamma')$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma \gamma' \rangle$ and $\text{Erase}(p, \gamma, g, p')$

Then, $(p, \gamma) \xrightarrow{W} (p', \gamma')$ iff $(p, \gamma) \Longrightarrow^* (p', \gamma')$ and $(p, \gamma) \xrightarrow{S} (p', \gamma')$ iff $(p, \gamma) \Longrightarrow^* \xrightarrow{\text{true}} \Longrightarrow^* (p', \gamma')$ where $\Longrightarrow^* \xrightarrow{\text{false}} \xrightarrow{\text{true}}$.

Thus the relations \xrightarrow{S} and \xrightarrow{W} are computable.

The following theorem, [12], shows that, given the above proposition, the emptiness problem for any Büchi push-down system is decidable. We present the proof here since its details inspire the definition of resource constrained model checking (Section 3).

Theorem 1 A Büchi PDS \mathcal{BP} accepts some word iff there are p, γ, p', γ' such that $p \in P_0$, $(p, \gamma) \xrightarrow{W} (p', \gamma')$ and $(p', \gamma') \xrightarrow{S} (p', \gamma')$.

Proof : The following observation is useful.

Observation: If $\langle p_0, \gamma_0 w_0 \rangle \xrightarrow{*} \langle p_n, \gamma_n w_n \rangle$ is a run where for each i with $0 \leq i \leq n$, $|\gamma_i w_i| \geq |\gamma_0 w_0|$, then $(p_0, \gamma_0) \xrightarrow{W} (p_n, \gamma_n)$ and further if this run involves a configuration with $p_i \in G$ then $(p_0, \gamma_0) \xrightarrow{S} (p_n, \gamma_n)$. (In either case given run itself serves as a witness to this membership.)

Let the accepting run of \mathcal{BP} be $S = \langle p_0, \gamma_0 \rangle, \langle p_1, \gamma_1 w_1 \rangle, \dots \langle p_n, \gamma_n w_n \rangle \dots$. The proof proceeds by considering two cases.

Case 1: For any integer d the set $\{w_i \mid |w_i| = d\}$ is finite (i.e. the stack size grows “monotonically”).

Let i_d be the largest integer such that $|w_{i_d}| = d$. Clearly, i_d is monotonic on d . Let $\langle q_i, \gamma_i v_i \rangle = \langle p_{i_d}, w_{i_d} \rangle, \forall i \geq 1$. Then, $\langle q_i, \gamma_i \rangle \xrightarrow{*} \langle q_j, \gamma_j w_j \rangle \forall j > i$ via the subrun of the given run and further at every point in this run the size of the stack is at least $|\gamma_i v_i|$. Thus, by the above observation, $(q_i, \gamma_i) \xrightarrow{W} (q_j, \gamma_j) \forall i < j$. Further since the set of control locations and the stack alphabet are finite, there must be an infinite sequence j_1, j_2, \dots with $q' = q_{j_1} = q_{j_2} = \dots$ and $\gamma' = \gamma_{j_1} = \gamma_{j_2} = \dots$ and clearly there is a k such that in the subrun from $\langle q_{j_1}, \gamma_{j_1} v_{j_1} \rangle$ to $\langle q_{j_k}, \gamma_{j_k} v_{j_k} \rangle$ at least one of the intermediate configurations involves a control location from G . Thus, from the above observation $(q', \gamma') \xrightarrow{S} (q', \gamma')$. Once again using the above observation, $(p, \gamma) \xrightarrow{W} (q_i, \gamma_i)$ for each $i \geq 0$ and hence $(p, \gamma) \xrightarrow{W} (q', \gamma')$ and this completes the proof of this case.

Case 2: Otherwise, there is a least d such that there are infinitely many i with $|w_i| = d$. Then, clearly there is an N such that $\forall j \geq N \ |w_j| \geq d$. Therefore, there is an infinite sequence $j_1 < j_2 < \dots$, with $d < j_1$ with $|w_{j_i}| = d$. Let $w_{j_i} = \gamma_{j_i} v_{j_i}$.

Further, there is a sequence $j_1 < j_2 < \dots$ such that $q' = q_{j_1} = q_{j_2} = \dots$ and $\gamma' = \gamma_{j_1} = \gamma_{j_2} = \dots$. Once again, using the above observation (since the size of the stack at any configuration beginning at $\langle q_{j_i}, \gamma_{j_i} \rangle \geq d$) we conclude that $\langle q', \gamma' \rangle \circ^S \rightarrow \langle q', \gamma' \rangle$ and the proof follows as above.

For the converse, $(p, \gamma) \circ^W \rightarrow (p', \gamma')$ and $(p', \gamma') \circ^S \rightarrow (p', \gamma')$, then it is easy to see that there is an accepting run of the form $\langle p = p_1, \gamma = \gamma_1 \rangle \xrightarrow{*} \langle p', \gamma' v_0 \rangle \xrightarrow{*} \langle p', \gamma' v_1 v_0 \rangle \xrightarrow{*} \langle p', \gamma' v_1 v_1 v_0 \rangle \dots$ \square

3 Resource-Constrained Model Checking

In Section 2, an accepting sequence in \mathcal{BP} is defined without regard to the size of the stack in that sequence. This allows accepting sequences where the stack may diverge denoting an unfeasible run in any implementation of the program modeled by a PDS \mathcal{P} . We now focus only on runs where the stack size remains finite. We call the problem of determining whether a Büchi PDS has a finite-stack accepting run as the resource constrained model checking problem. Note that we do not bound the stack size *a priori* but consider all runs that have finite stack size.

We define two relations $\circ^W \rightarrow_0$ and $\circ^S \rightarrow_0$ similar to those in Definition 2.

Definition 3 *Given a Büchi PDS \mathcal{BP} , we associate with it a binary relation $\circ^S \rightarrow_0$, over the set $P \times \Gamma$, as follows: $(p, \gamma) \circ^S \rightarrow_0 (p', \gamma')$ iff there is a run from $\langle p, \gamma \rangle$ to $\langle p', \gamma' \rangle$, that visits at least one configuration whose control location belongs to G . Further $\circ^S \rightarrow_0$ corresponds to finite runs without net change in the stack size.*

Hence we have the following theorem.

Theorem 2 *A given Büchi PDS \mathcal{BP} has a finite stack accepting run iff there is p, γ, p', γ' such that $p \in P_0$, $(p, \gamma) \circ^W \rightarrow (p', \gamma')$ and $(p', \gamma') \circ^S \rightarrow_0 (p', \gamma')$.*

In order to show that the resource constrained model checking problem is decidable we need to show that the $\circ^S \rightarrow_0$ relation is computable.

Proposition 3. *Given a Büchi PDS \mathcal{BP} we define a relation $\Longrightarrow_0 \subseteq P \times \Gamma \times \{\text{false}, \text{true}\} \times \{0, 1\} \times P \times \Gamma$ as the least relation satisfying:*

1. $(p_1, \gamma_1) \xrightarrow{p_1 \in G} (p', \gamma')$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma' \rangle$
2. $(p_1, \gamma_1) \xrightarrow{p_1 \in G \vee g} (p', \gamma')$ if $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p, \gamma \gamma' \rangle$ and $\text{Erase}(p, \gamma, g, p')$

Then, $(p, \gamma) \circ^S \rightarrow_0 (p', \gamma')$ iff $(p, \gamma) \Longrightarrow_0^ \xrightarrow{\text{true}}_0 \Longrightarrow_0^* (p', \gamma')$. Hence, $\circ^S \rightarrow_0$ is computable.*

Proof : Let $(p, \gamma) \circ \xrightarrow{S}_0 (p', \gamma')$ and let $\langle p = p_1, \gamma = \gamma_1 \rangle \rightarrow \langle p_2, \gamma_2 w_2 \rangle \dots \rightarrow \langle p' = p_n, \gamma' = \gamma_n \rangle$ be the derivation witnessing this. Thus, there is an i such that $p_i \in G$.

We show that $(p, \gamma) \Longrightarrow_0^* \xrightarrow{true}_0 \Longrightarrow_0^* (p', \gamma')$ by induction on n . For $n = 0$, it must be the case that $p = p', \gamma = \gamma'$ and $p \in G$ and thus there is nothing to prove.

Suppose the result holds for all computations of length less than n . Now, there are two cases, if $w_2 = \epsilon$, then, by induction hypothesis, either $p \in G$ and $(p_2, \gamma_2) \Longrightarrow_0^* (p_n, \gamma_n)$ or $(p_2, \gamma_2) \Longrightarrow_0^* \xrightarrow{true}_0 \Longrightarrow_0^* (p_n, \gamma_n)$. In either case we have the desired result.

Now, suppose $w_2 \neq \epsilon$. Then $w_2 = \hat{\gamma}$ for some $\hat{\gamma} \in \Gamma$. By the definition of a run for a PDS, it then follows that there is a least $j > 2$ such that $\gamma_j = \hat{\gamma}$ and $w_j = \epsilon$. Thus, p_2 erases γ_2 and reaches p_j . Hence, depending on whether $1 \leq i < j$ or not, we either have $(p_1, \gamma_1) \xrightarrow{true}_0 (p_j, \gamma_j) \Longrightarrow_0^* (p_n, \gamma_n)$ or $(p_1, \gamma_1) \Longrightarrow_0 (p_j \gamma_j) \Longrightarrow_0^* \xrightarrow{true}_0 \Longrightarrow_0^* (p_n, \gamma_n)$. In either case we have the desired result.

The converse is an easy induction on the iterative definition of \xrightarrow{g}_0 and the details are omitted. \square

Theorem 2 shows that resource constrained model checking of a Büchi PDS can be reduced to checking for cycles in a graph induced by finite relations $\circ \xrightarrow{W}$ and $\circ \xrightarrow{S}_0$. Such a graph called an R-graph is defined as follows.

Definition 4 An R-graph of \mathcal{BP} is defined as $\mathcal{R} = ((P \times \Gamma), \Longrightarrow)$ where nodes are labeled by pair of control location and stack alphabet and set of edges are labeled by a pair (goodness label, resource label) with goodness label $\in \{true, false\}$, resource label $\in \{0, 1\}$.

The edge relation is such that there is an edge between nodes s_1 and s_2 iff $s_1 \xrightarrow{g}_0 s_2$, where \Longrightarrow is as defined in Proposition 2. g is called the goodness label of the edge.

The resource label of the edge is 0 if $s_1 \xrightarrow{g}_0 s_2$ where \Longrightarrow_0 is as defined in Proposition 3, and 1 otherwise.

A cycle in R-graph is said to be good if there is at least one edge in the cycle with goodness label *true* and resource labels of all edges in the cycle are 0. A path in R-graph starting at (p, γ) is said to be good if it reaches a good cycle.

Proposition 4. A given Büchi PDS \mathcal{BP} has a finite stack accepting run iff there is a good path in the corresponding R-graph.

The R-graph corresponding to the Büchi PDS in Fig. 2(a) is shown in Fig. 2(c).

4 Implementation

We now describe the salient aspects of an implementation of the model checker developed in the previous sections using logic programming. Encoding the various relations such as Erase as a logic program, and evaluating the program in

$ \begin{aligned} P &= \{p, q\} \\ P_0 &= \{p\} \\ \Gamma &= \{m_0, m_1, s_0, s_1, s_2\} \\ G &= \{q\} \\ \Delta &= \langle p, m_0 \rangle \hookrightarrow \langle p, s_0 m_1 \rangle \\ &\quad \langle p, m_1 \rangle \hookrightarrow \langle p, m_1 \rangle \\ &\quad \langle q, m_1 \rangle \hookrightarrow \langle q, m_1 \rangle \\ &\quad \langle p, s_0 \rangle \hookrightarrow \langle p, s_1 \rangle \\ &\quad \langle p, s_1 \rangle \hookrightarrow \langle p, s_0 s_2 \rangle \\ &\quad \langle p, s_2 \rangle \hookrightarrow \langle q, \epsilon \rangle \end{aligned} $	$ \begin{array}{c} \text{Erase relation} \\ \hline (p, s_0, false, p) \\ (p, s_2, false, q) \\ (p, s_1, false, q) \\ (p, s_0, false, q) \end{array} $	$ \begin{array}{c} \hline \stackrel{g}{\Rightarrow}_r \text{ relation} \\ \hline (p, m_0) \xrightarrow{false}_1 (p, s_0) \\ (p, m_1) \xrightarrow{false}_0 (p, m_1) \\ (q, m_1) \xrightarrow{true}_0 (q, m_1) \\ (p, s_0) \xrightarrow{false}_0 (p, s_1) \\ (p, s_1) \xrightarrow{false}_1 (p, s_0) \\ (p, m_0) \xrightarrow{false}_0 (p, m_1) \\ (p, m_0) \xrightarrow{false}_0 (q, m_1) \\ (p, s_1) \xrightarrow{false}_0 (p, s_2) \\ (p, s_1) \xrightarrow{false}_0 (q, s_2) \end{array} $
(a)	(b)	(c)

Fig. 2. (a) Büchi PDS, (b) corresponding Erase relation and (c) its R-graph

a goal-directed fashion, we get a local (*exploring* only the needed states) and on-the-fly (*constructing* states on demand) model checker.

From program to R-graph. Given a control flow graph representation of a program, it is straightforward to construct the equivalent PDS. Following [11], the valuation of global variables form the control states while the current node label and the valuation of local variables form the stack alphabet. We illustrate the construction for a *call* statement below. A transition $\langle p, \gamma \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_2 \rangle$ is represented below as `pds_trans(p, γ , p1, [γ_1 , γ_2])`.

```

pds_trans(G1, f(S1, L1), G2, [f(S,FL), f(S2,L1)]) :-
  cfg_node(S1, call(Proc, Params)),
  entry(Proc, S, Params, FL),
  cfg_edge(S1, _, S2).

```

In the fragment above, `cfg_edge` denotes the edge relation of a CFG (the 2-nd argument is a guard on the edge) and `cfg_node` denotes the mapping between node labels and the statements. The relation `entry` associates with each procedure `Proc` its entry point, formal parameters and local variables (which include the formals). Values are transformed at a basic level by *transfer functions* describing the behavior of statements such as assignments; the other statements propagate these changes.

A Büchi automaton can also be encoded with rules similar to those encoding a PDS. In fact, Horn clauses can be used to describe the construction of an automaton from the negation of an LTL formula [13]. Product construction to derive Büchi PDS is also straightforward and omitted. We assume that the transitions of a Büchi PDS are given by a relation `bpds_trans(P1, Gamma1, P2, Dest)` where `Dest` is a list of up to two elements with `nil` representing ϵ transitions.

```

erase(B1, Gamma, Good, B2) :-
    bpds_trans(B1, Gamma, B2, []),
    good_buechi_state(B1, Good).
erase(B1, Gamma, Good, B2) :-
    bpds_trans(B1, Gamma, B3, [Gamma1]),
    erase(B3, Gamma1, G1, B2),
    good_buechi_state(B1, G2), or(G1, G2, Good).
erase(B1, Gamma, Good, B2) :-
    bpds_trans(B1, Gamma, B3, [Gamma1, Gamma2]),
    erase(B3, Gamma1, G1, B4),
    erase(B4, Gamma2, G2, B2),
    good_buechi_state(B1, G3), or(G1, G2, Gt), or(Gt, G3, Good).

edge(s(B1, Gamma1), l(Good, 0), s(B2, Gamma2)) :-
    bpds_trans(B1, Gamma1, B2, [Gamma2]),
    good_buechi_state(B1, Good).
edge(s(B1, Gamma1), l(Good, 1), s(B2, Gamma2)) :-
    bpds_trans(B2, Gamma1, B2, [Gamma2, _]),
    good_buechi_state(B1, Good).
edge(s(B1, Gamma1), l(Good, 0), s(B2, Gamma2)) :-
    bpds_trans(B1, Gamma1, B3, [Gamma, Gamma2]),
    erase(B3, Gamma, G1, B2),
    good_buechi_state(B1, G2), or(G1, G2, Good).

```

Fig. 3. Generation of R-graphs from PDS models

Finally, the Erase relation (Definition 1) as well as the edge relation of the R-graph(Definition 4) are directly encoded as logic programs, as shown in Fig. 3. We use the relation `good_buechi_state(B,G)` to determine if `B` is an accepting state of \mathcal{B} .

Complexity. The crucial predicate in the encoding is `erase`. When evaluated with tabled resolution [17, 6], `erase` can be computed in $O(|\Delta_{bp}| \times |P_{bp}|^2)$, where Δ_{bp} and P_{bp} are the number of transitions and control states, respectively, in the Büchi PDS. To derive the space and time complexity in terms of the input program's size, let c be the size of the control flow graph, b the size of the Büchi automaton, and g and l be the (maximum) number of global and local variables. Then the time complexity of computing `erase` is $O(c \times b^3 \times 2^{g+l})$. The cubic factor comes from the last rule of `erase` which performs a join and hence effectively iterates once over all states in the Büchi automaton (note that `B4` is drawn only from the states of \mathcal{B}) for each tuple in the relation. The size of `erase` relation is $O(c \times b^2 \times 2^{g+l})$.

It can also be readily seen that the time taken to completely evaluate `edge` is $O(c \times b \times 2^{g+l})$ once `erase` has been computed. The size of the R-graph is also $O(c \times b \times 2^{g+l})$. Good cycles in the R-graph can be detected in time proportional to the size of the graph and hence the overall time to model check is $O(c \times b^3 \times 2^{g+l})$, matching the best-known algorithms. The time complexities assume unit-time table lookups. Organizing the tuples of the relations as binary trees would increase the complexity by a factor of $O((\log(c) + \log(b))(g + l))$. In an implementation platform, such as the XSB logic programming system [19], the tuples are a factor of $O((\log(c) + \log(b))(g + l))$. In a realistic implementation platform, such as the XSB logic programming system, the tuples are organized

using trie data structures, giving close to unit-time lookups in practice. The tries sometimes induce parts of tuple representations to be shared, reducing the space complexity.

The analysis does not take into account the locality due to the goal-directedness of tabled evaluation, since it does not appear to reduce the worst case complexity. However, if the transfer functions were monotonic (as in data-flow analyses), the factor of 2^{g+l} will be brought to $g(g+l)^2$ with goal-directed evaluation. We now present a local cycle detection algorithm that exploits the locality, by invoking `edge` and, in turn, `erase` only until a good cycle is found.

Local detection of good cycles. The final step in model checking is determining if there is a reachable good cycle in the R-graph. Recall that a good cycle is defined as one which has at least one edge with goodness label being true while all edges in the cycle have resource labels 0. The first condition is a disjunctive property: a cycle has a good edge if and only if an SCC has a good edge. Tarjan's SCC algorithm [18] can be adapted to perform local good-cycle detection when the acceptance condition is a disjunctive property: e.g., Couvreur's algorithm in [9]. The second condition, however, cannot be cast as a property of SCC. We present a local algorithm that incorporates both conditions. The algorithm, presented in Fig. 4, uses a modification of Couvreur's algorithm as a subroutine.

```

1. Boolean good_path(v0)
2. begin
3.   pending := { v0 };
4.   while ( v ∈ pending )
5.     pending := pending - {v};
6.     DFSnum := 1;
7.     Sstack := empty;
8.     Lstack := empty;
9.     push(Lstack, false);
10.    if (good_cycle(v)) then
11.      return true;
12.    end while
13.    return false;
14. end

1. procedure mark(v)
2. begin
3. if not v.complete then
4.   v.complete := true;
5.   forall (w such that there is
6.     an edge from v to w)
7.     mark(w);
8.   end forall
9. end

1. Boolean good_cycle(v)
2. begin
3.   v.visited := true;
4.   v.dfsnum := DFSnum++;
5.   push(Sstack, v.dfsnum);
6.   forall (G,w such that there is an edge
7.     from v to w with goodness label G
8.     and resource label 0)
9.     if (not w.visited) then
10.      push(Lstack, G);
11.      if good_cycle(w) then
12.        return true;
13.      else if not w.complete then
14.        if (G) then
15.          return true;
16.        else
17.          while (top(Sstack) > w.dfsnum)
18.            if (top(Lstack)) then
19.              return true
20.            else
21.              pop(Lstack); pop(Sstack);
22.            end while
23.          end forall
24.          if (top(Sstack) = v.dfsnum) then
25.            pop(Sstack); pop(Lstack);
26.            mark(v);
27.          forall (w such that there is an edge
28.            from v to w with resource label 1
29.            and not w.visited)
30.            pending := pending + {w};
31.          end forall
32.          return false;
33.        return false;
34.      end

```

Fig. 4. Local Good-cycle detection algorithm

We handle the “all 0-edges” condition by partitioning the depth first search where we explore all edges with a 0 resource label before looking at any with a 1 resource label. Given a graph with nodes in set S and a starting node v_0 , this partitions the nodes into sets S_0 and S'_0 , where S_0 consists of all (and only) those nodes that are reachable from v_0 using edges with 0 resource labels, and $S'_0 = S - S_0$. We do this partitioning while looking for good cycles in the subgraph induced by S_0 using a modification of the algorithm in [9]. If no good cycles are found, we pick a node, say v_1 from S'_0 that is reachable from some node in S_0 via a edge with resource label 1. We use v_1 to partition S'_0 into S_1 and S'_1 , and so on. This procedure will partition the graph into subgraphs containing only 0-edges where the subgraphs are connected by 1-edges. If a good “all 0-edge” cycle exists it will be within one subgraph since there are no 0-edges from a node S_i to a node in S_j if $j > i$.

In the algorithm in Fig. 4, we use two global stacks: *Sstack*, the stack of DFS numbers of current SCC roots, and *Lstack* that summarizes the labels on edges in/between each of the components rooted in *Sstack*. These stacks guide the local detection of good cycles within a single subgraph. While exploring a subgraph, when a previously visited node in an incomplete SCC is seen via a 0-edge, say from v to w , then we combine the SCC roots of v and w . While doing so we update the status of labels in the combined SCC and return immediately if the summary indicates a *true* label (lines 15–23 in *good_cycle()*). We use a set *pending* to record nodes reachable via a 1-edge from the current subgraph. Thus, at the end of exploring a subgraph S_i , *pending* contains exactly the set of nodes in S_{i+1} . The algorithm also maintains various marks on each node: *visited* and *complete*, both initially *false* and *dfsnum* that records the node’s DFS number.

It is easy to show that the local algorithm is linear. Although the algorithm only determines whether or not a good path exists, it can be readily modified to output such a path. Finally, by organizing *pending* as a queue, we can ensure that we will find a path with the smallest amount of stack consumed in the initial segment leading up to the good cycle.

Performance. We tested the performance of our model checker on an example program, shown in Fig. 1(a,c) with one modification: `main` (Fig 1(a)) calls `flip(N)` instead of `flip`. The procedure in Fig. 1(c) is a concrete version of the one shown in Fig. 1(b) with the recursion control parameter left unabstracted. Note that in the concrete version calls to the procedure `flip` from `main` is done with the recursion depth parameter `N`. We verified the property `AGF reach` for different values of the recursion depth parameter `N`. Fig. 5(a) shows the running time and space statistics for model checking when the global variable `g` initialized to *false*. With this initial value the property is true, and there are no good cycles in the corresponding R-graph (recall that we check for negation of the property). Fig. 5(b) shows the performance of our model checker with `g` left uninitialized (thus exploring both *true* and *false* valuations). In this case, which is identical to the one reported in [2] and [11], the property is false, and we exit the model checker as soon as we see the first good cycle in the R-graph.

(g initially <i>false</i> : no good cycle)					(g initially <i>undefined</i> : \exists good cycle)				
N	CPU Time	Space			N	CPU Time	Space		
		Total	Table				Total	Table	
1K	0.5s	10M	5M		8K	0.6s	19M	13M	
2K	1.1s	19M	10M		16K	1.2s	37M	27M	
4K	2.1s	37M	20M		32K	2.2s	74M	54M	
8K	4.4s	74M	40M		64K	4.8s	147M	108M	
16K	8.9s	148M	81M		128K	9.6s	294M	216M	
32K	17.2s	295M	161M		256K	19.0s	587M	431M	

Fig. 5. Performance of our model checker on AGF `reach` for program in Fig. 1(a,c) for `g` initialized to *false* (a) and left uninitialized (b). Measurements taken using XSB2.4 & Mandrake Linux 8.1 running on a 1.7GHz Xeon with 2GB memory.

The performance numbers are preliminary and only serve to highlight the unique aspects of our model checker. First of all, the figures show the impact of local model checking on this problem, with more than 7-fold difference in running time. Secondly, even though the performance reported here and in [11] were collected on different hardware platforms, the raw times in Fig. 5(a) are about 5 times smaller than those given in [11], indicating that a local explicit state checker can offer performance comparable to a symbolic one even when the entire state space is explored. Thirdly, the time and space performance for both cases is linear in the size of the input program, indicating no hidden costs in computing over a logic programming engine.

Finally, we ran our model checker on the abstract program shown in Fig. 1(a,b): the time and space consumption was too small to measure. That experiment shows the utility of resource-constrained checking: we have in effect shown the validity of the AGF `reach` for all values of the recursion parameter `N` in negligible time. It should be noted that program in Fig. 1(a,b) is natural abstraction of the case whose verification results are shown in Fig. 5(a).

5 Discussion

As mentioned earlier, our formulation of resource-constrained model checking is closely related to the works of [12, 4, 10, 11], where efficient algorithms have been described for model checking PDSs. Apart from the annotation of resource consumption on the edges of the R-graph, we provide a considerably different implementation strategy. For instance, [10] presents a model checking technique where *Pre** relation (analogous to our *Erase*) is used in two phases: one to identify good cycles (repeating heads) and another to check if such cycles are reachable. The subsequent paper [11] presents a symbolic algorithm for model checking PDSs. In contrast, we encode our model checker so as to derive a local (explicit-state) algorithm, and avoid the second use of *Erase*.

Recent works in [1, 3], show that (*recursive* or *hierarchical*) state machines can be used to model control flow of sequential programs consisting of recursive

calls. Both works give model checking algorithms that, when used for model checking push-down systems, run in time cubic in the size of the Büchi automaton and linear in the size of the push-down system. Furthermore, [3] describes special classes of state machines for which the model checking algorithms have better complexity. The main essence of both these works is to compute *summary* edges that reveal the relationship between the entry and exit points of each state machine. In addition, [1] points out that identifying edges that lead to increase in stack size, model checking can be restricted to finite-stack paths. The important similarities between [1, 3] and our work are as follows:

- Summary edges are analogous to \Longrightarrow_0 relation as computed in Proposition 3.
- Edges F_a and F_u as computed in [1], revealing finite- and infinite-stack paths respectively, are identical to $\circ \xrightarrow{S} \rightarrow_0$ and \Longrightarrow_1 .
- Optimization techniques involving forward and backward analysis of summary edges as discussed in these papers can be directly incorporated in our work.

The distinguishing aspect of our work is that we concretely describe a high-level yet efficient implementation of a local, on-the-fly model checker that can distinguish finite-stack runs from arbitrary runs of a push-down system.

The idea behind of Erase and R-graph appears to be more universal than model checking of PDSs. For instance, inter-procedural data flow analysis techniques define summaries of calls, which are simply variants of Erase. Closer inspection of data flow techniques reveal striking (although not surprising) similarities. These similarities are best exhibited by [15] and related works, where data flow analysis is formulated in terms of graph-reachability. Some of the analogies are listed below:

- Same Level Inter-procedurally valid paths (SLIVP): All the calls in the path is matched by the corresponding return. This is analogous to $\circ \rightarrow_0$ that we use to define a good cycle in R-graph.
- Inter-procedurally valid path (realizable path IVP) : All returns are matched but not all calls. This is similar to $\circ \rightarrow_1$.
- Path Edge \subset SLIVP: This is \Longrightarrow_0^* and $\circ \rightarrow_0$
- Summary Edge \subset SLIVP: This is \Longrightarrow_0 restricted to call nodes.

Although the interplay between data flow analysis and model checking has been widely recognized (e.g. [16]), the closeness in the details of algorithms used indicates a potential for furthering the practice in both areas through a better understanding of the interactions. Finally, although model checking of recursive programs using mu-calculus has been explored [5], the techniques appear to have an exponential blowup to handle recursion. It will be interesting to explore the relationship between these techniques and the ones presented in this paper, and is a topic of current research.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Computer-Aided Verification (CAV 2001)*. Springer-Verlag, 2001.

2. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN00: SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130, 2000.
3. M. Benedikt, P. Godefroid, and T. Reps. Model checking unrestricted hierarchical state machines. In *Twenty-Eighth Int. Colloq. on Automata, Languages, and Programming (ICALP 2001)*. Springer-Verlag, 2001.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Concurrency Theory (CONCURR 1997)*, 1997.
5. O. Burkart and B. Steffen. Model checking the full-modal mu-calculus for infinite sequential processes. In *Proceedings of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429, 1997.
6. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
9. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, 1999.
10. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer-Aided Verification (CAV 2000)*, pages 232–247. Springer-Verlag, 2000.
11. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Computer-Aided Verification (CAV 2001)*, pages 324–336. Springer-Verlag, 2001.
12. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Second International Workshop on Verification of Infinite State Systems (INFINITY 1997)*, volume 9. Elsevier Science, 1997.
13. L.R. Pokorny and C.R. Ramakrishnan. LTL model checking using tabled logic programming. In *Workshop on Tabling in Parsing and Deduction*, 2000. Available from <http://www.cs.sunysb.edu/~cram/papers>.
14. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
15. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.
16. D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Static Analysis Symposium*, pages 351–380, 1998.
17. H. Tamaki and T. Sato. OLD T resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. MIT Press, 1986.
18. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
19. XSB. The XSB logic programming system. Available from <http://xsb.sourceforge.net>.