A BDD-based Model Checker for Recursive Programs *

Javier Esparza, Stefan Schwoon

Technische Universität München
Arcisstr. 21, 80290 München, Germany
{esparza,schwoon}@in.tum.de

Abstract. We present a model-checker for boolean programs with (possibly recursive) procedures and the temporal logic LTL. The checker is guaranteed to terminate even for (usually faulty) programs in which the depth of the recursion is not bounded. The algorithm uses automata to finitely represent possibly infinite sets of stack contents and BDDs to compactly represent finite sets of values of boolean variables. We illustrate the checker on some examples and compare it with the Bebop tool of Ball and Rajamani.

1 Introduction

Boolean programs are C programs in which all variables and parameters (call-by value) have boolean type, and which may contain procedures with recursion. In a series of papers, Ball and Rajamani have convincingly argued that they are a good starting point for investigating model checking of software [1, 2].

Ball and Rajamani have also developed Bebop, a tool for reachability analysis in boolean programs. As part of the SLAM toolkit, Bebop has been successfully used to validate critical safety properties of device drivers [2]. Bebop can determine if a point of a boolean program can be reached along some execution path. Using an automata-theoretic approach it is easy to extend Bebop to a tool for safety properties. However, it cannot deal with liveness or fairness properties requiring to examine the infinite executions of the program. In particular, it cannot be used to prove termination.

In this paper we overcome this limitation by presenting a model-checker for boolean programs and arbitrary LTL-properties. The input to the model checker are symbolic pushdown systems (SPDS), a compact representation of the pushdown systems studied in [4]. A translation of boolean programs into this model is straightforward. The checker is based on the efficient algorithms for model checking ordinary pushdown systems (PDS) of [4]. While SPDSs have the same expressive power as PDSs, they can be exponentially more compact. (Essentially, the translation works by expanding the set of control states with

* This work was partially supported by the project "Advanced Validation Techniques for Telecommunication Protocols" of the Information Societies Technology Programme of the European Union.

all the possible values of the boolean variables.) Therefore, translating SPDSs into PDSs and then applying the algorithms of [4] is very inefficient. We follow a different path: We provide symbolic versions of the algorithms of [4] working on SPDSs, and use BDDs to succintly encode sets of (tuples of) values of the boolean variables.

This paper (and its full version [5]) contribute symbolic versions of the algorithms of [4], tuned to minimise the number of required BDD variables; an efficient implementation including three heuristic improvements; some experimental results on different versions of Quicksort; and, finally, a performance comparison with Bebop using an example of [1].

The paper is structured as follows. PDSs and SPDSs are introduced in Section 2. The symbolic versions of the algorithms of [4] are presented in Section 4 and their complexity is analysed. In particular, we analyse the complexity in terms of the number of global and local variables. In Section 5 we discuss the improvements in the checker and present our results on verifying Quicksort; in particular we analyse the impact of the improvements. Section 6 contains the comparison with Bebop, and Section 7 contains conclusions.

2 Basic definitions

In this section we briefly introduce the notions of pushdown systems and linear time logic, and establish our notations for them.

2.1 Pushdown systems

We mostly follow the notation of [4]. A *pushdown system* is a four-tuple $\mathcal{P} = (P, \Gamma, c_0, \Delta)$ where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*. If $((q, \gamma), (q', w)) \in \Delta$ then we write $\langle q, \gamma \rangle \hookrightarrow \langle q', w \rangle$. A *configuration* 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*. c_0 is called the *initial configuration* of $\mathcal{P}$. The set of all configurations is denoted by $\mathcal{C}$.

If $\langle q, \gamma \rangle \hookrightarrow \langle q', w \rangle$, then for every $v \in \Gamma^*$ the configuration $\langle q, \gamma v \rangle$ is an *immediate predecessor* of $\langle q', wv \rangle$, and $\langle q', wv \rangle$ an *immediate successor* of $\langle q, \gamma v \rangle$. The *reachability relation* $\Rightarrow$ is the reflexive and transitive closure of the immediate successor relation. A *run* of $\mathcal{P}$ is a sequence of configurations such that for each two consecutive configurations $c_i c_{i+1}$, c_{i+1} is an immediate successor of c_i.

The predecessor function $pre\colon 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ of $\mathcal{P}$ is defined as follows: c belongs to $pre(C)$ if some immediate successor of c belongs to C. The reflexive and transitive closure of pre is denoted by pre^*. We define $post(C)$ and $post^*$ similarly.

In the next section, when we model boolean programs as pushdown systems, we will see that it is natural to consider a product form for P and G. More precisely, it is convenient to introduce sets P_0 and G such that $P = P_0 \times G$, and sets Γ_0 and L such that $G = \Gamma_0 \times L$. G and L are called sets of global and local values, since they are, loosely speaking, the possible valuations of the global and local variables of the program, respectively. So, for the rest of the paper, we assume

$$P = P_0 \times G \quad \text{and} \quad G = \Gamma_0 \times L \ .$$

A symbolic pushdown system is a pushdown system in which sets of transition rules are represented by symbolic transition rules. Formally, a *symbolic pushdown system* is a tuple $\mathcal{P}_S = (P, \Gamma, c_0, \Delta_S)$, where Δ_S is a set of *symbolic transition rules* of the form $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1 \ldots \gamma_n\rangle$, and $R \subseteq (G \times L) \times (G \times L^n)$ is a relation. A symbolic pushdown system corresponds to a normal pushdown system $(P_0 \times G, \Gamma_0 \times L, c_0, \Delta)$ in the sense that a symbolic rule $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1 \ldots \gamma_n\rangle$ denotes a set of transition rules as follows:

$$\text{if } (g, l, g', l_1, \ldots, l_n) \in R, \text{ then } \langle (p, g), (\gamma, l)\rangle \hookrightarrow \langle (p', g'), (\gamma_1, l_1) \ldots (\gamma_n, l_n)\rangle \in \Delta$$

In practice, R should have an efficient symbolic representation. In our applications we have $G = \{0, 1\}^n$ and $L = \{0, 1\}^m$ for some n and m, and so R can be represented by a BDD.

Given a pushdown system $\mathcal{P} = (P, \Gamma, c_0, \Delta)$, we use $\mathcal{P}$*-automata* to represent regular sets of configurations of $\mathcal{P}$. A $\mathcal{P}$-automaton uses Γ as alphabet, and P as set of initial states. Formally, a $\mathcal{P}$-automaton is a tuple $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ where Q is a finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is a set of *transitions*, P is the set of *initial states* and $F \subseteq Q$ is the set of *final states*. An automaton *accepts* or *recognises* a configuration $\langle p, w\rangle$ if $p \xrightarrow{w} q$ for some $p \in P$, $q \in F$. The set of configurations recognised by an automaton $\mathcal{A}$ is denoted by $\mathit{Conf}(\mathcal{A})$. A set of configurations of $\mathcal{P}$ is *regular* if it is recognized by some automaton.

A *symbolic* $\mathcal{P}_S$*-automaton* is a tuple $\mathcal{A}_S = (\Gamma_0, Q, \delta_S, P_0, F)$, where the *symbolic transition relation* is a function $\delta_S \colon (Q \times \Gamma_0 \times Q) \to (G \times L \times G)$. The relation δ_S should be seen as the symbolic representation of the transition relation δ: $\delta_S(q, \gamma, q')$ is the set of all (g, l, g') such that $((q, g), (\gamma, l), (q', g')) \in \delta$. If $R \subseteq (G \times L \times G)$, we denote by $q \xrightarrow[R]{\gamma} q'$ the set of transitions $((q, g), (\gamma, l), (q', g'))$ such that $(g, l, g') \in R$. In the sequel, $\mathcal{P}$*-automata* and *symbolic* $\mathcal{P}_S$*-automata* are just called automata and symbolic automata, respectively.

2.2 The model-checking problem for LTL

We briefly recall the results of [3] and [4]. Given a formula φ of LTL, the model-checking problem consists of deciding if c_0 violates φ, that is whether there is some run starting at c_0 that violates φ. The problem is solved in [4] using the automata-theoretic approach. First, a Büchi pushdown system is constructed as the product of the original pushdown system and a Büchi automaton $\mathcal{B}$ for the negation of φ. This new pushdown system has a set of final control states. We define a new reachability relation $\overset{r}{\Longrightarrow}$ with respect to this set; we write $c \overset{r}{\Longrightarrow} c'$ if c' can be reached from c while visiting some final control state along the way. Now, define the *head* of a transition rule $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ as the configuration $\langle p, \gamma\rangle$. A head $\langle p, \gamma\rangle$ is *repeating* if there exists $v \in \Gamma^*$ such that $\langle p, \gamma\rangle \overset{r}{\Longrightarrow} \langle p, \gamma v\rangle$

```
int x;

void main() {
   int z;
   ...
   f(x+z);
   ...
}

void f() {
   x=x+y;
}
```

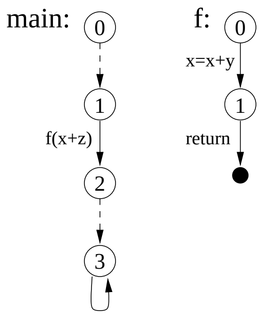

Fig. 1. An example program (left) and the associated flowgraph (right)

holds. We denote the set of repeating heads by Rh. It is shown in [4] that the model-checking problem reduces to either checking whether $c_0 \in pre^*(Rh\,\Gamma^*)$, or, equivalently, checking whether $post^*(\{c_0\}) \cap Rh\,\Gamma^* \neq \emptyset$. Furthermore, it is shown that the problem can be solved in $\mathcal{O}(|P|^2\,|\Delta|\,|\mathcal{B}|^3)$ time and $\mathcal{O}(|P|\,|\Delta|\,|\mathcal{B}|^2)$ space.

3 Modelling programs as symbolic pushdown systems

Pushdown systems find a natural application in the analysis of sequential programs with procedures (written in C or Java, for instance). We allow arbitrary recursion, even mutual procedure calls, between procedures; however, we require that the data types used in the program be finite. In the following, we present informally how to derive a symbolic pushdown system from such a program.

In a first step, we represent the program by a system of *flow graphs*, one for each procedure. The nodes of a flow graph correspond to control points in the procedure, and its edges are annotated with statements, e.g. assignments or calls to other procedures. Non-deterministic control flow is allowed and might for instance result from abstraction. Figure 1 shows a small C program and the corresponding flow graphs. The procedure `main` ends in an infinite loop to ensure that all executions are infinite. In the example, a finitary fragment of the type integer has to be chosen.

Given such a system of flow graphs, we derive a pushdown system and a corresponding symbolic pushdown system. For simplicity, we assume that all procedures have the same local variables. The sets G and L contain all the possible valuations of the global and local variables, respectively. E.g., if the program contains three boolean global variables and each procedure has two boolean local variables, then we have $G = \{0,1\}^3$ and $L = \{0,1\}^2$. P_0 contains one single element p, while Γ is the set of nodes of the flow graphs.

Program statements are translated to pushdown rules of three types.

Assignments An assignment labelling a flow-graph edge from node n_1 to node n_2 is represented by a set of rules of the form

$$\langle glob, (n_1, loc)\rangle \hookrightarrow \langle glob', (n_2, loc')\rangle.$$

where $glob$ and $glob'$ (loc and loc') are the values of the global (local) variables before and after the assigment. This set is represented by a symbolic rule of the form $\langle p, n_1\rangle \xhookrightarrow{R} \langle p, n_2\rangle$, where $R \subseteq (G \times L) \times (G \times L)$.

Procedure calls A procedure call labelling a flow-graph edge from node n_1 to node n_2 is translated into a set of rules with a right-hand side of length two according to the following scheme:

$$\langle glob, (n_1, loc)\rangle \hookrightarrow \langle glob', (m_0, loc')\, (n_2, loc'')\rangle$$

Here m_0 is the start node of the called procedure; loc' denotes initial values of its local variables; loc'' saves the local variables of the calling procedure. (Notice that no stack symbol contains variables from different procedures; hence the size of the stack alphabet depends only on the largest number of local variables in any procedure.) This set is represented by a symbolic rule of the form $\langle p, n_1\rangle \xhookrightarrow{R} \langle p, m_0 n_2\rangle$, where $R \subseteq (G \times L) \times (G \times L \times L)$.

Return statements A return statement has an empty right side:

$$\langle glob, (n, loc)\rangle \hookrightarrow \langle glob', \varepsilon\rangle$$

These rules correspond to a symbolic rule of the form $\langle p, n\rangle \xhookrightarrow{R} \langle p, \epsilon\rangle$, where $R \subseteq (G \times L) \times G$. Procedures which return values can be simulated by introducing an additional global variable and assigning the return value to it.

Notice that the size of the symbolic pushdown system may be exponentially smaller than the size of the pushdown system. This is the fact we exploit in order to make model-checking practically usable, at least for programs with few variables. Notice also that in the symbolic pushdown system we have $|P_0| = 1$ and Γ_0 is the set of nodes of the flow graphs.

Since a symbolic pushdown system is just a compact representation of an ordinary pushdown system, we continue to use the theory presented in [4]. In this paper we provide modified versions of the model-checking algorithms that take advantage of a more compact representation. In our experiments, we consider programs with boolean variables only and use BDDs to represent them. Integer variables with values from a finite range are simulated using multiple boolean variables.

4 Algorithms

According to Section 2 we can solve the model-checking problem by giving algorithms for the following three tasks:

– to compute the set $pre^*(C)$ for a regular set of configurations C (which will be applied to $C = Rh\,\Gamma^*$)
– to compute the set $post^*(C)$ for a regular set of configurations C (which will be applied to $C = \{c_0\}$)
– to compute the set of repeating heads Rh

In [4] efficient implementations for these three problems were proposed for ordinary pushdown systems. In this section, we sketch how the algorithms may be lifted to the case of symbolic pushdown systems. More detailed presentations are given in the full version of the paper [5]. We fix a symbolic pushdown system $\mathcal{P} = (P_0 \times G, \Gamma_0 \times L, c_0, \Delta_S)$ for the rest of the section.

4.1 Computing predecessors

Given a regular set C of configurations of $\mathcal{P}$, we want to compute $pre^*(C)$. Let $\mathcal{A}$ be a $\mathcal{P}$-automaton that accepts C. We modify $\mathcal{A}$ to an automaton that accepts $pre^*(C)$. The modification procedure adds only new transitions to $\mathcal{A}$, but no new states are created. Without loss of generality, we assume that $\mathcal{A}$ has no transitions ending in an initial state.

In ordinary pushdown systems, new transitions are added according to the following *saturation rule:*

If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma_1 \ldots \gamma_n\rangle$ and $p' \xrightarrow{\gamma_1} q_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} q$ in the current automaton, add a transition (p, γ, q).

The correctness of the procedure was proved in [3]. For the symbolic case, the corresponding rule becomes:

If $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1\gamma_2 \ldots \gamma_n\rangle$ and $p' \xrightarrow[R_1]{\gamma_1} q_1 \xrightarrow[R_2]{\gamma_2} \cdots \xrightarrow[R_n]{\gamma_n} q$ in the current automaton, replace $p \xrightarrow[R']{\gamma} q$ by $p \xrightarrow[R'']{\gamma} q$ where $R'' = R' \cup \{\, (g, l, g_n) \mid (g, l, g_0, l_1, \ldots, l_n) \in R \wedge \exists g_1, \ldots, g_{n-1} \colon \forall 1 \le i \le n \colon (g_{i-1}, l_i, g_i) \in R_i \,\}$.

The computation of R'' can be carried out using standard BDD operations. A detailed, efficient implementation of the procedure can be found in [5].

4.2 Computing the repeating heads

For ordinary pushdown systems [4] we construct a directed graph whose nodes are the heads of the transition rules (and so elements of $P \times \Gamma$). There is an edge from (p, γ) to (p', γ') iff there is a rule $\langle p, \gamma\rangle \hookrightarrow \langle p'', v_1\gamma' v_2\rangle$ where $\langle p'', v_1\rangle \Rightarrow \langle p', \varepsilon\rangle$ holds. The edge has label 1 iff either p is an accepting Büchi state, or $\langle p'', v_1\rangle \xRightarrow{r} \langle p', \varepsilon\rangle$. The edges are computed using pre^*. A head (p, γ) is repeating iff it belongs to a strongly connected component (SCC) containing a 1-labelled edge. The SCCs are computed in linear time using Tarjan's algorithm [9].

For symbolic pushdown systems we represent the graph compactly as a symbolic graph SG. The nodes of SG are elements of $P_0 \times \Gamma_0$, and its edges are annotated with a relation $R \subseteq (G \times L)^2$ (plus a boolean, which is easy to handle and is omitted in the following discussion for clarity). An edge $(p_0, \gamma_0) \xrightarrow{R} (p'_0, \gamma'_0)$ stands for the set of edges $(p_0, g, \gamma_0, l) \rightarrow (p'_0, g', \gamma'_0, l')$ such that $(g, l, g', l') \in R$. Unfortunately, when R is symbolically represented Tarjan's algorithm cannot be applied. A straightforward approach is to "saturate" SG instead according to the following two rules:

- If $(p_0, \gamma_0) \xrightarrow{R} (p''_0, \gamma''_0) \xrightarrow{R'} (p'_0, \gamma'_0)$, then add $(p_0, \gamma_0) \xrightarrow{R''} (p'_0, \gamma'_0)$, where $R'' := \{((g, l), (g', l')) \mid \exists (g'', l'') \colon ((g, l), (g'', l'')) \in R \wedge ((g'', l''), (g', l')) \in R'\}$.
- If $(p_0, \gamma_0) \xrightarrow{R} (p'_0, \gamma'_0)$ and $(p_0, \gamma_0) \xrightarrow{R'} (p'_0, \gamma'_0)$, then replace these two arcs by $(p_0, \gamma_0) \xrightarrow{R \cup R'} (p'_0, \gamma'_0)$

The saturation procedure terminates when a fixpoint is reached. It is easy to see that this algorithm has complexity $\mathcal{O}(n \cdot m)$ where n and m are the number of nodes and edges of G. Using this method, the model-checking problem for symbolic systems has a worse asymptotic complexity than for normal systems.

In practice, this disadvantage can be made up for, mainly due to the more succinct representation. Moreover, the straightforward approach can be replaced with more refined strategies that work better in practice (see the discussion in Section 5).

4.3 Computing successors

Given an automaton $\mathcal{A}$ accepting the set C, we modify it to an automaton accepting $post^*(C)$. Again we assume that $\mathcal{A}$ has no transitions leading to initial states, and moreover, that $|w| \leq 2$ holds for all rules $\langle p, \gamma \rangle \overset{R}{\hookrightarrow} \langle p', w \rangle$. This is not an essential restriction, as all systems can be transformed into one of this form with only a linear increase in size.

In the ordinary case, we allow ε-moves in the automaton. We write $\xrightarrow{\gamma}$ for the relation $(\xrightarrow{\varepsilon})^* \xrightarrow{\gamma} (\xrightarrow{\varepsilon})^*$. The algorithm works in two steps [4]:

- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$, add a state (p', γ') and a transition $(p', \gamma', (p', \gamma'))$.
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition (p', ϵ, q).
If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition (p', γ', q).
If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ and $p \overset{\gamma}{\Longrightarrow} q$ in the current automaton, add a transition $((p, \gamma), \gamma'', q)$.

For the symbolic case, the corresponding first step looks like this: For each symbolic rule $\langle p, \gamma \rangle \xhookrightarrow{R} \langle p', y'y'' \rangle$ we add a new state (p', γ'). We must adjust the symbolic transition relation slightly for these new states; e.g. when q and q' are such states, then $\delta_S(q, \gamma, q')$ is a subset of $(G \times L) \times L \times (G \times L)$. Moreover, for each such rule we add a transition $t = (p', y', (p', y'))$ s.t. $\delta_S(t) = \{\, (g', l', (g', l')) \mid \exists (g, l, g', l', l'') \in R \,\}$. Concerning ε-transitions, $\delta_S(q, \varepsilon, q')$ is a subset of $G \times G$. In the second step, we proceed as follows:

If $\langle p, \gamma \rangle \xhookrightarrow{R} \langle p', \varepsilon \rangle \in \Delta_S$ and $p \xRightarrow[R']{\gamma} q$ in the current automaton,
add to $\delta_S(p', \varepsilon, q)$ the set $\{\, (g', g'') \mid \exists (g, l, g') \in R, (g, l, g'') \in R' \,\}$.
If $\langle p, \gamma \rangle \xhookrightarrow{R} \langle p', \gamma' \rangle \in \Delta_S$ and $p \xRightarrow[R']{\gamma} q$ in the current automaton,
add to $\delta_S(p', \gamma', q)$ the set $\{\, (g', l', g'') \mid \exists (g, l, g', l') \in R, (g, l, g'') \in R' \,\}$.
If $\langle p, \gamma \rangle \xhookrightarrow{R} \langle p', \gamma'\gamma'' \rangle \in \Delta_S$ and $p \xRightarrow[R']{\gamma} q$, add to $\delta_S((p', \gamma'), \gamma'', q)$
the set $\{\, ((g', l'), l'', g'') \mid \exists (g, l, g', l', l'') \in R, (g, l, g'') \in R' \,\}$.

In [5] we present an efficient implementation of these rules.

4.4 Complexity analysis

Let $\mathcal{P} = (P, \Gamma, c_0, \Delta)$ be an ordinary pushdown system, and let $\mathcal{B}$ be a Büchi automaton corresponding to the negation of an LTL formula φ. Then, according to [4], the model-checking problem for $\mathcal{P}$ and $\mathcal{B}$ can be solved in $\mathcal{O}(|P|^2 \cdot |\Delta| \cdot |\mathcal{B}|^3)$ time and $\mathcal{O}(|P| \cdot |\Delta| \cdot |\mathcal{B}|^2)$ space.

Consider a pushdown system representing a sequential program with procedures. Let n be the size of a program's control flow, i.e. the number of statements. Let m_1 be the number of global (boolean) variables, and let m_2 be the maximum number of local (boolean) variables in any procedure. Assuming that the programs use deterministic assignments to variables, each statement translates to $2^{m_1+m_2}$ different pushdown rules. Since the number of control locations is 2^{m_1}, we would get an $\mathcal{O}(n \cdot 2^{3m_1+m_2} \cdot |B|^3)$ time and $\mathcal{O}(n \cdot 2^{2m_1+m_2} \cdot |B|^2)$ space algorithm by translating the program to an ordinary pushdown system.

When we use symbolic system, the complexity gets worse. The graph SG has $\mathcal{O}(|\Delta|)$ nodes and $\mathcal{O}(|P| \cdot |\Delta|)$ edges. So our symbolic algorithm for computing the SCCs has complexity $\mathcal{O}(|P| \cdot |\Delta|^2)$. We therefore get $\mathcal{O}(n^2 \cdot 2^{3m_1+2m_2} \cdot |B|^3)$ time in the symbolic case. (The space complexity remains the same.) However, as mentioned before, the more compact representation in the symbolic case compensates for this disadvantage in the examples we tried.

5 Efficient implementation

We have implemented the algorithms of Section 4 in a model-checking tool. Three refinements with respect to the abstract description of the algorithms are essential for efficiency.

Procedure for the model-checking problem As mentioned in section 2.2, the model-checking problem reduces to (a) checking whether $c_0 \in pre^*(Rh\,\Gamma^*)$, or (b) checking whether $post^*(\{c_0\}) \cap Rh\,\Gamma^* \neq \emptyset$. In order to compute (b) symbolically, we first compute the reachable configurations (i.e., $post^*(\{c_0\})$). Then, in each symbolic rule $\langle p, \gamma\rangle \xhookrightarrow{R} \langle p', \gamma_1 \ldots \gamma_n\rangle$ we replace R by a new relation R_{reach} defined as follows: $(g, l, g', l_1, \ldots l_n) \in R_{reach}$ if $(g, l, g', l_1, \ldots l_n) \in R$ and some configuration $\langle (p, g), (\gamma, l)w\rangle$ is reachable from c_0. This dramatically reduces the efforts needed for some computations if the number of reachable variable valuations is much smaller than the number of possible valuations. In this case, much of the work in (a) would be spent on finding cycles among unreachable valuations.

Efficient computation of the repeating heads As mentioned in section 4.2, the computation of the repeating heads reduces to determining the SCCs of a graph symbolically represented as a labelled graph SG. The nodes of SG are elements of $P_0 \times \Gamma_0$, and its edges are annotated with a relation $R \subseteq (G \times L)^2$ (and a boolean). In our implementation, we first compute the components "roughly", i.e., ignoring the Rs in the edges, using Tarjan's algorithm. Then we refine the search (including the Rs) within the components. For this problem a number of different approaches could be tried. The algorithm of Section 4.2 corresponds to computing the transitive closure of the edges. The transitive closure can be computed using a stepwise computation or iterative squaring (see also [7]); the stepwise method seems to work better in general. Xie and Beerel [10] suggest a more sophisticated approach for searching the components in a symbolic setting. Moreover, these possibilities can be combined with a preprocessing of the edge relation. The preprocessing looks for BDD variables that can change their values from only 0 to 1 (or vice versa), but not in the other direction and removes such edges for such variables, effectively limiting the length of the paths in the graph.

Variable ordering It is well known that the performance of BDD-based algorithms is very sensitive to the variable ordering. When checking the Quicksort example (see below) we found that a useful variable ordering was to place the inputs (i.e. the array of data to be sorted) at the end and the 'control variables' (i.e. indices into the array) at the beginning. Our intuition for this is that every instruction changes at most two elements of the array, and that such changes can be described with small BDDs. So we need one such BDD for each of the (relatively few) possible valuations of the control variables. If the input data was placed at the beginning, the BDDs would first branch into the (relatively many) possible valuations of the input data. While it is difficult to make a general assessment of variable orderings, there is hope that this ordering would also be useful in other examples where the same division between inputs and control variables can be made. Since the inputs are stored in global variables, this criterion corresponds to placing the local variables before the global variables.

In the rest of the section we give an idea of the performance of the algorithm by applying it to some versions of Quicksort. Then we show the impact of the three improvements listed above by presenting the running times when one of

```
void quicksort (int left,int right)
{
  int lo,hi,piv;

  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi) {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi];
      lo++;
    }
  }
  quicksort(left,hi);
  quicksort(lo,right);
}
```

n	locals	time	memory
		faulty version	
3	12	0.14 s	4.6 M
4	16	0.39 s	5.3 M
5	20	1.37 s	7.2 M
6	24	6.86 s	10.5 M
7	28	53 s	12.3 M
8	32	592 s	14.6 M
9	36	$>$ 3600 s	–
		corrected version	
3	15	0.22 s	4.8 M
4	20	0.67 s	6.1 M
5	25	3.63 s	9.4 M
6	30	48.67 s	14.7 M
7	35	1238 s	15.1 M
8	40	$>$ 3600 s	–

Fig. 2. Left: Faulty version of Quicksort. Right: Results for termination check.

the improvements is switched off. All computations were carried out on an Ultrasparc 60 with 1.5 GB memory. Operations on BDDs were implemented using the CUDD package [8].

5.1 Quicksort

We intend to sort the global array `a` in ascending order; a call to the `quicksort` function in figure 2 should sort the fragment of the array starting at index `left` and ending at index `right`. The program is parametrised by two variables: n, the number of bits used to represent the integer variables, and m, the number of array entries. We are interested in two properties: first, all executions of the program should terminate, and secondly, all of them should sort the array correctly.

Termination For this property we can abstract from the actual array contents and just regard the local variables. The program in figure 2 is faulty; it is not guaranteed to terminate (finding the fault is left as an exercise to the reader). A corrected version (containing one more integer variable) is easy to obtain from the counterexample provided by our checker. Figure 2 lists some experimental results. For each n, we list the number of resulting local variables in terms of booleans. Since the array contents are abstracted away here, there are no global variables, and m does not play a rôle.

Correctness of the sorting In this case we also need to model the array contents as global variables. Figure 3 lists the results for the corrected version of the algorithm in figure 2, as well as for a variant in which the pivot element is chosen randomly.

n	m	globals	locals	normal		randomised	
				time	memory	time	memory
3	4	12	18	1 s	7.2 M	1 s	8.0 M
3	5	15	18	4 s	14.5 M	8 s	15.2 M
3	6	18	18	38 s	22.3 M	82 s	29.9 M
4	4	16	24	3 s	12.1 M	6 s	12.3 M
4	5	20	24	24 s	18.7 M	48 s	25.1 M
4	6	24	24	193 s	77.4 M	531 s	134 M
4	7	28	24	1742 s	414 M	>3600 s	–

Fig. 3. Results for correctness of sorting.

	time	memory
NONE	1.02 s	7.2 M
VORD	49 s	6.8 M
PROC	624 s	60.6 M

	with preprocessing	w/o preprocessing
closure	0.40 s	213 s
method of [10]	35 s	14 s

Fig. 4. Impact of the improvements.

Impact of the improvements Figure 4 shows the impact of the three improvements in the task of checking the correctness of Quicksort. We consider the non-randomised version with $n = 3$, and $m = 4$. The line NONE contains the reference values when all improvements are present. The lines VORD and PROC give the time and space consumption when the improvements concerning variable ordering and procedure for solving the model-checking problem are "switched off", respectively. More precisely, in the VORD line we use a BDD ordering corresponding to the order *left*, *right*, *lo*, *hi*, *piv* (i.e. all BDD variables used for representing *left* before and after a program step come before those for representing *right* etc.) plus automatic reordering. In the PROC line we compute $pre^*(Rh\,\Gamma^*)$ instead of $post^*(\{c_0\}) \cap Rh\,\Gamma^*$.

In the right part of the figure we show results for different methods of computing the repeating heads. In all cases we first computed the 'rough' components based on control flow information. We tried the transitive closure approach and the method of [10], both with and without the preprocessing described earlier. The times are for the computation of the heads only. In these experiments, the preprocessing combined with a transitive closure computation worked best, followed by the method of [10] *without* preprocessing; interestingly, using [10] combined with preprocessing led to worse results.

In this example, the times achieved by the model checker would not be possible without the symbolic representation of the variables. The translation into a normal pushdown system would create thousands or even millions of rules, and in a test we made just creating these took far longer than the model-checking with the symbolic approach.

6 Comparison with Bebop

In [1], Ball and Rajamani used the following example (see figure 5) to test their reachability checker Bebop. The example consists of one `main` function and n functions called $\texttt{level}_i$, $1 \leq i \leq n$, for some $n > 0$. There is one global variable `g`. Function `main` calls $\texttt{level}_1$ twice. Every function $\texttt{level}_i$ checks `g`; if it is true, it increments a 3-bit counter to 7, otherwise it calls $\texttt{level}_{i+1}$ twice. Before returning, $\texttt{level}_i$ negates `g`. The checker is asked to find out if the labelled statement in `main` is reachable, i.e. if `g` can end with a value of false. Since `g` is not initialised, the checker has to consider both possibilities.

```
bool g;

void main() {
  level1();
  level1();
  if (!g) {
    reach: skip;
  }
}
```

```
void leveli() {
  int (0..7) i;
  if (g) {
    i = 0;
    while (i < 7) i++;
  } else if (i < n) {
    leveli+1();
    leveli+1();
  }
  g = !g;
}
```

n	time
200	0.50 s
400	0.94 s
600	1.46 s
800	1.99 s
1000	2.41 s
2000	4.85 s
5000	13.63 s

Fig. 5. Left: The example program. Right: Experimental results.

Despite the example's simplicity, some its features are worth pointing out. There is no recursion in the program, and so its state space is finite. However, typical finite-state approaches would flatten the procedure call hierarchy, blowing up the program to an exponential size. Moreover, the program has exponentially many states, yet we can solve the reachability question in time linear in n. Finally, there are $\mathcal{O}(n)$ different variables in the program; however, only two of them are in scope at any given time. For this reason, we can keep the stack alphabet very small, exploiting the locality inherent in the program's structure.

Running times for different values of n are listed in table 5. In [1] a running time of four and a half minutes using the CUDD package and one and a half minutes with the CMU package is reported for $n = 800$, but unfortunately the paper does not say on which machine. More significant is the comparison of space consumption. We have a peak number of 155 live BDD nodes, *independent of* n. On the contrary, Bebop's space consumption for BDDs increases linearly, reaching more than 200,000 live BDD nodes for $n = 800$. The reason of this difference is that our BDDs require 4 variables (one for the global variable g and three for the 3-bit counter in scope), while Bebop's BDDs require 2401 variables (one variable for g and 2400 for the 800 3-bit counters). Since [1] does not describe the model checking algorithm in detail, we cannot say if this difference in the number of BDD variables is inherent to the algorithms or due to a suboptimal implementation.

7 Conclusions

We have presented a model-checker to verify arbitrary LTL-properties of boolean programs with (possibly recursive) procedures. To the best of our knowledge this is the first checker able to deal with liveness properties. The Bebop model checker by Ball and Rajamani, the closest to ours, can also deal with recursive boolean programs, but it can only check safety properties [1].

Our checker works on a model called symbolic pushdown systems (SPDSs). While this model is definitely more abstract than Bebop's input language, a translation of the former into the latter is simple (see Section 3).

Moreover, having SPDSs as input allows us to make use of the efficient automata-based algorithms described in [4], which leads to some efficiency advantages. In particular, the maximal number of variables in our BDDs depends only on the maximal number of local variables of the procedures, and not on the recursion depth of the program.

Another interesting feature of the reachability algorithms of our checker is that they can be used to compute the set of reachable configurations of the program, i.e. we obtain a complete description of all the reachable pairs of the form (control point, stack content). This makes them applicable to some security problems of Java programs which require precisely this feature [6]. Even more generally, we can compute the set of reachable configurations from any regular set of initial configurations.

Acknowledgements Many thanks to Ahmed Bouajjani for helpful discussions on how to obtain symbolic versions of the algorithms of [4], and to one anonymous referee for interesting comments and suggestions.

References

1. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130, 2000.
2. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. Technical report, 2001.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of CONCUR '97*, LNCS 1243, pages 135–150, 1997.
4. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV '00*, LNCS 1855, 2000.
5. J. Esparza and S. Schwoon. A BDD-based Model Checker for Recursive Programs. Technical report, Institut für Informatik, Technische Universität München, 2001. Available at http://www7.in.tum.de/gruppen/theorie/publications/.
6. T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of 1999 IEEE Symposium on Security and Privacy*, IEEE Press, 1999.

7. J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
8. F. Somenzi. Colorado University Decision Diagram Package. Technical report, University of Colorado, Boulder, 1998.
9. R. E. Tarjan. Depth first search and linear graph algorithms. In *SICOMP 1*, pages 146–160, 1972.
10. A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of ICCAD*, pages 37–40, San Jose, CA, 1999.