SAT-Based Summarization for Boolean Programs

Gérard Basler*, Daniel Kroening, and Georg Weissenbacher**

Computer Systems Institute, ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. *Boolean programs* are frequently used to model abstractions of software programs. They have the advantage that reachability properties are decidable, despite the fact that their stack is not bounded. The enabling technique is *summarization* of procedure calls. Most model checking tools for Boolean programs use BDDs to represent these summaries, allowing for efficient fix-point detection. However, BDDs are highly sensitive to the number of state variables. We present an approach to over-approximate summaries using Bounded Model Checking. Our technique is based on a SAT solver and requires only few calls to a QBF solver for fix-point detection. Our benchmarks show that our implementation is able handle a larger number of variables than BDD-based algorithms on some examples.

1 Introduction

Boolean programs [1] are frequently used to model software programs. They provide the usual control-flow constructs of an imperative language such as C, but variables are exclusively of Boolean type. The use of Boolean programs as an abstract model has been promoted by the success of the SLAM project [2]. SLAM verifies control-flow dominated properties of Windows device drivers by abstracting an ANSI-C program into a Boolean program. The abstract model of the original program is obtained by means of *Predicate Abstraction* [3]. It contains the same procedures and control flow as the original program, and thus, Boolean programs are a natural formalization. The Boolean variables are used to keep track of predicates over the variables of the original program.

The main advantage of Boolean programs over finite-state transition systems is that their stack allows a precise representation of the behavior of procedure calls, including procedure-local variables and (possibly unbounded) recursive calls. Nevertheless, reachability properties for Boolean programs are decidable [4]: Procedures can access and modify only the topmost element of the stack. Therefore, *summarizing* the procedures prevents a re-evaluation of a call if the same calling context has already been considered before [5].

* Supported by the Swiss National Science Foundation.
** Supported by Microsoft Research through its European PhD Scholarship Programme.

D. Bošnački and S. Edelkamp (Eds.): SPIN 2007, LNCS 4595, pp. 131–148, 2007.

Most existing model checkers compute summaries incrementally at each call site. The reachable states are determined by means of a saturation procedure, which computes and adds summaries until no new states are discovered. Instances of such model checkers are Bebop [1], which is shipped with Slam, and Moped [6]. Both tools use a symbolic representation of states and summaries based on (ordered) Binary Decision Diagrams (BDDs) [7]. Ordered BDDs are a canonical representation of formulas and can be compared efficiently, thus enabling the detection of previously explored portions of the state space.

Unfortunately, in case of a large number of state variables, the BDDs become unmanageably large. Approaches based on SAT-solvers (e.g. [8]) are less sensitive to the number of variables. However, they suffer from the fact that the comparison of sets of states requires a decision procedure for quantified Boolean formulas (QBF), making fix-point detection significantly harder than with BDDs.

We propose a SAT-based summarization technique that reduces the required number of QBF calls significantly. We exploit the following observation about Boolean programs generated by tools such as Slam: They are *shallow*. Formally, this means that the length of the shortest path from an initial state to any reachable valuation of global and local variables is bounded by a small constant. This bound is called *sequential depth*. Only few programs written in a general purpose programming language such as ANSI-C have this property. A single loop with an integer counter variable may result in a Kripke structure with loop-free paths as long as 2^{32}.

Obviously, a Boolean program may just as well have a sequential depth that is exponential in the number of Boolean variables it contains. However, such a program *must* encode the equivalent of a binary counter using a propositional relation over the Boolean variables. Such models are typically not generated by any of the program analysis tools that we experimented with: Tools based on predicate abstraction generate at least one predicate per loop iteration [9].

In order to avoid the expensive fix-point detection, we generate a *universal* summary, which encodes *all* possible execution traces of the procedure up to its sequential depth, starting from an *uninitialized* calling context. A universal summary is an over-approximation of the fix-point that incremental summarization would yield.

Computing the exact sequential depth is as hard as model checking. To avoid this computation, we use an over-approximation, the *reachability recurrence diameter*, that is commonly used in bounded model checking (BMC) and can be computed by means of a SAT solver.

Related Work. Boolean programs have been introduced as a formalism to represent abstract models generated by Slam [10]. The success of the Slam project motivated many researchers to work on even faster model checking algorithms for Boolean programs. The formalism is equally expressive as Pushdown systems, which have been studied long before Slam was presented: Büchi proved already in 1964 that the set of reachable states of a pushdown system (represented by a string rewriting system) can be expressed in terms of a regular language [4]. This result implies that reachability of states of pushdown systems is decidable.

Sharir and Pnueli introduce summarization as an element of an iterative fix-point detection based dataflow analysis algorithm for a language slightly more expressive than Boolean programs [5]. Ball and Rajamani's model checker Bebop is based on this work, but uses BDDs to represent states symbolically [1]. Our QBF-based summarization approach (see Section 3) is similar to the algorithm implemented in Bebop, but uses SAT instead of BDDs. Finkel et al. present an automata-based saturation algorithm that constructs the regular set representing the reachable states of a pushdown system [11]. Esparza presents an optimized version of Finkel's algorithm [6], and Schwoon improves this approach using a BDD-based symbolic representation of pushdown systems [12]. Lal and Reps present a graph-theoretic approach for model checking (weighted) pushdown systems [13]. None of the approaches listed above is based on satisfiability solving techniques.

Recently, several algorithms based on rewriting have been proposed for model checking pushdown systems: Boujjani and Esparza survey approaches that use rewriting to solve the reachability problem for sequential as well as for concurrent pushdown systems [14]. Rewriting-based reachability analysis of concurrent pushdown system is also covered by [15] and [16]. However, the reachability problem for concurrent pushdown is undecidable.Various approaches have been considered to overcome this problem (e.g., [17,18,19] and [15]). We omit a discussion of this work, since our approach does not target concurrent Boolean programs.

Leino's model checker for Boolean programs Dizzy [20] uses SAT-based symbolic simulation. However, the fix-point detection is still done by computing BDDs representing the set of reachable states.

Kroening presents a SAT-based model checker called Boppo for concurrent pushdown systems *without* recursive procedures [8]. This work comes closest to our approach, since Boppo uses a SAT-solver for symbolic simulation and a QBF-solver for fix-point detection. Procedure calls can be simulated by dynamic threads. However, in the presence of threads, Boppo computes an over-approximation of the set of reachable states. Furthermore, Boppo does not generate summaries for procedures.

Our approach is different from the model checking algorithms listed above, since we compute an over-approximation of the *summaries* of a Boolean program, instead of computing the least fix-point of this set. Still, our algorithm does not yield false positives with respect to reachability properties. To compute this over-approximation, we use Bounded Model Checking (BMC).

BMC was introduced by Biere et al. [21] as a SAT-based alternative to finite-state model checking algorithms that use Binary Decision Diagrams (BDDs) [7]. BMC searches for counterexamples of length at most length k, which is increased iteratively. The approach is complete if k exceeds the *completeness threshold* $\mathcal{CT}$ [22]: If there is no counterexample of length at most $\mathcal{CT}$, then the property in question cannot be violated at all. The *recurrence diameter*, which can be computed using a SAT solver, is an over-approximation of the completeness threshold for reachability properties of finite-state transition systems [22].

Contribution and Outline. We present background on Boolean programs and BMC in Sec. 2. In Section 3, we present a SAT-based model checking algorithm which computes a set of summary edges for each procedure and finds the least fix-point of these sets using a QBF solver. This work is based on the algorithm presented in [1]. To the best of our knowledge, our tool is the first one that implements summarization using SAT and QBF.

In Section 4, we introduce the notion of a *universal summary*, which encodes the unwinding of all possible execution traces of a procedure up to its reachability recurrence diameter. We explain how the use of universal summaries can significantly reduce the number of calls to the QBF solver.

We have implemented these algorithms for model checking Boolean programs and provide experimental results comparing this implementation to a conventional BDD-based algorithm in Section 5.

2 Background

The construction of universal summaries is based on an over-approximation of what we call the *sequential depth* of a procedure. We borrow this idea from *Bounded Model Checking*.

2.1 Bounded Model Checking

BMC is a method for finding logical errors in finite-state transition systems. It is widely regarded as a complementary technique to symbolic BDD-based model checking, and frequently used in the hardware industry; see [23] for a survey of experiments with BMC conducted in industry.

Definition 1 (Finite-State Transition System). *A* Finite-State Transition System $\mathcal{M} = \langle S, T, s_0 \rangle$ *is defined by a finite set of states* S*, a transition relation* $T \subseteq S \times S$*, and an initial state* $s_0 \in S$*.*

Given a finite-state transition system $\mathcal{M}$, an LTL property φ, and a natural number k, a BMC procedure decides whether there exists a sequence of transitions of $\mathcal{M}$ of length k or less that violates φ. SAT-based BMC is performed by generating a propositional formula, which is satisfiable if and only if such a path exists. We write $\mathcal{M} \models_k \varphi$ if all sequences of transitions up to length k satisfy φ.

In practice, the application of BMC is typically restricted to the refutation of safety properties, and is conducted in an iterative manner: Starting with a small initial value of k, k is incremented until either 1) an error is found, or 2) the problem becomes intractable due to the complexity of solving the corresponding SAT instance.

Bounded Model Checking is *complete* iff k reaches a *completeness threshold* $\mathcal{CT}$, which indicates there exists no path in $\mathcal{M}$ that violates ϕ.

Definition 2 (Completeness Threshold [22]). *A* completeness threshold *of a transition system* $\mathcal{M}$ *with respect to a property* φ *is any natural number* $\mathcal{CT}$

such that, given that the property φ is not violated by any sequence of transitions of length up to $\mathcal{CT}$, then it cannot be violated at all, i.e.,

$$\mathcal{M} \models_{\mathcal{CT}} \phi \Longrightarrow \mathcal{M} \models \phi \tag{1}$$

holds.

Clearly, if $\mathcal{M} \models \varphi$, then the smallest $\mathcal{CT}$ is 0, and otherwise it is equal to the length of the shortest counterexample. This implies that finding the smallest $\mathcal{CT}$ is at least as hard as checking $\mathcal{M} \models \varphi$. Consequently, we concentrate on computing an over-approximation of the smallest $\mathcal{CT}$.

The *Reachability Diameter* of a finite-state transition system $\mathcal{M}$ is a completeness threshold for reachability properties of the form $\mathbf{G}p$:

Definition 3 (Reachability Diameter [22]). *Given a finite-state transition system $\mathcal{M}$, the* Reachability Diameter *$rd(\mathcal{M})$ of a $\mathcal{M}$ is the minimal number of steps required for reaching all reachable states.*

The Reachability Diameter corresponds to our notion of the *sequential depth.*

Definition 4 (Reachability Recurrence Diameter [22]). *The* Reachability Recurrence Diameter *with respect to a finite state transition system $\mathcal{M} = \langle S, T, s_0 \rangle$ is the longest loop-free path in $\mathcal{M}$ starting from the initial state s_0:*

$$rrd(\mathcal{M}) \stackrel{def}{=} max\{i | \exists s_1 \ldots s_i . \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \tag{2}$$

The Reachability Diameter and the Reachability Recurrence Diameter are only defined for transition systems with a finite state space. However, Boolean programs do not adhere to this restriction, since they contain procedures with call-by-value parameter passing and recursion.

2.2 Semantics of Boolean Programs

We define Boolean programs and their semantics in terms of the control-flow graph of a program. A Boolean program consists of a set of procedures, each of which is represented by its control flow graph (CFG).

As usual, a control flow graph is a directed graph with nodes corresponding to program locations. Without loss of generality, we assume that each procedure has exactly one entry node n_i and one exit node n_e.

In accordance to [1], a state comprises of the program counter n (which is a node in the CFG) and the valuation Ω of the variables *in scope*. Unlike the conventional notion of a program state, a state in a Boolean program does not contain the content of the call stack.

Each edge $\langle n_1, n_2 \rangle$ of the CFG corresponds to a transition $\langle n_1, \Omega_1 \rangle \rightarrow \langle n_2, \Omega_2 \rangle$ that relates the values Ω_1 of the variables in scope before the transition to those (Ω_2) after the transition.

We use the following notation to describe transition functions:

- $\Omega(e)$ denotes the evaluation of the expression e according to the valuation Ω of the variables in e. Expressions and their evaluation are defined the usual way.
- We refer to the state before the execution of a transition function as *current* state, and to the state afterwards as *next* state. We use *primed* versions of the variables to distinguish variables that refer to the next state from the variables in the current state. We allow expressions to range over variables in two different states $\langle n_1, \Omega_1 \rangle$ and $\langle n_2, \Omega_2 \rangle$. $(\Omega_1, \Omega_2)(e)$ denotes the evaluation of such an expression e.
- Expressions may also contain non-deterministic choice. While non-deterministic values are traditionally represented by "$*$", we use a set of non-deterministic choice variables $\iota_1, \ldots \iota_k$ instead. We use ξ to denote a valuation to these variables, and we use $[e]_\xi$ to denote the evaluation of the expression e under the mapping ξ.

In the given setting, only the topmost element of the stack has an immediate impact on the execution of a transition. Therefore, the outcome of a call to a procedure is exclusively determined by the values of the global variables and the actual parameters at the call site. Consequently, each actual call to a procedure **pr** can be *summarized* by a pair of states $\langle n_i, \Omega_i \rangle, \langle n_o, \Omega_o \rangle$, where n_i denotes the entry node of the control flow graph of **pr**, and n_o denotes the corresponding exit node. We use $\Sigma(\mathbf{pr})$ to denote the set of these pairs for a procedure **pr**. Furthermore, we assume in this section that $\Sigma(\mathbf{pr})$ contains the entries for all reachable call contexts. Clearly, $\Sigma(\mathbf{pr})$ is finite for Boolean programs.

For a given entry state, the corresponding exit states[1] are determined by the transition functions of the control flow graph of **pr**. The transition functions are in turn determined by the statements corresponding to the nodes of the control flow graph. We distinguish the following statements:

- The `skip` statement does not modify the variables, but increments the program counter by one.
- The `goto` $\ell_1, \ldots, \ell_m$ statement non-deterministically changes the program counter to one of the program locations $\ell_1, \ldots, \ell_m$ provided as argument. The valuation of the variables does not change.
- The `assume` e statement increases the program counter by one iff the condition e evaluates to `true` in the current state. Otherwise, the `assume` statement has no successor states, i.e., the program terminates.
- The constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e assigns the values of the expressions $e_1, \ldots, e_k$ to the variables $x_1, \ldots, x_k$. The expressions are evaluated in the current state and may contain a non-deterministic choice variables. The constraint e is a predicate that ranges over the variables of the current *and* the next state. It is evaluated in both states, and the statement has no successor state if e does not hold.
- The `return` statement corresponds to the exit node of the control flow graph of **pr**. Whenever it is reached, the current state determines the exit valuation

[1] The use of non-determinism may result in more than one exit valuation.

of the corresponding summary. We assume without loss of generality that all return values are passed to the caller via global variables, i.e., `return` has no parameters. Therefore, the variables are not modified. The program counter of the successor statement is determined by the *caller* of the corresponding procedure **pr**.

– The call $\mathbf{pr}(e_1, \ldots, e_k)$ modifies the global variables according to an *applicable summary* $\langle n_i, \Omega_i \rangle, \langle n_o, \Omega_o \rangle$ in $\Sigma(\mathbf{pr})$. A summary is applicable if a) Ω_i agrees with the current state on the global variables, and b) the evaluation of $e_1, \ldots, e_k$ matches the corresponding actual parameters in Ω_i. (The calling context determines the entry valuation of a summary.)

  Then, the call to $\mathbf{pr}(e_1, \ldots, e_k)$ modifies the global variables according to Ω_o. If more than one summary is applicable, one summary is chosen non-deterministically (analogously to the `goto` statement).

  In the case that an applicable summary exists, the call sets the program counter to the statement that succeeds the call. Otherwise, the statement succeeding the call is never reached.

The statements `skip`, `assume`, the constrained assignment, and procedure calls have a single successor node in the control flow graph (according to the structure of the program). The `return` statement has no successor in the control flow graph, since the program location that succeeds a return statement cannot be determined statically. Goto statements may have more than one successor. Conditional statements like `if-then-else` or `while` loops can be modeled using a combination of the `goto` and `assume` statements.

The recursive nature with respect to $\Sigma(\mathbf{pr})$ of the definition of the semantics indicates that the set of summaries $\Sigma(\mathbf{pr})$ of a Boolean program can be obtained by means of a fix-point computation. Several algorithms that compute the least fix-point of the set $\Sigma(\mathbf{pr})$ in order to determine the set of reachable states have been proposed [1,11,12].

We present a QBF-based algorithm to compute the least fix-point for $\Sigma(\mathbf{pr})$ in the following section, and propose to use a SAT-based over-approximation of this fix-point in Section 4.

3 Summarization Using QBF

In this section we describe how we compute the least fix-point of $\Sigma(\mathbf{pr})$ using forward symbolic execution and QBF-based fix-point detection.

The valuation of a symbolic state is represented in terms of a Boolean formula over non-deterministic choice variables $\iota_1, \ldots, \iota_k$, i.e., we use a parametric representation. Boolean formulas are defined the usual way.

Let N be the set of nodes in a CFG of a Boolean program, let V be the variables of that program.

Definition 5 (Symbolic State). *A* symbolic state *is a triple* $\langle n, \gamma, \omega \rangle$*, where* $n \in N$ *identifies the node in the CFG and is represented explicitly, and* γ *is a Boolean formula over the non-deterministic choice variables and represents the*

Table 1. Conditions on the symbolic transitions $\langle n_1, \gamma_1, \omega_1 \rangle \rightarrow \langle n_2, \gamma_2, \omega_2 \rangle$ for the statements `skip`, `goto`, `assume`, and the constrained assignment

Instruction	γ_2	ω_2
`skip`	$\gamma_2 = \gamma_1$	$\omega_2 = \omega_1$
`return`	$\gamma_2 = \gamma_1$	$\omega_2 = \omega_1$
`goto` $\ell_1, \ldots \ell_k$	$\gamma_2 = \gamma_1$	$\omega_2 = \omega_1$
`assume` e	$\gamma_2 = (\gamma_1 \wedge \omega_1(e))$	$\omega_2 = \omega_1$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$\gamma_2 = (\gamma_1 \wedge (\omega_1, \omega_2)(e))$	$\omega_2 = (\omega_1[x_1/\omega_1(e_1)] \ldots [x_k/\omega_1(e_k)])$

guard *of the state. The component* ω *maps the variables* V *to formulas over* $\iota_1, \ldots, \iota_k$*, representing a set of valuations for the variables in the symbolic state.*

Each symbolic state $\langle n, \gamma, \omega \rangle$ represents the set of explicit states

$$\{\langle n, \Omega \rangle | \exists \xi . [\gamma]_\xi \wedge \forall v \in V . \Omega(v) = [\omega(v)]_\xi\}$$

where $\omega(e)$ denotes the evaluation of the expression e according to the mapping ω (analogously to $\Omega(e)$ in Section 2). The symbolic state $\langle \ell_{10}, (\iota_1 \vee \iota_2), \{a \mapsto \iota_1, b \mapsto (\neg \iota_1 \wedge \iota_2)\} \rangle$, for instance, represents the explicit states $\langle \ell_{10}, \{a \mapsto 0, b \mapsto 1\} \rangle$ and $\langle \ell_{10}, \{a \mapsto 1, b \mapsto 0\} \rangle$. The valuation $\langle \iota_1, \iota_2 \rangle = \langle 0, 0 \rangle$ is ruled out by the guard, and the valuations $\langle \iota_1, \iota_2 \rangle = \langle 1, 0 \rangle$ and $\langle \iota_1, \iota_2 \rangle = \langle 1, 1 \rangle$ yield the same explicit state. An unsatisfiable guard indicates that there is no concrete state represented by $\langle n, \gamma, \omega \rangle$ [8].

Before we proceed to introduce our symbolic representation of summaries, we define the transition conditions for the statements `skip`, `goto`, `assume`, and the constrained assignment. In Table 1, we write $\omega[x/e]$ for the mapping that maps x to the formula e, while it agrees with the mapping ω on all other variables. We use γ_1 and ω_1 to refer to the components γ and ω of the current state, and γ_2 and ω_2 to refer to the next state. The program locations are omitted, since they change according to the rules presented in Section 2. The conditions in Table 1 are equivalent to those presented in [8] (except for the `return` statement). According to this table, the components γ and ω are modified as follows:

- In case of a `skip`, `return`, or `goto` statement, γ as well as ω do not change.
- Conditions contributed by `assume` statements are instantiated according to ω_1 and conjoined with the guard γ_1. The symbolic execution terminates if γ_2 is unsatisfiable.
- A constrained assignment updates the mapping ω_2 accordingly. If a constraining condition is present, it is instantiated using ω_1 *and* ω_2, and conjoined with γ_1.

An actual symbolic transition can be characterized by a pair of symbolic states $\langle n_1, \gamma_1, \omega_1 \rangle, \langle n_2, \gamma_2, \omega_2 \rangle$. The first state represents the concrete set of states *before* the transition, and the second the corresponding concrete states *afterwards*. By

construction (see Table 1), the components $\gamma_1, \gamma_2, \omega_1$, and ω_2 *share* sub-formulas. Therefore, given one of the concrete states $\langle n_1, \Omega_1\rangle \in \langle n_1, \gamma_1, \omega_1\rangle$, we can obtain the states that are reachable from $\langle n_1, \Omega_1\rangle$ via this transition by *constraining* the state $\langle n_2, \gamma_2, \omega_2\rangle$:

$$\langle n_2, \quad \left(\gamma_2 \wedge \bigwedge_{v \in V} \omega_1(v) = \Omega_1(v)\right), \quad \omega_2\rangle \tag{3}$$

Continuing our example, we construct the symbolic successor for the state $\langle \ell_{10}, (\iota_1 \vee \iota_2), \{a \mapsto \iota_1, b \mapsto (\neg\iota_1 \wedge \iota_2)\}\rangle$ and the following statement:

$$a := \neg a \ \texttt{constrain}\ (\neg b \vee a')$$

According to Table 1, we obtain $\omega_2 = \{a \mapsto \neg\iota_1, b \mapsto (\neg\iota_1 \wedge \iota_2)\}$ and $\gamma_2 = (\iota_1 \vee \iota_2) \wedge (\iota_1 \vee \neg\iota_2 \vee \neg\iota_1) \equiv (\iota_1 \vee \iota_2)$. We compute the successor states for the concrete state $\langle \ell_{10}, \{a \mapsto 0, b \mapsto 1\}\rangle$ by adding the constraint $\iota_1 = 0 \wedge (\neg\iota_1 \wedge \iota_2) = 1$ to γ_2, ruling out the successor state $\langle \ell_{11}, \{a \mapsto 0, b \mapsto 1\}\rangle$ and leaving us with $\langle \ell_{11}, \{a \mapsto 1, b \mapsto 0\}\rangle$.

If we constrain the transition $\langle n_1, \gamma_1, \omega_1\rangle, \langle n_2, \gamma_2, \omega_2\rangle$ with a *symbolic* state $\langle n_1, \gamma_0, \omega_0\rangle$ (analogously to Equation 3), we obtain a symbolic state that represents the set of states reachable from $\langle n_1, \gamma_0, \omega_0\rangle$ via this transition:

$$\langle n_2, \left(\gamma_2 \wedge \gamma_0 \wedge \bigwedge_{v \in V} \omega_1(v) = \omega_0(v)\right), \omega_2\rangle \tag{4}$$

In (4), we assume that the non-deterministic choice variables in $\langle n_1, \gamma_0, \omega_0\rangle$ differ from those in $\langle n_1, \gamma_1, \omega_1\rangle$ and $\langle n_2, \gamma_2, \omega_2\rangle$. This can always be achieved by renaming.

The representation of transitions by means of two symbolic states is not restricted to single transitions, but can be extended to sequences of transitions in the natural way. This representation enables the summarization of compound transitions, and is similar to the concept of *path edges* [1].

Definition 6 (Path edges and summary edges). *A pair of symbolic states* $\langle n_i, \gamma_i, \omega_i\rangle, \langle n_o, \gamma_o, \omega_o\rangle$ *is a* path edge *of procedure* **pr** *iff all of the following hold:*

- n_i *is the entry node of* **pr**.
- $\langle n_i, \gamma_i, \omega_i\rangle$ *is reachable from an initial state of the Boolean program.*
- $\langle n_o, \gamma_o, \omega_o\rangle$ *is reachable from* $\langle n_i, \gamma_i, \omega_i\rangle$ *by a sequence of statements that does* not *contain the* `return` *statement of* **pr** *(unless* n_o *happens to be the corresponding exit node).*

A summary edge *of* **pr** *is a path edge* $\langle n_i, \gamma_i, \omega_i\rangle, \langle n_o, \gamma_o, \omega_o\rangle$*, for which* n_i *corresponds to the entry node, and* n_o *corresponds to the exit node of* **pr**.

Using Definition 6 and Equation 4, we can give a symbolic transition function for the procedure call statement. Assume that we encounter a procedure call

$\mathbf{pr}(e_1, \ldots, e_k)$ and the current state is $\langle n_c, \gamma_c, \omega_c \rangle$. Let $\Sigma_s(\mathbf{pr})$ be the set of symbolic summaries for the procedure $\mathbf{pr}$. We use $g_1, \ldots, g_m$ to denote the global variables of the Boolean program, $l_1, \ldots, l_j$ to denote the local variables of the calling context, and $f_1, \ldots, f_k$ to denote the formal parameters of the procedure $\mathbf{pr}$.

A summary edge $\langle n_i, \gamma_i, \omega_i \rangle, \langle n_o, \gamma_o, \omega_o \rangle \in \Sigma_s(\mathbf{pr})$ is applicable to the calling context $\langle n_c, \gamma_c, \omega_c \rangle$ iff

1. for all reachable valuations of $g_1, \ldots, g_m, e_1, \ldots, e_k$ in $\langle n_c, \gamma_c, \omega_c \rangle$ there exists a matching valuation to $g_1, \ldots, g_m, f_1, \ldots, f_k$ in $\langle n_i, \gamma_i, \omega_i \rangle$, and
2. γ_o is still satisfiable when the global and formal variables are restricted to the global variables and parameter expressions of γ_c and ω_c according to Equation 4.

The universal quantification in the first condition requires us to use a QBF instance to decide applicability:

$$\forall \xi_c.\ [\gamma_c]_{\xi_c} \Rightarrow \exists \xi_o.\ \bigwedge_{s \in \{1..m\}} [w_i(g_s)]_{\xi_o} = [w_c(g_s)]_{\xi_c} \wedge \bigwedge_{t \in \{1..k\}} [w_i(f_t)]_{\xi_o} = [w_c(e_t)]_{\xi_c} \ \wedge\ [\gamma_o]_{\xi_o} \tag{5}$$

Again, we assume that the non-deterministic choice variables in $\langle n_c, \gamma_c, \omega_c \rangle$ are disjoint from those in the summary edge.

Assuming that (5) holds, we can restrict the symbolic state $\langle n_o, \gamma_o, \omega_o \rangle$ to the states reachable from the calling context $\langle n_c, \gamma_c, \omega_c \rangle$ analogously to (4). By applying the summary, we obtain a new symbolic state $\langle n_r, \gamma_r, \omega_r \rangle$ (the state after the `return` statement) with

$$\gamma_r := \gamma_o \wedge \gamma_c \wedge \bigwedge_{s \in \{1..m\}} w_i(g_s) = w_c(g_s) \ \wedge \bigwedge_{t \in \{1..k\}} w_i(f_t) = w_c(e_t) \tag{6}$$

and

$$\omega_r(g_1) = \omega_o(g_1), \ldots, \omega_r(g_m) = \omega_o(g_m), \omega_r(l_1) = \omega_c(l_1), \ldots, \omega_r(l_j) = \omega_c(l_j) \tag{7}$$

Now consider the case that (5) does not hold, i.e., $\Sigma_s(\mathbf{pr})$ contains no applicable summary. In that case, a new summary edge must to be computed for the calling context $\langle n_c, \gamma_c, \omega_c \rangle$. For this purpose, we construct a new symbolic state $\langle n_i, \gamma_i, \omega_i \rangle$ which agrees with $\langle n_c, \gamma_c, \omega_c \rangle$ on the global variables, and assign $e_1, \ldots, e_k$ to the formal parameters (using same transition function as the assignment statement). The symbolic state $\langle n_i, \gamma_i, \omega_i \rangle$ serves as entry node for a new path edge, and may eventually yield a new summary edge.

Fix-point Detection. In order to determine the least fix-point of $\Sigma_s(\mathbf{pr})$, our reachability checking performs symbolic simulation of a Boolean program. The algorithm maintains a set $\mathcal{P}$ of path edges and summary edges $\Sigma_s(\mathbf{pr})$ that

```
 1: procedure INSERT(π)
 2:     if π ⋢ 𝒫 then
 3:         insert π into 𝒫
 4:         insert π into 𝒲
 5:     end if
 6: end procedure

 7: Initialize 𝒫 to ∅;
 8: for all pr do Initialize Σ_s(pr) to ∅;
 9: end for
10: 𝒲 := {⟨n_0, true, ω_*⟩, ⟨n_0, true, ω_*⟩};        ▷ ⟨n_0, true, ω_*⟩ is initial state
11: 𝒲' := ∅;
12: while 𝒲 ≠ ∅ do
13:     remove π = ⟨n_i, γ_i, ω_i⟩, ⟨n_o, γ_o, ω_o⟩ from 𝒲
14:     if statement of n_o is skip, assume, or assignment then
15:         π' :=TRANS(π, n_o);
16:         INSERT(π');
17:     else if statement of n_o is goto ℓ_1, ..., ℓ_k then
18:         π_{ℓ_1}, ..., π_{ℓ_k} :=TRANS(π, n_o);                    ▷ split path edge
19:         for all t ∈ {1..k} do
20:             INSERT(π_{ℓ_t});
21:         end for
22:     else if statement of n_o is call pr(e_1, ..., e_k) then
23:         for all σ_pr ∈ Σ(pr) do
24:             if APPLICABLE(π, σ_pr) then
25:                 π' :=APPLY(π, σ_pr);
26:                 INSERT(π');
27:             end if
28:         end for
29:         if {σ_pr ∈ Σ(pr) | APPLICABLE(π, σ_pr)} = ∅ then
30:             construct entry state ⟨n_i, γ_i, ω_i⟩ for pr;
31:             π' := {⟨n_i, γ_i, ω_i⟩, ⟨n_i, γ_i, ω_i⟩};
32:             INSERT(π');
33:             insert π into 𝒲'                              ▷ postpone expansion
34:         end if
35:     else if statement of n_o is return then
36:         if π ⋢ Σ_s(pr of n_i) then
37:             σ := π;                          ▷ encountered new summary edge
38:             insert σ into Σ_s(pr);
39:             for all π_c ∈ 𝒲' s.t. APPLICABLE(π_c, σ) do
40:                 π' :=APPLY(π_c, σ);          ▷ perform postponed expansion
41:                 INSERT(π');
42:             end for
43:         end if
44:     end if
45: end while
```

Fig. 1. The SAT based model checking algorithm

have been constructed so far. Our algorithm is similar to the BDD-based model checking algorithm presented in [1]. However, unlike a BDD-based representation of path edges, our representation is not canonical. The price we pay for being able to apply transition functions efficiently is that we need to solve a QBF instance in order to determine whether a path edge is already an element of $\mathcal{P}$.

Let V be the variables of the procedure **pr**. Given two path edges $\langle n_i, \gamma_{i1}, \omega_{i1}\rangle$, $\langle n_o, \gamma_{o1}, \omega_{o1}\rangle$ and $\langle n_i, \gamma_{i2}, \omega_{i2}\rangle, \langle n_o, \gamma_{o2}, \omega_{o2}\rangle$, the latter is *at least as general as* the former iff

$$\forall \xi_1 .\ [\gamma_{o1}]_{\xi_1} \Rightarrow \exists \xi_2 .\ [\gamma_{o2}]_{\xi_2} \wedge \bigwedge_{v \in V} [\omega_{i1}(v)]_{\xi_1} = [\omega_{i2}(v)]_{\xi_2} \wedge [\omega_{o1}(v)]_{\xi_1} = [\omega_{o2}(v)]_{\xi_2} \tag{8}$$

where ξ_1 refers to the non-deterministic choice variables of the first path edge, and ξ_2 to those of the second. Equation 8 holds iff the set of pairs of concrete states represented by the first path edge is a subset of the corresponding set represented by the second. In that case, a further expansion of the path edge $\langle n_i, \gamma_{i1}, \omega_{i1}\rangle, \langle n_o, \gamma_{o1}, \omega_{o1}\rangle$ does not yield any states that are not discovered by expanding the more general path edge.

The pseudo code of our QBF-based algorithm is presented in Figure 1. It resembles the model checking algorithm presented in [1], but uses SAT and QBF instead of BDDs. In line 10, we use ω_* to indicate that the state is initialized non-deterministically.

We use APPLICABLE(π, σ) to denote the condition in Equation 5, where $\pi = \langle n_e, \gamma_e, \omega_e\rangle, \langle n_c, \gamma_c, \omega_c\rangle$ is a path edge that provides the calling context, and σ is a summary edge. Furthermore, APPLY(π, σ) denotes the path edge that we obtain by applying the summary according to equations (6) and (7). The condition in Equation 8 is expressed by $\pi_1 \subseteq \pi_2$ and holds if the path edge π_1 is subsumed by π_2. Finally, we use TRANS(π, n) to denote the application of the transition function of a node n as listed in Table 1.

The algorithm maintains a work-list $\mathcal{W}$ in which all path-edges that are currently explored are stored. Each path edge of this work-list is expanded according to the transition functions described above, until either the guard becomes unsatisfiable or the resulting path edge is already in $\mathcal{P}$. For convenience, we define a procedure `insert`(π), which we use to insert a path edge into the work-list, unless it is already contained in $\mathcal{P}$.

In line 14, the transition functions presented in Table 1 are applied. Whenever the algorithm encounters a `goto` statement, the current path edge is split (see line 17).

Procedure calls are handled in line 22. Matching summary edges are applied immediately. However, if there is no applicable summary edge, we construct an entry state for the called procedure and add a corresponding path edge to the work-list $\mathcal{W}$. Furthermore, we store the current path edge σ in $\mathcal{W}'$, which is examined whenever we add a new summary to $\Sigma(\mathbf{pr})$. Thus, we guarantee that any summary of **pr** that is eventually generated is applied to also σ (see line 39).

Path merging. Splitting the path edge in line 17 of our algorithm in Figure 1 may lead to an explosion of the of the number of path edges. Therefore, we *merge* path edges in our work-list $\mathcal{W}$ whenever possible. Two path edges $\langle n_{i1}, \gamma_{i1}, \omega_{i1}\rangle, \langle n_{o1}, \gamma_{o1}, \omega_{o1}\rangle$ and $\langle n_{i2}, \gamma_{i2}, \omega_{i2}\rangle, \langle n_{o2}, \gamma_{o2}, \omega_{o2}\rangle$ can be merged if $\langle n_{i1}, n_{o1}\rangle$ and $\langle n_{i2}, n_{o2}\rangle$ coincide. In that case, we construct a new path edge $\langle n_{i1}, \gamma_i, \omega_i\rangle, \langle n_o 1, \gamma_o, \omega_o\rangle$ such following conditions hold for γ_i and ω_i:

- $\gamma_i = \big(\gamma_{i1} \wedge \bigwedge_{v \in V} \omega_i(v) = \omega_{i1}(v)\big) \vee \big(\gamma_{i2} \wedge \bigwedge_{v \in V} \omega_i(v) = \omega_{i2}(v)\big)$
- for all $v \in V$, $\omega_i(v) = \omega_{i1}(v) \vee \omega_{i2}(v)$

Analogously, we construct similar conditions for γ_o and ω_o and name the procedure that merges a set of path edges Π MERGE(Π). We deploy a heuristic that postpones the application of certain transitions (e.g. at join nodes in the CFG) in order to increase the number of path edges in $\mathcal{W}$ that can be merged. A similar approach is used by the model checker DIZZY [20], but not in combination with summarization.

4 Universal Summaries

Solving the QBF instances is the primary performance bottleneck of the algorithm presented in the previous section. The majority of the QBF instances is generated by the fix-point detection algorithm (see Equation 8). These QBF instances cannot be avoided if we want to compute the least fix-point of $\Sigma_s(\mathbf{pr})$. However, if we settle for an over-approximation of this fix-point, we can reduce the number of calls to the QBF solver significantly.

The set of path edges consisting of *all* sequences of transitions of a procedure **pr** up to its reachability recurrence diameter (see Def. 3) is such an over-approximation. The definition of the reachability recurrence diameter requires adaption to be applicable to procedures of Boolean programs:

- The reachability recurrence diameter is only defined for finite state system and therefore not applicable to recursive procedure calls.
- The transition function T in Definition 1 is a compound, synchronous transition function that modifies all state variables in each step. A Boolean program is a disjunctive partitioning of *local* transitions, and the advantage of locality is lost if we treat it as a compound transition function.

We address these issues as follows:

- We replace the procedure calls in **pr** by a non-deterministic assignment to all global variables that are potentially changed by the callee. The set of these variables can be obtained by static analysis. The resulting procedure **pr*** is an over-approximation of the behavior of **pr**.
- We still split the path edges, but perform aggressive merging, i.e., we merge at every join node in the CFG. Thus, instead of unwinding the entire compound transition function, each cycle in the CFG is unwound separately. This corresponds to the loop unrolling algorithm used in the CBMC tool [24].

```
 1: procedure UNROLL(pr)
 2:     W := {⟨n_i, true, ω_*⟩, ⟨n_i, true, ω_*⟩};     ▷ ⟨n_i, true, ω_*⟩ is initial state
 3:     for all nodes n ∈ CFG(pr) do
 4:         P(n) := ∅;
 5:     end for
 6:     assign priorities no nodes: the closer to a return statement, the lower;
 7:     while W ≠ ∅ do
 8:         choose n_o with highest priority s.t. ∃⟨n_i, γ_i, ω_i⟩, ⟨n_o, γ_o, ω_o⟩ ∈ W;
 9:         W' := {⟨n'_i, γ'_i, ω'_i⟩, ⟨n'_o, γ'_o, ω'_o⟩ ∈ W | n'_o = n_o};
10:         W := W \ W';
11:         π' := MERGE(W');
12:         EXPAND(π', n_o);                    ▷ expands π' and adds result to W
13:     end while
14:     assert (statement of n_o is return);
15:     return π';
16: end procedure
```

Fig. 2. Expanding and merging path edges at every join node

The algorithm in Figure 2 performs aggressive merging by making sure that no path edge can proceed beyond a *join node* unless all other path edges in the work list have "caught up". We achieve this by assigning a priority to each node in the control flow graph. The priority depends on the distance to the exit node: Nodes closer to the `return` statement have a lower priority, and the exit node itself has the lowest priority.

Note that this algorithm fails to proceed beyond a join node of a cycle of the CFG if the procedure EXPAND (see call in line 12) perpetually generates new path edges for this cycle. We prevent this by restricting the path edges generated by EXPAND to the reachability recurrence diameter of **pr**:

Definition 7 (Reachability Recurrence Diameter of procedure). *The* reachability recurrence diameter *rrd*(**pr**) *of a Boolean procedure* **pr** *is the* longest *sequence of concrete transitions in* **pr*** *such that no state* $\langle n, \Omega \rangle$ *of the procedure is visited twice.*

Given a set of symbolic path edges $\langle n_i, \gamma_i, \omega_i \rangle, \langle n_o, \gamma_{o,j}, \omega_{o,j} \rangle$, $j \in \{1..n\}$ for a node n_o in procedure **pr***, we can construct a formula (similar to Equation 2) that is satisfiable iff there are n distinct states such that each path edge represents one of them:

$$\exists \xi. \left[\bigwedge_{j=0}^{n-1} \bigwedge_{k=j+1}^{n} \gamma_{o,j} \wedge \gamma_{o,k} \wedge \bigvee_{v \in V} \omega_{o,j}(v) \neq \omega_{o,k}(v) \right]_{\xi} \tag{9}$$

If we keep a record of all path edges that reach a node n (using a set $\mathcal{P}(n)$), we can use Equation 9 to check whether there still exists a sequence of concrete transitions that visits no state at n twice. We use RECURRING($\mathcal{P}(n)$) to denote Equation 9.

```
 1: procedure INSERT(π)
 2:     if ¬RECURRING(P(n_o) ∪ {π}) then
 3:         P(n_o) := P(n_o) ∪ {π};
 4:         W := W ∪ {π};
 5:     end if
 6: end procedure
 7: procedure EXPAND(π, n_o)
 8:     if statement of n_o is skip, assume, or assignment then
 9:         π' :=TRANS(π, n_o);
10:         INSERT(π');
11:     else if statement of n_o is goto ℓ_1, ..., ℓ_k then
12:         π_{ℓ_1}, ..., π_{ℓ_k} := TRANS(π, n_o);
13:         for all ℓ ∈ {ℓ_1, ... ℓ_k} do
14:             INSERT(π_ℓ);
15:         end for
16:     else if statement of n_o is a call pr'(e_1, ..., e_k) then
17:         let G ⊆ {g_1, ..., g_m} be globals changed in pr';
18:         π' := π with G assigned non-deterministically;
19:         INSERT(π');
20:     end if
21: end procedure
```

Fig. 3. Unrolling transitions of a Boolean procedure until states repeat

The pseudo-code for the procedure EXPAND is presented in Figure 3. The algorithm checks at each program location whether the reachability recurrence diameter is exceeded. The path edge of the current step is added to $\mathcal{P}(n)$ unless RECURRING$(\mathcal{P} \cup \{\pi\})$ holds (we implicitly assume that path edges with unsatisfiable guards are dropped, too).

Whenever the algorithm encounters a procedure call **pr'**$(\ldots)$, it replaces the transition by a non-deterministic assignment to the globals that are potentially changed by **pr'** (see line 18). Finally, we return the entirely merged path edge π that reaches the exit node of the CFG.

Again, we use $\langle n_i, \texttt{true}, \omega_* \rangle$ to denote the non-deterministically initialized state (see line 2 in Figure 2), i.e., we do not restrict the calling context. Therefore, our algorithm generates a path edge that contains *all* sequences of transitions that do not visit a state twice for an arbitrary calling context.

We use this observation to justify our initial claim: The path edge π obtained by computing UNROLL(**pr**) over-approximates Σ_s(**pr**). We call this π a universal summary.

Definition 8 (Universal summary). *The* universal summary Σ_u(**pr**) *of a procedure* **pr** *is the path edge that we obtain by merging the path edges of all sequences of transitions (the initial state being an unconstrained calling context) of* **pr*** *up to the reachability recurrence diameter of* **pr***, *i.e.,* $\Sigma_u(\mathbf{pr}) :=$ UNROLL(**pr**)*.*

Table 2. Comparison of performance of Bebop, QBF-based summarization, and universal summaries (timout: $> 2h$)

Benchmark	#vars	Bebop	QBF-summaries	univ. summ.	violation
Slam adddevice	434	4m37.4s	0m0.6s	0m1.8s	yes
Slam nulldevice	434	4m34.0s	0m8.6s	0m1.4s	yes
Slam pendedcompletedreq	86	0m30.9s	timeout	0m13.5s	yes
Slam targetrelationneedsref	37	0m0.4s	0m0.5s	0m2.74s	no
Slam markirppending	11	0m0.4s	0m3.0s	0m18.5s	no
Slam wmiforward	15	0m0.7s	0m2.0s	0m15.3s	no
Terminator 1	74	timeout	1m55.9s	1m55.9s	yes
Terminator 2	60	88m22.6s	timeout	timeout	yes

The universal summary of **pr** does not only over-approximate $\Sigma_s(\mathbf{pr})$, but also the set of feasible transition sequences. These spurious execution traces are eliminated when $\Sigma_u(\mathbf{pr})$ is applied at all call sites of **pr** according to equations (6) and (7).

However, in case of a cyclic dependency between procedures (i.e., in case of recursion or mutual recursion) the universal summaries cannot be applied. Therefore, we compute the universal summaries of all non-recursive procedures of a Boolean program, and use the algorithm in Figure 1 to handle the remaining recursive procedure calls.

5 Benchmarks

We used Slam [10] and Terminator [25] to generate eight Boolean programs from Windows device drivers. We compare the performance of our implementation on these examples to the model checker Bebop, which is part of Microsoft Research's Slam/SDV toolkit [10] (see Table 2). In addition, we compare the effect of summarization with universal summaries and entirely QBF-based summarization. The column labeled "violation" indicates whether the reachability property we are checking for can be violated or not.

The algorithm that uses universal summaries deploys a heuristic that switches back to QBF-summaries for procedures with universal summaries that are larger than a certain threshold. This typically happens if the procedure contains a lot of non-deterministic assignments.

The benchmarks are incoherent, and we have yet to investigate why this is the case. In some situations (like the `adddevice` benchmark with 434 variables), our implementation is significantly faster than Bebop. However, this cannot be generalized. Furthermore, in some cases it turns out to be disadvantageous to use universal summaries (see, for instance, the **markirppending** benchmark). The QBF-instances resulting from the combination of universal summaries and QBF-based summarization may become too large for the solver Skizzo [26]. In the Terminator 2 benchmark, this also happens without universal summaries.

We have a large number of small regression tests that indicate that BDD-based implementations are still faster for Boolean programs with a small number of variables. However, the reason for this may be that we did not profile and optimize our implementation, yet. We intend to make an updated set of benchmarks available as soon as we are able to explain the performance problems on small examples.

6 Conclusion

We present a SAT based model checking algorithm for Boolean programs that uses a QBF solver to compute the least fix-point of the set of summary edges of a procedure. Furthermore, we introduce the concept of *universal summaries*, an over-approximation of the summary edges our initial algorithm computes. By using universal summaries, we reduce the number of calls to the QBF solver significantly.

Our preliminary benchmarks do not allow us to conclude that our approach is in general superior to BDD based model checking. However, some of the results are very promising and indicate that it is worthwhile to further pursue the idea.

Acknowledgements. We would like to thank Vijay D'Silva, Angelo Brillout, and our anonymous reviewers for their valuable comments on this paper.

References

1. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN Model Checking and Software Verification. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
2. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
3. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
4. Büchi, J.R.: Regular canonical systems. Archive for Mathematical Logic 6, 91 (1964)
5. Sharir, M., Pnueli, A.: Two approaches to interprocedural data dalow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
6. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35, 677–691 (1986)
8. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous Boolean programs. In: Godefroid, P. (ed.) Model Checking Software. LNCS, vol. 3639, pp. 75–90. Springer, Heidelberg (2005)
9. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)

10. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, Springer, Heidelberg (2004)
11. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. ENTCS 9 (1997)
12. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
13. Lal, A., Reps, T.: Improving pushdown system model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
14. Bouajjani, A., Esparza, J.: Rewriting models of Boolean programs. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, Springer, Heidelberg (2006)
15. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: Principles of Programming Languages (POPL), pp. 62–73. ACM Press, New York (2003)
16. Bouajjani, A., Touili, T.: On computing reachability sets of process rewrite systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 484–499. Springer, Heidelberg (2005)
17. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–103. Springer, Heidelberg (2005)
18. Touili, T., Sighireanu, M.: Bounded communication reachability analysis of bounded communication reachability analysis of process rewrite systems with ordered parallelism. In: Verification of Infinite State Systems (INFINITY), Elsevier, Amsterdam (2007)
19. Cook, B., Kroening, D., Sharygina, N.: Over-approximating Boolean programs with unbounded thread creation. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 53–59. IEEE Computer Society Press, Los Alamitos (2006)
20. Leino, K.R.M.: A SAT characterization of Boolean-program correctness. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. LNCS, vol. 2648, pp. 104–120. Springer, Heidelberg (2003)
21. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
22. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 298–309. Springer, Heidelberg (2003)
23. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
24. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
25. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
26. Benedetti, M.: Evaluating QBFs via symbolic skolemization. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 285–300. Springer, Heidelberg (2005)