Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 388 (2007) 227–242
Theoretical Computer Science
www.elsevier.com/locate/tcs

Verification of Boolean programs with unbounded thread creation[☆]

Byron Cook[a], Daniel Kroening[b], Natasha Sharygina[c,*]

[a] *Microsoft Research, United Kingdom*
[b] *Computing Laboratory, Oxford University, United Kingdom*
[c] *Informatics Department, University of Lugano, Switzerland*

Received 30 October 2006; received in revised form 31 January 2007; accepted 6 July 2007

Communicated by M. Bonsangue and F. de Boer

Abstract

Most symbolic software model checkers use abstraction techniques to reduce the verification of infinite-state programs to that of decidable classes. *Boolean programs* [T. Ball, S.K. Rajamani, Bebop: A symbolic model checker for Boolean programs, in: SPIN 00, in: Lecture Notes in Computer Science, vol. 1885, Springer, 2000, pp. 113–130] are the most popular representation for these abstractions. Unfortunately, today's symbolic software model checkers are limited to the analysis of sequential programs due to the fact that reachability in Boolean programs with unbounded thread creation is undecidable. We address this limitation with a novel algorithm for over-approximating reachability in Boolean programs with unbounded thread creation. Although the Boolean programs are not of finite state, the algorithm always reaches a fix-point. The fixed points are detected by projecting the state of the threads to the globally visible parts, which are finite.
© 2007 Elsevier B.V. All rights reserved.

MSC: 68N30; 68Q55; 68Q60

Keywords: Program verification; Model checking; Boolean programs; Reachability

1. Introduction

All of today's symbolic model checkers for programs use some form of abstraction technique to reduce the verification of infinite-state programs to that of finite-state programs. The infinity in programs comes from three sources: unbounded arithmetic, heap-based data-structures, and unbounded thread creation. Numerous abstractions have been proposed in the literature for the first two forms of infinity, (e.g. [4] and [5,6]). However, to date, no useful abstraction has been proposed for unbounded thread creation. This is the purpose of this paper.

☆ This paper is based partially on [B. Cook, D. Kroening, N. Sharygina, Symbolic model checking for asynchronous Boolean programs, in: SPIN, in: Lecture Notes in Computer Science, vol. 3639, Springer, 2005, pp. 75–90] and [B. Cook, D. Kroening, N. Sharygina, Over-approximating Boolean programs with unbounded thread creation, in: Proceedings of FMCAD, IEEE, 2006, pp. 53–59].

* Corresponding author.
E-mail address: natasha.sharygina@unisi.ch (N. Sharygina).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.07.050

We tackle the problem by proposing an algorithm for over-approximating the reachability of *Boolean programs* [1] with unbounded thread creation. Boolean programs are a popular representation for the existing finite-state abstractions. Thus, using our algorithm, we can imagine a symbolic software model checker that could support programs with all the three forms of infinity.

As any abstraction procedure does, we lose some precision. That is, our Boolean program checker may now return counterexamples that are spurious within the abstraction itself. While refinement at the level of the abstraction could be performed, such a procedure is no longer guaranteed to terminate.

We have found that the spurious counterexamples that could be returned are not a practical problem. Model checkers that rely on abstractions commonly use automatic abstraction refinement techniques to obtain more precise abstractions. Counterexample-guided refinement [7] can thus be adapted to remove these false counterexamples as well as false counterexamples due to heap and arithmetic abstraction. This kind of refinement adds detail to the abstract model that is not available within the abstraction itself.

The paper is organized as follows. Section 2 defines syntax and semantics of concurrent Boolean programs. The contribution of this paper is contained in Sections 3 and 4, namely an algorithm for reachability analysis of programs with unbounded thread creation in Section 3 and their symbolic simulation in Section 4. Experimental results are discussed in Section 5.

Related work

Formal verification of multi-threaded programs is an area of active research; see [8] for an excellent survey. The development of static analysis tools for such programs is complicated due to the fact that reachability for interprocedural programs (that is, for programs that contain both communication and data-flow structures) is undecidable [9].

Pushdown automata have been used as tools for analyzing sequential programs with (recursive) procedures [10]. The expressive power of pushdown systems is equivalent to that of sequential programs with (possibly recursive) procedures where all variables have a finite data type. MOPED [11] and BEBOP [1], for example, are BDD-based symbolic model checkers for this class of language. DIZZY [12] uses symbolic simulation and a propositional satisfiability solver (SAT) in order to decide the resulting Boolean formulae. The fix-point detection is done by computing binary decision diagrams (BDDs) representing the set of reachable states. Our work uses a similar algorithm, but uses quantified Boolean formulae for the fix-point detection. As BEBOP and MOPED, DIZZY does not support multiple threads.

Several previous efforts have also applied symbolic model checking to Boolean programs with asynchronous threads. For example, Jain, Clarke and Kroening [13] use the BDD-based model checker NUSMV [14] to verify concurrent Boolean programs, but with only very limited success.

There has also been work on pushdown automata with multiple stacks, e.g., see [15]. As reachability is undecidable in this case, the existing implementations are not fully automated.

Unsound approaches have also proved successful in finding bugs in concurrent programs. For example, Qadeer and Rehof [16] note that many bugs can be found when the analysis is limited to execution traces with only a small set of context-switches. This analysis supports recursive programs. Our approach complements these techniques because, while they are unsound, they are able to analyze a larger set of programs.

The class of programs considered in this paper can be viewed as an instance of a *parameterized system*, i.e., a system with a number of identical processes (threads in our case). Many approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques, network invariants, predicate abstraction or system symmetry (see an excellent overview in [17]). Methods that are most closely related to our work are based on abstraction (for example, an extension of Murϕ uses abstraction for replicated identical components [18]). In contrast to our approach, many of these methods are only partially automated, requiring at least some human ingenuity to construct an invariant or a closure process (for example, the TLPVS tool [19] is based on manual theorem proving).

Henzinger et al. use predicate abstraction in order to construct environment models from threads [20]. When combined with a counter abstraction, an unbounded number of threads can be supported. Flanagan and Qadeer propose to use the idea of thread states in order to obtain environment models for *loosely-coupled* multi-threaded

programs [21]. In contrast to their algorithm, we address the spurious behavior introduced by this over-approximation by (safely) restricting the thread states that have been passed, and by an automatic refinement procedure.

One can also model concurrent Boolean programs as a set of rewriting rules and use rewriting techniques to prove safety. For example, [22] computes abstractions of program paths using the least solutions of a system of path language constraints. At this time it is not clear how our work compares to these techniques. One disadvantage of the term rewriting approach is that it requires translating programs written in general purpose languages into the term models. There have been no translation tools reported yet.

A number of tools for analysis of multi-threaded Java programs are available. While some of the tools compute abstract models automatically, most perform only explicit state-space exploration. Representative examples of model checkers for Java are [23] and JPF [24]. Yahav reports an implementation of a Model Checker for Java with an unbounded number of threads using three-valued logic [25]. Similar to our approach, an over-approximation is computed.

The reachability of concurrent programs with a restricted form of recursion is shown to be decidable and was implemented in ZING [26]. Here, recursive functions are partitioned into *atomic transactions*, which are only allowed to modify local variables. ZING, however, suffers from scalability problems, since its approach flattens concurrent programs to sequential programs to handle recursive procedures. BEACON [27] (an explicit-state model checker for concurrent Boolean programs) has similar scalability problems. In addition, tools implementing abstraction refinement produce abstractions that make non-trivial use of non-deterministic choice. For this reason, explicit-state model checkers perform poorly when used as the reachability procedure within a refinement loop. Kahlon and Gupta show decidability of reachability properties for a range of restrictions of concurrent pushdown systems, e.g., when communication is restricted to pairwise locks obtained in a specific discipline [28].

2. Boolean programs

In this section, we first describe how model checking for Boolean programs contributes to the problem of model checking for arbitrary software programs. We then provide syntax and a semantics of Boolean programs.

2.1. Boolean programs and predicate abstraction

The program that is to be analyzed is typically given in a language such as ANSI-C and is formally modeled by means of an infinite-state transition system.

Definition 1 (*Transition System*). A *Transition System* is a triple (S, S_0, R), where S is a (possibly infinite) set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation.

We denote the transition system (the model) of the program by M. A program state $s \in S$ comprises an encoding of the current program location and a valuation of the program variables. We denote the finite set of program locations by L.

The goal of abstraction is to compute an abstract model $\hat{M}$ from the concrete model M such that the size of the state-space is reduced while the property of interest is preserved. We denote the set of abstract states by $\hat{S}$. A concrete state is mapped to an abstract state by means of an *abstraction function*, which we denote by $\alpha : S \longrightarrow \hat{S}$. We also extend the definition of α to sets of states: Given a set $S' \subseteq S$, we denote $\{\hat{s} \in \hat{S} \mid \exists s \in S'.\alpha(s) = \hat{s}\}$ by $\alpha(S')$. The inverse of α, called the *concretization function*, maps abstract states back to the corresponding concrete states, and is denoted by γ:

$$\gamma(\hat{S}') := \{s \in S \mid \alpha(s) \in \hat{S}'\} \quad (1)$$

We restrict the presentation to *reachability properties*. The goal therefore is to compute an abstraction that preserves reachability: any program location that is reachable in M must be reachable in $\hat{M}$. *Existential Abstraction* is a form of abstraction that preserves reachability [29].

Definition 2 (*Existential Abstraction*). Given an abstraction function $\alpha : S \longrightarrow \hat{S}$, a model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R})$ is called an *Existential Abstraction* of $M = (S, S_0, R)$ iff the following conditions hold:

identifier	:	`[a − z A − Z _] [a − z A − Z 0 − 9 _]*`
expr	:	*expr* '∨' *expr* | *expr* '∧' *expr* | *expr* '⇒' *expr* | '¬' *expr*
	|	*identifier* | *identifier* ''' | ' $\star_j$' | '0' | '1'

Fig. 1. Grammar for expressions in Boolean programs.

(1) The abstract model can make a transition from an abstract state $\hat{s}$ to $\hat{s}'$ iff there is a transition from s to s' in the concrete model and s is abstracted to $\hat{s}$ and s' is abstracted to $\hat{s}'$:

$$\forall\langle\hat{s},\hat{s}'\rangle \in (\hat{S}\times\hat{S}).\hat{R}(\hat{s},\hat{s}') \iff (\exists s, s' \in (S\times S).R(s,s') \wedge \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}'). \tag{2}$$

(2) The set of abstract initial states $\hat{S}_0$ is simply the abstraction of the set of concrete initial states:

$$\hat{S}_0 = \alpha(S_0). \tag{3}$$

The fact that existential abstraction is a conservative abstraction with respect to reachability properties is formalized as follows:

Theorem 1. *Let $\hat{M}$ denote an existential abstraction of M, and let ϕ denote a reachability property. If ϕ holds on $\hat{M}$, it also holds on M:*

$$\hat{M} \models \phi \Longrightarrow M \models \phi.$$

Thus, for an existential abstraction $\hat{M}$ and any program location l that is not reachable in the abstract model $\hat{M}$, we may safely conclude that it is also unreachable in the concrete model M. Note that the converse does not hold, i.e., there may be locations that are reachable in $\hat{M}$ but not in M.

There are various possible choices for an abstraction function α. *Predicate Abstraction* is one of the most popular and widely applied methods for systematic abstraction of programs, and was introduced by Graf and Saïdi [4]. An automated procedure to generate predicate abstractions was introduced by Colón and Uribe [30]. Predicate abstraction abstracts data by only keeping track of certain predicates on the data. The predicates are typically defined by Boolean expressions over the concrete program variables. Each predicate is then represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting program is called a *Boolean Program*.

The only difference between the original program and the Boolean program are the program variables. All other aspects, in particular the control-flow structure, remain the same. In particular, if the original program is recursive and concurrent, so is the Boolean program. Due to the smaller state-space of the Boolean program, Model Checking becomes feasible. Predicate abstraction is promoted by the success of the SLAM project at Microsoft Research [31]. The SLAM tool checks control-flow dominated properties of Windows device drivers.

The predicates can be automatically inferred using *Counterexample-guided abstraction refinement* [32,33]: Abstract counterexamples that cannot be replayed in the concrete program serve as a guide for refining the abstract model. We postpone the discussion of refinement techniques for the abstract model to Section 3.2.

2.2. Syntax

The syntax of the control-flow statements is derived from C. We use the syntax introduced by Ball et al. [1] with slight modifications. We first discuss the syntax used for expressions, as Boolean programs permit a few non-standard constructs.

The grammar for expressions is given in BNF in Fig. 1. It permits the usual Boolean operators, and the following two extensions: 1) non-deterministic choice, and 2) next-state variables. The stars denote non-deterministic choice symbols. If multiple non-deterministic choices are to be used in one expression, we number them $\star_1, \star_2, \ldots$.[1] If an identifier is followed by a prime, the identifier is to be evaluated in the next state.

The grammar for Boolean programs is provided in Fig. 2. A Boolean program is a list of declarations and procedures. Declarations are comma-separated lists of identifiers. Procedures are lists of declarations and labeled statements, which are statements preceded by an optional label. A statement may be a `skip` statement, a `goto`

[1] The `schoose` non-deterministic choice operator implemented by BEBOP and similar tools can be transformed into an expression that uses $\star$.

program	:	(*declaration* | *procedure*)$^+$
declaration	:	'decl' *identifier* (',' *identifier*)* ;
procedure	:	*identifier* '(' *identifier** ')' 'begin'
		declaration*
		labeled-statement*
		'end'
labeled-statement	:	*statement* | *identifier* ':' *statement*
statement	:	'skip' ';'
	|	'goto' *identifier* ';'
	|	'return' ';'
	|	*identifier*$^+$ ':=' *expr*$^+$ 'constrain' *expr* ';'
	|	'lock' *identifier* ';'
	|	'unlock' *identifier* ';'
	|	'assume' '(' *expr* ')' ';'
	|	*identifier* '(' *expr** ')' ';'
	|	'start_thread' *identifier* ';'
	|	'end_thread' ';'

Fig. 2. Grammar for Boolean programs.

statement, a `return` statement, an assignment, an `if` statement, a procedure call, or a `start_thread` or `end_thread` statement. Tools that analyze Boolean programs permit syntactic sugar such as `while` loops, which may be replaced by corresponding `goto` statements. We assume that identifier conflicts related to scoping are resolved by means of renaming.

2.3. *Formal semantics*

2.3.1. *States and expressions*

We extend the semantics of Boolean Programs [1] to permit unbounded thread creation. Let V_g denote the set of global variables. For the sake of simplicity, we assume that all threads have the same set of local variables V_l and the same program code, i.e., there is only one set of program locations L. We denote the program by P, and the instruction at location $l \in L$ by $P(l)$. A program with threads that have different codes can easily be transformed into a program with identical threads. We denote the set of variables by $V = V_g \dot{\cup} V_l$. We assume that a subset $\mathcal{L} \subseteq V_l$ of the *local* variables is used exclusively as specially designated lock variables.[2] We write L for the set of program locations. For now, we assume that function calls are inlined. We extend our algorithm to support unbounded recursion in Section 4.3.

Definition 3 (*Explicit State*). An *explicit state* η of a Boolean program is a triple $\langle n, pc, \Omega\rangle$, where $n \in \mathbb{N}$ is the number of threads, $pc : \{1, \dots, n\} \mapsto L$ is the vector of program locations, $\Omega : (\{1, \dots, n\} \times V_l) \cup V_g \mapsto \mathbb{B}$ is the valuation of the program variables. We denote the set of explicit states by $\hat{S}$.

We denote the projection of a state η to the number of running threads in that state by $\eta.n$, the projection from a state to the values of the program counters by $\eta.pc$, and so on. The value of the program counter of thread $t \in \{1, \dots, n\}$ is denoted by $\eta.pc(t)$, the value of the local variable $v \in V_l$ of thread t is denoted by $\eta.\Omega(t, v)$.

Definition 4 (*Thread State*). The pair $\langle PC, \Omega\rangle$ with $PC \in L$ and $\Omega : V \longrightarrow \mathbb{B}$ is called a *thread state*. It is a valuation of the program counter, the local variables of a particular thread, and the globally shared variables. We use $\tilde{S}$ to denote the set of thread states.

Thus, the thread state is the set of values that are *visible* to a thread. Note that the thread states are drawn from a finite set, whereas the set of explicit states is infinite.

[2] We use local variables instead of global variables for locking in order to be able to identify the individual thread that holds a lock.

Definition 5 (μ_t). The *thread state projection function* $\mu_t : \hat{S} \longrightarrow \tilde{S}$ takes a state η of the full state-space and maps it to the state that is visible to thread $t \in \{1, \ldots, \eta.n\}$.

$$\mu_t(\eta).PC \quad := \eta.pc(t)$$
$$\mu_t(\eta).\Omega(v) := \begin{cases} \eta.\Omega(v) & : v \in V_g \\ \eta.\Omega(t, v) & : v \in V_l. \end{cases}$$

As the syntax for expressions permits a few non-standard constructs, we formally define their meaning. Besides Boolean operators and identifiers, expressions may contain next-state identifiers (primed identifiers), which refer to a second thread state of the same thread (and thus, to a second valuation of the variables). They may also contain non-deterministic choice denoted by the $\star$ symbols. We define three different evaluation functions, depending on which of these constructs are permitted in the context of the expression.

Definition 6 (*Meaning of Expressions*). Let Ω and Ω' denote a valuation of the variables V, and let ι denote a valuation of the $\star$ symbols. Given an expression e, we use $[\![e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!]$ to denote the evaluation of e. Let e, e_1, and e_2 denote expressions, and $v \in V$ be a variable. Formally, $[\![e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!]$ is defined recursively as follows:

$$\begin{aligned} [\![e_1 \vee e_2\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] &:= [\![e_1\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] \vee [\![e_2\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] \\ [\![\neg e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] &:= \neg[\![e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] \\ [\![v\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] &:= \Omega(v) \\ [\![v'\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] &:= \Omega'(v) \\ [\![\star_j\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] &:= \iota_j. \end{aligned}$$

We may write $[\![e\,,\ \Omega\,,\ \iota]\!]$ for $[\![e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!]$ if e contains no primed variables. We may write $[\![e\,,\ \Omega]\!]$ for $[\![e\,,\ \Omega\,,\ \iota]\!]$ if e contains neither $\star$ symbols nor primed variables.

For any function $f : D \rightarrow R$ and any $d \in D, r \in R$, we define $f[d/r] : D \rightarrow R$ as follows:

$$f[d/r](x) = \begin{cases} r & : d = x \\ f(x) & : \text{otherwise.} \end{cases}$$

As a shorthand, for any $\tilde{\eta} \in \tilde{S}$, $\tilde{\zeta} \in \tilde{S}$ we write $\tilde{\eta} \stackrel{G}{=} \tilde{\zeta}$ iff the values of the global, i.e., shared variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal, i.e., $\forall g \in V_g.\tilde{\eta}.\Omega(g) = \tilde{\zeta}.\Omega(g)$. Similarly, we write $\tilde{\eta} \stackrel{L}{=} \tilde{\zeta}$ iff the values of the local variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal.

2.3.2. Execution semantics

We provide operational semantics for Boolean programs, i.e., we define the sequences of states that constitute valid executions of a given program. We first define semantics for the sequential program, and later on extend it to the concurrent case.

We use $\tilde{\eta} \longrightarrow \tilde{\zeta}$ to denote the fact that a transition from thread state $\tilde{\eta}$ to thread state $\tilde{\zeta}$ is valid. The relation $\tilde{\eta} \longrightarrow \tilde{\zeta}$ is defined by a case-split on the instruction $\tilde{\eta}.PC$.

The rules for each instruction are shown in Table 1. The semantics of the `skip`, `goto`, constrained assignment, and `start_thread` statements are straightforward:

- The `skip` statement increments the program counter of the thread. The values of the variables do not change.
- The `goto` $\theta_1, \ldots, \theta_k$ statement changes the program counter to one of the program locations $\theta_1, \ldots, \theta_k$ that are given as argument. The choice is arbitrary, i.e., non-deterministic. The values of the variables do not change.
- The constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e changes the program counter like `skip`. It also updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state $\tilde{\eta}$. The expressions may contain choice variables $\iota_1, \ldots, \iota_m$. These variables allow a non-deterministic choice on data, and are quantified existentially.

  The transition also has an additional user-provided constraint e. The constraint e is a predicate in terms of the current state $\tilde{\eta}$ and the next state $\tilde{\zeta}$. It is evaluated in both states accordingly, where the next-state variables are primed. If there is no choice for ι that satisfies the constraint, state $\tilde{\eta}$ has no successor states.

Table 1
Conditions on the explicit thread state transition $\tilde{\eta} \longrightarrow \tilde{\zeta}$ for various statements, where $\theta, \theta_i \in L$, e is an expression and $l \in \mathcal{L}$

$$\frac{P(PC) = \texttt{skip}}{\langle PC, \Omega\rangle \longrightarrow \langle PC+1, \Omega\rangle} \tag{4}$$

$$\frac{P(PC) = \texttt{goto}\ \theta_1, \ldots, \theta_k}{\langle PC, \Omega\rangle \longrightarrow \langle \theta_i, \Omega\rangle} \quad i \in \{1, \ldots, k\} \tag{5}$$

$$\frac{\begin{array}{l} P(PC) = x_1, \ldots, x_k \texttt{:=} e_1, \ldots, e_k\ \texttt{constrain}\ e, \\ \quad \Omega' = \Omega[x_1/[\![e_1\,,\ \Omega\,,\ \iota]\!]]\ldots[x_k/[\![e_k\,,\ \Omega\,,\ \iota]\!]], \\ \quad [\![e\,,\ \Omega\,,\ \Omega'\,,\ \iota]\!] \end{array}}{\langle PC, \Omega\rangle \longrightarrow \langle PC+1, \Omega'\rangle} \tag{6}$$

$$\frac{P(PC) = \texttt{start_thread}\ \theta}{\langle PC, \Omega\rangle \longrightarrow \langle PC+1, \Omega\rangle} \qquad \frac{P(PC) = \texttt{start_thread}\ \theta}{\langle PC, \Omega\rangle \longrightarrow \langle \theta, \Omega\rangle} \tag{7}$$

$$\frac{P(PC) = \texttt{lock}\ l, \quad \Omega(l) = \text{false}}{\langle PC, \Omega\rangle \longrightarrow \langle PC+1, \Omega[l/\text{true}]\rangle} \tag{8}$$

$$\frac{P(PC) = \texttt{unlock}\ l, \quad \Omega(l) = \text{true}}{\langle PC, \Omega\rangle \longrightarrow \langle PC+1, \Omega[l/\text{false}]\rangle} \tag{9}$$

- A state in which the `start_thread` instruction is executed has two successors: in one successor, the PC is incremented, and in the other successor, the PC is set to the program location θ given as argument. The values of the variables do not change.

Note that `lock` and `unlock` are special cases of a constrained assignment. The `end_thread` instruction can be read as an infinite loop. The `assume` statement is a special case of a constrained assignment, and the rule is therefore omitted.

We write $\ell(\tilde{\eta}) \subseteq \mathcal{L} := \{l \in \mathcal{L} \mid \tilde{\eta}.\Omega(l)\}$ for the set of locks that are held in state $\tilde{\eta}$.

We now define the semantics of the concurrent program. Let $\hat{S}^0 \subseteq \hat{S}$ denote the set of initial states, which consists of the state that has a single thread, the PC points to the entry point of the `main` function, and all variables are zero.

$$\hat{S}^0 = \{\langle 1, \lambda t.\, main, \lambda v \in (\{1\} \times V_l) \cup V_g\,.\,0\rangle\}. \tag{10}$$

Assume that the scheduler picks a thread $t \in \{1, \ldots, \eta.n\}$ to execute in state η. We use $\eta \longrightarrow_t \zeta$ to denote the fact that a transition from state η is made to ζ by executing one statement of thread t. The statement that is executed is $P(\eta.pc(t))$. The relation $\eta \longrightarrow_t \zeta$ is defined by a case-split on this instruction.

The formal rules are given in Table 2. For all instructions but `start_thread`, we require that

- the number of threads does not change, i.e., $\zeta.n = \eta.n$,
- thread t makes a transition, i.e., $\mu_t(\eta) \longrightarrow \mu_t(\zeta)$,
- and the values of local variables and the program counters of the other threads $j \neq t$ remain unchanged, i.e., $\mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$,
- locks are held exclusively, i.e., $\ell(\mu_u(\zeta)) \cap \ell(\mu_v(\zeta)) = \emptyset$ for all $u \neq v$.

If $P(\eta.pc(t))$ is `start_thread` θ, we require that

- the number of threads increases by one, i.e., $\zeta.n = \eta.n + 1$,
- the program counter of the new thread is θ, and the program counter of thread t is $\eta.pc(t) + 1$, i.e., $\zeta.pc(\zeta.n) = \theta$ and $\zeta.pc(t) = \eta.pc(t) + 1$,
- thread t makes a transition into both changed states, i.e., $\mu_t(\eta) \longrightarrow \mu_t(\zeta)$ and $\mu_t(\eta) \longrightarrow \mu_{\zeta.n}(\zeta)$, and
- the values of the local variables of the other threads $j \neq t$ and $j \neq \zeta.n$ remain unchanged, i.e., $\mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$.

Table 2
Conditions on the explicit thread state transition $\eta \longrightarrow \zeta$

$$\frac{\begin{array}{l} P(\eta.pc(t)) \neq \texttt{start_thread}, \\ \mu_t(\eta) \longrightarrow \mu_t(\zeta), \quad \eta.n = \zeta.n, \\ \forall j \neq t.\, \mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta) \wedge \zeta.pc(j) = \eta.pc(j), \\ \forall u \neq v.\, l(\mu_u(\zeta)) \cap l(\mu_v(\zeta)) = \emptyset \end{array}}{\eta \longrightarrow \zeta} \tag{11}$$

$$\frac{\begin{array}{l} P(\eta.pc(t)) = \texttt{start_thread}\ \theta, \\ \mu_t(\eta) \longrightarrow \mu_t(\zeta), \quad \mu_t(\eta) \longrightarrow \mu_{\zeta.n}(\zeta), \quad \zeta.n = \eta.n + 1, \\ \zeta.pc(\zeta.n) = \theta, \zeta.pc(t) = \eta.pc(t) + 1, \\ \forall j \neq t \wedge j \neq \zeta.n.\ \mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta) \wedge \zeta.pc(j) = \eta.pc(j) \end{array}}{\eta \longrightarrow \zeta} \tag{12}$$

Syntactic sugar such as `if` or `while` can be easily transformed using `goto` and `assume`, as described in [2].

Finally, we write $\eta \longrightarrow \zeta$ if there exists a thread $t \in \{1, \ldots, \eta.n\}$ such that $\eta \longrightarrow_t \zeta$. In this case, we say that there is a transition from η to ζ, or that ζ is reachable from η with one transition. Let $\hat{S}^i \subseteq \hat{S}$ with $i \in \mathbb{N}$ denote the set of states reachable in i or less transitions, i.e., $\hat{S}^i := \hat{S}^{i-1} \cup \{\zeta \mid \exists \eta \in \hat{S}^{i-1}.\eta \longrightarrow \zeta\}$. The set of all reachable states is $\hat{S}^\infty$. The property we check is reachability of states with particular program locations. The set of these 'bad' locations is denoted by $\mathcal{B} \subset L$. Assertions can be re-written into this form, by introducing a designated error location, where the program branches to if the assertion fails.

3. Over-approximation and refinement

3.1. Over-approximating $\hat{S}^\infty$

Finite-state model checking algorithms are based on fix-point detection, that is, the model checker compares the new set of states computed using the transition relation with the states explored so far. The algorithm iterates until no new states are discovered.

This basic idea can be applied to programs with unbounded thread creation as well. For example, SPIN [34] permits dynamic creation of new threads by means of Promela's `run` statement. However, SPIN assumes that the program only creates a finite number of threads. If the thread creation is not actually bounded, the state enumeration of SPIN may not terminate.

We propose an algorithm that does not restrict thread creation to a finite number, i.e., we permit an infinite set $\hat{S}^\infty$ while still guaranteeing termination [3]. The classical fix-point detection algorithm is not readily applicable for this case.

Definition 7 (μ_*). Let $\mu_*(\eta)$ denote the set of the thread states $\mu_t(\eta)$ for all threads t. Let $\hat{S}' \subseteq \hat{S}$ be a set of states of the Boolean program. The *thread-visible states* are the states in $\hat{S}'$ projected to the thread states of all threads.

$$\mu_*(\eta) := \{\mu_t(\eta) : t \in \{1, \ldots, \eta.n\}\}$$
$$\mu_*(\hat{S}') := \bigcup_{\eta \in \hat{S}'} \mu_*(\eta).$$

The set of thread states reachable with at most i transitions is denoted by $\tilde{S}^i := \mu_*(\hat{S}^i) \subseteq \tilde{S}$. We propose to compute an over-approximation of $\tilde{S}^\infty$. This is sufficient to detect violations of reachability properties that are expressed in terms of the thread visible state,[3] e.g., assertions.

Definition 8 ($\rightsquigarrow$). Let $\tilde{A} \subseteq \tilde{S}$ denote a set of thread states, and $\tilde{\zeta} \in \tilde{S}$ denote a thread state. Let $\tilde{A} \rightsquigarrow \tilde{\zeta}$ hold iff any of the following two conditions holds:

(1) there is $\tilde{\eta} \in \tilde{A}$ such that there is a transition from $\tilde{\eta}$ to $\tilde{\zeta}$, i.e., $\tilde{\eta} \longrightarrow \tilde{\zeta}$,

[3] Note that the property still may depend on the behavior of multiple threads, due to the communication between the threads.

(2) or there exists $\tilde{\eta} \in \tilde{A}$ and another transition out of $\tilde{A}$ from states with disjoint sets of locks that changes the global state of $\tilde{\eta}$ to that of $\tilde{\zeta}$. Formally, we require $\tilde{\zeta}.PC = \tilde{\eta}.PC$, and $\tilde{\zeta} \overset{L}{=} \tilde{\eta}$ and there exist $\tilde{\eta}' \in \tilde{A}$ and $\tilde{\zeta}' \in \tilde{S}$ such that
   (a) $\tilde{\eta}' \longrightarrow \tilde{\zeta}'$ with $\tilde{\eta}' \overset{G}{\neq} \tilde{\zeta}'$, and
   (b) $\tilde{\eta}' \overset{G}{=} \tilde{\eta}$, and
   (c) $\tilde{\zeta}' \overset{G}{=} \tilde{\zeta}$, and
   (d) $\ell(\tilde{\eta}) \cap \ell(\tilde{\eta}') = \emptyset$ and $\ell(\tilde{\eta}) \cap \ell(\tilde{\zeta}') = \emptyset$.
   This case captures the communication between two threads.

We write $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ instead of the more cumbersome $\{\tilde{\eta}\} \rightsquigarrow \tilde{\zeta}$. Note that $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ implies $\tilde{A} \rightsquigarrow \tilde{\zeta}$ for any $\tilde{A}$ with $\tilde{\eta} \in \tilde{A}$. Let $\tilde{T}^0 := \mu_*(\hat{S}^0)$ denote the set of initial thread states, and $\tilde{T}^i$ for $i \in \mathbb{N}$ be defined recursively as follows:

$$\tilde{T}^i := \tilde{T}^{i-1} \cup \{\tilde{\zeta} \mid \tilde{T}^{i-1} \rightsquigarrow \tilde{\zeta}\}.$$

The following claim holds by construction of $\rightsquigarrow$.

Theorem 2. *For all $i \in \mathbb{N}_0$, the set $\tilde{T}^i$ is an over-approximation of the set of reachable thread states $\tilde{S}^i$.*

Proof. The claim is shown by induction on i. For $i = 0$, the claim is trivial.

We show that $\tilde{T}^i \supseteq \tilde{S}^i$ for the step from $i-1$ to i as follows. Let $\tilde{\zeta} \in \tilde{S}^i$. By definition of $\tilde{S}^i$, there is a full state $\zeta \in \hat{S}^i$ and $u \in \{1, \ldots, \zeta.n\}$ such that $\tilde{\zeta} = \mu_u(\zeta)$. Furthermore, there exists $\eta \in \hat{S}^{i-1}$ and $t \in \{1, \ldots, \eta.n\}$ such that $\eta \longrightarrow_t \zeta$. Let $\tilde{\eta}$ be a shorthand for $\mu_u(\eta)$. Using the induction hypothesis, we can conclude that $\mu_*(\eta) \subseteq \tilde{T}^{i-1}$, and in particular, $\tilde{\eta} \in \tilde{T}^{i-1}$.

If $u = t$, we have $\tilde{\eta} \longrightarrow \tilde{\zeta}$, which implies $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ (case 1 of Definition 8), and thus $\tilde{\zeta} \in \tilde{T}^i$, which concludes the claim.

If $u \neq t$, we make a case-split on the instruction $P(\eta.pc(t))$, which is executed in the transition from η to ζ (Table 1):

- If $P(\eta.pc(t))$ is `skip`, `goto`, or `assume`, only the PC of thread t changes, and thus, $\tilde{\zeta} = \tilde{\eta}$, which implies $\tilde{\zeta} \in \tilde{T}^{i-1}$, and thus, $\tilde{\zeta} \in \tilde{T}^i$.
- If $P(\eta.pc(t))$ is `start_thread` and $u \neq \zeta.n$ (i.e., u is not the newly created thread), we also have $\tilde{\zeta} = \tilde{\eta}$, which concludes the claim. If $u = \zeta.n$, we have $\mu_t(\eta) \longrightarrow \tilde{\zeta}$, which concludes the claim.
- If $P(\eta.pc(t))$ is $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e, let $k = 1$ without loss of generality. If $x_1 \in V_l$, only data local to thread t is modified, and the claim is shown as in case of `skip`.

  If $x_1 \in V_g$, let $v := \tilde{\zeta}.\Omega(x_1)$ denote the value that is assigned to x_1 by thread t. The new thread state $\tilde{\zeta}$ is equal to $\tilde{\eta}$ up to the assignment to x_1, i.e., $\tilde{\zeta}.PC = \tilde{\eta}.PC$ and $\tilde{\zeta}.\Omega = \tilde{\eta}.\Omega[x_1/v]$. Also, $\mu_t(\eta) \overset{G}{=} \tilde{\eta}$, and threads t and u hold a disjoint set of locks in state η. We therefore have $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ using case 2 of Definition 8.
- The statements `lock` and `unlock` are special cases of constrained assignments. ■

As the sequence $\tilde{T}^0, \tilde{T}^1, \ldots$ is monotonic and is taken from a finite set (the set of thread states, Definition 4), it has a fixed-point, and thus, $\tilde{T}^\infty$ is easily computable. The theorem above therefore gives rise to an algorithm (Fig. 3). If the algorithm terminates with "Property holds", the property is guaranteed to hold on the Boolean program. However, if an error state is found, there is no guarantee that the state is actually reachable. A counterexample trace can be computed by recording the one or two states that are used to compute a new thread state.

In order to illustrate the benefit of condition (2d) in Definition 8, consider the Boolean program in Fig. 4(a), and assume a definition of $\rightsquigarrow$ without condition (2d). Let $\rightsquigarrow^-$ denote this relation. Suppose we initialize `g` and `l` with 0 and start an unbounded number of threads that execute the function `f()`. The set of reachable thread states is shown in Fig. 4(b). The lock protects the global variable, and thus, the assertion in `L2` holds.

We denote a state by (PC, g, l). However, because of {(L5, 0, 1),(L2, 0, 1)} $\rightsquigarrow^-$ (L2, 0, 0) $\longrightarrow$ (L3, 0, 0) and {(L3, 0, 0),(L2, 0, 0)} $\rightsquigarrow$ (L2, 1, 0), the set $\tilde{T}^\infty$ contains a state that violates the assertion, and we obtain a spurious error trace.

// Input: Boolean Program P with locations L,
// set of bad locations $\mathcal{B} \subset L$
UNBOUNDEDTHREADAPPROXIMATION(P, b)
1 $\tilde{T} := \mu_*(\hat{S}^0)$; // Initial States
2 **while** (true)
3 **if** ($\exists \tilde{\eta} \in \tilde{T}.\tilde{\eta}.PC \in \mathcal{B}$) **return** "Error state found";
4 $\tilde{F} := \{\tilde{\zeta} \in \tilde{S} \mid \tilde{T} \rightsquigarrow \tilde{\zeta}\}$;
5 **if** ($\tilde{F} \subseteq \tilde{T}$) **return** "Property holds";
6 $\tilde{T} := \tilde{T} \cup \tilde{F}$;
7 **end**

Fig. 3. High level description of the approximation algorithm for reachability in Boolean programs with unbounded threads.

```
decl g, l;

void f() begin
 L1: lock l;
 L2: assert(!g);
 L3: g:=1;
 L4: g:=0;
 L5: unlock l;
 L6: skip;
end
```

(a)

PC	$\Omega(g)$	$\Omega(l)$
L1	0	0
L2	0	1
L3	0	1
L4	1	1
L5	0	1

(b)

Fig. 4. Boolean Program with critical section.

Remark 1. As an additional optimization, we keep track of whether a thread state was generated before or after a `start_thread` command. Thread states that are generated before the execution of any `start_thread` command need not participate in case 2 of Definition 8. This optimization results in fewer spurious error traces, as the set of states of the program reachable up to the first `start_tread` command is no longer over-approximated. This case is omitted from the proof.

3.2. Refinement

If the procedure above concludes that no state with an error location is reachable, this also holds on the original program, and we can conclude that the property holds (Theorem 1).

The procedure above may also terminate by discovering a thread state that has a PC that corresponds to an error location, i.e., a location in the set of 'bad' locations $\mathcal{B}$. In this case, we compute an abstract counterexample.

Definition 9 (*Abstract Counterexample*). An *abstract counterexample* is a sequence of explicit states $\hat{s}_0, \ldots, \hat{s}_n$ with $\hat{s}_i \in \hat{S}$, $\hat{s}_i \longrightarrow \hat{s}_{i+1}$, and $\hat{s}_n.pc(t) \in \mathcal{B}$ for some t.

The abstract counterexample is computed by tracking the thread states that are used to infer a new thread state. Recall the definition of the $\rightsquigarrow$ relation (Definition 8). In the first case, the predecessor of any state $\tilde{\zeta}$ with $\tilde{A} \longrightarrow \tilde{\zeta}$ is well-defined, and recorded appropriately. In the second case, there are two thread states that are used to infer $\tilde{\zeta}$. We record the transition that corresponds to the second pair of states.

The abstract counterexample is passed to the abstraction refinement procedure that generated the Boolean program. The next step is to simulate it on the original model. An abstract trace is *feasible* in M iff the following formula is satisfiable:

$$s_0 \in S \wedge \bigwedge_{i=0}^{n} \alpha(s_i) = \hat{s}_i \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}). \quad (13)$$

A satisfying assignment to this formula contains a valuation of $s_0, \ldots, s_n$, which is a counterexample for $M \models \varphi$. In practice, as any concrete counterexample suffices, a weaker criterion is typically checked, which disregards the data:

$$s_0 \in S \wedge \bigwedge_{i=0}^{n} s_i.pc = \hat{s}_i.pc \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}). \tag{14}$$

If the formula is satisfiable, the counterexample is reported, and the algorithm terminates. If not so, the counterexample is *spurious*, and refinement has to be performed.

Such a simulation corresponds to an incremental series of SAT instances, and is commonly performed by program analysis tools that implement abstraction refinement, e.g., SLAM and BLAST. In the previous work, spurious counterexamples are always due to lack of precision of the abstract model. Two cases are distinguished: spurious transitions and spurious prefixes.

(1) A spurious transition is an abstract transition with $\hat{s}_i \longrightarrow \hat{s}_{i+1}$ but there exists no pair s_i, s_{i+1} with $R(s_i, s_{i+1})$ and $\alpha(s_i) = \hat{s}_i$, $\alpha(s_{i+1}) = \hat{s}_{i+1}$. Such counterexamples are removed by strengthening assignment statements by means of the `constrain` construct.
(2) A spurious prefix is a prefix of an abstract counterexample without spurious transitions that does not satisfy Eq. (14).

The drawback of the over-approximation we perform to verify $\hat{M}$ is that it may produce *additional* spurious counterexamples that do not fall into either category. These counterexamples are characterized by the fact that there is an abstract transition from $\hat{s}_i$ to $\hat{s}_{i+1}$ with $\hat{s}_i \not\longrightarrow \hat{s}_{i+1}$, i.e., the trace does not fulfill the conditions of Definition 9. This case can only be caused by an application of the second case in Definition 8.

In this case, we refine the abstract model $\hat{M}$ as in the case of a spurious prefix: we add additional predicates by either forward-simulating using strongest postconditions or backward-simulating using weakest preconditions. The additional predicate correspond to new variables, which *split* the states that were falsely merged, and thus, eliminate the spurious counterexample.

4. Symbolic simulation

In this section we describe how we represent a set of states symbolically using formulae, and then how to transform Boolean programs into such formulae.

4.1. Symbolic state representation

Definition 10. A symbolic *formula* is defined using the following syntax rules:

(1) The Boolean constants true and false are formulae.
(2) The non-deterministic choice identifiers $\star_1, \star_2, \ldots$ are formulae.
(3) If f_1 and f_2 are formulae, then $f_1 \wedge f_2$, $f_1 \vee f_2$, and $\neg f_1$ are formulae.

The set of such formulae is denoted by $\mathcal{F}$.

A symbolic formula may be evaluated to multiple values due to the choice identifiers. As an example, the pair of formulae $\langle \star_1, \star_2 \wedge \neg\star_1 \rangle$ may be evaluated to $\langle 0, 0\rangle$, $\langle 1, 0\rangle$, $\langle 0, 1\rangle$, but not to $\langle 1, 1\rangle$. Given a particular valuation ι for the non-deterministic choices $\star_i$, we denote the value of a symbolic formula f as $[\![f]\!]^\iota$, i.e., $\iota \models f \iff [\![f]\!]^\iota = \mathsf{true}$.

We use these symbolic formulae in order to represent sets of explicit thread states:

Definition 11. A *symbolic thread state* $\tilde{\sigma}$ is a triple $\langle PC, \omega, \gamma \rangle$, with $PC \in L$, $\omega : V \mapsto \mathcal{F}$, and $\gamma \in \mathcal{F}$.

The first component of a symbolic thread state $\tilde{\sigma}$, namely PC, is identical to the first component of an explicit thread state (Definition 4). The second component, called ω, is a mapping from the set of variables into the set of formulae. It denotes the *symbolic* valuation of the state variables. The last component, called γ, is a formula that represents the *guard* of the state symbolically, i.e., a constraint over the variables. Note that the program counter is represented explicitly, while the program variables are represented symbolically.

Table 3
Conditions on the symbolic thread state transition $\tilde{\sigma} \longrightarrow \tilde{\zeta}$ with $\tilde{\sigma} = (PC, \omega, \gamma)$ and $\tilde{\zeta} = (PC', \omega', \gamma')$, for various statements $P(PC)$, where $\theta_i \in L$, e is an expression and $l \in \mathcal{L}$

$P(\tilde{\sigma}.PC)$	ω	γ
`skip`	$\omega' = \omega$	$\gamma' = \gamma$
`goto` $\theta_1, \ldots, \theta_k$	$\omega' = \omega$	$\gamma' = \gamma$
`assume` e	$\omega' = \omega$	$\gamma' = (\gamma \wedge [\![e\,,\,\tilde{\sigma}]\!])$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$\exists \iota.\ \Omega' = \Omega \quad [x_1/[\![e_1\,,\,\tilde{\sigma}\,,\,\tilde{\zeta}]\!]] \ \ldots \quad [x_k/[\![e_k\,,\,\tilde{\sigma}\,,\,\tilde{\zeta}]\!]]$	$\gamma' = (\gamma \wedge [\![e\,,\,\tilde{\sigma}\,,\,\tilde{\zeta}]\!])$

We can define the symbolic evaluation $[\![e\,,\,\tilde{\sigma}\,,\,\tilde{\tau}]\!]$ of an expression e in the symbolic thread state $\tilde{\sigma}$ and a next state $\tilde{\tau}$ in analogy to the definition for explicit states. The set of explicit-thread states represented by a symbolic thread state $\tilde{\sigma}$ are those states $\tilde{\eta} \in \tilde{S}$ that satisfy the following conditions:

- They have matching PCs: $\tilde{\eta}.PC = \tilde{\sigma}.PC$.
- There exists a valuation ι of the choice variables that satisfies the guard γ and assigns values to the variables that match the values given by $\tilde{\eta}.\Omega$.

$$\exists \iota.\iota \models \gamma \wedge \forall v \in V.[\![v\,,\,\tilde{\eta}]\!] = [\![v\,,\,\tilde{\sigma}]\!]^{\iota}. \tag{15}$$

Note that the set of explicit states corresponding to a symbolic state is defined using a predicate in the parameter ι. Therefore, we have a *parametric representation* of the state-space. Parametric representations of sets of states have been used in formal verification before, e.g., in [35–37], but mostly in the context of hardware verification.

4.2. Symbolic execution

In analogy to the explicit-state model described in Section 2, we use $\tilde{\sigma} \rightarrow \tilde{\zeta}$ to denote the fact that a transition is made from state $\tilde{\sigma}$ to $\tilde{\zeta}$ by executing one statement of thread t. Again, the statement that is executed is $P(\tilde{\sigma}.PC)$. The definition of the relation $\tilde{\sigma} \rightarrow \tilde{\zeta}$ is done using a case-split on this instruction. The conditions for each statement are shown in Table 3. The column describing the constraints on the program counters PC and PC' is identical to the column in Table 1, and therefore is not repeated here. We explain the symbolic execution of each statement as follows:

- The definitions of the `skip` and `goto` statements follow the definitions for the explicit-state case. The formulae for the guards are not changed by these statements.
- In the symbolic case, the `assume` e statement does not have the precondition that e is true. Instead, the condition e is instantiated in the state $\tilde{\sigma}$. This results in a symbolic formula. The symbolic formula is conjoined with the guard γ, forming the formula γ'.
- In the symbolic case, a constrained assignment statement $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e updates the values of the variables using the expressions $e_1, \ldots, e_k$. The expressions are evaluated in state $\tilde{\sigma}$. It is no longer necessary to enumerate the possible values of the non-deterministic choice variables ι, as $\omega(v)$ is now a formula, and not a Boolean value. Thus, the fresh choice variables become part of the formula. Also, the additional constraint e is added to the guard, in analogy to an `assume` statement.

The fixed-point loop using symbolic thread states is identical to the loop for the explicit representation shown in Fig. 3. The only non-trivial part is the detection of the saturation condition, which corresponds to a QBF (quantified Boolean formula) instance.

We also implement partial order reduction. In the context of the algorithm in Fig. 3, this corresponds to strengthening $\rightsquigarrow$ such that modifications to global variables are only applied to thread states $\tilde{\zeta}$ that have a program counter $\tilde{\zeta}.PC$ that points to an instruction that either (1) reads one of the global variables that are modified or (2) writes a global variable.

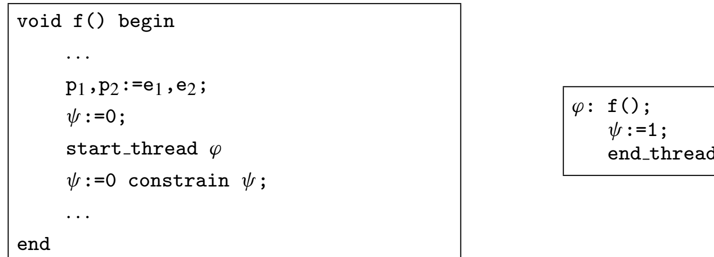

Fig. 5. Over-approximating a recursive call $f(e_1, e_2)$ with thread creation.

4.3. Recursive functions

Reachability for programs with recursion and concurrency (even with only two threads) is undecidable [9]. In order to model recursive programs we further extend the idea of conservative over-approximation.

Let f denote the function that is called, and let $p_1, \ldots, p_k \in V_l$ denote the parameters of the function. The expression e_i is passed as argument of the call for p_i.

- As first step, an assignment $p_1, \ldots, p_k := e_1, \ldots, e_k$ is performed.
- For synchronization upon return of the function, we introduce a new global variable ψ. An assignment statement is inserted before the function call that sets ψ to zero.
- The function call is replaced by a `start_thread` φ command, where φ is a program location in which f is called.
- After the `start_thread` θ command, the statement $\psi := 0$ `constrain` ψ is inserted. It sets ψ to false, but waits for ψ to become true before doing so.
- When f returns (using `return`), it sets ψ to true. The return values are passed by means of global variables.

The approximation of a recursive call $f(e_1, e_2)$ using thread creation is illustrated in Fig. 5.

This reduction is similar to an encoding of recursion commonly done in SPIN that uses a new channel for synchronization. In contrast to this reduction used for SPIN, we use a finite set of global variables for synchronization (one call per site), and therefore loose precision. The termination of the second recursion may synchronize with the call site of the first recursion and so on.

5. Experimental results

We have implemented the technique described in this paper in a tool called BOPPO, which is available on-line.[4] For the purpose of experimentation we have integrated BOPPO as the Boolean program checker for SATABS [38,39] and used the prototype to verify safety properties of programs with (possibly unbounded) `while` loops that contain thread creating statements, e.g., the `pthread_create()` command. We compared the results of this integration to the only existing model checker that supports all the features needed in these examples.

The experiments have been performed on an Intel Xeon Processor with 2.8 GHz running Linux. The results are summarized in Table 4. We use MiniSAT as our SAT-solver, and Quantor as QBF solver for the fixed-point detection. The run-time results are reported for our tool with symbolic partial order reduction. We also report the number of *symbolic* thread states that are explored, i.e., $|\tilde{T}|$. Note that one symbolic thread state typically corresponds to many explicit thread states, in particular if non-determinism is used heavily. On all experiments with non-trivial run-time, the run-time is dominated by the QBF solver.

We focus on the evaluation of the scalability of the implementation. We have two classes of benchmarks: artificial ones to measure scalability (ART series), and benchmarks extracted from the Apache httpd web-server package (AP series). The ART-PC-n series benchmarks are scaled in the number of program locations. Each benchmark generates an unbounded number of threads (using an infinite loop containing `start_thread`). Each thread then executes n non-deterministic assignments to global variables. The QBF instances generated for the fixed-point detection contain a number of quantified variables that is linear in n. The ART-V-n series benchmarks are parameterized in the number

[4] Available for download at http://www.verify.ethz.ch/boppo/.

Table 4
Summary of results: A star denotes that the 1gig spaceout threshold was exceeded

Benchmark	SATABS/BOPPO		Zing	Spin
	Time (s)	$\#\tilde{\sigma}$	Time	Time
ART-PC-10	<0.1	21	⋆	⋆
ART-PC-20	<0.1	31	⋆	⋆
ART-PC-100	0.2	111	⋆	⋆
ART-PC-1000	8.3	1011	⋆	⋆
ART-V-10	0.1	29	⋆	⋆
ART-V-20	0.3	49	⋆	⋆
ART-V-100	5.3	209	⋆	⋆
ART-V-1000	3508.1	2009	⋆	⋆
AP1	242.7	8009	⋆	⋆
AP2	269.5	10766	⋆	⋆
AP3	288.9	11422	⋆	⋆
AP4	155.1	5453	⋆	⋆
AP5	1130.9	43812	⋆	⋆

The columns under $\#\tilde{\sigma}$ contain the number of symbolic thread states.

of variables, where n denotes the number of global (and thus, also thread-visible) variables. The number of variables in the QBF instances grows quadratically in n.

The APn series of benchmarks is extracted from an ANSI-C program using SATABS. While the original program generates a finite number of threads using the POSIX `pthread_create` command, the abstraction of the program generates an unbounded number of threads. The POSIX `pthread_mutex_lock` and `unlock` functions are mapped to `lock` and `unlock` in the Boolean programs. The various benchmarks correspond to different properties of the same program.

Apache (like most other programs) does not use locking during initialization, i.e., before it starts the worker threads. The algorithm as described above results in states with inconsistent global predicates, which produce a large number of spurious counterexamples. For this benchmark, we therefore extend the algorithm to distinguish two different types of thread states using a flag as suggested in Remark 1 above. The flag is false in the initial state, and is set to true upon execution of `start_thread`. Case 2 of Definition 8 is changed such that global data is only passed between thread states that have the same value of the flag. After the initialization phase, most writes to global data are protected by means of locks, which prevents spurious error traces.

On the artificial examples, the regular refinement using weakest preconditions (WPs) works finely to eliminate the spurious counterexamples, as it adds more Boolean variables, which makes the states different (and only states with equal values of the global variables participate in case 2 of Definition 5).

As the table highlights, our SATABS/BOPPO integration is the first software model checker to realistically support programs with unbounded thread creation. On these examples both ZING and SPIN spaceout due to the large amounts of non-deterministically initialized data used to model the environment. SATABS/BOPPO is able to support this form of infiniteness due to its support for abstraction together with its symbolic model checking engine.

Direct comparison. The experiments described above, of course, do not compare BOPPO *directly* to competing tools. Note that, even if thread creation is bounded in the original unabstracted program, SATABS's abstraction procedure is likely to produce abstractions in which thread creation is not bounded. For this reason both ZING and SPIN timed-out on all of the abstractions produced by SATABS.

6. Conclusion

Symbolic model checkers that implement abstraction refinement have proven tremendously useful for sequential programs. For shared-memory concurrent software they have been essentially useless. This is due to the fact that reachability on the abstraction of a concurrent program is undecidable. Existing approaches use algorithms that surrender completeness, and thus, the refinement loop may get stuck even on simplistic abstract models.

This paper introduces a new algorithm for checking abstractions of software (Boolean programs) that supports arbitrary thread creation while guaranteeing termination. This algorithm can potentially return spurious counterexamples, but it *is always able* to produce one. The existing refinement techniques can be used to refine abstract models with such counterexamples.

Acknowledgements

The second author was supported in part by an award from IBM Research. The second and third authors were supported by a grant from the Swiss National Science Foundation.

References

[1] T. Ball, S.K. Rajamani, Bebop: A symbolic model checker for Boolean programs, in: SPIN 00, in: Lecture Notes in Computer Science, vol. 1885, Springer, 2000, pp. 113–130.
[2] B. Cook, D. Kroening, N. Sharygina, Symbolic model checking for asynchronous Boolean programs, in: SPIN, in: Lecture Notes in Computer Science, vol. 3639, Springer, 2005, pp. 75–90.
[3] B. Cook, D. Kroening, N. Sharygina, Over-approximating Boolean programs with unbounded thread creation, in: Proceedings of FMCAD, IEEE, 2006, pp. 53–59.
[4] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, vol. 1254, Springer, 1997, pp. 72–83.
[5] A. Podelski, T. Wies, Boolean heaps, in: Static Analysis (SAS), in: Lecture Notes in Computer Science, vol. 3672, Springer, 2005, pp. 268–283.
[6] G. Yorsh, T.W. Reps, M. Sagiv, R. Wilhelm, Logical characterizations of heap abstractions, ACM Transactions on Computational Logic 8 (1) (2007).
[7] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, Journal of the ACM 50 (5) (2003) 752–794.
[8] M.C. Rinard, Analysis of multithreaded programs, in: Static Analysis (SAS), in: Lecture Notes in Computer Science, vol. 2126, Springer, 2001, pp. 1–19.
[9] G. Ramalingam, Context-sensitive synchronization-sensitive analysis is undecidable, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 416–430.
[10] O. Burkart, B. Steffen, Composition, decomposition and model checking of pushdown processes, Nordic Journal of Computing 2 (1995) 89–125.
[11] J. Esparza, S. Schwoon, A BDD-based model checker for recursive programs, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, vol. 2102, Springer, 2001, pp. 324–336.
[12] K.R.M. Leino, A SAT characterization of Boolean-program correctness, in: Model Checking Software (SPIN), in: Lecture Notes in Computer Science, vol. 2648, Springer, 2003, pp. 104–120.
[13] H. Jain, E. Clarke, D. Kroening, Verification of SpecC and Verilog using predicate abstraction, in: Proceedings of MEMOCODE 2004, IEEE, 2004, pp. 7–16.
[14] A. Cimatti, et al., NuSMV 2: An opensource tool for symbolic model checking, in: Computer Aided Verification (CAV), in: Lecture Notes in Computer Science, Springer, 2002, pp. 359–364.
[15] S. Chaki, E.M. Clarke, N. Kidd, T.W. Reps, T. Touili, Verifying concurrent message-passing C programs with recursive calls, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), in: Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 334–349.
[16] S. Qadeer, J. Rehof, Context-bounded model checking of concurrent software, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), in: Lecture Notes in Computer Science, vol. 3440, Springer, 2005, pp. 93–107.
[17] E. Clarke, M. Talupur, T. Touili, H. Veith, Verification by network decomposition, in: CONCUR 04, in: Lecture Notes in Computer Science, vol. 3170, Springer, 2004, pp. 276–291.
[18] C. Ip, D. Dill, Verifying systems with replicated components in murϕ, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, vol. 1102, Springer, 1996, pp. 147–158.
[19] A. Pnueli, T. Arons, TLPVS: A PVS-based LTL verification system, in: Verification—Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday, in: Lecture Notes in Computer Science, Springer, 2003, pp. 84–98.
[20] T.A. Henzinger, R. Jhala, R. Majumdar, Race checking by context inference, in: PLDI, ACM, 2004, pp. 1–13.
[21] C. Flanagan, S. Qadeer, Thread-modular model checking, in: SPIN, in: Lecture Notes in Computer Science, vol. 2648, Springer, 2003, pp. 213–224.
[22] A. Bouajjani, J. Esparza, T. Touili, Reachability analysis of synchronized PA systems, Electronic Notes in Theoretical Computer Science 138 (3) (2005) 153–178.
[23] S.D. Stoller, Model-checking multi-threaded distributed Java programs, in: SPIN Model Checking and Software Verification, in: Lecture Notes in Computer Science, vol. 1885, Springer, 2000, pp. 224–244.
[24] K. Havelund, T. Pressburger, Model checking Java programs using Java PathFinder, International Journal on Software Tools for Technology Transfer 2 (4) (2000) 366–381.

[25] E. Yahav, Verifying safety properties of concurrent Java programs using 3-valued logic, in: POPL, ACM Press, 2001, pp. 27–40.
[26] T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, Y. Xie, Zing: Exploiting program structure for model checking concurrent software, in: CONCUR 04, in: Lecture Notes in Computer Science, vol. 3170, Springer, 2004, pp. 1–15.
[27] T. Ball, S. Chaki, S.K. Rajamani, Parameterized verification of multithreaded software libraries, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), in: Lecture Notes in Computer Science, vol. 2031, Springer, 2001, pp. 158–173.
[28] V. Kahlon, A. Gupta, On the analysis of interacting pushdown systems, in: Principles of Programming Languages (POPL), ACM, 2007, pp. 303–314.
[29] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, in: Principles of Programming Languages (POPL), ACM, 1992, pp. 343–354.
[30] M.A. Colón, T.E. Uribe, Generating finite-state abstractions of reactive systems using decision procedures, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, vol. 1427, Springer, 1998, pp. 293–304.
[31] T. Ball, S. Rajamani, Boolean programs: A model and process for software analysis, Tech. Rep. 2000-14, Microsoft Research, February 2000.
[32] R. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton University Press, 1995.
[33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, Springer, 2000, pp. 154–169.
[34] G. Holzmann, D. Peled, The State of SPIN, in: Computer-Aided Verification (CAV), in: Lecture Notes in Computer Science, vol. 1102, Springer, 1996, pp. 385–389.
[35] P. Jain, G. Gopalakrishnan, Efficient symbolic simulation-based verification using the parametric form of Boolean expressions, IEEE Transactions on CAD of ICs and Systems 13 (8) (1994) 1005–1015.
[36] O. Coudert, J. Madre, A unified framework for the formal verification of sequential circuits, in: ICCAD, IEEE, 1990, pp. 78–82.
[37] M.D. Aagaard, R.B. Jones, C.-J.H. Seger, Formal verification using parametric representations of boolean constraints, in: DAC, ACM Press, 1999, pp. 402–407.
[38] E. Clarke, D. Kroening, N. Sharygina, K. Yorav, Predicate abstraction of ANSI-C programs using SAT, Formal Methods in System Design 25 (2004) 105–127.
[39] E. Clarke, D. Kroening, N. Sharygina, K. Yorav, SATABS: SAT-based predicate abstraction for ANSI-C, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), in: Lecture Notes in Computer Science, vol. 3440, Springer, 2005, pp. 570–574.