Over-Approximating Boolean Programs with Unbounded Thread Creation

Byron Cook
Microsoft Research
Cambridge
Email: bycook@microsoft.com

Daniel Kroening
Computer Systems Institute
ETH Zurich
Email: daniel.kroening@inf.ethz.ch

Natasha Sharygina
Computer Science Department
University of Lugano
Email: natasha.sharygina@unisi.ch

Abstract— **This paper describes a symbolic algorithm for over-approximating reachability in Boolean programs with unbounded thread creation. The fix-point is detected by projecting the state of the threads to the globally visible parts, which are finite. Our algorithm models recursion by over-approximating the call stack that contains the return locations of recursive function calls, as reachability is undecidable in this case. The algorithm may obtain spurious counterexamples, which are removed iteratively by means of an abstraction refinement loop. Experiments show that the symbolic algorithm for unbounded thread creation scales to large abstract models.**

I. Introduction

All scalable symbolic model checkers for software are currently based on counterexample-guided abstraction refinement (CEGAR) (e.g., BLAST [1], SLAM [2], MAGIC [3], SATABS [4], DIVER [5]). To date, none of these model checkers supports unbounded thread creation together with shared memory cross-thread communication. This gap is not due to lack of need: much of the software that these tools are used to verify are actually shared memory concurrent programs with unbounded thread creation. Static Driver Verifier (SDV) [2], for example, is used to verify Windows device drivers—which are tremendously concurrent pieces of software. SDV's analysis is unsound because it ignores the side-effects caused by other threads.

The cause for this gap between the software model checkers and the software that they are intended to verify is a technical one: CEGAR is effective only if the underlying reachability procedure is guaranteed to terminate—and terminate quickly. When unbounded thread creation is added into the mixture, today's reachability engines often do not terminate.

We address this problem with a new symbolic model checker for *Boolean programs* (the most common form of abstractions used within CEGAR-based tools for software) that supports unbounded thread creation while guaranteeing termination. What we lose is precision—the Boolean program checker may now return counterexamples that are spurious within the abstraction itself. The experimental results show that this is not a practical problem: the CEGAR refinement mechanism can be adapted to remove these false counterexamples as well as the counterexamples that are spurious only in the unabstracted software. Furthermore, the experimental results demonstrate that the algorithm scales in practice to large concurrent programs.

The contribution of this paper is contained in Sections III and IV, namely an algorithm for reachability analysis of programs with unbounded thread creation in Section III and their symbolic simulation in Section IV. Experimental results are discussed in Section V.

Related Work

Formal verification of multi-threaded programs is an area of active research; see [6] 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 [7].

Pushdown automata have been used as tools for analyzing sequential programs with (recursive) procedures [8]. 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 [9] and BEBOP [10], for example, are BDD-based symbolic model checkers for this class of language. There has also been work on pushdown automata with multiple stacks, e.g., see [11]. 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 & Rehof [12] 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 [13]). 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 [14]). In contrast to our approach, many of these methods are only partially automated, requiring at least some human ingenuity to construct a process invariant

or a closure process (for example, the TLPVS tool [15] is based on manual theorem proving).

Henzinger et al. use predicate abstraction in order to construct environment models from threads [16]. 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 [17]. In contrast to their algorithm, we address the spurious behavior introduced by this over-approximation by (safely) restricting the thread-states that are 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, [18] 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 are no translation tools reported yet.

A number of tools for analysis of multi-threaded Java programs is 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 [19] and JPF [20]. Yahav reports an implementation of a Model Checker for Java with an unbounded number of threads using three-valued logic [21]. Similarly 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 implemented in ZING [22]. 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 [23] (an explicit-state model checker for concurrent Boolean programs) has similar scalability problems. Additionally, CEGAR-based tools produce abstractions that make non-trivial use of under-specified values. For this reason, explicit-state model checkers perform poorly when used as the reachability procedure within a CEGAR loop.

II. BOOLEAN PROGRAMS

A. Syntax

The syntax of the control flow statements is derived from C, and can be found in [10]. The syntax for expressions permits the usual Boolean operators, and the following two extensions: 1) non-deterministic choice, and 2) next-state variables.

$$\begin{array}{rcl} \textit{expression} & : & \textit{expression} \text{ '} \vee \text{' } \textit{expression} \\ & | & \textit{expression} \text{ '} \wedge \text{' } \textit{expression} \\ & | & \text{'}\neg\text{'}\textit{expression} \\ & | & \textit{atom} \\ \textit{atom} & : & \textit{Identifier} \mid \textit{Identifier} \text{ '}'\text{' } \mid \text{ '}\star_j\text{'} \end{array}$$

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.[2]

B. Formal Semantics

We extend the semantics of Boolean Programs [10] 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. A program with threads that have different code 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 for locking exclusively.[3]

Definition 1 (Explicit State): An *explicit state* η of a Boolean program is a triple (n, pc, Ω), where $n \in \mathbb{N}$ is the number of threads, $pc : \{1, \ldots, n\} \mapsto L$ is the vector of program locations, $\Omega : (\{1, \ldots, n\} \times V_l) \cup V_g \mapsto \mathbb{B}$ is the valuation of the program variables. We denote the set of explicit states by 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 $t \in \{1, \ldots, n\}$ 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 2 (Thread State): The tuple (PC, Ω) 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 global shared variables. We use $\tilde{S}$ to denote the set of thread states.

Thus, the thread state is the set of values that is *visible* to a thread.

Definition 3 (μ_t): The *thread state projection function* $\mu_t : S \longrightarrow \tilde{S}$ takes a state η of the full state space and maps it to the state visible to thread $t \in \{1, \ldots, \eta.n\}$.

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

Given a thread state $\tilde{\eta} \in \tilde{S}$ and an expression e over the variables V, we use $[\![e\,,\,\tilde{\eta}]\!]$ to denote the evaluation of e by a thread in state $\tilde{\eta}$. Let e, e_1, and e_2 denote expressions, and $v \in V$ be a variable. Formally, $[\![e\,,\,\tilde{\eta}]\!]$ is defined recursively as follows:

$$\begin{array}{rcl} [\![e_1 \vee e_2\,,\,\tilde{\eta}]\!] & := & [\![e_1\,,\,\tilde{\eta}]\!] \vee [\![e_2\,,\,\tilde{\eta}]\!] \\ [\![\neg e\,,\,\tilde{\eta}]\!] & := & \neg[\![e\,,\,\tilde{\eta}]\!] \\ [\![v\,,\,\tilde{\eta}]\!] & := & \tilde{\eta}.\Omega(v) \end{array}$$

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

[2] Tools such as BEBOP expect the prime *before* the identifier.

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

The next-state identifiers (primed identifiers) refer to the next thread state $\tilde{\zeta} \in \tilde{S}$. The semantics of expressions containing such primed identifiers is defined using the evaluation function $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!]$. The definition of $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!]$ is identical to the definition of $[\![e\,,\,\tilde{\eta}]\!]$ above, unless e is a primed identifier:

$$[\![v'\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!] \quad := \quad \tilde{\zeta}.\Omega(v)$$

The semantics of expressions containing non-deterministic choice symbols is given by $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]$, where ι denotes the valuation of the $\star$ symbols. The definition is identical to the definition above, unless e is $\star_j$ for some j:

$$[\![\star_j\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!] \quad := \quad \iota_j$$

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

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

As a shorthand, we write $\tilde{\eta} \overset{G}{=} \tilde{\zeta}$ iff the values of the global, i.e., shared variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal, i.e., $\forall g \in V_g.[\![g\,,\,\tilde{\eta}]\!] = [\![g\,,\,\tilde{\zeta}]\!]$. Similarly, we write $\tilde{\eta} \overset{L}{=} \tilde{\zeta}$ iff the values of the local variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal.

Execution Semantics: We use $\tilde{\eta} \longrightarrow \tilde{\zeta}$ to denote the fact that a transition from state $\tilde{\eta}$ is made to $\tilde{\zeta}$ by executing the statement $\tilde{\eta}.PC$. The relation $\tilde{\eta} \longrightarrow \tilde{\zeta}$ is defined by a case-split on this instruction. The conditions for each case are shown in Table I. The description of the semantics of the `skip`, `goto`, `assume`, and constrained assignment statements are identical to the description found in [24]. The definitions of `lock` and `unlock` are straight-forward. Note that `lock` and `unlock` are special cases of a constrained assignment.

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

The semantics of the concurrent program is defined as follows: Assume 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. 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., $\mu_j(\eta) \overset{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$,
- locks are held exclusively, i.e., $l(\mu_u(\zeta)) \cap l(\mu_v(\zeta)) = \emptyset$ 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., $\mu_j(\eta) \overset{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$.

Syntactic sugar such as `if` or `while` can be easily transformed using `goto` and `assume`, as described in [24]. For now, we assume that function calls can be inlined. We extend our algorithm to support unbounded recursion in Section IV-B.

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 $S^0 \subseteq S$ denote the set of initial states, and let $S^i \subseteq S$ with $i \in \mathbb{N}$ denote the set of states reachable in i or less transitions. The set of all reachable states is S^∞. The property we check is reachability of states with particular program locations.

III. Over-Approximation and Refinement

A. Over-approximating S^∞

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 [25] 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 never terminates.

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

Definition 4 (μ_*): Let $\mu_*(\eta)$ denote the set of the thread states $\mu_t(\eta)$ for any thread t. Let $S' \subseteq S$ be a set of states. The *thread-visible states* are the states in S' projected to the thread states of all threads.

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

The set of thread states reachable in i transitions is denoted by $\tilde{S}^i := \mu_*(S^i) \subseteq \tilde{S}$. 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[4], e.g., assertions.

Definition 5 ($\rightsquigarrow$): 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., $\tilde{\eta} \longrightarrow \tilde{\zeta}$,

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

$P(\tilde{\eta}.PC)$	PC	Ω
`skip`	$PC' = PC + 1$	$\Omega' = \Omega$
`goto` $\theta_1, \ldots, \theta_k$	$\bigvee_{i=1}^{k} PC' = \theta_i$	$\Omega' = \Omega$
`assume` e	$PC' = PC + 1$	$\Omega' = \Omega \ \wedge [\![e\,,\,\tilde{\eta}]\!] = \mathsf{true}$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$PC' = PC + 1$	$\exists \iota.\ \Omega' = \Omega \quad [x_1/[\![e_1\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]] \ \ldots \quad [x_k/[\![e_k\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]] \ \wedge [\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]$
`start_thread` θ	$PC' = PC + 1$ $\vee PC' = \theta$	$\Omega' = \Omega$
`lock` l	$PC' = PC + 1$	$\Omega(l) = \mathsf{false} \wedge \Omega' = \Omega[l/\mathsf{true}]$
`unlock` l	$PC' = PC + 1$	$\Omega(l) = \mathsf{true} \wedge \Omega' = \Omega[l/\mathsf{false}]$

TABLE I

CONDITIONS ON THE EXPLICIT THREAD STATE TRANSITION $\tilde{\eta} \longrightarrow \tilde{\zeta}$ WITH $\tilde{\eta} = (PC, \Omega)$ AND $\tilde{\zeta} = (PC', \Omega')$, FOR VARIOUS STATEMENTS $P(PC)$, WHERE $\theta_i \in L$, e IS AN EXPRESSION AND $l \in \mathcal{L}$.

2) or there exists $\tilde{\eta} \in \tilde{A}$ and another transition out of $\tilde{A}$ with a disjoint set of locks that changes the global state of $\tilde{\eta}$ to that of $\tilde{\zeta}$. Formally, we require $\tilde{\zeta}.PC = \tilde{\eta}.PC$, $\tilde{\zeta} \stackrel{L}{=} \tilde{\eta}$ and there exist $\tilde{\eta}' \in \tilde{A}$ and $\tilde{\zeta}' \in \tilde{S}$ such that
   a) $\tilde{\eta}' \longrightarrow \tilde{\zeta}'$ with $\tilde{\eta}' \stackrel{G}{\neq} \tilde{\zeta}'$,
   b) $\tilde{\eta}' \stackrel{G}{=} \tilde{\eta}$, and
   c) $\tilde{\zeta}' \stackrel{G}{=} \tilde{\zeta}$,
   d) $l(\tilde{\eta}) \cap l(\tilde{\eta}') = \emptyset$ and $l(\tilde{\eta}) \cap l(\tilde{\zeta}') = \emptyset$.

   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 $\tilde{T}^0 := \mu_*(S^0)$ denote the set of initial thread states, and $\tilde{T}^i$ for $i \in \mathbb{N}$ be defined recursively as follows:

$$\tilde{T}^i := \tilde{T}^{i-1} \cup \{\tilde{\zeta} \,|\, \tilde{T}^{i-1} \rightsquigarrow \tilde{\zeta}\}$$

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

Theorem 1: 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 $\tilde{T}^i \supseteq \tilde{S}^i$ 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 $\zeta \in S^i$ and $u \in \{1, \ldots, \zeta.n\}$ such that $\tilde{\zeta} = \mu_u(\zeta)$. Furthermore, there exists $\eta \in S^{i-1}$ 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 Def. 5), 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 I):

- 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, $\mu_t(\eta) \stackrel{G}{=} \tilde{\eta}$, 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 Def. 5.
- The statements `lock` and `unlock` are special cases of constrained assignments.

■

```
   // Input: Boolean Program P with locations L,
   // bad location b ∈ L
   UNBOUNDEDTHREADAPPROXIMATION(P, b)
1  T̃:=μ*(S⁰);      // Initial States
2  while (true)
3        if (∃η̃ ∈ T̃.η̃.PC = b) return "Error state found";
4        F̃:={ζ̃ ∈ S̃ | T̃ ⇝ ζ̃};
5        if (F̃ ⊆ T̃) return "Property holds";
6        T̃:=T̃ ∪ F̃;
7  end
```

Fig. 1. High level description of the approximation algorithm for reachability in Boolean programs with unbounded threads

As the sequence $\tilde{T}^0, \tilde{T}^1, \ldots$ is monotonic and taken from a finite set, it has a fixed-point, and thus, $\tilde{T}^\infty$ is easily computable. The theorem above therefore gives rise to an algorithm (Fig. 1). 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.

To illustrate the benefit of condition 2d) in Def. 5, consider the Boolean program in Fig. 2a, and assume a definition of $\rightsquigarrow$ without condition 2d). Suppose we start an unbounded number of threads that execute `f()`. The set of reachable thread states is shown in Fig. 2b. The lock protects the global variable, and

```
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. 2. Boolean Program with critical section

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), $\tilde{T}^{\infty}$ contains a state that violates the assertion, and we obtain a spurious error trace.

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 5. 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.

B. Refinement

The drawback of the over-approximation is that it may produce additional spurious counterexamples. Thus, for reachability properties φ, we may obtain $M' \not\models \varphi$ even though $M \models \varphi$ holds. If the algorithm generates an error trace, the error trace is simulated on the full model in order to rule out spurious error traces due to the imprecision introduced by $\rightsquigarrow$. 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.

If the error trace is spurious, the over-approximation is refined. Note that we assume that this refinement is performed outside of the model checker as part of an abstraction refinement loop. Our algorithm may introduce spurious counterexamples due to the over-approximation caused by case 2 of Def. 5. The refinement algorithms in the existing predicate abstraction tools remove spurious traces by adding predicates to the model. This refinement strategy is effective for the over-approximation performed by our analysis algorithm, as the additional variables split states into two or more states $\tilde{\eta}, \tilde{\zeta}$ such that $\tilde{\eta} \overset{G}{\neq} \tilde{\zeta}$, which violates condition 2 of Def. 5.

IV. SYMBOLIC SIMULATION

A. Symbolic State Representation

This section presents how thread states are represented symbolically. It extends the algorithm described in [24] to support an unbounded number of threads.

Definition 6: 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 evaluate 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 evaluate 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 7: 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 2). 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.

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 the same PC: $\tilde{\eta}.n = \tilde{\sigma}.n \,\wedge\, \tilde{\eta}.PC = \tilde{\sigma}.PC$
- There exists a valuation ι that satisfies the guard γ and assigns values to the variables that match the values given by $\tilde{\eta}.\Omega$.

$$\exists \iota . \iota \models \gamma \;\wedge\; \forall v \in V.\; [\![v\,,\,\tilde{\eta}]\!] = [\![v\,,\,\tilde{\sigma}]\!]^{\iota} \qquad (1)$$

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 [26], [27], [28], but mostly in the context of hardware verification.

The construction of the symbolic thread states and the fixed-point loop with fixed-point detection using QBF follows the principle described in [24].

We also implement partial order reduction. In the context of algorithm 1, this corresponds to strengthening $\rightsquigarrow$ such that transitions are only propagated 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 begin modified or 2) writes a global variable.

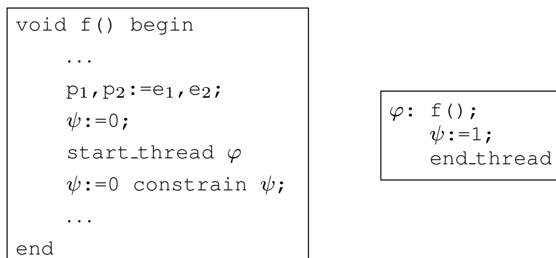

Fig. 3. Over-approximating a recursive call $f(e_1, e_2)$ with thread creation

B. Recursive Functions

Reachability for programs with recursion and concurrency (even with only two threads) is undecidable [7]. 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` θ command, where θ denotes the first program location of f.
- After the function call, 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. 3.

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 per call site), and therefore loose precision. The termination of the second recursion may synchronize with the call site of the first recursion and so on.

V. Experimental Results

We have implemented the technique described in this paper in a tool called BoPPo. To the best of our knowledge, there is no other model checker available for either Boolean programs with unbounded thread creation or concurrent Boolean programs with recursion. An implementation based on symbolic simulation has been compared to Moped, SPIN, Bebop, and Zing in [24]. However, none of these model checkers supports the class of programs the algorithm described in this paper aims at, which prevents experimental comparison. We make our implementation available to other researchers for experimentation[5].

[5]`http://www.verify.ethz.ch/boppo/`

The BoPPo is integrated as model checker for abstract models into SatAbs, which is an implementation of SAT-based predicate abstraction [4], [29]. In this configuration, SatAbs can verify safety properties of programs with (possibly unbounded) `while` loops that contain thread creating statements, e.g., the `pthread_create()` command. SatAbs is also available for download[6].

The experiments have been performed on an Intel Xeon Processor with 2.8 GHz running Linux. The results are summarized in Table II. 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 and without 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 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 are 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 produces 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 Def. 5 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

[6]`http://www.verify.ethz.ch/satabs/`

Benchmark	Without PO		With PO	
	Time	$\#\tilde{\sigma}$	Time	$\#\tilde{\sigma}$
ART-PC-10	6.0s	893	<0.1s	21
ART-PC-20	21.0s	2723	<0.1s	31
ART-PC-100	$\star$		0.2s	111
ART-PC-1000	$\star$		8.3s	1011
ART-V-10	17.2s	801	0.1s	29
ART-V-20	111.7s	2571	0.3s	49
ART-V-100	$\star$		5.3s	209
ART-V-1000	$\star$		3508.1s	2009
AP1	$\star$		242.7s	8009
AP2	$\star$		269.5s	10766
AP3	$\star$		288.9s	11422
AP4	$\star$		155.1s	5453
AP5	$\star$		1130.9s	43812

TABLE II

SUMMARY OF RESULTS: A STAR DENOTES THAT THE TWO HOUR TIMEOUT WAS EXCEEDED. THE COLUMNS UNDER $\#\tilde{\sigma}$ CONTAIN THE NUMBER OF SYMBOLIC THREAD STATES.

prevents spurious error traces.

On the artificial examples, the regular refinement with WPs works fine to eliminate the spurious CEs, as it adds more Boolean variables, which make the states different (and only states with equal values of the global variables participate in case 2 of Def. 5).

VI. CONCLUSION

CEGAR-based symbolic model checkers have proven themselves tremendously useful for sequential programs. For shared-memory concurrent software they have been effectively useless. This is due to fact that the underlying tool that checks the abstractions must always return an answer—something that no tool has been able to guaranteed when applied to abstractions that can support arbitrary thread creation. This paper introduces a new symbolic model checker for software abstractions (Boolean programs) that supports arbitrary thread creation while guaranteeing termination. This checker can potentially return spurious counterexamples, but it *is always able* to produce one.

REFERENCES

[1] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, "Thread modular abstraction refinement," in *CAV*, ser. LNCS. Springer, 2003, pp. 262–274.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *EuroSys*, 2006.

[3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *International Conf. on Software Engineering*, 2003, pp. 385–395.

[4] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT," *Formal Methods in System Design*, vol. 25, pp. 105–127, September–November 2004.

[5] M. K. Ganai, A. Gupta, and P. Ashar, "DiVer: SAT-based model checking platform for verifying large scale systems," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 575–580.

[6] M. Rinard, "Analysis of multithreaded programs," in *SAS*, ser. LNCS, vol. 2126. Springer, 2001. [Online]. Available: citeseer.ist.psu.edu/article/rinard01analysis.html

[7] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 416–430, March 2000.

[8] O. Burkart and B. Steffen, "Composition, decomposition and model checking of pushdown processes," *Nordic Journal of Computing*, vol. 2, pp. 89 – 125, 1995.

[9] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," in *CAV*, ser. LNCS 2102. Springer, 2001, pp. 324–336.

[10] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for Boolean programs," in *SPIN 00*, ser. LNCS 1885. Springer, 2000, pp. 113–130.

[11] S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili, "Verifying concurrent message-passing C programs with recursive calls." in *TACAS*, ser. LNCS, vol. 3920. Springer, 2006, pp. 334–349.

[12] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *TACAS 05*. Springer, 2005.

[13] E. Clarke, M. Talupur, T. Touili, and H. Veith, "Verification by network decomposition," in *CONCUR 04*, ser. LNCS, vol. 3170. Springer, 2004, pp. 276–291.

[14] C. Ip and D. Dill, "Verifying systems with replicated components in murϕ," in *CAV*, ser. LNCS, vol. 1102. Springer, 1996, pp. 147–158.

[15] A. Pnueli and 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*, ser. LNCS. Springer, 2003, pp. 84–98.

[16] T. A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference." in *PLDI*. ACM, 2004, pp. 1–13.

[17] C. Flanagan and S. Qadeer, "Thread-modular model checking." in *SPIN*, ser. LNCS, vol. 2648. Springer, 2003, pp. 213–224.

[18] J. E. Ahmed Bouajjani and T. Touili, "Reachability analysis of synchronized PA systems," in *Infinity 04: International Workshop on Verification of Infinite-State Systems*, 2004.

[19] S. Stoller, "Model-checking multi-threaded distributed Java programs," in *SPIN 00*. Springer, 2000.

[20] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2(4), 2000.

[21] E. Yahav, "Verifying safety properties of concurrent java programs using 3-valued logic." in *POPL*. ACM Press, 2001, pp. 27–40.

[22] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: Exploiting program structure for model checking concurrent software," in *CONCUR 04*, 2004.

[23] T. Ball, S. Chaki, and S. K. Rajamani, "Parameterized verification of multithreaded software libraries," in *TACAS*. Springer, 2001.

[24] B. Cook, D. Kroening, and N. Sharygina, "Symbolic model checking for asynchronous boolean programs," in *SPIN*. Springer, 2005, pp. 75–90.

[25] G. Holzmann and D. Peled, "The State of SPIN," in *CAV*, ser. LNCS. Springer, 1996, vol. 1102, pp. 385–389.

[26] P. Jain and G. Gopalakrishnan, *IEEE Transactions on Computer-Aided Design*, vol. 13, 1994.

[27] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD*. IEEE, 1990, pp. 78–82.

[28] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of boolean constraints," in *DAC*. ACM Press, 1999, pp. 402–407.

[29] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3440. Springer Verlag, 2005, pp. 570–574.