Infinite-State Backward Exploration of Boolean Broadcast Programs

Peizun Liu and Thomas Wahl
Northeastern University, Boston, USA {lpzun|wahl}@ccs.neu.edu

***Abstract*—Assertion checking for non-recursive unbounded-thread Boolean programs can be performed in principle by converting the program into an infinite-state transition system such as a Petri net and subjecting the system to a *coverability* check, for which sound and complete algorithms exist. Said conversion adds, however, an additional heavy burden to these already expensive algorithms, as the number of system states is exponential in the size of the program. Our solution to this problem avoids the construction of a Petri net and instead applies the coverability algorithm directly to the Boolean program. A challenge is that, in the presence of advanced communication primitives such as broadcasts, the coverability algorithm proceeds backwards, requiring a backward execution of the program. The benefit of avoiding the up-front transition system construction is that "what you see is what you pay": only system states backward-reachable from the target state are generated, often resulting in dramatic savings. We demonstrate this using Boolean programs constructed by the SATABS predicate abstraction engine.**

I. INTRODUCTION

Infinite-state system verification continues to be an active field of research. A highly sought-after target are algorithms for the reachability of state sets "upward-closed" with respect to a given well quasi-order; a problem referred to as *coverability*. Recent years have seen intense work on designing practical coverability algorithms that attempt to defy the high computational lower bounds known for this problem.

The application of these algorithms to *programs* — with variable assignments and control flow — rather than state transition systems, is more involved. The data complexity of programs is typically addressed via predicate abstraction. Recent work has pushed the limits of this technique to encompass multi-threaded software [1]. The abstractions are finite-state "Boolean" programs executed concurrently by a possibly unbounded number of threads.

What remains is to close the gap between these programs and the framework of *well quasi-ordered systems (WQOS)* [2], for which coverability problems are decidable. In principle, this can be achieved by formally translating the (symmetric) unbounded-thread Boolean programs into transition systems such as forms of Petri nets: thread-local variable valuations become local states, which in turn are converted into unbounded counter variables, recording the number of threads occupying the corresponding local state at a given time.

In practice, however, this naive method only works for programs with few variables, since the number of states in the resulting WQOS is of course exponential in the size of the program. This explosion — before any kind of system analysis has been performed — makes subsequent coverability analysis intractable but for small programs. In finite-state model checking, the classical method to curb the explosion incured during the program-to-system translation is to avoid the translation altogether and instead build the transition system on the fly: system states are converted to program states, the program is simulated one step, and the resulting program state is converted back into a system state.

In this paper, we build on this idea and present a coverability algorithm — a variant of infinite-state backward search [2] — that operates *directly on the Boolean program*. What makes the classical on-the-fly technique challenging in this context is:

1) the WQOS constructed on the fly does not encode the simulated multi-threaded program directly, but a *counting abstraction* of it: WQOS states store numbers of threads in certain local states. This additional level of indirection must be unraveled before the program can be executed on a system state; and
2) the coverability algorithm [2] proceeds backwards. That is, after unfolding a system state into a program state, we have to execute the program backwards in order to find predecessors. Moreover, the algorithm computes preimages consisting not only of direct predecessors, but also of *cover predecessors*: predecessors of states "larger" than the current state.

The computation of cover predecessors is a consequence of the infinite-state operation of the algorithm; how to do this for Boolean programs is a main technical contribution of this paper. The backward direction of the algorithm is essential to be able to handle *broadcasts*, such as produced by a recent predicate abstraction method [1]. Alternative, forward-directed infinite-state algorithms such as the Karp-Miller procedure [3] are known not to extend naturally to broadcast programs [4].

To summarize, we present in this paper the first, to our knowledge, coverability algorithm for the broad class of Boolean broadcast programs that avoids an up-front construction of (broadcast|Petri) nets or other transition systems. The exploration cost is thus proportional to the backward-reachable system states, rather than the size of the conceivable state space. We show experimental results on 30 predicate-abstracted C programs that convincingly demonstrate how our method speeds up algorithms otherwise known to be well-performing coverability checkers.

This work is supported by NSF grant no. 1253331.

ISBN: 978-0-9835678-4-4.

II. PREPARATIONS

This paper presents an approach to applying Abdulla's infinite-state backward search algorithm [2], designed for well quasi-ordered transition systems (WQOS), to a Boolean program family. In this section we sketch syntax and semantics of Boolean programs, the notion of WQOS and their relation to Boolean program families, and the basics of Abdulla's algorithm to decide certain reachability questions over WQOS.

A. Boolean Broadcast Programs

Boolean programs typically arise from predicate abstractions of `C` or Java code. All variables are of type `bool`. Control flow constructs are optimized for automated analysis, rather than ease of programming.

An overview of the syntax of Boolean programs is given in Fig. 1 and mostly compatible with that used in the CPROVER toolkit[1]. A program is a top-level **decl**aration of Boolean variables — called *shared* — with compile-time computable, possibly nondeterministic initial values, followed by a list of function definitions. A function definition is an initializing **decl**aration of Boolean variables called *local*, followed by a list of labeled statements.

prog ::= **decl** *initvarlist*; *func**

func ::= *name* (*varlist*) { **decl** *initvarlist*; [*label*: *stmt*;]* }

stmt ::= *seqstmt*
| **start_thread** *label*
| **atomic** { [*stmt*;]* }
| **wait**
| **broadcast**

seqstmt ::= **skip**
| **goto** *labellist*
| **assume** (*expr*)
| *varlist* **:=** *exprlist* [**constrain** *expr*]
| **if** (*expr*) **then** *seqstmt* **else** *seqstmt* **fi**
| **assert** (*expr*)

Fig. 1: Boolean program syntax (partial; slightly simplified)

A formal description of the semantics of Boolean program statements is beyond the scope of this paper. We sketch here the main concepts; for some details see Table I, for more details see [5]. The **skip** statement advances the program counter (pc); **goto** *labellist* nondeterministically chooses one of the given labels as the next pc; **assume** terminates executions that do not satisfy the given expression. The **:=** statement assigns the values of the given expressions to the respective variables, in parallel, but terminates the execution if the result does not satisfy the **constrain** expression, if any. The semantics of **if** is standard; **assert** indicates assertions for verification and otherwise acts like **skip**. In all cases, *expr* is a Boolean expression over shared and local variables of the program, the constants 0 and 1, and the choice symbol $\star$; the latter nondeterministically evaluates to 0 or 1. For example, the statement `assume` $(b \wedge \star)$ behaves like `skip` in states where $b = 1$, and terminates the execution in states where $b = 0$. Function calls and **return** statements are omitted; they have standard semantics.

The remaining statements in Fig. 1 support threading in Boolean programs. Their intuitive semantics is as follows:

start_thread *label* (i) advances the pc of the executing thread to the next statement, and (ii) creates a new thread whose local variables are copied from those of the executing thread and whose pc is given by *label*.

atomic $\{$*stmt*$^*\}$ denotes atomic execution: a thread executing inside an atomic section cannot be preempted.

wait blocks the execution of a thread (see next).

broadcast advances the pc of the executing thread, and wakes up all threads currently blocked at a **wait** statement, if any, i.e. it advances their pc as well. A **broadcast** is thus non-blocking. (More general models may offer distinct pairs of **wait/broadcast** statements, using *condition variables*.)

Thread termination is omitted, as — for the purposes of reachability analysis — it can be simulated by trapping the terminating thread in a self loop. Fig. 2 (left) shows a Boolean program with an assertion. We are interested in this paper in detecting assertion violations: does there exist a multi-threaded execution of the program in which some thread reaches a failing assertion?

B. From Programs to Infinite-State Transition Systems

Let $\mathcal{B}$ be a Boolean program defined over sets of shared and local Boolean variables V_S and V_L, respectively, and let $\{1, \ldots, pc_{\max}\}$ be the set of program locations. $\mathcal{B}$ gives rise to an infinite-state transition system M^∞ as follows. The states of M^∞ have the form $(s, \ell_1, \ldots, \ell_n)$, where s is a valuation of the shared variables of $\mathcal{B}$ and is called the *shared state*. Symbol ℓ_i is a valuation of the pc and the local variables of $\mathcal{B}$ and is called the *local state of thread* i. We write $s.v$ ($\ell_i.v$) for the value of shared (local) variable v in shared (local) state s (ℓ_i), and $\ell_i.pc$ for thread i's current pc value. Finally, n is a positive integer, intuitively the number of threads currently running. The state space of M^∞ is therefore the infinite set

$$S^\infty = \{0,1\}^{|V_S|} \times \bigcup_{n=1}^{\infty} \Big(\{1, \ldots, pc_{\max}\} \times \{0,1\}^{|V_L|} \Big)^n \ .$$

A transition of M^∞ is of the form

$$(s, \ell_1, \ldots, \ell_n) \quad \rightarrow \quad (s', \ell'_1, \ldots, \ell'_{n'})$$

such that one of the following conditions holds:

1) $n' = n$ and there exists $i \in \{1, \ldots, n\}$ such that (i) the statement at $\ell_i.pc$ is a *seqstmt*; executing it *atomically* from the variable valuation given by (s, ℓ_i) results in the variable valuation given by (s', ℓ'_i), and (ii) for $j \in \{1, \ldots, n\} \setminus \{i\}$, $\ell'_j = \ell_j$.
2) $n' = n + 1$, $s' = s$ and there exists $i \in \{1, \ldots, n\}$ such that (i) the statement at $\ell_i.pc$ is of the form `start_thread` x, (ii) $\ell'_i.pc = \ell_i.pc + 1$ and $\ell'_i.v =$

[1] http://www.cprover.org/boolean-programs/grammar.pdf

$\ell_i.v$ for $v \in V_L$, (iii) $\ell'_{n'}.pc = x$ and $\ell'_{n'}.v = \ell_i.v$ for $v \in V_L$, and (iv) for every $j \in \{1,\ldots,n\} \setminus \{i\}$, $\ell'_j = \ell_j$.

3) $n' = n$, $s' = s$ and there exists $i \in \{1,\ldots,n\}$ such that (i) the statement at $\ell_i.pc$ is `broadcast`, (ii) $\ell'_i.pc = \ell_i.pc + 1$ and $\ell'_i.v = \ell_i.v$ for $v \in V_L$, (iii) for every $j \in \{1,\ldots,n\} \setminus \{i\}$ such that the statement at $\ell_j.pc$ is **wait**, $\ell'_j.pc = \ell_j.pc + 1$ and $\ell'_j.v = \ell_j.v$ for $v \in V_L$, and (iv) for every $j \in \{1,\ldots,n\} \setminus \{i\}$ such that the statement at $\ell_j.pc$ is *not* **wait**, $\ell'_j.pc = \ell_j.pc$ and $\ell'_j.v = \ell_j.v$ for $v \in V_L$.

In each case, thread i is called *active*, the others *passive*. We omit the precise formalization of **atomic** blocks, which is, however, straightforward. The initial states of M^∞ are given by (i) $n = 1$ and (ii) s and ℓ_1 determined by the (nondeterministically) initializing declarations in $\mathcal{B}$ and by $\ell_1.pc = 1$.

Transition system M^∞ thusly defined is a *well quasi-ordered transition system* (WQOS) [2]. That is, there exists a well-quasi order $\succeq$ on S^∞ that satisfies a *monotonicity* property: for states x, y, x' with $x \to x'$ and $y \succeq x$, we can find y' such that $y' \succeq x'$ and $y \to y'$. This order is the *covers* relation:

$$(\bar{s}, \bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}) \succeq (s, \ell_1, \ldots, \ell_n)$$

whenever $\bar{s} = s$ and $[\bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}] \supseteq [\ell_1, \ldots, \ell_n]$, where $[\cdot]$ denotes a *multiset*. The well-quasi orderedness follows from properties of $\supseteq$ and Dixon's lemma; the monotonicity of $\to$ with respect to $\succeq$ follows since actions of a thread in a state cannot be disabled by adding threads to the state; see semantics of M^∞. These are standard concepts.

The multi-threaded assertion violation question can now be phrased as a *coverability problem* for the derived WQOS M^∞: let Q be the set of (shared, local) state pairs (s, ℓ) such that the statement at $\ell.pc$ is an assertion that is violated by the variable valuation given by (s, ℓ). Coverability of the "bad-states set" Q asks whether a state z is reachable such that, for some $q \in Q$, $z \succeq q$. Coverability is decidable but of high complexity, e.g. *Ackermann*-complete for Petri nets with broadcasts (a form of WQOS), which means that the complexity grows as fast as the Ackermann function [6].

C. Backward Search

A sound and complete algorithm to decide coverability for WQOS is the *backward search* algorithm by Abdulla et al. [2], [7], a high-level version of which is shown in Alg. 1. In this listing, symbol $\uparrow U$ stands for the *upward closure* of U: $\uparrow U = \{\bar{u} : \exists u \in U : \bar{u} \succeq u\}$. Input to Alg. 1 is a set of initial states $I \subseteq S^\infty$, and a target set $Q \subseteq S^\infty$. The algorithm maintains a work set $W \subseteq S^\infty$ of unprocessed states, and a set $U \subseteq S^\infty$ of minimal encountered states. It successively computes minimal *cover predecessors*

$$\mathsf{CovPre}(w) = \min\{p : \exists \bar{w} \succeq w : p \to \bar{w}\} \qquad (1)$$

starting from elements in Q, and terminates either by backward-reaching an initial state (thus proving coverability of some $q \in Q$), or when no unprocessed vertex remains (thus proving uncoverability).

Algorithm 1 BWS(I, Q)

Input: initial states I, target set Q disjoint from I

```
1: W := Q ; U := Q
2: while ∃w ∈ W
3:     W := W \ {w}
4:     for p ∈ CovPre(w) \ ↑U
5:         if p ∈ I then
6:             "some q ∈ Q coverable"
7:         W := min(W ∪ {p})
8:         U := min(U ∪ {p})
9: "no q ∈ Q coverable"
```

III. Boolean Program Backward Search: Overview

We illustrate our approach using the Boolean program $\mathcal{B}$ in Fig. 2 (left). The program is started by one thread; the nondeterministic `goto` in Line 1 determines whether to launch an additional thread in Line 2. Suppose not (we proceed in Line 3). Then the left branch starting at the node corresponding to Line 4 (Fig. 2, right) is not executable since $t = 0$ violates `assume(`t`)`. Along the right branch, local variable m is not modified, `assume(!`t`)` passes, and so does the `assert(!`m`)` in Line 10.

```
decl t := ⋆;
main() {
      decl m := 0;
  1:  goto 2,3;
  2:  start_thread 3;
  3:  t := 0;
  4:  goto 5,8;
  5:  assume(t);
  6:  m := 1;
  7:  goto 9;
  8:  assume(!t);
  9:  t := !t;
 10:  assert(!m);
}
```

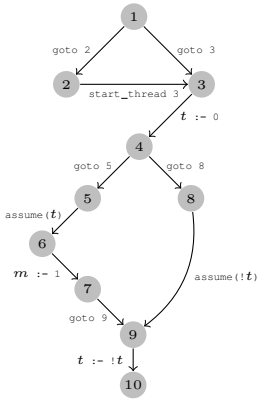

Fig. 2: A Boolean program with $V_S = \{t\}$, $V_L = \{m\}$ (left); its control flow graph (right)

Nonetheless, program $\mathcal{B}$ permits an assertion violation, as the backward trace in Fig. 3 shows. The target set is $\{(0|^{10}/_1), (1|^{10}/_1)\}$; the trace shown starts from $(0|^{10}/_1)$. Our backward search algorithm proceeds from a given state w in two steps: (i) we select a thread in w as active and compute all direct predecessors that $\mathcal{B}$ permits; (ii) we try to find *expanded* predecessors of w, explained below.

Let us first consider a direct predecessor example, using state $(1|^7/_1)$ in the first row. The algorithm first consults the control flow graph, shown in Fig. 2 (right), for possible control predecessors of $pc' = 7$. There is only one, $pc = 6$. The statement along this edge is m `:= 1`. We therefore now compute the weakest precondition of state $t = 1, m = 1$ under this statement, i.e.

$$\mathsf{WP}_{m\ :=\ 1}(t \wedge m) \quad = \quad t\ ,$$

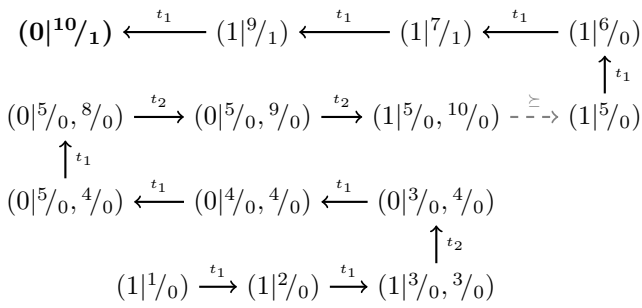

Fig. 3: Coverability analysis using backward search, applied to the program in Fig. 2 (left). Targets are the states (s, ℓ) satisfying $\ell.pc = 10, \ell.m = 1$. Notation $(t_0|^{pc_1}/_{m_1}, \ldots)$ denotes a global state with shared variable $t = t_0$ and the given values for the local variables pc and m, for the various threads. Labels atop transitions indicate the active thread; $\succeq$ indicates expansion

indicating $\{(1|^6/_0), (1|^6/_1)\}$ as the set of direct predecessors. Both are recorded in our algorithm; the trace shown in Fig. 3 continues with state $(1|^6/_0)$.

Direct predecessor computation alone will never increase the number of involved threads and thus cannot detect multi-threaded assertion violations. Alg. 1 involves a step we call *expansion* of w to a larger state $\bar{w}$, by adding to w a thread in a suitable local state not present in w. Expanded predecessors of w are then the (minimal) direct predecessors of $\bar{w}$, obtained by backward-executing the added thread. Sections IV and V present the details of this step, especially that adding a single thread to w is sufficient, what a "suitable" local state is, and that direct and expanded predecessors constitute exactly the set of all cover predecessors of w (Eq. (1)).

Consider the example of state $(1|^5/_0)$ in Fig. 3. The only direct predecessor is $(1|^4/_0)$, from which there is no further direct predecessor: the only CFG edge entering Line 4 is labeled with statement t := 0, which does not permit $t = 1$ in the current state. We thus try to expand. As we will see, expansion is only useful if the added thread gives rise to a predecessor *with a modified shared state*. Only few statements in $\mathcal{B}$ change t; one is t := !t in Line 9. Expansion therefore adds a thread with $pc = 10$, namely in local state $^{10}/_0$. The direct global predecessor state $(0|^5/_0, ^9/_0)$ has changed t by backward-executing t := !t. From now on the backward search proceeds with two threads until we encounter the `start_thread` command; backward-executing it eliminates thread 2. At the end, the search reaches the initial state $(1|^1/_0)$, proving reachability of the violated assertion.

IV. Boolean Program Backward Search

This section presents our infinite-state backward search algorithm, applied to an unbounded-thread Boolean program $\mathcal{B}$.

A. Data Structures and Prerequisites

While exploring $\mathcal{B}$, the algorithm builds — on the fly — the infinite-state structure M^∞. States $\tau = (s, m_1, \ldots, m_n)$ of this structure are stored in the form

$$\tau = \langle s, \{(\ell_1, n_1), \ldots, (\ell_k, n_k)\}\rangle \quad (2)$$

where $\ell_1, \ldots, \ell_k$ are the *distinct* local states occurring in τ, and for $i \in \{1, \ldots, k\}$, $n_i = |\{j : m_j = \ell_i\}|$. That is, instead of listing the local states of all n threads in τ, (2) collapses multiple occurrences of local states and lists their count. Threads in the same local state are equivalent under standard symmetry equivalence; their order in τ and their identities are immaterial. By construction, $n_i > 0$ for all i.

The algorithm assumes the control flow graph (CFG) of $\mathcal{B}$ is given as $G = (\{1, \ldots, pc_{\max}\}, E)$. The CFG is a directed graph over the program locations of $\mathcal{B}$. Each edge $e \in E$, with source and target $source(e)$ and $target(e)$, resp., is labeled with the statement $e.stmt$ of $\mathcal{B}$ that carries the control from $source(e)$ to $target(e)$. For example, Line 1 of the program in Fig. 2 (left) induces **two** edges in G, as shown on the right.

For a statement $stmt$ and states (s, ℓ) and (s', ℓ') of $\mathcal{B}$, let $\mathsf{WP}_{stmt}(s, \ell, s', \ell')$ be a Boolean formula asserting that state (s, ℓ) satisfies the *weakest precondition* for state (s', ℓ') under statement $stmt$. That is, $\mathsf{WP}_{stmt}(s, \ell, s', \ell')$ holds exactly if executing $stmt$ from state (s, ℓ) results in state (s', ℓ').[2] Examples for sequential statements are given in Table I.

Statement $stmt$	$\mathsf{WP}_{stmt}(s, \ell, s', \ell')$
`skip`	$\ell'.pc = \ell.pc + 1 \wedge invar$
`assume` $(t = m)$	$s.t = \ell.m \wedge \ell'.pc = \ell.pc + 1 \wedge invar$
t := $\star$	$\ell'.m = \ell.m \wedge \ell'.pc = \ell.pc + 1$

TABLE I: Examples for weakest precondition formulas for a program $\mathcal{B}$ with $V_S = \{t\}$ and $V_L = \{m\}$. Symbol $invar$ stands for "data invariance": $s'.t = s.t \wedge \ell'.m = \ell.m$

B. Cover Predecessor Computation

Our algorithm is an instance of the high-level scheme shown in Alg. 1. The only (albeit substantial) modification is the computation of the cover predecessor function in Line 4. The definition of this function, Eq. (1), poses the following challenges for an implementation:

1) given w, expanded elements $\bar{w} \succeq w$ need to be explored;
2) given $\bar{w}$, a preimage needs to be computed.

Regarding 2), our algorithm will use the CFG of $\mathcal{B}$ and weakest precondition transformers to compute preimages of states. In order to meet challenge 1), we need an "upper bound" on the expanded elements $\bar{w}$. This is accomplished by the following lemma. We denote by $|w|$ the number of threads in state $w \in S^\infty$, e.g. $|(s, \ell_1, \ldots, \ell_n)| = n$.

Lemma 1. *For a transition relation $\rightarrow$ induced by a Boolean broadcast program, and states $p, w, \bar{w} \in S^\infty$,*

$$p \in \mathit{CovPre}(w) \wedge \bar{w} \succeq w \wedge p \rightarrow \bar{w} \quad \Rightarrow \quad |\bar{w}| \leq |w| + 1.$$

Proof: Since p is a *minimal* cover predecessor of w, there are no states $o \prec p$ and $\bar{v}$ such that $\bar{v} \succeq w$ and $o \rightarrow \bar{v}$. Note that $o \prec p$ abbreviates $o \preceq p \wedge \neg(o \succeq p)$.

[2]We note that all atomic (non-compound) statements of $\mathcal{B}$ are terminating.

We prove $|\bar{w}| \leq |w| + 1$ via contraposition: assume $|\bar{w}| \geq |w|+2$. From this assumption and $\bar{w} \succeq w$, we conclude that $\bar{w}$ contains, in some permutation, all $|w|$ local states that w also contains, and at least two more local states, say at positions j_1 and j_2 with $j_1 \neq j_2$. Let i be the index of the thread active during transition $p \rightarrow \bar{w}$. We observe that, since $j_1 \neq j_2$, there exists $j \in \{j_1, j_2\}$ such that $j \neq i$. Let now o and $\bar{v}$ be the same states as p and $\bar{w}$, respectively, except that thread j is dropped. Then: (i) $o \prec p$; (ii) $o \rightarrow \bar{v}$, because thread j is *not* active in $p \rightarrow \bar{w}$, and dropping j does not invalidate the transition; and (iii) $\bar{v} \succeq w$, since $\bar{w} \succeq w$ and $|\bar{w}| \geq |w| + 2$, and we have dropped only one thread from $\bar{w}$ to obtain $\bar{v}$. Properties (i)–(iii) contradict the minimality of p, as stated at the beginning of the proof. ■

We now turn to the main result in this paper: the procedure for cover predecessor computation (Alg. 2). Input is a state τ' in format (2) (we attach a prime $'$ to the input symbols to suggest that we are computing preimages). The results are collected in a set $\mathcal{C}$.

The computation of cover predecessors according to Eq. (1) involves finding an element $\bar{w}$ satisfying $\bar{w} \succeq w$, and then determining predecessors of $\bar{w}$. Condition $\bar{w} \succeq w$ is tantamount to $\bar{w} \preceq\succeq w \vee \bar{w} \succ w$ (where $\bar{w} \preceq\succeq w$ means *equivalence*: $\bar{w} \succeq w \wedge w \succeq \bar{w}$). The algorithm deals with the two cases $\bar{w} \preceq\succeq w$ and $\bar{w} \succ w$ separately, as follows.

1) Direct predecessors: Condition $\bar{w} \preceq\succeq w$ means that there exists a thread permutation π such that $w = \pi(\bar{w})$ (π reorders threads in the local state vector representation of $\bar{w}$). By thread symmetry, w and $\bar{w}$ therefore have the same sets of $\rightarrow$ predecessors, up to applying local state permutations. Now observe that states that are identical up to local state permutations have the same thread counter representation (2). This means that, in the case $\bar{w} \preceq\succeq w$, we can ignore the "detour" through $\bar{w}$ and directly compute predecessors of w: those are the elements of $\mathsf{CovPre}(w)$. Naturally, we call such elements *direct predecessors*.

To compute direct predecessors, Alg. 2 iterates through all local states ℓ'_i (thus, implicitly, threads) present in τ'. It then consults the CFG for edges e leading to the current program location $\ell'_i.pc$ of any of the threads in ℓ'_i (Lines 2 and 3). Reversing edge e, i.e. executing it backwards on (s', ℓ'_i), gives us the desired predecessors.

To this end, the algorithm switches over the possible types of statement $e.stmt$:

`start_thread` x: this is possible exactly if the current state τ' contains a "started" thread in some local state ℓ'_j with $\ell'_j.pc = x$ and same data as the thread in ℓ'_i (Line 6). If so, in the predecessor state τ, the thread in ℓ'_i is unchanged except that its pc is the previous program location (Line 7; notation $\ell'_i[pc \mapsto Y]$ returns ℓ'_i except pc replaced by Y). To construct τ, we update the thread counters: those for ℓ'_i and ℓ'_j are decremented; that of the predecessor local state ℓ_i is incremented (the thread in ℓ'_j has just been created, so going backwards it "disappears"). These updates are done in Line 8 via a function UPDATE-COUNTERS explained below. The updates are performed for all eligible local states ℓ'_j; the results are added to $\mathcal{C}$.

`broadcast`: since broadcasts are non-blocking, they are executable from any predecessor state. The broadcasting thread's pc is decreased by one; the change is recorded in a temporary variable Y. Line 13 selects all threads with program counter pc such that the statement at $pc - 1$ is `wait`: these threads *may* have just been released by the broadcast. However they may also have resided in location pc *before* the broadcast was issued — the exact subset J of indices of threads that are released cannot be determined when exploring $\mathcal{B}$ backwards.

Therefore, the algorithm iterates through all such sets $J \subseteq \mathcal{J}$ and threads $j \in J$ (Line 16): these threads are "unreleased", i.e. their pc is set back to the `wait` location. The updates to Z in Lines 18–19 (see Alg. 4 for the MERGE function) perform counter updates for the synchronous state change of all unreleased threads.

default: this case takes care of all sequential statements: using the weakest precondition function WP, we generate all possible predecessor program states (s, ℓ), update the counters, and add the results τ to $\mathcal{C}$. Solving the Boolean formula $\mathsf{WP}_{e.stmt}(s, \ell, s', \ell'_i)$ for (s, ℓ) is done with the aid of a SAT solver (see Sect. VI).

Note that the **switch** in Line 4 does not process `wait` statements: these cannot backward-execute by themselves, as releasing waiting threads happens in synchrony with broadcasts.

2) Expanded predecessors: We now consider the case $\bar{w} \succ w$. We have to expand state w by adding threads to it, followed by the computation of predecessors of the expanded state $\bar{w}$. The following observations render this step feasible:

- by Lemma 1, adding a **single** thread to w is sufficient;
- when computing predecessors p of $\bar{w}$, those obtained when the single added thread is active are sufficient: predecessors triggered by threads already present in w are handled as direct predecessors of w.

Alg. 2 implements the expanded predecessor computation along those principles. In Line 25 we determine states (s, ℓ) of $\mathcal{B}$ such that there exists a local state m' (= that of the added thread) not present in τ' such that the following holds: (i) the pc values of ℓ and m' form an edge $e \in E$, and (ii) executing $e.stmt$ from (s, ℓ) leads to (s', m').

The solutions to the constraint in Line 25 are determined using a SAT solver, which is passed the constraint as an existentially quantified Boolean formula. In this formula, all of $s', \ell'_1, \ldots, \ell'_k$ are Boolean constants; the only variables are s, ℓ (free) and m' (quantified). The solutions (s, ℓ) we are looking for are the assignments satisfying this formula, projected to s and ℓ. To avoid excessive enumeration, we discuss in Sect. V how the selection of candidate local states m' for expansion can be substantially and soundly restricted.

We conclude this algorithm description by explaining function UPDATE-COUNTERS(T, T', Z'), shown in Alg. 3. It increments/decrements the counters for all local states in T/T', using the MERGE function from Alg. 4.

Algorithm 2 CovPre(τ')

Input: $\tau' = \langle s', Z' \rangle$, where $Z' = \{(\ell'_1, n'_1), \ldots, (\ell'_k, n'_k)\}$
Output: cover predecessors of τ'

1: $\mathcal{C} := \emptyset$
2: **for each** $i \in \{1, \ldots, k\}$ ▷ direct predecessors
3: **for each** $e \in E$ s.t. $target(e) = \ell'_i.pc$
4: **switch** $e.stmt$:
5: **case** `start_thread` x, for some x**:**
6: **for each** $j \in \{1, \ldots, k\} \setminus \{i\}$ s.t. $\ell'_j.pc = x \wedge \forall v \in V_L : \ell'_j.v = \ell'_i.v$
7: $\ell_i := \ell'_i\,[\,pc \mapsto \ell'_i.pc - 1\,]$
8: $\tau := \langle s', \textsc{Update-Counters}(\{\ell_i\}, \{\ell'_i, \ell'_j\}, Z') \rangle$
9: $\mathcal{C} := \mathcal{C} \cup \{\tau\}$
10: **case** `broadcast`**:**
11: $\ell_i := \ell'_i\,[\,pc \mapsto \ell'_i.pc - 1\,]$
12: $Y := \textsc{Update-Counters}(\{\ell_i\}, \{\ell'_i\}, Z')$
13: $\mathcal{J} := \{j \in \{1, \ldots, k\} \setminus \{i\}$ s.t. stmt. at $\ell'_j.pc - 1$ is `wait`$\}$
14: **for each** $J \subseteq \mathcal{J}$
15: $Z := Y$
16: **for each** $j \in J$
17: $\ell_j := \ell'_j\,[\,pc \mapsto \ell'_j.pc - 1\,]$
18: $Z := Z \setminus \{(\ell'_j, n'_j)\}$
19: $\textsc{Merge}(\ell_j, n'_j, Z)$
20: $\mathcal{C} := \mathcal{C} \cup \{\langle s', Z \rangle\}$
21: **default:**
22: **for each** (s, ℓ) s.t. $\mathsf{WP}_{e.stmt}(s, \ell, s', \ell'_i)$
23: $\tau := \langle s, \textsc{Update-Counters}(\{\ell\}, \{\ell'_i\}, Z') \rangle$
24: $\mathcal{C} := \mathcal{C} \cup \{\tau\}$
25: **for each** (s, ℓ) s.t. $\exists m' \notin \{\ell'_1, \ldots, \ell'_k\} : \; e := (\ell.pc, m'.pc) \in E \;\wedge\; \mathsf{WP}_{e.stmt}(s, \ell, s', m')$ ▷ expanded predecessors
26: $\tau := \langle s, \textsc{Update-Counters}(\{\ell\}, \emptyset, Z') \rangle$
27: $\mathcal{C} := \mathcal{C} \cup \{\tau\}$
28: **return** $\mathcal{C}$

Algorithm 3 Update-Counters(T, T', Z')

Input: T: local states whose counter is to be incremented
T': local states whose counter is to be decremented
Z': thread counter vector

1: **for each** $\ell \in T$
2: $\textsc{Merge}(\ell, 1, Z')$
3: **for each** $\ell' \in T'$
4: **let** n' be such that $(\ell', n') \in Z'$ ▷ n' is unique
5: $Z := Z' \setminus \{(\ell', n')\} \cup (\; n' > 1 \;?\; \{(\ell', n' - 1)\} : \emptyset \;)$
6: **return** Z

Algorithm 4 Merge(ℓ, n, Z)

Input: ℓ: local state, n: counter, Z: thread counter vector

1: **if** there exists n' such that $(\ell, n') \in Z$ **then**
2: $Z := Z \setminus \{(\ell, n')\} \cup \{(\ell, n' + n)\}$
3: **else**
4: $Z := Z \cup \{(\ell, n)\}$

V. Efficiency

In Sect. IV-B2 we saw two specializations of the generic backward coverability Alg. 1 that apply to the computation of expanded cover predecessors for Boolean programs. In particular, the bound on the size of expanded states $\bar{w}$ makes CovPre(w) effectively computable. In this section we describe two improvements that are essential to, among others, curb the number of candidate local states m' in Line 25.

The first improvement is that m' can be restricted such that the statement along edge $e = (\ell.pc, m'.pc)$ changes the shared state. The justification is as follows. Suppose $e.stmt$ does not change the shared state, i.e. $s = s'$. The cover predecessor state τ is thus of the form $(s', \ldots, \ell, \ldots)$. Any cover predecessor of τ that is obtained by backward executing $\mathcal{B}$ from program state (s', ℓ) is also a cover predecessor of the original τ': we simply expand τ' by local state ℓ instead of m'.

This improvement is easy to implement: we change Line 25 in Alg. 2 to the following *two* lines:

for each $e \in E$ s.t. $e.stmt$ may modify the shared state
for each (s, ℓ) s.t. $s \neq s' \;\wedge\; \exists m' \notin \{\ell'_1, \ldots, \ell'_k\} :$
$target(e) = m'.pc \;\wedge\; \mathsf{WP}_{e.stmt}(s, \ell, s', m')$
$\tau := \ldots$ ▷ (continue with Line 26 of Alg. 2)

That is, we first select edges e that have the *potential* to change the shared state. Such are edges that assign a variable

in V_S; they can be identified inexpensively up front, while building the CFG. We then determine states (s, ℓ) of $\mathcal{B}$ such that (i) shared state s is actually different from s', and (ii) there exists a local state m' not present in τ' whose pc is the target of e and such that $\mathsf{WP}_{e.stmt}(s, \ell, s', m')$.

This improvement is also highly effective: only few of the syntactically possible statements may actually change the shared state, and their frequency in Boolean programs is proportionately small, as we demonstrate in Table II (Sect. VI).

The second improvement exploits that local states m' that are not *forward-reachable* from an initial state of M^∞ can be omitted, since they obviously are not part of any legal execution. While the exact determination of reachability of a local state is of course a coverability problem in itself, we can employ very inexpensive overapproximating analyses that soundly provide *unreachability* information.

One such analysis, adapted from [8], is to execute $\mathcal{B}$ essentially as a single-threaded program. That is, statements related to multi-threading are ignored (`start_thread` in particular). Sequential statements are honored, except that:

1) assignments to shared variables are ignored
2) conditionals that depend on shared variables are replaced by $\star$, i.e. in `assume`, `constrain`, and `if` statements.

The set $\mathcal{L}$ of local states reachable in this single-threaded program is cheap to compute and overapproximates the precise set of local states reachable in the multi-threaded execution of $\mathcal{B}$. That is, local states not in $\mathcal{L}$ are unreachable. We exploit this information by adding the requirement $m' \in \mathcal{L}$ to the second **for each** statement in the above modification to Alg. 2.

In fact, this insight not only applies to the selection of states m' during expanded predecessor computation: we can simply ignore local states generated during the backward search (Lines 7, 11, 17, 22 in Alg. 2) that do not belong to $\mathcal{L}$. This technique can be seen as an instance of combining forward and backward analysis for increased efficiency (executing $\mathcal{B}$ is tantamount to forward reachability analysis). This idea was used in several other works, such as [9], where an incomplete forward-searching Karp-Miller procedure assists a slower but complete backward coverability analysis. Incidentally, the Karp-Miller implementation used in [9] may not terminate and may thus *underapproximate* the set of coverable configurations when applied to broadcast nets. In contrast, we need an overapproximation, as our goal is to prune encountered states that are guaranteed to be **un**reachable.

VI. Empirical Evaluation

In this section, we evaluate our verifier Ucob[3] on a set of 30 non-recursive concurrent C programs. Threads synchronize through diverse communication primitives, such as shared variables, mutex variables, and broadcasts. All programs contain procedures executed by an arbitrary number of threads, which are dynamically spawned by the initial thread.

[3] "**U**nbounded-thread **co**verability analysis for **b**oolean programs"

For each benchmark, we consider verification of a safety property, specified via an assertion. In total, the programs include roughly 2300 lines of code; on average they feature 3 shared and 6 local variables (cf. Table II). The programs are available from the homepage of the second author.

***01–10: thread-safe algorithms*:** atomic counters (1–2); concurrent pseudo-random number generator (3–4); maximum element finding algorithm (5–8); stack data structure with concurrent operations (9-10).

***11–17: OS code*:** code from the FreeBSD (11–12), NetBSD (13), Solaris (14) and Linux (15–17) open-source operating systems.

***18–22: pthread programs*:** several programs that use the C Posix Threads library.

***23–28: mutex algorithms*:** test-and-set lock (23); multiple locks control access to a shared resource (24–26); two ticket algorithms (27–28).

***29–30: misc*:** two simple examples from [1].

Implementation: Ucob uses SatAbs [10] to construct Boolean programs from C. To compare with coverability tools that don't accept Boolean programs as input, we also use SatAbs to generate thread transition system (TTS) models as input (option `--build-tts`). Finally, we use miniSAT [11] to solve WP formulas. Ucob offers both optimizations presented in Sect. V: the shared-variable restriction, and the single-thread forward-reachable local states computation. The latter employs the tool Boom [12], which we call a "forward oracle" in this context (a terminology suggested in [9]).

The experiments are performed on a 2.3GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux.

Comparison: For $1 \leq k \leq 30$, Fig. 4 plots the total time (log-scale) taken to solve the k easiest of our benchmark problems, for the following tools:

Ucob: our tool with shared-variable optimization;
Ucob/Boom: Ucob with Boom as forward oracle;
Mcov: Mcov without forward oracle [9];
Mcov/Gkm: Mcov with forward oracle [9];
Boom-Km: Karp-Miller implementation in Boom [12].

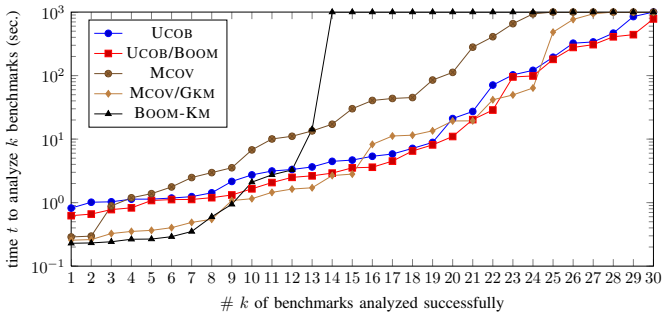

Fig. 4: Cactus plot comparing Ucob with other coverability tools. For each curve, entry (k, t) shows the time t it took to solve the k easiest — for the method associated with that curve — benchmarks (order varies across methods).

TABLE II: Benchmark characteristics and results: SV / LV / *LOC* = # of shared / local C program variables / lines of code; *Mtx?* / *Bc?* = presence of mutex vars / broadcasts (● = "yes"); $|V_S|$ / $|V_L|$ / *Its.* = # of shared / local Boolean variables / CEGAR iterations; *Mod.Sh.* = percentage of statements that may modify the shared state; *Safe?* = program safety

ID/Program	C Program					Boolean Program				*Safe?*
	SV	*LV*	*LOC*	*Mtx?*	*Bc?*	$\|V_S\|$	$\|V_L\|$	*Its.*	*Mod.Sh.*	
01/INC-L	2	1	46	●	○	3	1	2	7.5	●
02/INC-C	1	3	57	○	○	0	4	4	0	●
03/PRNSIMP-L	2	4	63	●	○	2	3	2	7.7	●
04/PRNSIMP-C	1	5	95	○	○	0	5	2	0	●
05/MAXSIM-L	3	3	59	●	○	1	0	2	3.7	●
06/MAXSIM-C	2	5	79	○	○	0	1	2	0	●
07/MAXOPT-L	3	4	69	●	○	1	1	2	3.1	●
08/MAXOPT-C	2	6	86	●	○	0	2	2	0	●
09/STACK-L	4	2	79	●	○	1	3	3	3.8	●
10/STACK-C	3	3	89	○	○	3	1	2	6.4	●
11/BSD-AK	1	7	90	●	●	3	1	15	11.7	●
12/BSD-RA	2	21	87	●	●	3	0	19	12.3	●
13/NETBSD	1	28	152	●	●	3	1	30	10.1	●
14/SOLARIS	1	56	122	●	●	5	1	14	10.9	●
15/BOOP	5	2	89	○	○	5	2	4	11.4	○
16/QRCU-2	7	6	120	○	○	3	0	16	10.1	●
17/QRCU-4	8	8	182	○	○	5	2	28	9.8	●
18/BS-LOOP	0	6	24	○	○	0	7	1	0	○
19/COND	1	3	56	●	○	0	3	2	0	●
20/FUNC-P	2	1	67	●	○	2	6	3	8.3	●
21/S-LOOP	5	0	60	●	○	4	0	20	22.8	●
22/PTHREAD	5	0	85	●	○	7	0	5	17.1	○
23/TAS-L	2	2	58	○	○	3	1	2	14.9	●
24/DOUBLE-1	3	0	70	●	○	7	1	10	16.4	●
25/DOUBLE-2	3	0	73	●	○	6	1	23	18.2	●
26/DOUBLE-3	3	0	66	●	○	4	1	3	15.3	●
27/TICKET-HC	3	1	61	○	○	5	1	5	18.4	●
28/TICKET-LO	3	1	46	○	○	5	1	5	20.8	●
29/UNVEREX	2	1	25	○	○	4	0	3	8.9	●
30/SPIN	2	0	37	●	○	3	0	2	15.5	●

The results in the chart demonstrate that UCOB solves almost all of the benchmarks (29), and does so in less time for most programs. Furthermore, the results for UCOB/BOOM show that the forward oracle, despite being an overapproximation, can accelerate the backward search by pruning unreachable cover predecessors. MCOV/GKM is the most competitive proof tool. For the small-scale benchmarks, where the TTS construction does not blow up, it takes the least amount of time. However, the efficiency drops sharply with increasing cost of either TTS generation or verification. The only other unbounded-thread on-the-fly verifier we are aware of, BOOM-KM, benefits from forward search and is thus competitive "early", but reports runtime errors for some of the more complex benchmarks. Others it cannot solve: the Karp-Miller implementation in BOOM-KM does not support broadcasts.

VII. Related Work and Concluding Remarks

The issue of the blow-up incurred when translating a program model $\mathcal{B}$ into a transition system model M is classically addressed using an on-the-fly exploration. In the context of symmetric concurrent systems, things are more complicated as the transition system (a WQOS) does not model $\mathcal{B}$ directly, but via a counting abstraction. In [13], this issue was addressed for the finite-state case, by interleaving the counting abstraction and transition system construction. We have borrowed the state representation (2) and (mostly) the counter update function UPDATE-COUNTERS (Alg. 3) from that work.

We are aware of very few attempts to address the issue in connection with (much more complex) *infinite-state* verification techniques. While these techniques have been applied to programs directly [14], [15], the application is typically preceded by a static compilation of the program into an explicit transition system, which only works for small local state spaces, for example when predicates used for abstraction are hand-picked.

An on-the-fly implementation of the Karp-Miller algorithm is available in BOOM [12]. This algorithm proceeds forward, making the implementation much easier. On the other hand, due to theoretical limitations of Karp-Miller (see e.g. [4]), this tool cannot handle broadcast programs, our target language. On non-broadcast programs, it suffers from the notoriously high space complexity of the Karp-Miller procedure. We have compared against this tool in Sect. VI.

Recent research has established that, for the rich class of Boolean broadcast programs, forward search tends to be efficient but incomplete (or unsound), while backward search guarantees correctness but lags behind. The solution is to combine both searches, which we have done here in a somewhat shallow fashion. A goal for future work is therefore to implement our on-the-fly strategy directly on an advanced WQOS coverability algorithm such as [9].

References

[1] A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, "Counterexample-guided abstraction refinement for symmetric concurrent programs," *Formal Methods in System Design*, 2012.

[2] P. A. Abdulla, "Well (and better) quasi-ordered transition systems," *Bulletin of Symbolic Logic*, 2010.

[3] R. M. Karp and R. E. Miller, "Parallel program schemata," *J. Comput. Syst. Sci.*, 1969.

[4] J. Esparza, A. Finkel, and R. Mayr, "On the verification of broadcast protocols," in *LICS*, 1999.

[5] B. Cook, D. Kroening, and N. Sharygina, "Symbolic model checking for asynchronous Boolean programs," in *SPIN*, 2005.

[6] P. Schnoebelen, "Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets," in *MFCS*, 2010.

[7] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *LICS*, 1996.

[8] A. Emerson and T. Wahl, "Efficient reduction techniques for systems with many components," *Electr. Notes Theor. Comput. Sci.*, 2005.

[9] A. Kaiser, D. Kroening, and T. Wahl, "Efficient coverability analysis by proof minimization," in *CONCUR*, 2012.

[10] G. Basler, A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, "SATABS: A bit-precise verifier for C programs," in *TACAS*, 2012.

[11] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.

[12] G. Basler, M. Hague, D. Kroening, L. Ong, T. Wahl, and H. Zhao, "Boom: Taking Boolean program model checking one step further," in *TACAS*, 2010.

[13] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Context-aware counter abstraction," *Formal Methods in System Design*, 2010.

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

[15] G. Delzanno, J.-F. Raskin, and L. V. Begin, "Towards the automated verification of multithreaded Java programs," in *TACAS*, 2002.