Accelerating Backward Coverability Analysis using Symbolic Pruning

Peizun Liu
Northeastern University, Boston, United States

Abstract. We present a method for accelerating explicit-state backward search algorithm for thread transition systems, a model can be naturally derived from programs running by arbitrarily many finite-state threads. The idea is to prune uncoverable states and their cover predecessors using an approximate coverability analysis. The later is built atop thread-state equations. We demonstrate performance gain of our acceleration method applied to backward search for proving and refuting safety properties of an extensive set of benchmarks.

1 Introduction

We target in this paper unbounded-thread programs where each thread executes a non-recursive finite-state procedure and communicates via shared memory. This model is popular, as it connects to multi-threaded programs via predicate abstraction [4], a technique that has enjoyed progress for multi-threaded programs in the past few years [6]. The model is also popular since basic program state reachability is decidable. They are also, however, of high complexity: the equivalent coverability problem for Petri nets was shown to be EXPSPACE hard [5]. The motivation for our work is therefore to improve the efficiency of existing algorithms.

A sound and complete method for coverability analysis for well quasi-ordered systems (WQOS) is the backward search algorithm by Abdulla [1]. Coverability for WQOS subsumes state reachability analysis for a wide class of multi-threaded Boolean programs. Starting from the target state whose reachability is under investigation, the algorithm proceeds backward by computing cover predecessors, until either an initial state is reached, or a fixpoint.

In this paper we propose a method for accelerating explicit-state backward search algorithms like Abdulla’s. The intuition of acceleration is straightforward: whenever a state turns out uncoverable, so are all its cover predecessors. This state therefore is a bad choice and should never have been expanded. Our method hence prunes the entire tree rooting at that state. The remaining question is: given a state, how to determine its uncoverability? Is it the very goal of the backward search?

The resolution of the paradox is that we do not need to compute the uncoverability precisely and neither to prune all uncoverable states. An approximate uncoverability analysis is typically sufficient to help us prune a large number of uncoverable states and hence accelerate the backward search dramatically. To this end, we equip the backward search with a symbolic uncoverability proof engine. The later can in turn be obtained from thread-state equations [3], a a recent SMT based coverability analysis.
We conclude this paper with experiments on an extensive set of benchmarks that investigate the performance gain of our acceleration method applied to backward search. The results showcase that our method dramatically improves the efficiency of backward search on safe examples. It also showcases its success on unsafe examples with large sizes. An extensive performance comparison with existing tools is left for future work.

2 Setting the Stage

We assume multi-threaded programs are given in the form of an abstract state machine called thread-transition system (TTS) [9]. Such a system reflects the replicated nature of the programs we consider: those consisting of threads executing a given procedure defined over shared and local variables. A TTS is a tuple $P = (T, R)$, where

- $T \subseteq S \times L$ is a finite set of thread states, with $S/L$ are finite sets of shared/local states.
- $R \subseteq T \times T$ is a finite set of transitions.

A TTS gives rise to a family, parameterized by $n$, of transition systems $P_n = (T_n, R_n)$ over the state space $T_n = S \times L^n$, whose states are written as $\tau = (s|l_1, \ldots, l_n)$. This notation defines a global system state with shared component $s$, and $n$ threads in local states $l_i$ for $i \in \{1, \ldots, n\}$. The transitions of $P_n$, forming the set $R_n$, are written as $(s|l_1, \ldots, l_n) \rightarrow (s'|l'_1, \ldots, l'_n)$, its semantics can be found in [3]. Our execution model is asynchronous: each transition of $R_n$ affects the shared state, and the local state of at most one thread.

Let $L_I \subseteq L$ be a set of initial local states and $s_I$ be the unique initial shared state; initial (global) states of $P_n$ hence have the form $\tau_I = (s_I|l_1, \ldots, l_n)$ where $l_i \in L_I$ for all $i$. We denote $I$ as the set of initial states. A path in $P_n$ is a finite sequence of states in $T_n$ starting from any $\tau_I$ whose adjacent elements are related by $R_n$.

A TTS also gives rise to an infinite-state transition system $P_\infty = (T_\infty, R_\infty)$ whose set of states/transitions/paths is the union of the sets of states/transitions/paths of $P_n$, for all $n \in \mathbb{N}$. We can reduce $P_n$ to a well quasi-ordered system (WQOS): let the covers relation $\succeq$ over $T_\infty$ be defined as follows:

$$(s|l_1, \ldots, l_n) \succeq (s'|l'_1, \ldots, l'_n)$$

whenever $s = s'$ and $[l_1, \ldots, l_n] \supseteq [l'_1, \ldots, l'_n]$, where $[\cdot]$ denotes a multiset.

Problem definition. We are tackling in this paper the coverability problem: given a TTS $P$ and a state $\tau_F \in T_\infty$: is $\tau_F$ coverable? In particular, we are asking if there exists a path starting from an initial state $\tau_I$ and reaching a state $\tau$ such that $\tau \succeq \tau_F$.

The coverability problem is decidable, but of high complexity: the equivalent coverability problem for Petri nets was shown to be EXPSPACE hard [5]. In this work, we aim at improving the efficiency of existing algorithms.

Backward coverability analysis. A sound and complete method to decide coverability
for WQOS is the backward search algorithm by Abdulla et al. [2, 1], a simple version of which is shown on the right. Input is a WQOS $P$, a set of initial states $I$, and a non-initial final state $\tau_F$. The algorithm maintains a work set $W$ of unprocessed states, and a set $X$ of minimal encountered states. It iteratively computes minimal cover predecessors

$$\text{CovPre}(w) = \min \{ p : \exists w' \geq w : p \rightarrow w' \}$$

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

### Algorithm 1 BWS($P, I, \tau_F$)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$W := {\tau_F} ; X := {\tau_F}$</td>
</tr>
<tr>
<td>2</td>
<td>while $\exists w \in W$</td>
</tr>
<tr>
<td>3</td>
<td>$W := W \setminus {w}$</td>
</tr>
<tr>
<td>4</td>
<td>for $p \in \text{CovPre}(w) \setminus \uparrow X$</td>
</tr>
<tr>
<td>5</td>
<td>if $p \in I$ then</td>
</tr>
<tr>
<td>6</td>
<td>return “coverable”</td>
</tr>
<tr>
<td>7</td>
<td>$W := \min(W \cup {p})$</td>
</tr>
<tr>
<td>8</td>
<td>$X := \min(X \cup {p})$</td>
</tr>
<tr>
<td>9</td>
<td>return “uncoverable”</td>
</tr>
</tbody>
</table>

Alg. 1: infinite-state backward search. $\uparrow X = \{x' : \exists x \in X : x' \geq x\}$, stands for the upward closure of $X$.

### Thread-State Equation

A thread-state equation [3], a variant of marking equation from Petri nets [7], is a set of linear constraints that can be easily derived from a TTS, whose set of solutions overapproximates the set of reachable states. We denote $\phi$ as the conjunction of thread-state equation. Thread-state equation can be supplemented with linear constraints specifying a final state $\tau_F$; we denote the conjunction of linear constraints as $\psi(\tau_F)$. Thus, $\phi \land \psi(\tau_F)$ can be solved using standard linear or integer programming. The unsatisfiability of $\phi \land \psi(\tau_F)$ implies the unreachability of any state covering the final state $\tau_F$ and hence concludes the safety of TTS. Please refer to [3] for more details.

## 3 Symbolic Pruning with Thread State Equation

In this section, we present an algorithm for accelerating the backward search by pruning some uncoverable states. Our idea is pretty simple: we run backward search as normal. But for each state, before computing its minimal cover predecessors, we detect if it is uncoverable by using thread-state equation [3]. We discard it if the uncoverability is proved; we expand the state otherwise.

### The Algorithm

Input of Alg. 2 is a TTS $P$, a set $I$ of initial states, and a final state $\tau_F$. The algorithm maintains a work set $W$ of unprocessed states, a set $X$ of minimal encountered states, and a set $U$ of minimal uncoverable states. Starting from $\tau_F$, it successively computes cover predecessors, and terminates either by reaching an initial state (thus proving the coverability of $\tau_F$) or when $W$ is empty (then proving the uncoverability of $\tau_F$). Line 2 with subroutine TSE generates the thread-state equation $\phi$ of $P$. Notice that $\phi$ is fixed in the rest of computation.

The while loop in Line 3 now steps through all unprocessed states $w$. Three possibilities arise depend on the value of $w$:

- **Line 5–6** if there exists an expanded state $x \in X$ such that $w \geq x$, then we discard $w$ because $w \in \uparrow x$ and hence all $w$’s cover predecessors are also $x$’s.
Algorithm 2 Backward Coverability Analysis with Symbolic Pruning

\begin{algorithm}
\KwIn{a TTS $P$; initial states $I$; final states $\tau_F$}
\KwOut{\{ coverable + witness \mid uncoverable \}}

1: $W := \{ \tau_F \}, X := \emptyset, U := \emptyset$

2: $\phi := \text{TSE}(P)$ \Comment{thread-state equation}

3: while $\exists w \in W$

4: 

5:   if $\exists x \in X$ s.t. $w \succeq x$ then

6:     “discard $w$”

7:   else if $\exists u \in U$ s.t. $w \succeq u$ then \Comment{uncoverability}

8:     “discard $w$”

9:   else if $\phi \land \psi(w) = \text{unsat}$ then \Comment{symbolic pruning}

10:     $U := \min(U \cup \{w\})$

11: else

12:   for each $p \in \text{COVPRE}(w)$ \Comment{cover predecessors}

13:     if $p \in I$ then

14:         return “coverable” + witness

15:     $W := \min(W \cup \{p\})$ \Comment{minimize $W$ w.r.t $p$}

16:     $X := \min(X \cup \{w\})$ \Comment{minimize $X$ w.r.t $w$}

17: return “uncoverable”

\end{algorithm}

Line 7–8 if there exists a state $u \in U$ such that $w \succeq u$, then we discard $w$ since the uncoverability of $u$ derives the uncoverability of $w$ in trivial.

Line 9–16 if none of above is true, the algorithm first attempts to prove the uncoverability of $w$ with the conjunction of thread-state equation $\phi$ and final state equations $\psi(w)$. If the conjunction is unsatisfiable, then Alg. 2 proves $w$’s uncoverability. The explicit exploration will not compute $w$’s cover predecessors and hence prunes the potential exploration tree rooting at $w$. Line 10 adds $w$ to $U$ and minimizes the uncoverable set $U$ by removing the elements larger than $w$ with respect to $\succeq$. Otherwise, Alg. 2 steps into the else branch in Line 11. Line 12–16 run the normal exploration as in Alg. 1.

Soundness and completeness. As thread-state equations are an approximate coverability analysis, exact uncoverability proofs in Line 9 is impossible. Thus, false positives might be reported. Our approach, however, is still sound as it always returns a correct result assuming the input is valid. It is complete as in the worst case Alg. 2 will fall back to Alg. 1 which guarantees to terminate.

Empirical Evaluation. We experimentally evaluate our approach in a verifier named BSSP. The goal of the evaluation is to measure the performance impact of the presented approach compared to Alg. 1 BWS.

We collected an extensive set of benchmarks, 352 in total, which is organized into two suites. The first suite contains 220 TTSs obtained from Boolean programs (BPs) (taken from [3]). As TTSs are equivalent in expressiveness to certain forms of Petri nets

\footnote{BSSP "=" Backward Search with Symbolic Pruning}
We use Z3 (v4.3.2) [10] as the backend solver. For each benchmark, we consider verification of a safety property. All experiments are performed on a 2.3GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux. Execution time is limited to 30min, and memory to 4 GB. All benchmarks and our tool are available online.

Tool comparison. The comparison results are give in Fig. 1. We observe that BSSP performs much better than BWS on the safe cases in both suites. On unsafe cases, BSSP beats BWS on large examples and remains competitive on small examples. In general, we observe that BWS outperforms BSSP on small examples which are solved within few seconds; an effect can be attributed to the overhead added by Z3. With the results, we can safely conclude that BSSP successfully accelerates backward search.

4 Conclusion

In this paper we have proposed a method for accelerating explicit-state backward search algorithm for thread transition systems by employing a symbolic pruning engine resulting from thread-state equations. We showcased the power of the method for proving and refuting safety properties in an extensive collection of benchmarks. In the future, we plan to evaluate the efficiency of our method by comparing against more state of the art algorithms and tools.

2 Webpage: https://github.com/lpzun/bssp
References