Unbounded-Thread Program Verification using Thread-State Equations

Konstantinos Athanasiou(✉), Peizun Liu, and Thomas Wahl

Northeastern University, Boston, USA
konathan@ccs.neu.edu

Abstract. Infinite-state reachability problems arising from unbounded-thread program verification are of great practical importance, yet algorithmically hard. Despite the remarkable success of explicit-state exploration methods to solve such problems, there is a sense that SMT technology can be beneficial to speed up the decision making. This vision was pioneered in recent work by Esparza et al. on SMT-based coverability analysis of Petri nets. We present here an approximate coverability method that operates on *thread-transition systems*, a model naturally derived from predicate abstractions of multi-threaded programs. In addition to successfully proving uncoverability for *all* our safe benchmark programs, our approach extends previous work by the ability to decide the *unsafety* of many unsafe programs, and to provide a witness path. We also demonstrate experimentally that our method beats all leading explicit-state techniques on safe benchmarks and is competitive on unsafe ones, promising to be a very accurate and fast coverability analyzer.

1 Introduction

Unbounded-thread program verification continues to attract the attention it deserves: it targets programs designed to run on multi-user platforms and web servers, where concurrent software threads respond to service requests of a number of clients that can usually neither be predicted nor meaningfully bounded from above a priori. To account for these circumstances, such programs are designed for an unspecified and unbounded number of parallel threads.

We target in this paper unbounded-thread shared-memory programs where each thread executes a non-recursive, finite-data procedure. This model is popular, as it connects to multi-threaded `C` programs via predicate abstraction, a technique that has enjoyed progress for concurrent programs in recent years [5]. The model is also popular since basic program state reachability questions are decidable, although of high complexity: the corresponding *coverability problem* for Petri nets was shown to be EXPSPACE-complete [4,21].

Owing to the importance of this problem, much effort has since been invested into finding practically viable algorithms [1,3,10,11,15–17]. The vast majority of these are flavors of explicit-state exploration tools. Given the impressive advances

This work is supported by NSF grant no. CCF-1253331.

N. Olivetti and A. Tiwari (Eds.): IJCAR 2016, LNAI 9706, pp. 516–531, 2016.
DOI: 10.1007/978-3-319-40229-1_35

that SMT technology has made, and its widespread "infiltration" of program verification, an obvious question is to what extent such technology can assist in solving the coverability problem.

An encouraging answer to this question was given in a recent symbolic implementation of the Petri net *marking equations* technique for coverability checking [6]. The equations are expressed as integer linear arithmetic constraints and passed to an SMT solver. While the constraints overapproximate the coverability condition, causing the technique to produce false positives, its success rate was very convincing.

Building on the promise of this technique, in this paper

1. we develop a similar approach that applies to a computational model more fitting for software verification, called *thread-transition systems* (TTS). This model makes shared and local thread storage explicit and is designed for encodings of shared-variable concurrent programs. It enjoys a one-to-one correspondence with multi-threaded *Boolean programs*. The latter in turn is a widely used software abstraction employed in concurrency-capable tools such as SatAbs [5] and Bfc [15]. Naturally, we dub our constraint sets *thread-state equations*;
2. we equip our approach with a straightforward but effective component to detect spurious assignments, and to refine the constraints if needed. This component enables the approach to prove systems *unsafe* and generate counterexamples; a feature that was not addressed in [6].

Our method is sound but theoretically incomplete. We implemented it in a tool called Tse; Sect. 5 contains an extensive evaluation on a large number of Boolean program benchmarks. We give a preview of our findings here:

– Notwithstanding said incompleteness, Tse was able to correctly decide 98 % of all TTS instances; this includes safe and unsafe ones.
– Comparing to the most competitive *complete* coverability checker for replicated Boolean programs, Bfc [15], Tse proves to be very close in efficiency on unsafe benchmarks, and *much more efficient* than Bfc on safe ones. (The gap is even larger with other explicit-state explorers.)

In summary, we envision our work to introduce the power of constraint-based coverability analysis to the world of unbounded-thread program verification. Our results showcase Tse as a very capable and highly successful replicated Boolean program verifier.

2 Thread-Transition Systems

We assume multi-threaded programs are given in the form of an abstract state machine called *thread-transition system* (TTS) [15]. Such a system reflects the replicated nature of programs we consider: programs consisting of threads executing a given procedure defined over shared and (thread-)local variables. A thread-transition system is defined over a set of *thread states* $T = S \times L$,

where S and L are the finite sets of *shared* and *local* states respectively. $R \subseteq T \times T$ is the transition relation on T, partitioned into $R = \mapsto \cup \looparrowright$; the two partitions intuitively represent *thread transitions* and *spawn transitions*, respectively (semantics below). We refer to elements of R as *edges*. A TTS can now be defined as $\mathcal{P} = (T, R)$. Figure 1 shows an example.

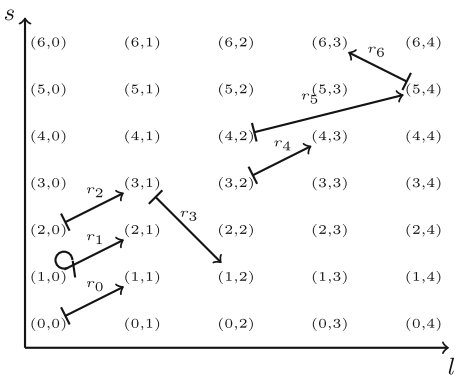

Fig. 1. A thread-transition system with thread ($\mapsto$) and spawn ($\looparrowright$) transitions.

A TTS induces an infinite-state transition system $\mathcal{P}_\infty = (V_\infty, R_\infty)$, as follows. For a positive integer n, let $V_n = S \times L^n$ and $V_\infty = \cup_{i=1}^{\infty} V_i$. We write $v = (s|l_1, \ldots, l_n)$ to denote a (global) system state with a shared component s, and n threads in local states l_i ($i \in \{1, ..., n\}$).

A transition, written as $(s|l_1, \ldots, l_n) \rightarrowtail (s'|l'_1, \ldots, l'_{n'})$, belongs to relation R_∞ exactly if one of the following conditions holds:

Thread Transition: $n' = n$ and there exists $(s, l) \mapsto (s', l') \in R$ and i such that $l_i = l$, $l'_i = l'$, and for all $j \neq i$, $l'_j = l_j$.

Spawn Transition: $n' = n + 1$ and there exists $(s, l) \looparrowright (s', l') \in R$ and i such that $l_i = l$, $l'_{n'} = l'$, and for all $j < n'$, $l'_j = l_j$.

Thus, a transition in R_∞ affects the shared state, and the local state of at most one thread. It may fire only if one thread—the *active* thread—is currently at the corresponding TTS edge's source thread state. We denote by $w \rightarrowtail_{(\mapsto)} w'$ the fact that the thread active in $w \rightarrowtail w'$ fires a $\mapsto$ edge; similarly for $\rightarrowtail_{(\looparrowright)}$.

Let $L_I \subseteq L$ be a set of initial local states and s_I be the unique initial shared state; initial states of $\mathcal{P}_\infty$ hence have the form $v_I = (s_I|l_1, \ldots, l_n)$ where $l_i \in L_I$ for all i. A *path* in $\mathcal{P}_\infty$ is a finite sequence of states in V_∞ starting from any v_I whose adjacent elements are related by R_∞.

In order to state the problem we are tackling, define the *covers* relation $\succeq$ over V_∞ as $(s|l_1, \ldots, l_n) \succeq (s'|l'_1, \ldots, l'_{n'})$ if $s = s'$ and $[l_1, \ldots, l_n] \supseteq [l'_1, \ldots, l'_{n'}]$, where $[\cdot]$ denotes a *multi-set*. We are solving in this paper the *coverability* problem for a given (final) state $v_F \in V_\infty$: is v_F coverable, i.e. does there exists a path in $\mathcal{P}_\infty$ leading to a state v that covers v_F : $v \succeq v_F$? We denote the final shared state by s_F, i.e. $v_F = (s_F| \ldots)$. As an example, state $(1|0)$ is coverable in the 2-thread system derived from the TTS in Fig. 1 with the unique initial thread state $(0, 0)$; the path consists of one thread firing the edge $(0, 0) \mapsto (1, 1)$.

The coverability problem is decidable: relation $\succeq$ is a well-quasi order with respect to which the system $\mathcal{P}_\infty$ is *monotone* [15]. Algorithms for deciding coverability over such systems exist [2,11] but are of high complexity, e.g. EXPSPACE-complete for standard Petri nets [4,21], which are equivalent in expressiveness to infinite-state transition systems obtained from TTS [15].

3 Safety Proofs via Thread-State Equations

In this section we describe how, given a coverability problem, we derive a set of equations whose inconsistency (unsatisfiability of their conjunction) implies the unreachability of any global state covering the final state v_F, and hence the safety of the infinite-state system. We do so by determining constraints on the number of threads in each local state when a global state is reached, as well as constraints that encode the synchronization that shared states enforce among the threads.

3.1 Thread and Transition Counting

Given an initial global state, a finite path p in $\mathcal{P}_\infty$ can be succinctly and unambiguously represented as a sequence of pairs (r, i), where $r \in R$ is a TTS edge and i is a thread index. An abstraction of such a sequence is given by the number of times each edge in R fires along p. This "counting abstraction", which can be seen as simplifying an edge sequence to a multi-set, is rather crude, as it ignores the order of edges fired along p. On the other hand, it allows us to express the coverability condition: from the numbers of times each edge fires, we can obtain the number of threads per local state in the final global state of p. We now require that they match or exceed the thread counts in v_F. Along with the obvious non-negativity constraints for counters, we obtain a first approximation of our thread-state equations, as follows.

Given a TTS $\mathcal{P} = (T, R)$ and a final state v_F, we fix a total order on all edges, and a total order on all local states. We further introduce:

- an integer vector $\mathbf{r}$ of $|R|$ variables, representing the number of occurrences of each edge along p (the edges appear in $\mathbf{r}$ in the given total order);
- an integer vector $\mathbf{l}_I$ of $|L|$ variables, representing the number of threads per local state in the initial state of p (the local states appear in $\mathbf{l}_I$ in the given total order);
- an integer vector $\mathbf{l}_F$ of $|L|$ variables, representing the number of threads per local state in the final state of p (the local states appear in $\mathbf{l}_F$ in the given total order);
- an $|L| \times |R|$ integer matrix $\mathbf{c}$ (a constant) that captures the effect of each edge on each local state, as follows:

$$\mathbf{c}(l, r) = \begin{cases} +1 & \text{if edge } r \textit{ ends} \text{ in local state } l \\ -1 & \text{if } r \in \mapsto \text{ and } r \textit{ starts} \text{ in local state } l \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

(For simplicity, we identify l with the local state with number l in the total order, similarly for r.) We assume R has no self-loops (which are irrelevant for coverability), hence the quantity $\mathbf{c}(l, r)$ is well-defined. Note that the -1 case only applies to standard thread transition edges ("$\mapsto$"), not to spawns: the latter affect only the number of threads in the target local state. Also note that $\mathbf{c}$ does not capture shared-state changes.

With these variables, we define the following system of *local-state constraints* $\mathcal{C}_L$:

$$\mathcal{C}_L \;=\; \bigwedge \begin{cases} \mathbf{r} \geq 0 & \text{non-negative edge counters} \\ \left.\begin{array}{l} \mathbf{l}_I \geq 0 \\ \mathbf{l}_F \geq 0 \end{array}\right\} & \text{non-neg. local state ctrs.} \\ \bigwedge_{l \notin L_I} \mathbf{l}_I(l) = 0 & \text{initial state condition} \\ \mathbf{l}_F = \mathbf{l}_I + \mathbf{c} \cdot \mathbf{r} & \text{final state condition} \\ \bigwedge_{l \in L} \mathbf{l}_F(l) \geq |\{i : v_F(i) = l\}| & \text{coverability condition} \end{cases} \tag{2}$$

The notation $\mathbf{r} \geq 0$ means "pointwise non-negative"; similarly for $\mathbf{l}_I$ and $\mathbf{l}_F$. Symbol $\mathbf{l}_I(l)$ refers to the component of $\mathbf{l}_I$ corresponding to local state l; similarly for $\mathbf{l}_F(l)$. Operator $\cdot$ denotes matrix multiplication, $|\{\ldots\}|$ is set cardinality, and $v_F(i)$ stands for the local state of thread i in state v_F. These constraints stipulate that all edge and local state counters be natural numbers; that no thread start out in a non-initial local state; that the final local state counters account for the effect of all edges; and that the final global state covers v_F.

3.2 Shared State Synchronization

The thread and transition counting constraints reflected in $\mathcal{C}_L$ ignore the order in which edges fire along a path p, since distinguishing ordered edge sequences symbolically is prohibitively expensive. Some of the ordering information can, however, be recovered, by taking shared state changes into account (which have also been ignored so far): consecutive edges along p must synchronize on the shared state "in the middle".

This requirement can be formalized as follows. Consider an assignment to $(\mathbf{r}, \mathbf{l}_I, \mathbf{l}_F)$ satisfying the constraints $\mathcal{C}_L$. We call an edge $r \in R$ *active* if $\mathbf{r}(r) > 0$, and a shared state *active* if at least one of its adjacent edges is active.

Observation 1. *Let $G_{\mathbf{r}}\big|_S$ be the directed **multi-**graph with node set S and edge **multi-**set $[r \in R : \mathbf{r}(r) > 0]\big|_S$. That is, $G_{\mathbf{r}}\big|_S$ is defined over the active edges in the multiplicity given by $\mathbf{r}$, projected to S. An edge sequence p*

1. uses exactly the edges in the multiplicity given by $\mathbf{r}$, and
2. has consecutive edges that synchronize on the shared state,

*exactly if p is an **Euler path** in $G_{\mathbf{r}}\big|_S$.*

This observation is easily seen to hold: the Euler criterion guarantees that exactly all edges in $G_{\mathbf{r}}\big|_S$ (= the active edges, in the given multiplicity) are used. The "pathness" in the S-projection guarantees the synchronization condition.

We are thus looking for an Euler path in $G_{\mathbf{r}}\big|_S$. To formalize its existence, we use the following standard adjacency notions from graph theory:

$$in(s) = \{r \in R \,|\, r \text{ ends in shared state } s\}, \qquad adj(s) = in(s) \cup out(s),$$
$$out(s) = \{r \in R \,|\, r \text{ starts in shared state } s\}.$$

Note that edges that leave the shared state invariant (denoting thread-internal transitions) are contained in both the *in* and *out* sets.

The existence of an Euler path from s_I to s_F in $G_{\mathbf{r}}\big|_S$ is known to be equivalent to the conjunction of the following two conditions (see, e.g., [8]):

Flow: each shared state except s_I and s_F is entered and exited the same number of times (along with some special conditions on s_I and s_F),

Connectivity: the *undirected* subgraph of $G_{\mathbf{r}}\big|_S$ induced by the active shared states is *connected* (has a path between any two nodes).

We now describe how we formalize these conditions as symbolic constraints.

Flow Condition. We write shared state s's *flow constraints* as

$$\mathit{flow}(s) \quad :: \quad \sum_{r \in \mathit{in}(s)} \mathbf{r}(r) \; - \; \sum_{r \in \mathit{out}(s)} \mathbf{r}(r) \; = \; N \tag{3}$$

where N is defined depending on the relationship between s, s_I, and s_F:

$$N \; = \; \begin{cases} 0 & \text{if } s \notin \{s_I, s_F\} \textbf{ or } s = s_I = s_F \\ -1 & \text{if } s = s_I \neq s_F \\ +1 & \text{if } s = s_F \neq s_I \end{cases} \tag{4}$$

Our overall flow condition enforces flow constraints (3) for all shared states: $\mathcal{C}_F = \bigwedge_{s \in S} \mathit{flow}(s)$.

Connectivity Condition. For an Euler path to exist in $G_{\mathbf{r}}\big|_S$, the undirected graph induced by its active shared state nodes must be connected. This is equivalent to the existence of a simple *undirected* path between the initial shared state s_I and s, for each shared state s. To this end we introduce, for each $s \in S$,

- a vector $\mathbf{e}_s$ of $|R|$ integer variables. These variables, later constrained to be in $\{0, 1\}$, encode, in unary, the set of undirected edges of $G_{\mathbf{r}}\big|_S$ participating in the simple path between s_I and s.
- a predicate for the existence of such a path to s:

$$\begin{aligned} p(s) \quad :: \quad & \textstyle\sum_{r \in adj(s_I)} \mathbf{e}_s(r) = 1 \; \wedge \; \sum_{r \in adj(s)} \mathbf{e}_s(r) = 1 \\ & \wedge \; \forall s' \in S \setminus \{s_I, s\} \quad \textstyle\sum_{r \in adj(s')} \mathbf{e}_s(r) \; \in \; \{0, 2\} \end{aligned} \tag{5}$$

  The first two sums ensure that the initial (s_I) and target (s) shared states of the simple path have exactly one adjacent transition (and thus degree 1). The last two ensure that each other shared state is either part of the simple path (and has degree 2) or it is not (and has degree 0).
- a predicate characterizing active shared states: $\mathit{act}(s) :: \sum_{r \in adj(s)} \mathbf{r}(r) > 0$.

We now formulate the following system of *connectivity constraints* $\mathcal{C}_C$:

$$\mathcal{C}_C \;=\; \bigwedge \begin{cases} \bigwedge_{s\in S} \bigwedge_{r\in R} \; \mathbf{e}_s(r) \in \{0,1\} \\ \bigwedge_{r\in R} \; (\mathbf{r}(r) = 0 \implies \bigwedge_{s\in S} \mathbf{e}_s(r) = 0) \\ \bigwedge_{s\in S\setminus\{s_I,s_F\}} \; act(s) \implies p(s) \\ s_I \neq s_F \wedge act(s_F) \implies p(s_F) \end{cases} \tag{6}$$

These constraints state that the $\mathbf{e}_s$ are bitvectors (used to encode the edge set of $G_{\mathbf{r}}\big|_S$ in unary); that inactive edges are excluded from the connected subgraph; and that each active shared state except s_I and s_F is connected by a simple path to the initial shared state; the last line requires the same of s_F unless $s_I = s_F$.

Just like $\mathcal{C}_L$, constraints $\mathcal{C}_F$ and $\mathcal{C}_C$ are expressible in the decidable theory of integer linear arithmetic (ILA). Formulas $\mathcal{C}_L$ and $\mathcal{C}_F$ require a number of variables linear in the size of the input TTS, namely $|R|+2|L|$, while $\mathcal{C}_C$ requires a quadratic number of variables, namely $|S|\times|R|$. This larger number of variables has consequences for deciding the $\mathcal{C}_C$ constraints; a fact that is taken into account by the coverability algorithm proposed in Sect. 4.1.

We finally remark that satisfiability of all conditions together, i.e. $\mathcal{C}_L \wedge \mathcal{C}_F \wedge \mathcal{C}_C$, does not guarantee that the edges given by $\mathbf{r}$ can be sequenced to a proper path through $\mathcal{P}_\infty$. Figure 2 shows a TTS and a satisfying assignment to $(\mathbf{r}, \mathbf{l}_I, \mathbf{l}_F)$ that suggests to form a path consisting of exactly one occurrence of each edge in the TTS. The S-projection of these edges is connected. However, it is easy to see that no permutation of the three edges constitutes a valid firing sequence.

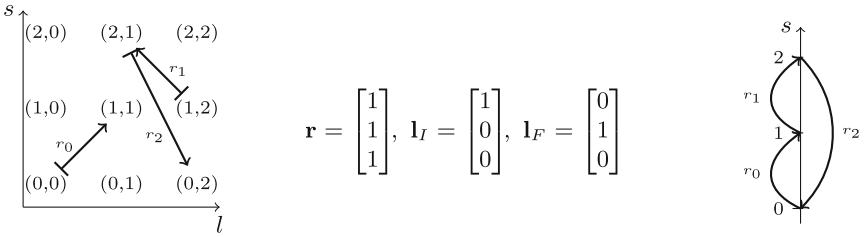

Fig. 2. A TTS (left) with $s_I = 0$, $L_I = \{0\}$, and $v_F = (0|1)$, an assignment satisfying $\mathcal{C}_L \wedge \mathcal{C}_F \wedge \mathcal{C}_C$ (middle), and its S-projection $G_{\mathbf{r}}\big|_S$ (right)

3.3 Thread-State Equations by Example

We use the TTS of Fig. 3 to showcase how our approach attempts to symbolically solve the coverability problem, by reducing it to a conjunction of integer linear constraints. We consider the case where $s_I = 0$, $L_I = \{0\}$ and therefore the initial state of $\mathcal{P}_\infty$ is of the form (0|0,...,0). We would like to confirm safety with respect to the "bad" final thread state t_F = (1,1). It is not coverable, i.e. there exists no state reachable from any initial state that covers the final state $v_F = (1|1)$.

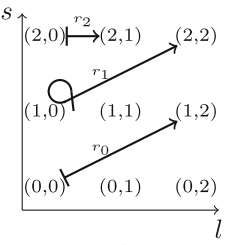

Fig. 3. A TTS

We start by formulating the $\mathcal{C}_L$ constraints, described in Eq. (2). The first equation on the left column of Fig. 4 is the final state condition for local state 0. No edges enter local state 0 but two edges exit it: r_0 and r_2; hence counters $\mathbf{r}(0)$ and $\mathbf{r}(2)$ are subtracted from $\mathbf{l}_I(0)$ to yield $\mathbf{l}_F(0)$. (Recall that spawn edges leave the active thread's local state intact; hence r_1 does not affect $\mathbf{l}_F(0)$.) We derive the final state constraints for the remaining local states similarly; for these the entries of $\mathbf{l}_I$ are 0 by the initial state condition of (2). The coverability condition of (2) for final state $v_F = (1 \mid 1)$ translates into $\mathbf{l}_F(1) \geq 1$.

$$\begin{aligned} \mathbf{l}_I(0) - \mathbf{r}(0) - \mathbf{r}(2) &= \mathbf{l}_F(0) & -\mathbf{r}(0) &= -1 \\ \mathbf{r}(2) &= \mathbf{l}_F(1) & \mathbf{r}(0) - \mathbf{r}(1) &= 1 \\ \mathbf{r}(0) + \mathbf{r}(1) &= \mathbf{l}_F(2) & \mathbf{r}(1) &= 0 \\ \\ \mathbf{l}_F(1) &\geq 1 & \mathbf{r}(1) = 0 &\implies \mathbf{e}_2(1) = 0 \wedge \mathbf{e}_1(1) = 0 \end{aligned}$$

$$p(2) \;::\; \mathbf{e}_2(0) = 1 \wedge \mathbf{e}_2(1) = 1 \wedge \big(\mathbf{e}_2(0) + \mathbf{e}_2(1) = 0 \vee \mathbf{e}_2(0) + \mathbf{e}_2(1) = 2\big)$$

Fig. 4. Thread-state equations for the TTS of Fig. 3. Notation $\mathbf{l}_F(i)$ stands for the counter variable for local state $l_i \in \{0, 1, 2\}$; $\mathbf{r}(i)$ for the counter variable for edge r_i. The left column shows the local state constraints $\mathcal{C}_L$, the right column the synchronization constraints $\mathcal{C}_F$ and the inactive edge condition for r_1; the last row shows the path predicate $p(2)$.

Next we show in Fig. 4 (right) the flow constraints as defined by in (3). The first equation deals with shared state 0, which has only one adjacent transition: r_0. Since it exists 0 and 0 is initial, we obtain $-\mathbf{r}(0) = -1$. The next equation deals with shared state 1, which has two adjacent transitions: r_0 (entering) and r_1 (exiting); since 1 is final, we obtain $\mathbf{r}(0) - \mathbf{r}(1) = 1$. Regarding shared state 2, edge r_2 leaves it invariant, while r_1 enters it; we obtain $\mathbf{r}(1) = 0$.

Finally we write the constraints for the active edge condition for transition r_1, and the predicate $p(2)$. If r_1 occurs 0 times then the values $\mathbf{e}_1(1)$ and $\mathbf{e}_2(1)$ are set to 0 so that the undirected edges they encode cannot be part of simple paths between the initial shared state and shared states 1 and 2 respectively. $p(2)$ checks for existence of a simple, undirected path between shared states 0 (initial) and 2. The values encoding their adjacent edges, $\mathbf{e}_2(0)$ and $\mathbf{e}_2(1)$, are set to 1 so that shared states 0 and 2 serve as source and target of the path. For shared state 1, the sum of their adjacent edges is set to either 0 or 2 to allow it to either be part of the simple path or not.

The above TSE are unsatisfiable, confirming the uncoverability of t_F. The $\mathcal{C}_F$ constraints enforce that $\mathbf{r}(1)$ is 0, implying that $\mathbf{e}_2(1)$ is 0, which prevents a path between shared states 0 and 2. It turns out that *without* the connectivity condition, the TSE permit the spurious two-thread solution $\mathbf{r}(2) = 1$, $\mathbf{r}(0) = 1$: firing these edges in some order would cover local state 1 (local-state constraints), and the flow constraints are satisfied as well; note that edge r_2, once projected to S, is a self-loop and thus irrelevant for Eq. (3). The two edges do not, however, synchronize on the shared state, no matter which order they fire (the S-projection permits no Euler path). This failure is caught by Eq. (6).

4 Coverability Analysis via Thread-State Equations

We are now ready to incorporate our thread-state equations into an algorithm for establishing system safety. We also present a simple refinement scheme that, very often in practice, enables our algorithm to prove unsafety.

4.1 Coverability via TSE: The Algorithm

Overview. Our algorithm employs the local-state, flow, and connectivity constraints given by $\mathcal{C}_L$, $\mathcal{C}_F$, and $\mathcal{C}_C$, respectively. Constraints $\mathcal{C}_C$, formulating the (non-trivial) connectedness condition for graph $G_{\mathbf{r}}\big|_S$, use a number of variables quadratic in the size $|S|+|L|+|R|$ of the TTS (see Sect. 3.2). As we have determined empirically, they tend to be more expensive to check for satisfiability than $\mathcal{C}_L$ and $\mathcal{C}_F$. Our algorithm is therefore composed of two sub-processes, as follows. Process **A** implements the main algorithm and is described in detail below. Process **B** runs in parallel with **A** and attempts to prove safety using the full set of constraints $\psi = \mathcal{C}_L \wedge \mathcal{C}_F \wedge \mathcal{C}_C$, including $\mathcal{C}_C$. If it proves ψ unsatisfiable, it kills **A** and returns "uncoverable" as the overall answer. If ψ is satisfiable, or **B** runs out of memory, it exits without returning an answer, and process **A** continues alone.

The composition of processes **A** and **B** is shown in Algorithm 1, which attempts to decide the reachability, in $\mathcal{P}_\infty$, of a global state covering v_F. We describe in the following the implementation of process **A**, which uses the (more lightweight) counting and flow constraints to prove safety, and a witness generation scheme to prove unsafety. Process **A** begins by building the thread-state equations $\varphi = \mathcal{C}_L \wedge \mathcal{C}_F$ for the given $\mathcal{P}$, and passing it to a model-building SMT solver capable of deciding integer linear arithmetic formulas (Line **A**1). If the solver decides φ is unsatisfiable, the algorithm returns "uncoverable".

Otherwise let m be a model, i.e. an assignment to $(\mathbf{r}, \mathbf{l}_I, \mathbf{l}_F)$ (Line **A**2). From these assignments we can extract the number n_m of threads that exist at the beginning of the path to be built as is the sum of all $\mathbf{l}_I$ variables, and the number s_m of threads spawned along the path as the sum of all $\mathbf{r}$ variables that correspond to spawn edges (Lines **A**3 and **A**4).

Process **A** now needs to check whether the assignment obtained in m is spurious, or whether it can be turned into a proper witness path in $\mathcal{P}_\infty$. To do this efficiently, we generalize this task and ask whether v_F is coverable along *any* path, but given limited resources, namely n_m initial threads and at most s_m spawns. The key is that this is a finite-state search problem. We have built our own, reasonably efficient and complete, counterexample-producing explorer for this purpose; it is invoked in Line **A**5. If this search is successful, we have a solution to the infinite-state search problem as well: we return the witness path generated by $\textsc{Fss}(\mathcal{P}, n_m, s_m)$ as the answer produced by Algorithm 1.

If the finite-state search is unsuccessful, it shows that, if a state covering v_F is reachable, then only along a path that starts with more than n_m initial threads ("$n > n_m$") **or** spawns more than s_m threads along the way ("$s > s_m$"). This condition is enforced in Line **A**7, thus strengthening φ. In contrast to Lines

Algorithm 1. Coverability$(\mathcal{P}, s_I, L_I, v_F)$.
The **return** statements kill off the respective other process before returning

Input: TTS $\mathcal{P}$; initial shared state s_I; initial local states set L_I; final state v_F
Output: "uncoverable", or "coverable" + witness path

Process A

1: $\varphi := \mathcal{C}_L \wedge \mathcal{C}_F$
2: **while** $\exists m : m \models \varphi$
3: $\quad n_m := \sum_{l \in L} \mathbf{l}_I(l)(m)$
4: $\quad s_m := \sum_{r \in \rightsquigarrow} \mathbf{r}(r)(m)$
5: $\quad$ **if** Fss$(\mathcal{P}, n_m, s_m)$ = "coverable" + witness p
6: $\quad\quad$ **return** "coverable" + p
7: $\quad \varphi := \varphi \wedge (n > n_m \vee s > s_m)$
8: **return** "uncoverable"

||

Process B

1: $\psi := \mathcal{C}_L \wedge \mathcal{C}_F \wedge \mathcal{C}_C$
2: **if** ψ is unsat
3: $\quad$ **return** "uncoverable"

A3 and **A**4, the strengthening is expressed *symbolically* over the variables in $\mathbf{l}_I$ and $\mathbf{r}$. More precisely, $n > n_m$ abbreviates the formula

$$\mathbf{l}_I(0) \; + \quad \ldots \quad + \; \mathbf{l}_I(|L| - 1) \; > \; n_m \; ,$$

where the $\mathbf{l}_I(i)$ are variables, and n_m is the constant computed in Line **A**3. The formula abbreviated by $s > s_m$ is built similarly; here the sum expression for s is formed over the variables in $\mathbf{r}$ that correspond to spawn edges.

Given the strengthening to φ computed in Line **A**7, process **A** returns to the beginning of the loop and checks φ for satisfiability.

Finite-State Search. A breadth-first style algorithm for routine Fss is shown on the right. It maintains a worklist W and an *explored* set E, both initialized to the of initial states I_n, which covers all combinations of initial threads with size n. Each state w maintains a counter s to record the remaining number of spawns that can be fired from w; for $w \in I$, $w.s = s$. In each step, Fss removes a state w from W and expands it to w' if $w.s$ allows so. It returns coverable if $w' \succeq v_F$; otherwise steps forward. w' decreases the value of s inherited from w if the transition is due to a spawn.

Algorithm 2. Fss$(\mathcal{P}, n, s)$

1: $W := I_n$; $E := I_n$
2: **while** $\exists w \in W$
3: $\quad W := W \setminus \{w\}$
4: $\quad$ **for each** $w' \notin E$: $w \rightarrowtail_{(\mapsto)} w'$
$\quad\quad\quad \vee \; (w \rightarrowtail_{(\rightsquigarrow)} w' \wedge \; w.s > 0)$
5: $\quad\quad$ **if** $w' \succeq v_F$ **then**
6: $\quad\quad\quad$ **return** "coverable"
7: $\quad\quad$ **if** $w \rightarrowtail_{(\rightsquigarrow)} w'$ **then**
8: $\quad\quad\quad w'.s$--
9: $\quad\quad W := W \cup \{w'\}$; $E := E \cup \{w\}$
10: **return** "uncoverable"

4.2 Coverability via TSE: Analysis

We first prove the soundness (partial correctness) of Algorithm 1, and then discuss its termination. We assume that Lines **A**2 and **B**2 use a sound, complete, and model-building ILA solver; we use Z3 [20] in our experiments.

Partial Correctness. We begin our analysis with the following property.

Lemma 2. *If v_F is coverable in $\mathcal{P}_\infty$, then φ built in Lines **A**1 and **A**7 and ψ built in Line **B**1 are satisfiable.*

Theorem 3 (Soundness). *If Algorithm 1 returns "coverable", v_F is coverable in $\mathcal{P}_\infty$. If Algorithm 1 returns "uncoverable", v_F is uncoverable in $\mathcal{P}_\infty$.*

Proof. If Algorithm 1 returns "coverable", v_F is coverable, as procedure Fss running on a finite state space is sound and complete. If Algorithm 1 returns "uncoverable", triggered by the unsatisfiability of φ in Line **A**2 or ψ in Line **B**2, then v_F is uncoverable in $\mathcal{P}_\infty$ by Lemma 2. □

Termination. In general, Algorithm 1 is not guaranteed to terminate: neither of the two processes **A** and **B** may return. Two different scenarios can lead to non-termination. The first is that despite an uncoverable final state, **A** keeps finding spurious assignments, and **B** does the same or times out. Consider again the scenario and the assignment $(\mathbf{r}, \mathbf{l}_I, \mathbf{l}_F)$ shown in Fig. 2. As discussed in Sect. 3.2, this assignment is spurious, as will be confirmed by the invocation of $\textsc{Fss}(\mathcal{P}, 1, 0)$, which fails to reach a state covering v_F. φ is strengthened by $\mathbf{l}_I(0) > 1$. The result is again satisfiable, this time with a model that sets all of $\mathbf{r}(0)$, $\mathbf{r}(1)$, $\mathbf{r}(2)$ and $\mathbf{l}_F(1)$ to 2. We see that, for any n_m, there exists a model of φ satisfying $\mathbf{r}(i) = n_m$ for $i \in \{0 \dots 2\}$, $\mathbf{l}_I(0) = n_m$, $\mathbf{l}_F(1) = n_m$, which never translates to a genuine path in $\mathcal{P}_\infty$. Therefore Algorithm 1 will not terminate.

The other non-termination scenario is that of a *coverable* final state that is overlooked as the search diverges in the wrong direction. The problem is that increments applied to the initial thread count n and the spawn count s by the solver may not be *fair*: Line **A**7 only requires one of them to go up. As a special case, if the TTS has no spawn transitions ($\not\mapsto = \emptyset$), we can tighten Line **A**7 to $\varphi := \varphi \ \wedge \ n > n_m$, in which case the algorithm is (in principle) *complete for unsafe instances.*

5 Empirical Evaluation

The technique presented in this paper is implemented in a coverability checker named Tse (for "Thread-State Equation"). Tse is written in C++ and uses Z3 (v4.3.1) as the back-end ILA solver. It takes as input coverability problems for TTS. We used a benchmark suite of concurrent Boolean programs to evaluate Tse. We ran Tse on Boolean programs in order to compare with the following state-of-the-art checkers[1]:

Petrinizer: An SMT-based coverability checker described in [6] (v1.0)
Bfc: A coverability checker with forward oracle presented in [15] (v2.0)
Bfc-Km: A generalized Karp-Miller procedure presented in [15] (v1.0)
IIC: Incremental, inductive coverability algorithm [17]
Mist-Ar: An abstraction refinement method presented in [10] (v1.1)
Mist-Eec: Forward analysis with enumerative refinement [11] (v1.1)

[1] Available at www.cprover.org/bfc/; github.com/pierreganty/mist; and http://www.mpi-sws.org/~fniksic/cav2014/repository.tgz.

Benchmarks. Our benchmark set contains 339 concurrent Boolean programs generated from concurrent `C` programs (taken from [15,19]), 135 of which are safe. For each example, we consider a reachability property that is specified via an assertion. The table on the right shows the size ranges of the BPs.

BP	min.	max.
$\|S\|$	5	32769
$\|L\|$	17	55
$\|R\|$	18	584384

To apply Tse to `C` programs, we use SatAbs to transform those programs to TTS (option `--build-tts`) via intermediate Boolean programs [5]. When SatAbs requires several CEGAR iterations over the `C` programs until the abstraction permits a decision, the same `C` source program gives rise to several Boolean programs and TTSs.

Experimental Setup. The main objective of our experiments with BPs is to measure the competitiveness of Tse against state-of-the-art infinite-thread BP checkers; this is mostly variants of the Bfc tool. We also investigated how Tse fares against tools targeting Petri nets, of which there are many; most interesting for us is the Petrinizer tool, as it implements an idea similar to the one used in (and inspirational for) Tse. Petri net tools can be used for BP verification by converting those programs to Petri nets. We have experimented with two translators: one used in [6,15][2], and one by Pierre Ganty et al. github.com/pevalme/bfc_fork, which tries to alleviate the blowup incurred by shared state conversion. As different tools accept different translations, we used both translators in our experiments. The running times we report in the results *ignore* translation time, which ranges from almost nothing up to dozens of seconds.

All experiments are performed on a 2.3 GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux. The timeout is set to 30 min and the memory limit to 4 GB. All benchmarks and our tool are available online [18].

Precision. Table 1 compares the results of precision on BPs for all tools. Tse successfully decides *all* BPs except 5 unsafe instances, where the SMT solver runs out of memory. Both Bfc and Bfc-Km prove 4 out of these 5 instances. As for the safe instances, it was interesting to observe that the connectivity constraints $\mathcal{C}_C$ were never required to conclude unsatisfiability, i.e. the constraints $\mathcal{C}_L$ and $\mathcal{C}_F$ were already inconsistent. This means that process **B** in Algorithm 1 never ran to completion.

Table 1. Precision results for all tools. Note that Petrinizer decides only safe benchmarks

suite \ tools	Tse	Petrinizer	Bfc	Bfc-Km	IIC	Mist-Ar	EEC	# instances
safe BP (%)	100	100	57.04	2.22	81.48	94.07	34.81	135
*un*safe BP (%)	97.55	–	99.02	98.04	62.75	12.75	18.63	204
total (%)	98.53	–	82.60	59.88	70.21	45.13	25.07	339

[2] www.cprover.org/bfc/.

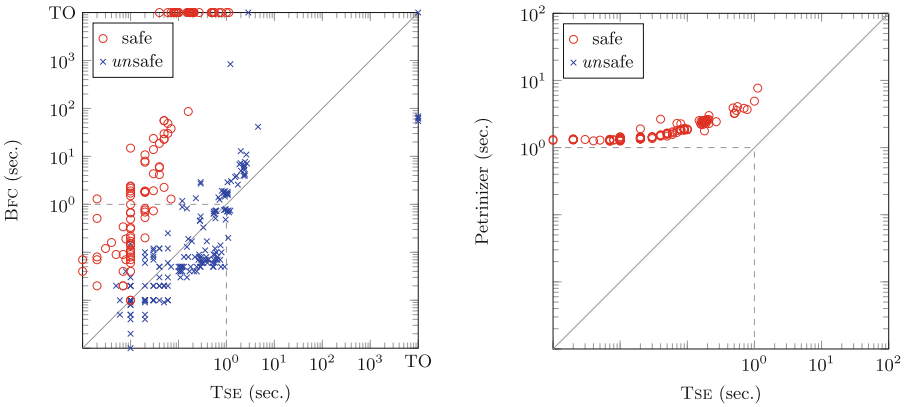

Fig. 5. Performance comparison for Boolean Programs: Tse vs. Bfc (left) and vs. Petrinizer (right). Each dot represents execution time for one program (TO = timeout) (Color figure online)

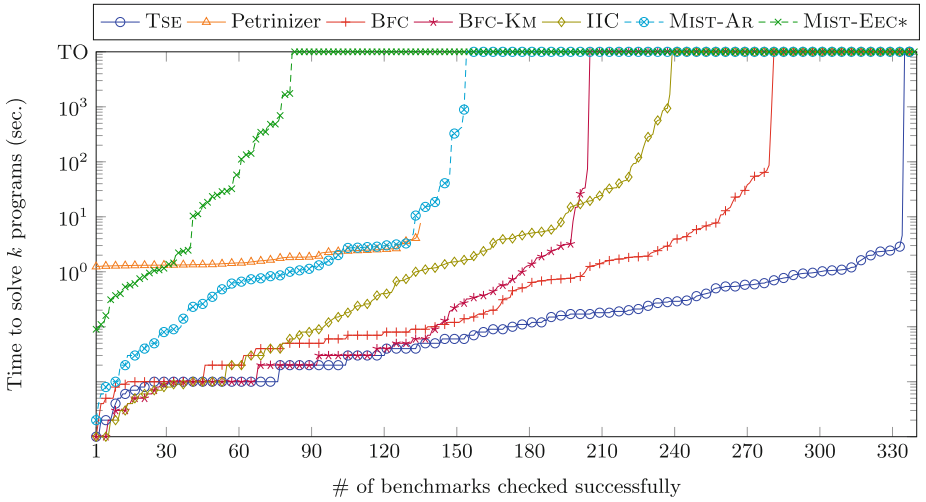

Fig. 6. Comparison on Boolean Programs: cactus plot comparing Tse with prior coverability tools. An entry of the form (k, t) for some curve shows the time t it took to solve the k easiest — for the method associate with that curve — benchmarks (order varies across methods). $*$ indicates that inputs to this tool are Petri nets from Pierre Ganty's translator. (Color figure online)

Efficiency. Figure 5 (left) plots the detailed comparison against Bfc (the most efficient of the competing tools, according to [15]) over each benchmark. Tse clearly beats Bfc on safe instances and remains competitive on unsafe ones. In general, we observe that Bfc outperforms Tse on very small benchmarks which are solved within one second, an effect that can be attributed to the overhead

added by the solver. Figure 5 (right) plots the comparison with Petrinizer[3] [6]. Since Petrinizer does not handle unsafe instances, we focus on safe ones. Tse is invariably faster. Figure 6 is a cumulative plot showing the total time (log-scale) taken to solve the k, for $1 \leq k \leq 339$, easiest of our benchmark problems, for all tools. The results demonstrate that in most cases Tse terminates within 5 seconds. Bfc is the most competitive among other tools.

Summary. We summarize the precision and efficiency results as follows. Given that our tool is sound (it never gives an incorrect answer), and that it does give an answer in the vast majority of the benchmarks we have used, it is prudent to base the comparison on the efficiency results, even against exact tools. Here we observe the strength of Tse especially as a safety prover, i.e. on uncoverable instances. The aggressive search for counterexamples used in Bfc gives that tool a nominal advantage for coverable instances, which is, however, hardly decisive as the running times on those instances tend to be very small.

6 Related Work and Discussion

Groundbreaking results in infinite-state system analysis include the decidability of coverability in *vector addition systems* (VAS) [16], and the work by German and Sistla on modeling communicating finite-state threads as VAS [13]. Numerous results have since improved on the original procedure in [16] in practice [11,12,22,23]. Others extend it to more general computational models, including *well-structured* [9] or *well-quasi-ordered* (wqo) transition systems [2,3].

Explicit-state techniques that combine forward and backward exploration (IIC,Bfc) [15,17] or apply abstraction refinement (Mist-Eec,Mist-Ar) [10,11] have been shown to efficiently decide the coverability problem for large instances, like the ones we consider in our work.

Contrary to the above mentioned complete methods for coverability, [6] follows the direction of trading completeness for performance, by reducing the coverability problem to linear constraint solving and discharging it to a SMT solver, and serves as the inspiration of our work. The thread-state equations we present can be viewed as an instantiation of the marking equation – a classical Petri net technique – in the domain of TTS. In addition to TSE, we extend the approach of [6] by equipping our method with a refinement scheme and a straightforward finite-state search in order to efficiently discover unsafe instances and provide coverability witnesses.

Another incomplete symbolic approach for coverability analysis utilizing the marking equation is presented in [24]. CEGAR is applied on top of the marking equation and is used to guide the solution space of the integer linear constraints. More complex strategies for guiding the solution space were recently introduced in [14]. Such schemes differ from ours, as the solutions to TSE are used as the starting point of the finite state space exploration. If the latter is unsuccessful, TSE are strengthened to allow a simple but efficient refinement scheme.

[3] Petrinizer offers four methods; we use the most powerful: refinement over integers.

Conclusions. Our experimental results demonstrate the trade-off between complete, explicit state exploration and incomplete, symbolic approaches. Verifying safe instances often becomes infeasible when trying to retain completeness, but is shown to be very efficient when posed as a constraint solving problem, as also pointed out in [6]. Our approach aims at *continuing* this trend of devising incomplete yet practical methods for problems of high computational cost [7], by providing an algorithm that fills in the gap of verification of unsafe instances, and efficiently solves the coverability problem in software verification for almost all of our instances.

References

1. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few (parameterized verification through view abstraction). In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013)
2. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symbolic Logic **16**(4), 457–515 (2010)
3. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
4. Cardoza, E., Lipton, R.J., Meyer, A.R.: Exponential space complete problems for petri nets and commutative semigroups: preliminary report. In: STOC, pp. 50–54 (1976)
5. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 356–371. Springer, Heidelberg (2011)
6. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P., Niksic, F.: An SMT-based approach to coverability analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 603–619. Springer, Heidelberg (2014)
7. Esparza, J., Meyer, P.J.: An SMT-based approach to fair termination analysis. In: FMCAD, pp. 49–56 (2015)
8. Even, S.: Graph Algorithms. W. H. Freeman & Co., New York (1979)
9. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theor. Comput. Sci. **256**(1–2), 63–92 (2001)
10. Ganty, P., Raskin, J.F., Van Begin, L.: From many places to few: automatic abstraction refinement for petri nets. Fundam. Inf. **88**(3), 275–305 (2008)
11. Geeraerts, G., Raskin, J.F., Begin, L.V.: Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. J. Comput. Syst. Sci. **72**(1), 180–203 (2006)
12. Geeraerts, G., Raskin, J.-F., Van Begin, L.: On the efficient computation of the minimal coverability set for petri nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 98–113. Springer, Heidelberg (2007)
13. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
14. Hajdu, Á., Vörös, A., Bartha, T.: New search strategies for the petri net CEGAR approach. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 309–328. Springer, Heidelberg (2015)

15. Kaiser, A., Kroening, D., Wahl, T.: A widening approach to multithreaded program verification. ACM Trans. Program. Lang. Syst. **36**(4), 14 (2014)
16. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969)
17. Kloos, J., Majumdar, R., Niksic, F., Piskac, R.: Incremental, inductive coverability. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 158–173. Springer, Heidelberg (2013)
18. Liu, P.: www.ccs.neu.edu/home/lpzun/tse/
19. Liu, P., Wahl, T.: Infinite-state backward exploration of Boolean broadcast programs. In: FMCAD, pp. 155–162 (2014)
20. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
21. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. **6**, 223–231 (1978)
22. Reynier, P.-A., Servais, F.: Minimal coverability set for petri nets: Karp and Miller algorithm with pruning. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 69–88. Springer, Heidelberg (2011)
23. Valmari, A., Hansen, H.: Old and new algorithms for minimal coverability sets. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 208–227. Springer, Heidelberg (2012)
24. Wimmel, H., Wolf, K.: Applying CEGAR to the petri net state equation. Log. Methods Comput. Sci. **8**(3), 827–846 (2012)