Concolic Unbounded-Thread Reachability via Loop Summaries

Peizun Liu(✉) and Thomas Wahl

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

Abstract. We present a method for accelerating explicit-state backward search algorithms for systems of arbitrarily many finite-state threads. Our method statically analyzes the program executed by the threads for the existence of simple loops. We show how such loops can be collapsed *without approximation* into Presburger arithmetic constraints that symbolically summarize the effect of executing the backward search algorithm along the loop in the multi-threaded program. As a result, the subsequent explicit-state search does not need to explore the summarized part of the state space. The combination of concrete and symbolic exploration gives our algorithm a *concolic* flavor. We demonstrate the power of this method for proving and refuting safety properties of unbounded-thread programs.

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. Such programs are therefore designed for an unspecified and unbounded number of parallel threads as a system parameter.

We target in this paper unbounded-thread shared-memory programs where each thread executes a non-recursive Boolean (finite-data) procedure. This model is popular, as it connects to multi-threaded `C` programs via predicate abstraction [4,14], a technique that has enjoyed progress for concurrent programs in recent years [7]. The model is also popular since basic program state reachability questions are decidable. They are also, however, of high complexity: the equivalent *coverability problem* for Petri nets was shown to be EXPSPACE hard [6]. 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 program state reachability analysis for a wide class of multi-threaded Boolean programs. Starting from the target state whose reachability

This work is supported by US National Science Foundation grant no. 1253331.

K. Ogata et al. (Eds.): ICFEM 2016, LNCS 10009, pp. 346–362, 2016.
DOI: 10.1007/978-3-319-47846-3_22

is under investigation, the algorithm proceeds backward by computing *cover preimages*, until either an initial state is reached, or a fixpoint. This search principle is used in several variants, such as the widening-based approach in [16].

In this paper we propose an idea to accelerate backward search algorithms like Abdulla's. The goal is to symbolically *summarize* parts of the finite-state transition system $\mathcal{P}$ (our formal model for Boolean programs) executed by each thread, in a way that reachability in the summarized parts can be reduced to satisfiability of the summary formulas. Prime candidates for such symbolic summaries are *loops* in $\mathcal{P}$. The exploration algorithm may have to traverse them multiple times before a loop fixpoint is reached. We instead wish to summarize the loop statically, obtaining a formula parameterized by the number κ of loop iterations, for the global state reached after κ traversals of the loop.

In order to enable loop summarization, our approach first builds an abstraction $\overline{\mathcal{P}}$ of the transition graph $\mathcal{P}$ (i) that is *acyclic*, and (ii) whose *single-threaded* execution overapproximates the execution of $\mathcal{P}$ by *any* number of threads. Thus, if there is no single-threaded path to the final state in $\overline{\mathcal{P}}$, the algorithm returns "unreachable" immediately. Otherwise, since $\overline{\mathcal{P}}$ is acyclic, there are only finitely many paths that require investigation.

For each such path, we now determine whether it is "summarizable". This is the case if the path either features no loops, or only *simple* loops: single cyclic paths without nesting. We show in this paper how a *precise* summary of the execution of standard backward search across such a path can be obtained as a formula in Presburger arithmetic, the decidable theory over linear integer operations. Conjoined with appropriate constraints encoding the symbolic initial and final states, reachability is then equivalent to the satisfiability of this summary.

Our algorithm can be viewed as separating the branching required in the explicit-state traversal in Abdulla's algorithm [1], and the arithmetic required to keep track (via counting) of the threads in various local states. Structure $\overline{\mathcal{P}}$ is loop-free and can thus be explored path by path. Paths with only simple loops are symbolically summarized into a Presburger formula. The question whether the target state is reachable along this path can then often be answered quickly, in part since the formulas tend to be easy to decide. Other parts are explored using standard explicit-state traversal, restricted to the narrow slice of the state space laid out by this path, which gives our algorithm a *concolic* flavor.

We conclude this paper with experiments that investigate the performance gain of our acceleration method applied to backward search. The results demonstrate that transition systems obtained from Boolean programs, which feature "execution discipline" enforced by the control flow, are better suited to path-wise acceleration than Petri nets, which often encode rule-based (rather than program-based) transition systems and thus feature fewer summarizable paths.

Proofs to claims made in this paper can be found in the Appendix of [21].

2 Thread-Transition Diagrams and Backward Search

We assume multi-threaded programs are given in the form of an abstract state machine called *thread transition diagram* [16]. Such a diagram reflects the

replicated nature of programs we consider: programs consisting of threads executing a given procedure defined over shared ("global") and (procedure-)local variables. A thread transition diagram (TTD) is a tuple $\mathcal{P} = (S, L, R)$, where

- S is a finite set of *shared* states;
- L is a finite set of *local* states;
- $R \subseteq (S \times L) \times (S \times L)$ is a (finite) set of *edges*.

An element of $V = S \times L$ is called *thread state*. We write $(s_1, l_1) \to (s_2, l_2)$ for $((s_1, l_1), (s_2, l_2)) \in R$. We assume the TTD has a *unique* initial thread state, denoted $t_I = (s_I, l_I)$; the case of multiple initial thread states is discussed in Appendix A of [21]. An example of a TTD is shown in Fig. 1(a).

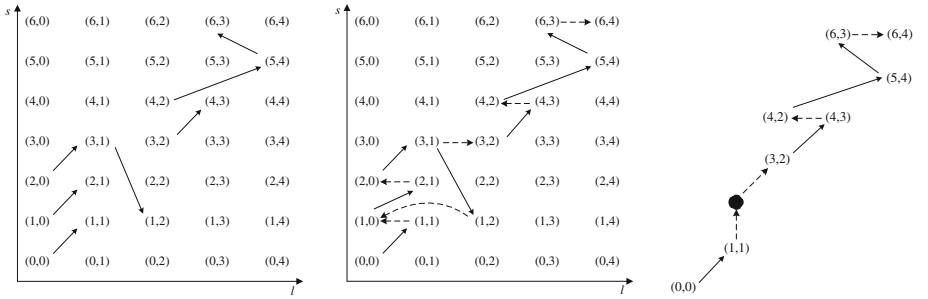


Fig. 1. (a) A thread transition diagram $\mathcal{P}$ (initial state $t_I = (0, 0)$); (b) the Expanded TTD $\mathcal{P}^+$ with a path σ^+; (c) the SCC quotient graph $\overline{\mathcal{P}}$ of $\mathcal{P}^+$, with quotient path $\overline{\sigma}$. The black disc represents the loop in σ^+ (the other SCCs are trivial)

A TTD gives rise to a family, parameterized by n, of transition systems $\mathcal{P}_n = (V_n, R_n)$ over the state space $V_n = S \times L^n$, whose states we write in the form $(s|l_1, \ldots, l_n)$. This notation represents 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 $\mathcal{P}_n$, forming the set R_n, are written in the form $(s|l_1, \ldots, l_n) \rightarrowtail (s'|l'_1, \ldots, l'_n)$. This transition is defined exactly if there exists $i \in \{1, \ldots, n\}$ such that $(s, l_i) \to (s', l'_i)$ and for all $j \neq i$, $l_j = l'_j$. That is, our execution model is asynchronous: each transition affects the local state of at most one thread.[1]

The initial state set of $\mathcal{P}_n$ is $\{s_I\} \times \{l_I\}^n$. A *path* of $\mathcal{P}_n$ is a finite sequence of states in V_n whose first element is initial, and whose adjacent elements are related by R_n. A thread state $(s, l) \in S \times L$ is *reachable in* $\mathcal{P}_n$ if there exists a path in $\mathcal{P}_n$ ending in a state with shared state s and some thread in local state l.

A TTD also gives rise to an infinite-state transition system $\mathcal{P}_\infty = (V_\infty, R_\infty)$, whose set of states/transitions/initial states/paths is the union of the sets of states/transitions/initial states/paths of $\mathcal{P}_n$, for all $n \in \mathbb{N}$. We are tackling in this

[1] *Dynamic thread creation* is discussed at end of Sect. 6.

paper the *thread state reachability problem*: given a TTD $\mathcal{P}$ and a *final* thread state (s, l_F), is (s, l_F) reachable in $\mathcal{P}_\infty$? It is easy to show that this question is decidable, by reducing $\mathcal{P}_\infty$ to a *well quasi-ordered system* (WQOS) [1]: let the *covers* relation $\succeq$ over V_∞ be defined as follows:

$$(s|l_1, \ldots, l_n) \succeq (s'|l'_1, \ldots, l'_{n'})$$

whenever $s = s'$ and for all $l \in L$, $|\{i : l_i = l\}| \geq |\{i : l'_i = l\}|$. The latter inequality states that the number of threads in local state l "on the left" is at least the number of threads in local state l "on the right". Relation $\succeq$ is a well quasi-order on V_∞, and $(\mathcal{P}_\infty, \succeq)$ satisfies the definition of a WQOS, in particular the *monotonicity* property required of $\succeq$ and $\rightarrowtail$. The proof of this property exploits the *symmetry* of the multi-threaded system: the threads execute the same program $\mathcal{P}$: a state $(s|l_1, \ldots, l_n)$ can be compressed without loss of information into the counter notation $(s|n_1, \ldots, n_{|L|})$, where $n_l = |\{i : l_i = l\}|$.

The thread state reachability question can now be cast as a *coverability problem*, which is decidable but of high complexity, e.g. EXPSPACE-hard for standard Petri nets [6], which are equivalent in expressiveness to infinite-state transition systems obtained from TTD [16].

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

$$\textsc{CovPre}(w) = \min\{p : \exists w' \succeq w : p \rightarrowtail w'\} \quad (1)$$

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

Algorithm 1. $\textsc{Bws}(M, I, q)$

Input: initial states I,
final state $q \notin I$

1: $W := \{q\}$; $U := \{q\}$
2: **while** $\exists w \in W$
3: $W := W \setminus \{w\}$
4: **for** $p \in \textsc{CovPre}(w) \setminus \uparrow U$
5: **if** $p \in I$ **then**
6: "q coverable"
7: $W := \min(W \cup \{p\})$
8: $U := \min(U \cup \{p\})$
9: "q not coverable"

Algorithm 1: Infinite-state backward search. Symbol $\uparrow U$ stands for the *upward closure* of U: $\uparrow U = \{u' : \exists u \in U : u' \succeq u\}$.

Strongly Connected Components. In this paper we also frequently make use of the following standard notions. Given a directed graph G, a *strongly connected component* (SCC) is a maximal set C of vertices such that for any two vertices c_1 and c_2 in C, there is a path in C from c_1 to c_2. If the subgraph of G induced by C has no edge, C is called *trivial*.

The *SCC quotient graph* $\overline{G}$ of G has exactly one vertex for each SCC of G, and no other vertices; we identify each vertex of $\overline{G}$ with the SCC it represents. An edge (C_1, C_2) exists in $\overline{G}$ whenever $C_1 \neq C_2$ and there is a G-edge from some vertex in C_1 to some vertex in C_2. For a vertex v, we denote by $\overline{v}$ the unique SCC that v belongs to (hence, by identification, $\overline{v}$ is also a vertex in $\overline{G}$). Since each cycle of G is contained entirely in one SCC, and nodes in $\overline{G}$ have no self-loops, $\overline{G}$ is *acyclic*.

3 Pathwise Unbounded-Thread Reachability: Overview

Our approach for accelerating backward reachability analysis is two-phased. The first phase constructs from $\mathcal{P}$ an abstract structure $\overline{\mathcal{P}}$, with the property that any thread state reachable in $\mathcal{P}_\infty$ (i.e., for any number of threads) is also reachable in $\overline{\mathcal{P}}$ **when executed by a single thread**. Structure $\overline{\mathcal{P}}$ thus overapproximates the thread-state reachability problem for $\mathcal{P}$ to a much simpler sequential reachability problem. Technically, the abstraction first adds certain edges to $\mathcal{P}$, and then collapses strongly connected components to obtain $\overline{\mathcal{P}}$, which is hence acyclic. Note that this first phase performs no exploration and is in fact independent of the underlying reachability algorithm being accelerated.

In the second phase, we analyze each path $\overline{\sigma}$ in the acyclic structure $\overline{\mathcal{P}}$ from t_I to t_F separately, if any. We now distinguish: if $\overline{\sigma}$ visits only *simple* SCCs, by which we mean SCCs that represent simple loops, then we call $\overline{\sigma}$ simple, and we precisely summarize the effect of traversing the path using Presburger formulas.[2] Instead of executing Algorithm 1, we solve these Presburger constraints, in effect accelerating the algorithm, losslessly, along loop-free path segments and simple loops. If $\overline{\sigma}$ visits at least one *spaghetti SCC* — an SCC that represents more than a simple loop (e.g. a loop nest) — then we call $\overline{\sigma}$ *spaghetti* as well and explore it using Algorithm 1, restricted to the edges along $\overline{\sigma}$.

At the end of this section we illustrate the overall process in more detail. We first introduce the acyclic quotient structure $\overline{\mathcal{P}}$.

A Single-Threaded Abstraction of $\mathcal{P}_\infty$. A key operation employed during backward search is what we call *expansion* of a global state: the addition of a thread in a suitable local state during the computation of the cover preimage (1). We can simulate the effect of such expansions *without adding threads*, by allowing a thread to change its local state in certain disciplined ways. To this end, we expand the TTD data structure as follows.

Definition 1. *Given a TTD $\mathcal{P} = (S, L, R)$, an **expansion edge** is an edge of the form $((s, l), (s, l'))$ (same shared state) such that $l \neq l'$ and the following holds:*

- *there exists an edge of the form $\ldots \to (s, l)$ in R,* ***and***
- *there exists an edge of the form $(s, l') \to \ldots$ in R, or $(s, l') = (s_F, l_F)$.*

*The **Expanded TTD (ETTD)** of $\mathcal{P}$ is the structure $\mathcal{P}^+ = (S, L, R^+)$ with $R^+ = R \cup \{e : \ e \text{ is an expansion edge}\}$.*

To distinguish the edge types in $\mathcal{P}^+$, we speak of *real edges* (in R) and expansion edges. Intuitively, expansion edges close the gap between two real edges whose target and source, respectively, differ only in the local state. This can be seen in Fig. 1(b), which shows the ETTD generated from the TTD in Fig. 1(a). In

[2] Simple SCC nodes (representing a simple loop) are not to be confused with *trivial* SCC nodes (representing a single node). Simple nodes are by definition non-trivial.

the graphical representation, expansion edges run horizontally and are shown as dashed arrows $(s, l) \dashrightarrow (s', l')$.

To facilitate the identification and treatment of loops, we collapse the ETTD $\mathcal{P}^+$ into its (acyclic) SCC quotient graph, denoted $\overline{\mathcal{P}}$. An example is shown in Fig. 1(c). For ease of presentation, we assume that both the initial and final thread states t_I and t_F of $\mathcal{P}$ form single-node SCCs in $\overline{\mathcal{P}}$, i.e. loops occur only in the interior of a path. This can be enforced easily using artificial states.

Being acyclic, the quotient graph $\overline{\mathcal{P}}$ contains only finitely many paths between any two nodes. It also has another key property that makes it attractive for our approach. Let us interpret $\overline{\mathcal{P}}$ as a sequential transition system. That is, when we speak of *reachability of a thread state* and *paths* in $\overline{\mathcal{P}}$, we assume $\overline{\mathcal{P}}$ is executed by a single thread from t_I. (In contrast, the semantics of $\mathcal{P}$ is defined via the unbounded-thread transition system $\mathcal{P}_\infty$.) Given these stipulations, $\overline{\mathcal{P}}$ overapproximates $\mathcal{P}$, in the following sense:

Lemma 2. *If thread state t_F is reachable in $\mathcal{P}_\infty$, then t_F is also reachable in $\overline{\mathcal{P}}$.*

By Lemma 2, if t_F is not reachable from t_I in $\overline{\mathcal{P}}$ (a simple sequential reachability problem), it is not reachable in $\mathcal{P}_\infty$. In that case our algorithm immediately returns "unreachable" and terminates. If t_F is reachable in $\overline{\mathcal{P}}$, we cannot conclude reachability in $\mathcal{P}_\infty$, as can be seen from Fig. 1: thread state $t_F := (6, 4)$ is easily seen to be unreachable in $\mathcal{P}_\infty$, no matter how many threads execute the diagram $\mathcal{P}$ in (a). But t_F is obviously sequentially reachable in $\overline{\mathcal{P}}$ (c). In the rest of this paper we describe how to decide, for each path $\overline{\sigma}$ in $\overline{\mathcal{P}}$ from t_I to t_F, whether it actually witnesses reachability of t_F in $\mathcal{P}_\infty$.

To give an overview of this process, consider a quotient path $\overline{\sigma}$ with one simple SCC node. One such path is schematically depicted in Fig. 2, where we have zoomed in on the SCC node ℓ_i in order to show the simple loop of $\mathcal{P}^+$ collapsed inside it. To analyze reachability of t_F in $\mathcal{P}_\infty$, we first consider the path segment from t_F to the *exit point* of the loop (see Fig. 2). The exit point is the unique node of $\mathcal{P}^+$ abstracted by SCC node ℓ_i that is first encountered when the quotient path $\overline{\sigma}$ is explored *backward*.

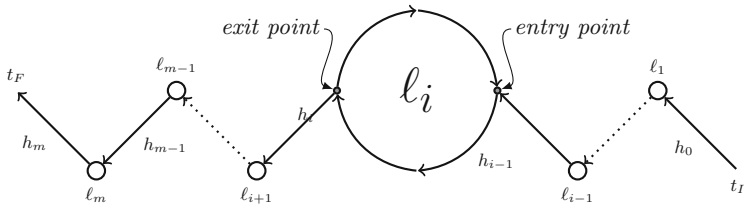


Fig. 2. A path $\overline{\sigma}$ in the acyclic structure $\overline{\mathcal{P}}$ with a non-trivial and magnified SCC node ℓ_i, representing some kind of loop structure in $\mathcal{P}^+$

Our approach builds a symbolic summary for this path segment. We then do the same for the simple loop collapsed inside ℓ_i, and for the path from the *entry point* of the loop back to t_I. These summaries are combined conjunctively

into a single Presburger expression φ over a parameter κ that represents the number of iterations through the loop represented by ℓ_i. We now conjoin φ with the constraint that, when backward-reaching t_I along $\overline{\sigma}$, no thread resides in any local state *other than* l_I. This condition ensures that the global state constructed via symbolic backward execution is of the form $\{s_I\} \times \{l_I\}^n$, i.e. it is initial. The claim that t_F is reachable in $\mathcal{P}_\infty$ is then equivalent to the satisfiability of the overall formula; a satisfying assignment to κ specifies how many times the loop in ℓ_i needs to be traversed.

In Sects. 4 and 5 we describe how loop-free path segments and simple loops, respectively, are summarized, to obtain a symbolic characterization.

4 Presburger Summaries for Loop-Free Path Segments

Consider a path segment $\overline{\sigma}$ in $\overline{\mathcal{P}}$ with only trivial (singleton) SCC nodes in its interior; we call such segments *loop-free*. (The start and end state of $\overline{\sigma}$ may still be non-trivial SCC nodes; the loops contracted by these SCC nodes are not considered in this section.) The real and expansion edges along $\overline{\sigma}$ suggest a *firing sequence* of edges during an exploration of $\mathcal{P}_\infty$ using Algorithm 1. Each real edge corresponds to a thread state change for a single thread; each expansion edge corresponds to the expansion of the current global state. More precisely, given a global state of the form $(s'|l'_1, \ldots, l'_n)$, Algorithm 1 computes cover preimages (Eq. (1)), by first firing edges of R backward whose targets equal one of the thread states (s', l'_i). Second, for each edge e whose target (s', l') (with shared state s') does not match any of the thread states (s', l'_i), Algorithm 1 expands the global state, by adding one thread in local state l', followed by firing e backward, using the added thread.[3]

The steps performed by Algorithm 1 can be expressed in terms of updates to local-state counters. Let edge e be of the form $(s, l) \to (s', l')$. If the current global state $(s'|l'_1, \ldots, l'_n)$ contains a thread in local state l', firing e backward amounts to decrementing the counter $n_{l'}$ for the target l', and incrementing the counter n_l for the source l. If the current global state does not contain a thread in local state l', we first expand the state by adding such a thread, followed by firing e backward. Together the step amounts exactly to an increment of n_l.

We can execute these steps *symbolically*, instead of concretely, by traversing path segment $\overline{\sigma}$ backward and encoding the corresponding counter updates described in the previous paragraph as logical constraints over the local-state counters. The constraints are expressible in *Presburger* (linear integer) arithmetic. To demonstrate this, we introduce some light notation. For $x, y \in \mathbb{Z}$ and $b \in \mathbb{N}$, let $x \oplus_b y = \max\{x+y, b\}$. Intuitively, $x \oplus_b y$ is "$x+y$ but at least b". When $b = 0$, we omit the subscript. We also use $x \ominus_b y$ as a shorthand for $x \oplus_b (-y)$ $(= \max\{x - y, b\})$. For example, $x \ominus 1$ equals $x - 1$ if $x \geq 1$, and 0 otherwise. Neither $\oplus_b$ nor $\ominus_b$ are associative: $(1 \oplus 2) \oplus -3 = 0 \neq 1 = 1 \oplus (2 \oplus -3)$. We therefore stipulate: these operators associate from left to right, and they have the same binding power as $+$ and $-$.

[3] We exploit the fact that cover preimages in systems induced by TTDs increase the number of threads in a state by at most 1 (see [20, Lemma 1] for a proof).

Algorithm 2. Summary of a loop-free path segment

Input: path $\overline{\sigma} = t_1, \ldots, t_k$ in $\overline{\mathcal{P}}$, i.e. $(t_i, t_{i+1}) \in R^+$ for $1 \leq i < k$; local state l

```
1: e_i := (t_i, t_{i+1}) for 1 ≤ i < k , (s_i, l_i) := t_i for 1 ≤ i ≤ k
2: summary := "n_l"                                   ▷ summary is a string
3: for i := k − 1 downto 1
4:    if e_i ∈ R and l_i = l then
5:       summary := summary."+1"                      ▷ . = string concatenation
6:    if e_i ∈ R and l_{i+1} = l then
7:       summary := summary."-1"
8:    if e_i ∈ R^+ \ R and l_i = l then
9:       summary := summary."⊖1+1"
10: return summary
```

Operators $\oplus/\ominus$ in Presburger formulas are syntactic sugar: we can rewrite a formula Γ containing $x \oplus_b y$, using a fresh variable v per occurrence:

$$\Gamma \quad \equiv \quad (\Gamma|_{(x \oplus_b y) \to v}) \ \wedge \ ((x + y \geq b \wedge v = x + y) \vee (x + y < b \wedge v = b)) \qquad (2)$$

where $\alpha|_{\beta \to \gamma}$ denotes substitution of γ for β in α.

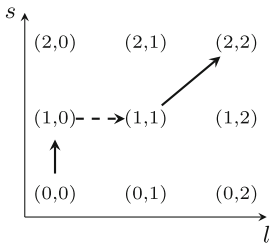
The *summary* of loop-free path segment $\overline{\sigma}$ is computed separately for each local state l: Algorithm 2 symbolically executes $\overline{\sigma}$ backward; for certain edges a "contribution" to counter n_l is recorded, namely for each edge of R^+ that is adjacent to local state l, but only if it is real, or it is an expansion edge that starts in local state l. Note that the three **if** clauses in Algorithm 2 are not disjoint: the first two both apply when edge e_i is "vertical": it both enters and exits local state l. In this case the two contributions cancel out.

The summary of path $\overline{\sigma}$ for local state l defines a function $\Sigma_l \colon \mathbb{N} \to \mathbb{N}$ that summarizes the effect of path $\overline{\sigma}$ on counter n_l. The summary functions for the short path in Fig. 3 are shown next to the figure. These examples illustrate how we can encode a loop-free quotient path into a quantifier-free Presburger formula. The formula for $\Sigma_0(n_0)$ implies that if we traverse the path backward from a state with $n_0 = 0$ threads in local state 0, at the end there will be $\Sigma_0(0) = 0 \ominus 1 + 1 = 1$ thread in local state 0. If we start with $n_0 = 1$, we also end up with $n_0 = 1$. Note that the path cannot be traversed backward starting with $n_2 = 0$, since its endpoint is thread state $(2, 2)$.

Non-trivial SCC nodes along $\overline{\sigma}$ are contractions of loops in the expanded structure $\mathcal{P}^+$, to the effect that paths in $\mathcal{P}^+$ are no longer finite; their summaries cannot be obtained by symbolic execution. Instead we will determine a precise summary of simple loops that is parameterized by the number κ of times the loop is executed. Spaghetti loops are discussed in Sect. 6.

5 Presburger Summaries for Simple Loops

In this section we generalize path summaries to the case of simple SCCs, formed by a single *simple loop*, i.e., a single cyclic path without repeated inner nodes.



Summary functions for local states $l = 0, 1, 2$:

$$\Sigma_0(n_0) = n_0 \ominus 1 + 1 - 1 + 1 = n_0 \ominus 1 + 1$$
$$\Sigma_1(n_1) = n_1 + 1$$
$$\Sigma_2(n_2) = n_2 - 1$$

Examples:

$$\Sigma_0(0) = 1,\ \Sigma_0(1) = 1,\ \Sigma_1(0) = 1,\ \Sigma_2(1) = 0\ .$$

Fig. 3. A loop-free quotient structure $\overline{\mathcal{P}}$ with a vertical real edge

We aim at an exact solution in the form of a closed expression for the value of local state counter n_l after Algorithm 1 traverses the loop some number of times κ.

In this section, since we need to "zoom in" to SCCs collapsed into single nodes in $\overline{\mathcal{P}}$, we instead look at paths in $\mathcal{P}^+$. Recall that for a loop-free path σ^+, the value of counter n_l after Algorithm 1 traverses σ^+ can be computed using σ^+'s path summary function Σ_l, determined via symbolic execution (Algorithm 2). In the case that σ^+ is a loop, we would like to obtain a summary formula parameterized by the number κ of times the loop is executed (we cannot replicate σ^+'s summary function κ times, since κ is a variable).

To this end, let $\sigma^+ = t_1, \ldots, t_k$ with $t_k = t_1$ be a loop in $\mathcal{P}^+$, and define $(s_i, l_i) := t_i$ for $1 \leq i \leq k$. Let

$$\begin{aligned}\delta_l = |\{i : 1 \leq i < k :\ (t_i, t_{i+1}) \in R \wedge l_i \quad = l\}| \ - \\ |\{i : 1 \leq i < k :\ (t_i, t_{i+1}) \in R \wedge l_{i+1} = l\}|\end{aligned} \tag{3}$$

be the *real-edge summary* $\delta_l \in \mathbb{Z}$ of σ^+, i.e. the number of *real* edges along σ^+ that start in local state l, minus the number of *real* edges along σ^+ that end in l. Value δ_l summarizes the total contribution by real edges to counter n_l as path σ^+ is traversed backward: real edges starting in l increment the counter, those ending in l decrement it. Let further $b_l = \Sigma_l(1)$ if σ^+ ends in local state l (in this case the backward traversal must start with at least 1 thread in l), and $b_l = \Sigma_l(0)$ otherwise.

Theorem 3. *Let superscript* $^{(\kappa)}$ *denote* κ *function applications. Then, for* $\kappa \geq 1$,

$$\Sigma_l^{(\kappa)}(n_l) = n_l \oplus_{b_l} \delta_l \oplus_{b_l} (\kappa - 1) \cdot \delta_l. \tag{4}$$

Recall that $\oplus$ is not associative (it associates from left to right); the right-hand side of Eq. (4) can generally not be simplified to $n_l \oplus_{b_l} \kappa \cdot \delta_l$. Intuitively, the term $n_l \oplus_{b_l} \delta_l$ marks the contribution to counter n_l of the first loop traversal, while $(\kappa - 1) \cdot \delta_l$ marks the contribution of the remaining $\kappa - 1$ traversals.

Example. We show how the *un*reachability of thread state $(6, 4)$ for the TTD in Fig. 1 is established. For each local state $l \in \{0, \ldots, 4\}$, the following constraints

are obtained (after simplifications) from summaries of the loop-free path segment from $(6,4)$ to $(3,1)$ ("loop exit point"), the loop inside the SCC node (black disc) using Theorem 3, and the loop-free path segment from $(1,0)$ ("loop entry point") via $(1,1)$ to the initial thread state $(0,0)$. Parameter κ is the number of times the loop is executed:

$$\begin{array}{lllllcc}
n_0 : 0 \oplus_0 & 0 \oplus_2 & 2 \oplus_2 & (\kappa - 1) \cdot & 2 \oplus_3 & 3 & \geq 1 \\
n_1 : 0 \oplus_1 & 0 \oplus_1 & -1 \oplus_1 & (\kappa - 1) \cdot & -1 \oplus_0 & -3 & = 0 \\
n_2 : 0 \oplus_2 & 2 \oplus_0 & -1 \oplus_0 & (\kappa - 1) \cdot & -1 \oplus_0 & 0 & = 0 \\
n_3 : 0 \oplus_0 & -2 \oplus_0 & 0 \oplus_0 & (\kappa - 1) \cdot & 0 \oplus_0 & 0 & = 0 \\
n_4 : 1 \oplus_1 & 0 \oplus_0 & 0 \oplus_0 & (\kappa - 1) \cdot & 0 \oplus_0 & 0 & = 0
\end{array}$$

The equation for n_4 simplifies to $1 = 0$ and thus immediately yields unsatisfiability. Since there is only one path in $\overline{\mathcal{P}}$, we conclude unreachability of $t_F = (6,4)$. In contrast, for target thread state $(6,3)$, the equations for n_3 and n_4 both reduce to *true*. The conjunction of all five equations reduces to $1 \oplus_0 (\kappa - 1) \cdot (-1) = 0$. This formula is satisfied by $\kappa = 2$, witnessing reachability of $(6,3)$ via a path containing two full iterations of the loop inside the SCC.

6 Pathwise Unbounded-Thread Reachability

Consider an SCC along quotient path $\overline{\sigma}$ that represents several distinct simple loops in $\mathcal{P}^+$. An example is an SCC with two loops A and B that have one point in common and form an "eight" ∞. Such a double loop features paths of the form $(A|B)^*$, where in each iteration there is a choice between A and B. Our loop acceleration technique from Sect. 5 does not apply to such paths.

To solve this problem, we exploit the synergy between the pathwise analysis suggested by the acyclic structure $\overline{\mathcal{P}}$, and the fact that certain — namely, simple — paths can be processed using the technique described in Sects. 4 and 5. Spaghetti paths are explored by Algorithm 1, but restricted to the narrow "slice" of $\mathcal{P}$ marked by the quotient path in $\overline{\mathcal{P}}$.

This algorithm is shown in Algorithm 3. It takes as input the TTD $\mathcal{P}$, as well as the initial and final thread states, t_I and t_F. The algorithm begins by building the quotient structure $\overline{\mathcal{P}}$. This acyclic structure is now analyzed pathwise. For each path $\overline{\sigma}$ from t_I to t_F in $\overline{\mathcal{P}}$, we first decide whether it is spaghetti or simple.

- If $\overline{\sigma}$ is spaghetti (visits some spaghetti SCCs), we explore it using Algorithm 1 (Line 4). More precisely, let $\mathcal{P}\big|_{\overline{\sigma}}$ be the restriction of the given TTD to the edges along $\overline{\sigma}$, including any edges collapsed inside SCCs. Let further $(\mathcal{P}\big|_{\overline{\sigma}})_\infty$ be the infinite-state transition system derived from $\mathcal{P}\big|_{\overline{\sigma}}$ as described in Sect. 2. We pass this transition system to procedure Bws (Algorithm 1), along with the unchanged set of initial states, and the unchanged final state (which is also the end-point of $\overline{\sigma}$). If this invocation results in "coverable", t_F is reachable in $\mathcal{P}_\infty$ from t_I, which is hence returned in Line 5.

Algorithm 3. Pathwise Reachability

Input: TTD $\mathcal{P}$, thread states t_I, t_F
1: $\mathcal{P}^+$:= expanded TTD, $\overline{\mathcal{P}}$:= SCC quotient graph of $\mathcal{P}^+$
2: **for all** path $\overline{\sigma}$ in $\overline{\mathcal{P}}$ from t_I to t_F
3: **if** $\overline{\sigma}$ is spaghetti **then**
4: **if** $\textsc{Bws}((\mathcal{P}|_{\overline{\sigma}})_\infty, \cup_{n\in\mathbb{N}}\{s_I\}\times\{l_I\}^n, t_F)$ = "t_F coverable" **then**
5: **return** "t_F reachable in $\mathcal{P}_\infty$ from t_I"
6: **else**
7: m := number of non-trivial SCCs visited by $\overline{\sigma}$ ▷ these SCCs are all simple
8: $\phi(\kappa_1, \ldots, \kappa_m)$:= *Presburger summary for* $\overline{\sigma}$ ▷ Sect. 4, 5
9: **if** $\phi(\kappa_1, \ldots, \kappa_m)$ satisfiable **then**
10: **return** "t_F reachable in $\mathcal{P}_\infty$ from t_I"
11: **return** "t_F unreachable in $\mathcal{P}_\infty$ from t_I"

- If $\overline{\sigma}$ is simple (does not visit any spaghetti SCCs), we can accelerate exploration along it using the techniques introduced in Sects. 4 and 5. We build a Presburger summary for the path, parameterized by the loop iteration counts κ_i, one for each loop.[4] If this formula is satisfiable, again we have that t_F is reachable in $\mathcal{P}_\infty$ from t_I. The assignment to the κ_i gives the number of times each loop needs to be traversed; from this data a multi-threaded path through $\mathcal{P}$ can easily be constructed.

If none of the paths $\overline{\sigma}$ results in the answer "coverable" by either concrete or symbolic exploration, t_F is unreachable in $\mathcal{P}_\infty$ from t_I, which is hence returned as the answer. Note that this happens in particular if there is no path at all from t_I to t_F in $\overline{\mathcal{P}}$.

Correctness. Algorithm 3 terminates since $\overline{\mathcal{P}}$ is acyclic, so the loop in Line 2 goes through finitely many iterations. Partial correctness follows from the following two claims. Let $\overline{\sigma}$ be a quotient path considered in Line 2.

1. If $\overline{\sigma}$ is spaghetti, then Algorithm 3 outputs "reachable" in Line 5 iff t_F is reachable in $\mathcal{P}_\infty$ along the edges represented by $\overline{\sigma}$.
2. If $\overline{\sigma}$ is simple, then Algorithm 3 outputs "reachable" in Line 10 iff t_F is reachable in $\mathcal{P}_\infty$ along the edges represented by $\overline{\sigma}$.

Claim 1 is proved using soundness and completeness of Algorithm 1. Claim 2 is proved using Theorem 3. Given these claims, we obtain:

Corrolary 4 (Soundness). *If Algorithm 3 returns "reachable" (Line 5 or Line 10) or "unreachable" (Line 11), then t_F is reachable or unreachable, respectively, in $\mathcal{P}_\infty$.*

[4] Loop-free paths ($m = 0$) can be processed either using Algorithm 1, or via summaries.

Proof

- If Algorithm 3 returns "reachable", then it does so for some $\overline{\sigma}$, in Lines 5 or 10. The fact that t_F is actually reachable in $\mathcal{P}_\infty$ follows from one of the two claims above, depending on whether $\overline{\sigma}$ is spaghetti or simple.
- If Algorithm 3 returns "unreachable", then it does not reach Lines 5 or 10, for *any* $\overline{\sigma}$. By the above two claims, t_F is not reachable in $\mathcal{P}_\infty$ along the edges represented by any quotient path. The fact that then $\overline{\sigma}$ is not reachable in $\mathcal{P}_\infty$ at all follows from the proof of Lemma 2: the proof shows that, if t_F is reachable, then there exists a quotient path in $\overline{\mathcal{P}}$ from t_I to t_F such that t_F is reachable in $\mathcal{P}_\infty$ along the edges represented by that quotient path. □

Implementation. Our technique is implemented in a reachability checker named CUTR[5]. We discuss some details on the implementation of Algorithm 3 in CUTR.

Line 2 selects potential paths in $\overline{\mathcal{P}}$. Since we can abort the algorithm once a path is found that witnesses reachability, it makes sense to rank the paths by "promise" of ease of processing: we begin with loop-free paths, i.e. those with only trivial SCCs, followed by paths with simple SCCs whose edges are all real, followed by paths with simple SCCs that feature expansion edges. Finally we select paths with spaghetti loops inside SCCs. The length of a path is secondary.

In order to call BWS in Line 4 on the TTD restricted to the edges represented by $\overline{\sigma}$, there is no need to construct $\mathcal{P}|_{\overline{\sigma}}$ a priori. Instead, when computing cover preimages, we make sure to only fire TTD edges belong $\overline{\sigma}$ and its loops.

To keep our computational model simple, we have excluded from the formalization in Sect. 2 *dynamic thread creation*, where threads are spawned during the execution of the program. This feature does not formally add expressive power, but is often included for its presence in multi-threaded software. Our implementation does support thread creation. Symbolically backward-executing a thread creation edge is straightforward: the counter of the local state of the spawned thread must be decreased, since that thread does not exist in the source state. Our implementation performs some book-keeping to ensure the backward-executability of such an edge: both the local state of the spawned thread, as well as that of the spawning thread must exist in the successor state, since the spawning thread does not change its state (it only side-effects the thread creation).

7 Empirical Evaluation

In this section we provide experimental results obtained using CUTR. The goal of the experiments is to measure the performance impact of the presented approach compared to the backward search Algorithm 1. We expect our approach to improve the latter, as it is short-cutting standard backward exploration across simple loops and linear path segments. The question is whether solving Presburger equations instead of concretely exploring loops actually amounts to speed-up.

[5] CUTR "=" **C**oncolic **U**nbounded-**T**hread **R**eachability analysis.

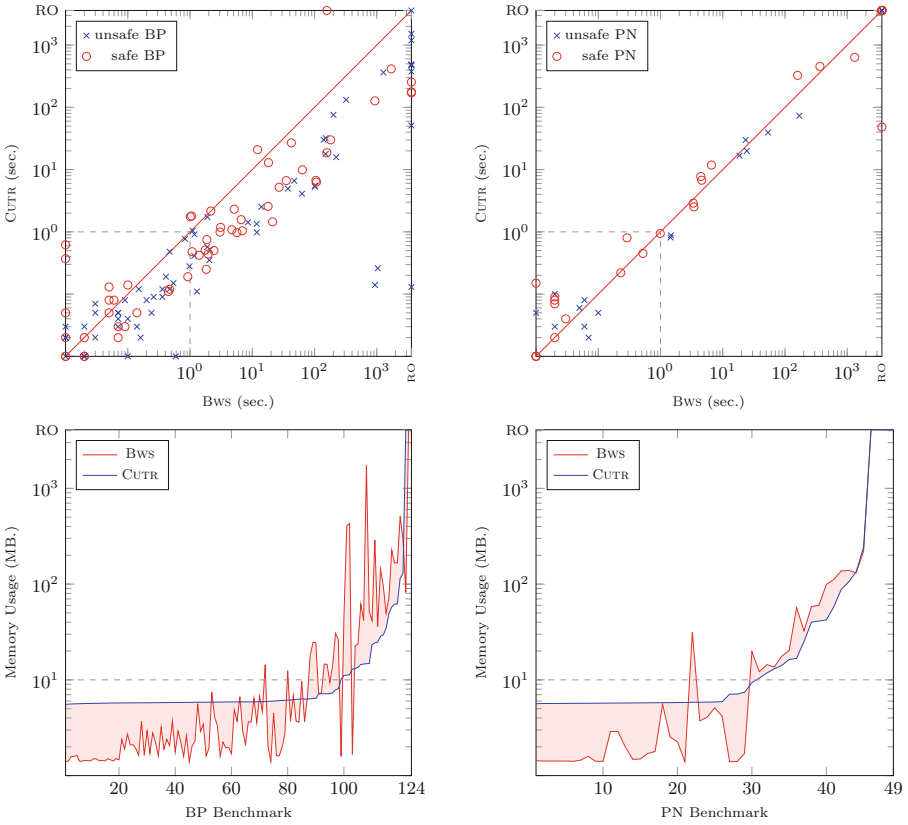


Fig. 4. Performance impact (BP/PN = TTD from BP/PN). RO stands for "out of resources": the run reached the time or the memory limit before producing a result.
• Top row shows the comparisons of execution time. Left: comparison on BPs; Right: comparison on PNs. Each plot represents execution time on one example.
• Bottom row shows the comparison of memory usage. Left: comparison on BPs; Right: comparison on PNs. The curves are sorted by the memory usage of Cutr.

Experimental Setup. We collected an extensive set of benchmarks, 173 in total, which is organized into two suites. The first suite contains 124 TTDs obtained from Boolean programs (BPs), which are in turn obtained from `C` source programs (taken from [16]) via predicate abstraction. As TTDs are equivalent in expressiveness to certain forms of Petri nets [8,16], we include PN examples in our benchmark collection. The second suite therefore contains 49 TTDs obtained from PNs (taken from [8]). While PNs are not the main focus of this work, we were hoping to get insights into how complex concurrency affects our approach, as the PNs available to us exhibited more challenging concurrent behaviors than the BPs. The table on the right shows size ranges of the benchmarks.

We use Z3 (v4.3.2) [22] as the Presburger solver. For each benchmark, we consider verification of a safety property. In the case of BP examples, the property is specified via an assertion. There are 87 safe instances in total: 56 of the BPs, and 31 of the PNs. All experiments are performed on a 2.3 GHz Intel Xeon machine with 64 GB memory, running 64-bit Linux. Execution time is limited to 30 min, and memory to 4 GB. All benchmarks and our tool are available online[6].

BP	Min.	Max.
$\|S\|$	5	257
$\|L\|$	14	4097
$\|R\|$	18	20608
PN	Min.	Max.
$\|S\|$	6	18234
$\|L\|$	6	332
$\|R\|$	13	27724

Comparison. We first consider TTDs obtained from BPs, the target of this work. The runtime comparison results are given in the top left part of Fig. 4. The results demonstrate that Cutr performs much better than Bws. In some cases, runs that time out in Bws can be successfully solved by Cutr within 30 min; in contrast, there is only one example such that Bws successfully completes while Cutr runs time out. The latter situation can be explained by the path explosion: there are more than 5000 paths for this example.

We now consider TTDs obtained from PNs; see top right of Fig. 4. Here we see little performance difference. Investigating this further, we found that for Petri nets the density of TTD edges is higher (explained by their more complex structure and the relatively less organization and control in the concurrent systems represented by Petri nets, compared to programs). This has two consequences: (i) there are few but large SCCs, and (ii) most of them are spaghetti. As a result, there are few paths through the quotient structure, and almost all of them are explored via calling Bws. This makes the whole process essentially equivalent to a single call to Bws.

The curves at the bottom left of Fig. 4 illustrate that Bws utilizes less memory on small BP benchmarks, an effect can be explained by the overhead of pathwise analysis and Z3. On large examples Cutr tends to need less memory resource than Bws. The memory comparison for PNs shows similar results.

The performance impact of our acceleration approach, on both runtime and memory, can be summarized as follows. Our method analyzes a specific path at a time. If t_F is reachable, there is a good chance Cutr can find a solution early, due to the ranking of paths, some of which permit quick decisions. Although Cutr relies on backward search to cross nested loops, the cost of that is limited as such exploration is confined to a small fragment of the TTD. In the extreme, the entire TTD contains only one path with spaghetti loops. In this case Cutr falls back on backward search.

8 Related Work

Groundbreaking results in infinite-state system analysis include the decidability of coverability in *vector addition systems* (VAS) [17], and the work by German

[6] Webpage: http://www.ccs.neu.edu/home/lpzun/cutr.

and Sistla on modeling communicating finite-state threads as VAS [13]. Numerous results have since improved on the original procedure in [17] in practice [11,12,23,24]. Others extend it to more general computational models, including *well-structured* [10] or *well quasi-ordered* transition systems [1,2].

Recent theoretical work by Leroux employs Presburger arithmetic to solve the VAS global configuration reachability (not coverability) problem. In [18], it is shown that a state is *unreachable* in the VAS iff there exists an "inductive" Presburger formula that separates the initial and final states. The theoretical complexity of this technique is mostly left open. Practicality is not discussed and doubted later by the author in [19], where a more direct approach is presented that permits the computation of a Presburger definition of the reachability set of the VAS in some cases, e.g. for *flatable* VAS. Reachability can then be cast as a Presburger decision problem. The question under what exact conditions the VAS reachability set is Presburger-definable appears to be undecided.

The results referenced above are mainly foundational in nature and target generally harder (even undecidable) reachability questions than we do in this paper. We emphasize that our motivation for acceleration is not to ensure convergence of (otherwise possible diverging) fixpoint computations. Instead, our goal here was to show, for the decidable problem of TTS thread state reachability, (i) how to practically compute a Presburger encoding whose satisfiability implies reachability of the thread state, and (ii) that the resulting (quantifier-free) formulas are often easy to decide, thus giving rise to an efficient algorithm. Existing (typically forward) acceleration techniques for infinite-state systems [5,9,15] were inspirational for this paper.

In recent work, Petri net *marking equations* are used to reduce the coverability problem to linear constraint solving [8]. Follow-up work investigates a similar approach for thread-transition systems [3]. Like the present work, these approaches benefit from advances in SMT technology and in fact have proved to be efficient. On the other hand, they are incomplete (the constraints overapproximate coverability). Our goal here was to retain (soundness and) completeness, and to investigate at what cost this can be achieved.

9 Conclusion

In this paper, we have presented an approach for accelerating a widely-applicable infinite-state search algorithm for systems of unbounded numbers of threads. A key ingredient is the construction of an acyclic quotient of the input program, which in turn enables a finite path-by-path analysis. Loop-free paths and paths with only simple loops can be collapsed *without approximation* into Presburger arithmetic constraints that symbolically summarize the effect of executing the backward search algorithm along these paths in the multi-threaded program. Each path passing through loop nests is processed via standard explicit-state backward search but confined to this particular path. We have demonstrated the power of this method for proving and refuting safety properties of an extensive set of TTDs obtained from Boolean program benchmarks. We conclude that

partial but exact symbolic acceleration of existing sound and complete infinite-state search algorithms is very much feasible, and in fact very beneficial.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symb. Log. **16**(4), 457–515 (2010)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
3. Athanasiou, K., Liu, P., Wahl, T.: Unbounded-thread program verification using thread-state equations. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 516–531. Springer, Heidelberg (2016)
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI, pp. 203–213 (2001)
5. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)
6. 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)
7. 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)
8. 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)
9. Finkel, A., Leroux, J.: How to compose presburger-accelerations: applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theor. Comput. Sci. **256**(1–2), 63–92 (2001)
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. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV, pp. 72–83 (1997)
15. Jonsson, B., Saksena, M.: Systematic acceleration in regular model checking. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 131–144. Springer, Heidelberg (2007)
16. Kaiser, A., Kroening, D., Wahl, T.: A widening approach to multithreaded program verification. ACM Trans. Program. Lang. Syst. **36**(4), 14 (2014)

17. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969)
18. Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. In: LICS, pp. 4–13 (2009)
19. Leroux, J.: Presburger vector addition systems. In: LICS, pp. 23–32 (2013)
20. Liu, P., Wahl, T.: Infinite-state backward exploration of Boolean broadcast programs. In: FMCAD, pp. 155–162 (2014)
21. Liu, P., Wahl, T.: Concolic unbounded-thread reachability via loop summaries (extended technical report). CoRR abs/1607.08273 (2016). http://arxiv.org/abs/1505.02637
22. 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)
23. Reynier, P.A., Servais, F.: Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. In: Petri Nets, pp. 69–88 (2011)
24. 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)