Efficient Coverability Analysis by Proof Minimization ⋆

Alexander Kaiser[1], Daniel Kroening[1], and Thomas Wahl[2]

[1] University of Oxford, United Kingdom
[2] Northeastern University, Boston, United States

Abstract. We consider multi-threaded programs with an unbounded number of threads executing a finite-state, non-recursive procedure. Safety properties of such programs can be checked via reduction to the *coverability problem* for *well-structured transition system*s (WSTS). In this paper, we present a novel, sound and complete yet empirically much improved solution to this problem. The key idea to achieve a compact search structure is to track *uncoverability* only for *minimal* uncoverable elements, even if these elements are not part of the original coverability query. To this end, our algorithm examines elements in the downward closure of elements backward-reachable from the initial queries. A downside is that the algorithm may unnecessarily explore elements that turn out coverable and thus fail to contribute to the proof minimization. We counter this effect using a forward search engine that simultaneously generates (a subset of all) coverable elements, e.g. a generalized Karp-Miller procedure. We demonstrate in extensive experiments on `C` programs that our approach targeting minimal uncoverability proofs outperforms existing techniques by orders of magnitude.

1 Introduction

In anticipation of the prominent role concurrency is predicted to play in future software, popular systems languages like `C` and Java readily embrace support for multiple threads of execution. Communication among threads is naturally enabled via shared variables, mutexes, but also via non-blocking sleep/wake-up mechanisms. The correct use of these communication mechanisms is largely up to the user. The inevitable frustration caused by attempts to find and reproduce concurrency bugs through conventional program testing strongly encourages the use of automated formal techniques.

In this paper, we consider finite-state, non-recursive procedures executed by an *unspecified* number of threads. This scenario is highly relevant in practice. For example, the number of processes concurrently requesting I/O services in an operating system environment cannot be determined a priori. For settings like this, we are interested in detecting, or proving the absence of, assertion failures, mutual-exclusion violations, etc.

Despite the arbitrary number of threads, problems of this kind have long been known to be decidable [2], for instance by reduction to the *coverability problem* for the rich class of *well-structured transition systems* (WSTS) [23, 16]. "Coverability" of an erroneous configuration of threads (e.g. violating mutual exclusion) is tantamount to the existence of a reachable program state exhibiting such an error.

⋆ This research is supported by the EPSRC project EP/G026254/1 and ERC project 280053.

While decidable, checking coverability for WSTS incurs a high computational cost. For example, for the subclass of *vector addition systems* the problem was shown to be EXPSPACE-complete [6]. Extensions such as *transfer transitions*, which allow several threads to change their local state simultaneously and are essential to model broadcast primitives and predicate abstractions of broadcast-free programs [10, 9], render the problem Ackermann-hard [33].

The significance of the coverability problem both as a theoretical challenge, as well as in practical concurrent program verification, has led to a flurry of related activity in recent years [29, 2, 21, 20, 3, 25]. The most general solution to the coverability problem was presented in a paper by Abdulla et al. [2], which backward-explores states starting from the target state.

In this paper, we introduce a new, sound and complete solution to the coverability problem in WSTS. In contrast to existing techniques, our method relies on sequences of many inexpensive uncoverability proofs that build upon one another. We compute such proofs by searching the downward-closure of states encountered during backward exploration from the target state for smallest uncoverable members. Elements encountered during that search that are not currently known to be coverable give rise to "uncoverability candidates". If a candidate proves uncoverable, so are all elements in its upward closure, which in the end contributes to the decision for the target state. Otherwise, the coverable candidate is retained to prevent it from being expanded again.

The downside of such a "speculative exploration" is that *coverable* exploration candidates mean wasted effort. This effort can, however, be largely reduced using a simultaneously running forward search engine that labels states as coverable and thus prevents them from being explored in the (futile) hope of finding an uncoverability proof. The key is that such a forward engine acts only as a "catalyst": it affects the speed of the overall algorithm, not its result. Thus, we can use incomplete procedures such as generalizations of the (forward-directed) Karp-Miller algorithm [26, 11, 14].

To summarize, this work makes the following contributions:

- We present a novel approach to coverability checking in WSTS that combines forward propagation of under approximations with backward propagation of over approximations.
- We provide an implementation (publicly available; see Section 5) that accepts Petri nets with transfer arcs, and an extension to verifying `C` programs with unbounded thread counts in a predicate abstraction-based CEGAR loop [4, 7]. Our algorithm outperforms the best known coverability approach by orders of magnitude, enabling the analysis of programs which are out of scope of the previous technology. The experiments also reveal that our approach is able to guide the search far more effectively than existing structural invariant heuristics [13, 8].

These improvements are possible thanks to the compactness of the uncoverability proofs generated during exploration. On our `C` benchmarks, we observe reductions of up to 95% in the proof size.

2 System Model and Problem Definition

Our algorithms operate on *well-structured transition systems* (WSTS) [16]. A WSTS is a transition system equipped with a well-quasi-ordering $\succeq$ on its states that satisfies the following monotonicity property: for all states u, u', r, if u' is a successor of u and $r \succeq u$, then there exists a successor r' of r such that $r' \succeq u'$. In other words, $\succeq$ is a simulation relation for the transition system. A state q is *coverable* if there exists a reachable state v such that $v \succeq q$; the definition of "reachable (with respect to a set of initial states)" is standard.

Let now $(M, \succeq)$ be a WSTS. We denote by Cover the *coverability set*, consisting of all states covered by some reachable state. The *coverability problem* is: given a state $q \in V$, determine whether q is coverable.

Thread transition systems. We give an example of a class of WSTS called *thread transition system* (TTS) that are motivated by the task of verifying multi-threaded asynchronous software. A TTS is a machine model that gives rise to transition systems equal in expressiveness to Petri nets [25, 28]. We use TTS in examples throughout this paper.

Let S and L be finite sets of *shared* and *local* states, respectively. The elements of $T = S \times L$ are called *thread states*. Formally, a *thread transition system* (TTS) is a pair (T, Δ), where $\Delta \subseteq T \times T$ models thread transitions. Let $V = \cup_{n=0}^{\infty}(S \times L^n)$. The elements of V are called *states*; we write them in the form $(s \,|\, l_1, \ldots, l_n)$. A TTS gives rise to a transition system $M = (V, \rightarrowtail)$ with

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

exactly if, for some $i \in \{1, \ldots, n\}$, $(s, l_i, s', l'_i) \in \Delta$ and for all $j \neq i$, $l_j = l'_j$. That is, transitions may affect the shared state, and the local state of exactly one thread in local state l.

Given sets $I_s \subseteq S$ and $I_l \subseteq L$ of initial shared and local states, respectively, we define the set of initial states to be $I = I_s \times (\cup_{n=0}^{\infty} I_l^n)$. An *execution* of transition system M is a finite or infinite sequence of states in V whose adjacent states are related by $\rightarrowtail$; the first state must be initial. A state is *reachable* if it appears in some execution.

To show that M is a WSTS, 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 $\langle l_1, \ldots, l_n \rangle \supseteq \langle l'_1, \ldots, l'_{n'} \rangle$, where $\langle x \rangle$ denotes the *multiset* with the elements from x. Let further $v \succ v'$ whenever $v \succeq v'$ and $v \neq v'$. If $0 \in S$ and $0, 1, 2 \in L$, then for example $(0 \,|\, 0, 2, 0, 1) \succeq (0 \,|\, 2, 0)$, but $(0 \,|\, 0, 2, 0, 1) \not\succeq (0 \,|\, 0, 2, 0, 0)$. Relation $\succeq$ is neither symmetric nor anti-symmetric: states that cover each other are identical *up to permutations of the threads* and thus form a classical thread symmetry equivalence class. Relation $\succeq$ is thus a quasi-order, and in fact a *well-quasi-order* (wqo) on V: any infinite sequence $v_1, v_2, \ldots$ of elements from V contains an increasing pair $v_i \preceq v_j$ with $i < j$. It is easy to see that $(M, \succeq)$ fulfills the definition of a WSTS.

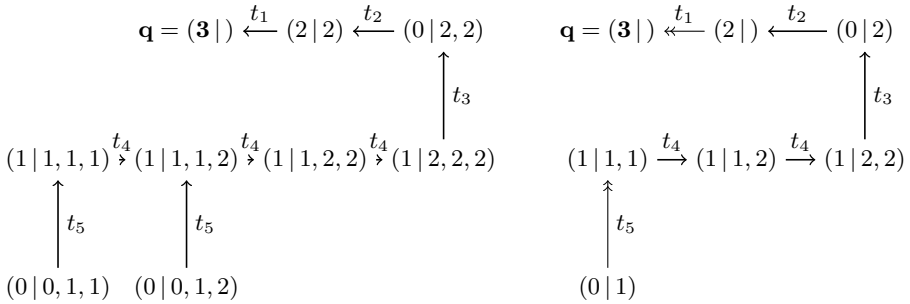

Fig. 1. Standard and minimal uncoverability proof for target q. Arrows $\rightarrow$ visualize covering predecessor relations, subscripted by the inducing thread transition; note that $p \rightarrow w$ implies $p \in \mathsf{CPre}(w)$. Arrows $p \twoheadrightarrow w$ indicate that there exists some $v \in \mathsf{CPre}(w)$ such that $v \succ p$

3 Compact Backward Reachable Sets

We introduce some basic definitions and sketch the idea underlying our approach. A set $P \subseteq V$ of states is *upward-closed* if, for any $v \in P$ and any v', $v' \succeq v$ implies $v' \in P$. We write $\uparrow P$ for the *upward closure* of P, i.e. the least upward-closed set that contains P, and $\min P$ for the set of minimal elements of P, i.e. the least subset M of P such that $\uparrow P = \uparrow M$. Every upward-closed set is representable by its minimal elements, of which only a finite number exists due to the wqo properties of $\preceq$. The term and symbol *downward-closed* and $\downarrow P$ are defined analogously.

Let the *covering predecessors* of a state $v \in V$, denoted by $\mathsf{CPre}(v)$, be all the minimal states that have a successor covering v:

$$\mathsf{CPre}(v) = \min\{p \in V \mid \exists v' \in V : p \rightarrowtail v' \wedge v' \succeq v\}\,.$$

Note that for TTS a state $(s \mid \ldots) \in \mathsf{CPre}(v)$ involves an additional thread if no thread in v is responsible for the transition to shared state s (we will see an example later on).

Backward Reachability Algorithm 1 shows a refined version of the classical backward search for WSTS [2, 1]. Input is a target state $q \in V$. The algorithm maintains a set $U \subseteq V$ with *vertices* that are labeled and identified with encountered states, and a *work set* $W \subseteq U$ of yet unprocessed vertices.

Algorithm 1 $\mathsf{Bc}(q \in V)$

```
1: W := {q} ; U := {q}
2: while ∃w ∈ W : w ∈ min(U) do
3:     W := W \ {w}
4:     for all p ∈ CPre(w) : p ∉ ↑U do
5:         if p ∈ I then
6:             return "q ∈ Cover"
7:         W := W ∪ {p} ; U := U ∪ {p}
8: return "q ∉ Cover"
```

The algorithm performs an iterative search over covering predecessors starting from q. It terminates either by finding an execution leading to a state that covers q, or when no minimal and unprocessed vertex remains (this eventually happens since $\succeq$ is a wqo), thus proving the uncoverability of the target state.

Minimal Uncoverability Proofs Let us assume for the rest of this section that the target q is uncoverable. In this case Algorithm 1 computes a representation (in terms of minimal states in U) of all states that have an emanating execution leading to a state that covers q. This set, which we denote by Brs, is upward-closed due to monotonicity.

Instead of computing this set precisely we can, however, also prove the target state uncoverable by any over approximation $\mathsf{Brs}^\sharp$ of this set that still enjoys disjointness from the initial states. The crux is, given that upward-closed sets are represented by their minimal elements, over approximating these sets by adding smaller ("$\preceq$") elements leads to fewer and smaller minimal elements, hence to a **more succinct representation**. The following lemma reveals how to exploit this property and settle the uncoverability more efficiently.

Lemma 1 *Let v and v' be two states such that $v' \preceq v$. Then for all $m \in \mathsf{CPre}(v)$ there exists $m' \in \mathsf{CPre}(v')$ such that $m' \preceq m$.*

The proof of Lemma 1 is straightforward. Applied to Algorithm 1 this property generalizes to paths of arbitrary length through the search: the smaller the covering predecessors we examine in Line 4, the shorter the paths we need to explore.

Definition 2 *An **uncoverability proof** for an element q is an upward-closed set of states $\mathsf{Brs}^\sharp$ such that $q \in \mathsf{Brs}^\sharp$, $\mathsf{Brs}^\sharp \supseteq \mathsf{CPre}(\mathsf{Brs}^\sharp)$, and $\mathsf{Brs}^\sharp \cap I = \emptyset$. An uncoverability proof $\mathsf{Brs}^\sharp$ for q is **minimal** if $\min \mathsf{Brs}^\sharp \subseteq \min(V \setminus \mathsf{Cover})$, and every upward-closed subset $X \subset \mathsf{Brs}^\sharp$ is not an uncoverability proof for q.*

Minimal uncoverability proofs thus consist solely of smallest uncoverable states and cannot be reduced by removing the upward-closure of some of its minimal states. Note that multiple minimal uncoverability proofs may exist.

Bearing Lemma 1 in mind, we observe that minimal uncoverability proofs are an interesting means for proving the uncoverability of target q, as they minimize the maximum length of paths Algorithm 1 needs to traverse.

Example. To illustrate this idea, let us consider the TTS with shared and local states $0, \ldots, 3$ and thread transitions $t_1 = (2, 2, 3, 0)$, $t_2 = (0, 2, 2, 0)$, $t_3 = (1, 2, 0, 0)$, $t_4 = (1, 1, 1, 2)$ and $t_5 = (0, 0, 1, 1)$; the initial shared and local state sets are $I_s = I_l = \{0\}$. Assume we wish to check whether shared state 3 can be reached, i.e. the target is $(3 \mid)$. Figure 1 (left) depicts the minimal states of the corresponding set Brs computed by Algorithm 1. If we search, however, the downward-closure of encountered states for smallest uncoverable members, we obtain the minimal uncoverability proof shown on the right: the covering predecessors $(2 \mid 2)$ and $(0 \mid 0, 1, 2)$ give rise to candidate $(2 \mid)$ and $(0 \mid 1)$, respectively. Comparing both uncoverability proofs, we observe reductions in various dimensions: the number of minimal states drops from 9 to 7, the longest traversed path from 7 to 6, and the maximum thread count from 3 to 2. □

In Section 5 we present experimental evidence that show the potential for compressing the proof size along these dimensions in practice: for our concurrent C program benchmarks we observed **reductions by 95%, 67% and 50%**, respectively. This potential is the key for the efficiency of our approach.

4 Minimal Uncoverability Proof Algorithm

In this section, we develop our approach to compute minimal uncoverability proofs. An obstacle is the determination of "helpful" candidates. We begin by illustrating it on the TTS from the previous example; we omit non-minimal states for sake of brevity.

Example. Again, we start from target $(3\,|\,)$. However, before exploring covering predecessors, we check whether a helpful candidate for a smaller state exists. Since this is not the case (no smaller state exists), we proceed as usual and obtain predecessor $(2\,|\,2)$ which gives rise to candidate $(2\,|\,)$. If we find a path showing $(2\,|\,)$ coverable, we will withdraw the candidate and proceed with the former state as usual. From $(2\,|\,)$ we encounter predecessor $(0\,|\,2)$; although $(0\,|\,2)$ strictly covers $(0\,|\,)$, we do not create a corresponding candidate as it is initial. However, for its predecessor $(1\,|\,2,2)$ in turn we do so, and create candidate $(1\,|\,)$. Further exploring this state we obtain path $(0\,|\,0) \to (1\,|\,)$, proving the candidate's coverability. We withdraw the candidate and mark the downward-closure of all states along the execution as coverable, so that these elements are not expanded again. With path $(0\,|\,0) \to (1\,|\,1) \to (1\,|\,2)$ the next candidate proof attempt also fails. From the original predecessor $(1\,|\,2,2)$ we arrive at $(1\,|\,1,2)$, of which we can rule out the existence of smaller uncoverable states from the collected coverability results; the same holds for the next predecessor $(1\,|\,1,1)$. We finally arrive at predecessor $(0\,|\,0,1)$ and create the candidate $(0\,|\,1)$. Since no new (with respect to $\uparrow$) predecessor exists, we terminate with the tree shown in Figure 1 on the right. □

4.1 Backward-constructed Minimal Proofs

In addition to the data structures used by Algorithm 1, namely a set $U \subseteq V$ with vertices that are labeled and identified with encountered states, and a work set $W \subseteq U$ of unprocessed vertices, our algorithm maintains

i) a set E storing (directed) *edges* between vertices, $E \subseteq U \times U$;
ii) a mapping ζ *associating* each vertex with a unique vertex, $\zeta\colon U \to U$;
iii) a downward-closed set D storing collected *coverability results*, $D \subseteq V$.

As already indicated in Figure 1, we write $u \to r$ for $(u, r) \in E$, and $\to^*$ for the reflexive transitive closure of $\to$. We call a vertex $u \in U$ *candidate vertex* if $\zeta(u) = u$, and *predecessor vertex* otherwise. A *path* of (U, E) is a finite sequence of vertices from U whose adjacent vertices are related by $\to$; the *last* state must be a candidate vertex. The mapping ζ (extended to sets X by $\zeta(X) = \{\zeta(x) | x \in X\}$) clusters the vertices into $|\,\zeta(U)|$ *partitions*, one per candidate vertex (vertices that are associated with that candidate vertex). The set D stores states that were shown to be coverable.

The algorithm takes a target q as input and ensures at all times that restricting the partitioned graph (U, E, ζ) to any equivalence class of vertices with the same associated candidate vertex, say u, forms a tree with u as root, and all other vertices being predecessor vertices. Each tree represents an attempt to prove the corresponding candidate (as done by Algorithm 1 for input u). Edges and the mapping ζ enable the withdrawal

Fig. 2. Effect of routine Backtrack in the presence of candidate vertices s, r and t, each with a single primed predecessor vertex in their partition; the partition of candidate vertex r we wish to remove is highlighted, and the single conflicting edge is marked with "$\circ$" (left). After the call, the partition of r is removed and its former predecessor vertex r' is associated with s (right)

of unhelpful candidates in a way that preserves parts of their partition that are shared with remaining candidates.

The algorithm consists of three routines: Enlarge creates a new candidate vertex, Backtrack removes partitions of candidates and Mcov is the main routine.

Enlargement routine The Enlarge routine takes a candidate u we wish to add as input. If u is a new vertex ($u \notin U$), it is inserted in the work and vertex set. In all cases, the graph is repartitioned by adjusting ζ and associating every vertex in the set

$$\Lambda(u) = \{r \in U | u = r \vee (r \rightarrow^* u \wedge \zeta(r) = \zeta(u))\}$$

with u. This repartitioning (observe $u \in \Lambda(u)$) ensures that $r \in \Lambda(u)$ now entails $\zeta(r) = u$. The graph thus contains the new candidate vertex u, with a partition in the shape of a tree.

Backtracking routine The purpose of the Backtrack routine, shown in Algorithm 2, is to remove unhelpful candidate vertices $P \subseteq \zeta(U)$ and their partitions. An obstacle is that paths $u \rightarrow^* r \notin P$ to remaining candidate vertices may have segments in partitions that will be removed (paths can traverse multiple partitions). To ensure soundness, we need to preserve them.

Algorithm 2 Backtrack($P \subseteq \zeta(U)$)

```
1: while ∃(r, s) ∈ E : (r, s) is P-confl. do
2:     for all t ∈ Λ(r) do
3:         ζ(t) := ζ(s)
4: for all r ∈ U : ζ(r) ∈ P do
5:     W := W \ {r} ; U := U \ {r}
6:     for all (t, r) ∈ E do
7:         E := E \ {(t, r)}
```

Definition 3 *Consider a set P of candidate vertices and an edge $(r, s) \in E$. The edge (r, s) is called P-**conflicting** if $\zeta(r) \in P$ and $\zeta(s) \notin P$.*

Hence, P-conflicting edges induce segments of the above kind. To preserve them, we exhaustively resolve conflicts in a first step (Lines 1–3): for a conflicting edge, say $r \rightarrow s$, we do this by reassociating vertices in $\Lambda(r)$ to $\zeta(s)$.

Once all conflicts are resolved and thus $r \rightarrow s$ and $\zeta(r) \in P$ entails $\zeta(s) \in P$, remaining vertices and edges of partitions in P are removed in Lines 4–7. Figure 2 sketches both steps.

Algorithm 3 Minimal Uncoverability Proof Algorithm: $\mathsf{Mcov}(q \in V)$

```
1: W := {q} ; U := {q} ; D := I ; E := ∅ ; ζ : q ↦ q
2: select n ∈ min C(q) ; Enlarge(n)    // create candidate vertex
3: while ∃w ∈ W : w ∈ min(U) do
4:     W := W \ {w}
5:     for all p ∈ CPre(w): p is ζ(w)-minimal do
6:         if p ∉ D then
7:             E := E ∪ {(p, w)}
8:             if p ∉ U then
9:                 W := W ∪ {p} ; U := U ∪ {p} ; ζ(p) := w    // add covering predecessor
10:                select n ∈ min C(p) ; Enlarge(n)    // create candidate vertex
11:        else if q ∉ ⌊p then
12:            D := D ∪ ⌊p    // mark coverable states
13:            Backtrack(ζ(⌊p))    // call backtrack routine
14:            while ∃u ∈ min(U) ∩ ↑P do
15:                select n ∈ min C(u) ; Enlarge(n)
16:            break    // skip forward to next iteration of while
17:        else
18:            return "q ∈ Cover"
19: return "q ∉ Cover"
```

Main routine We introduce some terminology:

Definition 4 *Let $v \in V$, and $u \in \zeta(U)$. State v is u-**minimal** if $v \not\succeq u$ and for all $s, s' \in U$ such that $s \to s'$ and $\zeta(s') = u$, we have $v \not\succeq s$.*

That is, state v is u-minimal if it covers neither the candidate vertex u nor any predecessor vertex in u's partition (observe that a predecessor vertex may yet belong to a partition other than $\zeta(u)$).

Definition 5 *Let $X \subseteq V$. X is **lower successor-closed** if, for any $p \in X$ and any v, $(p \to v \vee p \succeq v)$ entails $v \in X$.*

That is, a lower successor-closed set is both "successor-closed" (where successors are formed according to $\to$) and downward-closed. We write $\lfloor v$ for the least lower successor-closed set containing v. This set is obtained by closing $\{v\}$ under $\to$ successors and downward until fixpoint. The point of this definition is that, if v is coverable, so is every vertex in $\lfloor v$: coverability itself is closed under $\to$ successors and downward.

Algorithm 3 shows the main routine, Mcov, of our approach. The algorithm works as follows. Initially W and U contain one candidate vertex (target q), D is the set of initial states, the set E of edges is empty, and ζ associates q to itself (Line 1). If target q gives rise to a candidate we create a minimal candidate vertex (Line 2). The set of potential candidates $\mathcal{C}(p) \subseteq V$ is given by

$$\mathcal{C}(p) = \{v \in V | v \prec p \text{ and } v \notin D\} .$$

The set contains all the states that are strictly covered by p but not yet marked coverable. If $p = (0\,|\,0, 0, 1)$, and $D = \{(0\,|\,), (0\,|\,0), (0\,|\,1)\}$, then for example $\mathcal{C}(p) = \{(0\,|\,0, 1), (0\,|\,0, 0)\}$. We tacitly assume that Line 2 has no side-effect if $\mathcal{C}(p) = \emptyset$.

The algorithm now picks and removes a minimal and unprocessed vertex w from the work set, or returns "$q \notin \mathsf{Cover}$" (Line 19) if no such vertex remains. In the former case, the **for** loop in Line 5 steps through all covering predecessors p of w that are $\zeta(w)$-minimal and processes them as follows:

Lines 6–10 If p is not currently known to be coverable, then the graph is expanded. If p is a new vertex ($p \notin U$, Line 8), then we ensure that p will be processed when it turns minimal among the vertices by adding it as predecessor vertex to w's partition. Finally, we call the Enlarge routine to create new minimal candidate vertices.

Lines 11–16 If p is found to be coverable but not q, we add $\lfloor p$ (which is coverable as well) to D and invoke the Backtrack routine to remove partitions of coverable candidate vertices. Since this may remove candidate vertices of remaining predecessor vertices, we have to ensure that their downward-closure is further searched for minimal, yet helpful candidates. We therefore create new minimal candidate vertices (Lines 14–15). Again, we tacitly assume that Line 15 has no side-effect if $\mathcal{C}(p) = \emptyset$. Then, the **break** instruction skips forward to the next iteration of the **while** loop. As a consequence of backtracking, unprocessed vertices that were previously *not* minimal may now be.

Lines 17–18 Otherwise we return "$q \in \mathsf{Cover}$", since the coverability of target q has been settled (in the affirmative).

Example. We continue with the example from the beginning of this section. In this case routine Enlarge is called four times: predecessor vertices $(2\,|\,2)$, $(1\,|\,2,2)$ and $(0\,|\,0,1)$ give rise to candidates $(2\,|\,)$, $(1\,|\,)$ (and after its removal to $(1\,|\,2)$) and $(0\,|\,1)$, respectively. Routine Backtrack is called once after candidate vertices $(1\,|\,2)$ and $(1\,|\,2,2)$ turn out unhelpful. The mapping ζ shown in Figure 1 on the right has three partitions, one for each of the candidate vertices $(3\,|\,)$, $(2\,|\,)$ and $(0\,|\,1)$. The collected coverability results are $D = {\downarrow}\{(0\,|\,0), (1\,|\,1), (1\,|\,2)\}$, and the mapping ζ is: $\zeta(u) = (2\,|\,)$ if $u \in \{(0\,|\,2), (1\,|\,2,2), (1\,|\,1,2), (1\,|\,1,1)\}$, and $\zeta(u) = u$ if $u \in \{(3\,|\,), (2\,|\,), (0\,|\,1)\}$. □

Due to the finiteness of downward closures (we create a finite number of candidate vertices) the algorithm eventually terminates. Completeness follows from that of Algorithm 1, and the fact that we only remove conflicting edges during backtracking. When Mcov terminates for an uncoverable target q, the remaining minimal nodes represent an uncoverability proof for q: $\mathsf{Brs}^\sharp = {\uparrow}U$ (cmp. Definition 2).

In its current form Algorithm 3 computes uncoverability proofs with the property $\min \mathsf{Brs}^\sharp \subseteq \min(V \setminus \mathsf{Cover})$, but not necessarily minimal ones. This is attributed to two factors. First, if a covering predecessor gives rise to a candidate and we later remove this predecessor, then a created uncoverability candidate may turn irrelevant for the coverability of target q. Second, when we add a candidate vertex that is incomparable to existing candidate vertices, this may still turn some of the latter irrelevant as well. In order to obtain truly minimal uncoverability proofs, we remove candidate vertices that are no longer needed during calls to Backtrack, and after every call to Enlarge.

4.2 Balancing the Search via Supplementary Coverability Results

If candidates are chosen unwisely, the search may incur extra work to identify and eliminate the coverable elements. To reduce this overhead, we have to prevent unhelpful candidates from being created. In its current form, Algorithm 3 does so by incorporating collected coverability results when it creates a new candidate. These coverability results may also, however, come from any external source, which we call a *coverability oracle*. A coverability oracle *a*) needs to report states that are provably coverable and should thus reasonably search in a *forward direction*; *b*) is not required to find *all* coverable states: creating some unhelpful candidates does not harm the search.

This flexibility allows us to use any under approximating forward-directed search: a standard or random reachability analysis works just as well as generalizations of the Karp-Miller procedure to broadcast synchronization [11], which are known not to guarantee termination for WSTS.

We finally remark: since detecting coverable elements is one of the main goals of Algorithm 3, the coverability results reported by the coverability oracle directly benefit the algorithm itself. The coverability oracle and Algorithm 3 run in parallel and synchronize via the set D: the coverability oracle populates this set while maintaining $D \subseteq \mathsf{Cover}$. Receiving such updates, Algorithm 3 terminates if $q \in D$, or otherwise invokes the Backtrack routine on now known-to-be-coverable candidate vertices in regular intervals to restore the invariant $D \cap U = \emptyset$.

5 Experimental Evaluation

In this section, we evaluate our algorithms on 21 concurrent C programs. The programs feature a diverse set of communication primitives, such as shared variables, mutexes, condition variables and broadcasts. For each benchmark, we consider verification of a safety property, specified via an assertion. The C programs, ranging from 40 to 1000 lines of code, are:

1–4 broadcast-based code from FreeBSD, NetBSD and Solaris that is related to RDMA ZFS file system support and interface/system monitoring;
5–9 programs using several basic language features and the pthread library;
10–12 programs using multiple locks to control access to a shared resource;
13,14 blocking and non-blocking pseudo-random number generators [31, 10];
15 a program used in [17] to illustrate thread-modular model checking [24];
16,17 lock-based and lock-free stack described in [31], supporting concurrent pushes and pops (adapted from an IBM implementation) [10];
18,19 a Linux driver skeleton and a Mozilla vulnerability fix [27, 24];
20,21 algorithms to establish mutual exclusion [24].

We implemented our Mcov routine (Algorithm 3) for TTSs and transfer Petri nets in our tool Breach, equipped with a generalization of the Karp-Miller procedure (Gkm) as coverability oracle; our tool (we used v1.0) and all benchmarks are available online at www.cprover.org/bfc. The oracle reports coverability results to a data pool our Mcov routine taps into at regular intervals; both run in parallel. In order to measure the impact

Table 1. Comparison of classical coverability approaches to our Mcov algorithm; buggy benchmarks in **bold**, run times in seconds, or **TO** (**MO**) in case the time (memory) limit is hit

			Classical approaches			Our new approach			
C Programs	Final TTS		Gkm	Bc		Mcov		Mcov/Gkm	
id/Name	$\|T\|$	$\|\Delta\|$	*Time*	*Iter.*	*Time*	*Iter.*	*Time*	*Iter.*	*Time*
1/BSD-ABDD	82	288	**MO**	23476	19.1	328	0.1	184	0.0
2/BSD-RDMA-ADDR	101	304	1.6	12479	7.6	295	0.1	146	0.0
3/NETBSD-SYSMON-PWR	291	704	**MO**	–	**TO**	124	0.1	126	0.0
4/SOLARIS-SPACE-MAP	539	992	**MO**	10348	5.8	3412	2.2	2834	1.0
5/**BS-LOOP**	11616	20485	0.1	1483	1.5	1049	1.1	–	0.1
6/COND	280	1045	0.0	809	0.2	4660	88.4	–	0.0
7/FUNCTION-POINTER	9216	746770	**MO**	–	**TO**	–	**TO**	23139	592.0
8/S-LOOP	516	2813	0.0	3567	1.5	1567	1.4	–	0.5
9/PTHREAD	17920	135300	**MO**	–	**TO**	70841	1521.0	51265	189.7
10/DOUBLE-LOCK1	34880	233025	**MO**	–	**TO**	–	**MO**	90488	1146.5
11/DOUBLE-LOCK2	17216	114752	**MO**	–	**TO**	–	**MO**	46012	285.9
12/DOUBLE-LOCK3	3264	19250	**MO**	–	**TO**	24161	75.8	9514	14.5
13/PRNG (NON-BL.)	142	954	**MO**	191	0.0	4791	6.9	64	0.0
14/PRNG	788	5650	**MO**	–	**TO**	–	**TO**	9168	33.9
15/SPIN2003	188	984	0.0	6436	1.7	699	0.2	–	0.1
16/STACK (NON-BL.)	352	2550	**MO**	34046	133.7	18603	128.6	8249	12.5
17/STACK	648	3626	**MO**	35500	38.7	7616	20.2	2723	2.3
18/**BOOP**	7488	25929	0.0	1446	7.8	10776	361.1	–	0.1
19/MOZILLA-VUL.-FIXED	1648	8050	0.0	77053	84.2	3723	4.3	–	1.7
20/PETERSON	2048	8988	0.0	22951	15.5	2373	2.3	–	1.2
21/SZYMANSKI	8448	35896	0.1	–	**TO**	9597	35.8	–	11.0

of our new approach, the oracle can be deactivated, turning Breach into the refined version of the classical backward search (Algorithm 1). Due to efficiency limitations of the underlying data structures, we do not add candidate vertices that involve two threads or more (which we found to be a good trade-off between efficiency and proof minimization). To apply Breach to the C programs, we extended the abstract language interface of the C software model checker SatAbs to TTS. SatAbs implements the CEGAR loop based on a symmetry-aware predicate-abstraction technique [10], and handles function calls by inlining. All experiments are performed on a 3GHz Intel Xeon machine with 20 GB memory, running 64-bit Linux, with a timeout of 30 minutes.

Evaluation Table 1 presents results for various configurations of our implementation. Columns on the left show the benchmark id and name, and the total number of thread states and transitions emerged in the last, and always most expensive, CEGAR iteration. Remaining columns show details for:

Gkm: Our coverability oracle (stand-alone);
Bc: Refined version of the classical backward algorithm (Algorithm 1);
Mcov: Our Mcov algorithm (Algorithm 3);
Mcov/Gkm: The Mcov algorithm equipped with the coverability oracle Gkm.

For each approach we show the total model checking run time, and in addition for backward-directed algorithms the number of iterations.

The results demonstrate that our new approach outperforms the classical algorithms: Mcov/Gkm solves *all* 21 programs, and Mcov 17 instances, compared to 13 and 9 for the classical backward algorithm and the coverability oracle, respectively. Comparing

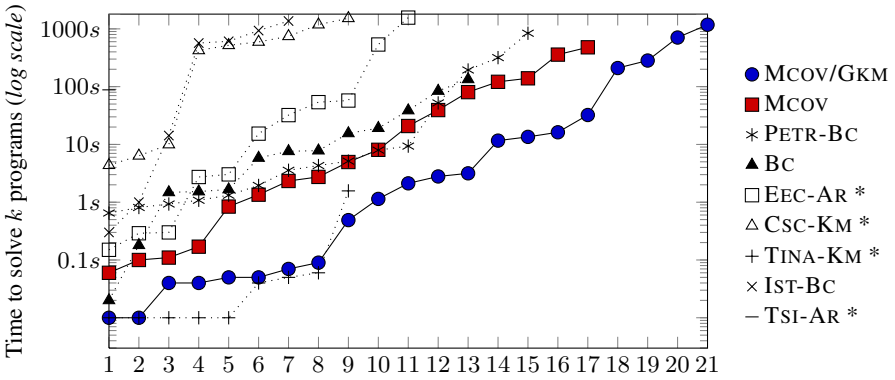

Fig. 3. Cactus plot on the 21 multi-threaded C programs comparing our Mcov and Mcov/Gkm approaches to various existing ones; due to broadcasts, the limit for tools marked with * is $k = 17$

the results for Bc and Mcov clearly shows that the uncoverability proofs the latter generates are much smaller. This is reflected by a strict decrease of the iteration count in 17 cases. In the majority of cases, this improvement manifests in the running time: Mcov outperforms Bc on 13 programs (often significantly), compared to 4 the other way around. Furthermore, the results for Mcov/Gkm show that the coverability oracle can substantially reduce the cost of unhelpful candidates, showing their synergies (observe that running Gkm and Bc stand-alone in parallel is not helpful). As a result, the positive effect is amplified: compared to Bc, the iteration count strictly decreases on *all* programs.

To measure the difference between standard and minimal uncoverability proofs, we removed the bound on candidate vertices (in return for longer runtimes). In this setup, we observed the following reductions (averaged): the the longest traversed path drops from 28 to 14 (-50%), the threads included in the proof from 6 to 2 (-67%), and the proof size in terms of minimal states from 22518 to 1222 (-95%). While the classical backward-approach includes up to eight threads in a proof, our approach always generates minimal uncoverability proofs which involve no more than *two threads*. With the bound on candidate vertices mentioned above and used for Table 1, the reductions are only marginally smaller (e.g. the previous thread number increases by one).

Comparison There exist a number of other approaches to the coverability problem. We compare to the following tools (all available online):

Ist − Bc**:** Classical backward search using interval sharing-trees (v1.0.3) [18];
Petr − Bc**:** Refined backward search with structural invariants (v0.1) [29, 30];
Tina − Km**:** Karp-Miller procedure (v3.0.0) [5];
Csc − Km**:** Refined Karp-Miller procedure using interval sharing-trees (v0.1) [22];
Eec − Ar**:** Pure forward algorithm with enumerative refinement (v1.0.3) [21];
Tsi − Ar**:** Variant of [21] using backward under approx. for refinement (v1.0.3) [19].

Only Ist − Bc and Petr − Bc support broadcast primitives. In order to allow for a meaningful comparison, we translated abstract TTS templates generated by SatAbs into (transfer) Petri nets and replaced the model checker back-end.

Figure 3 depicts total model checking run times (scaled logarithmically) for all methods as "cactus plot": the horizontal axis represents the number of programs the respective method could successfully handle, and the vertical axis the time needed to solve this number if they were ran in parallel. The results demonstrate significant improvements over all previous methods: only Mcov/Gkm is able to solve all 21 programs, followed by Mcov stand-alone (17), Petr − Bc (15), Bc (13), Eec − Ar (11), Tina − Km and Csc − Km (9), Ist − Bc (7) and Tsi − Ar (1).

The improvement over the best previous approach (Petr − Bc) shows that our new approach is able to guide the search more effectively than structural invariant heuristics, which are know to often yield invariants that are irrelevant to the safety property or too imprecise [15]. The inferior performance of our underlying classical backward algorithm (Bc) to Petr − Bc indicates that the observed improvements are not just owed to clever implementation, but rather the result of our novel approach.

6 Related Work

Algorithmic solutions to coverability analysis were first proposed for *vector addition systems* in a landmark paper by Karp and Miller [26]. The solution constructs a pseudo-reachability tree by forward exploration and replaces newly discovered states that are strictly greater than predecessors by their limit. It has a non-primitive recursive worst-case complexity [32]. The purpose there was mainly to show decidability of the coverability problem for VASes and the equivalent Petri nets. The technique is implemented in the tool Tina − Km [5]. It cannot be extended to broadcast primitives [12]. An improvements of this procedure that computes minimal coverability sets is [22].

To afford more flexibility in modeling parametrized programs, various algorithms were later proposed for WSTS, originally in a pure backward fashion [2], which was implemented in the tools Ist − Bc and Petr − Bc [29], later as forward exploration [14, 34]. The paradigm presented in [21] (and implemented in the tool Eec − Ar) is also a pure forward algorithm; it constructs abstractions of increasing precision. In contrast to the paradigm of Eec − Ar, the implementation itself does not support broadcasts. Other approaches are the backward and forward unfolding algorithms from [3] and [25].

Solutions combining forward and backward exploration are rare; we are only aware of the methods described in [15] and [19]. The authors of [15] propose to use a Csc − Km-like approach to compute over approximations of the coverability set, which are then used in a *subsequent* backward exploration to prune the search space. Our experimental results demonstrate, however, that this approach cannot cope with programs of the sizes we consider. In [19], the authors combine over approximations computed in a forward fashion, which are refined by using backward under approximations; the approach is implemented in the tool Tsi − Ar. On an abstract level, our algorithm can be seen as the dual of this approach. To the best of our knowledge, our approach is the first to combine forward propagation of *under* approximations with backward propagation of *over* approximations to the coverability problem in WSTS.

7 Conclusion

We introduced a new approach to the coverability problem in WSTS. The novelty of our algorithm is the way it proves uncoverable instances via a sequence of many inexpensive uncoverability proofs. Our algorithm can be used to check assertion failures, mutual exclusion violations and many other properties for parametrized programs communicating via mutexes, shared variables or common concurrency primitives such as broadcasts.

We demonstrated in extensive experiments on large benchmarks, generated by the software model checker SatAbs from C programs, that our algorithm outperforms the best known coverability approach by orders of magnitude, enabling the verification of programs which are out of scope of the previous technology. The experiments also reveal that our approach is able to guide the search far more effectively than existing structural invariant heuristics [13, 8]. We conclude from our experiments that programs tend to feature minimal uncoverability proofs with fewer and smaller minimal elements compared to those targeted by existing methods.

The ideas we have presented, supported by the simplicity of their implementation, are naturally applicable to coverability methods in general. We believe, for example, that while our method outperforms techniques based on structural net invariants, even more practical benefit is achievable by combining these strategies.

Acknowledgments. We wish to thank Michael Tautschnig for assistance with SatAbs, and Pierre Ganty, Leopold Haller, Philipp Rümmer and Emelie Vollmer for their insightful comments on earlier drafts of this work.

References

1. P. A. Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 16(4), 2010.
2. P. A. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. General decidability theorems of infinite-state systems. In *Logic in Computer Science (LICS)*, 1996.
3. P. A. Abdulla, S. P. Iyer, and A. Nylén. SAT-solving the coverability problem for Petri nets. In *Formal Methods in System Design (FMSD)*, 2004.
4. T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, 2002.
5. B. Berthomieu and F. Vernadat. *The Tina tool, release 2.9.6, November 2009*. LAAS/CNRS, `http://homepages.laas.fr/bernard/tina/`.
6. E. Cardoza, R. J. Lipton, and A. R. Meyer. Exponential space complete problems for Petri nets and commutative semigroups: Preliminary report. In *STOC*, pages 50–54, 1976.
7. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, 2005.
8. G. Delzanno, J.-F. Raskin, and L. V. Begin. Attacking symbolic state explosion. In *Computer-Aided Verification (CAV)*, 2001.
9. G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, pages 173–187, 2002.
10. A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Computer-Aided Verification*, 2011.

11. A. Emerson and K. K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Logic in Computer Science (LICS)*, pages 70–80, 1998.
12. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Logic in Computer Science (LICS)*, 1999.
13. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. In *Formal Methods in System Design (FMSD)*, 2000.
14. A. Finkel and J. Goubault-Larrecq. Forward analysis for WSTS, part II: Complete WSTS. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2009.
15. A. Finkel, J.-F. Raskin, M. Samuelides, and L. V. Begin. Monotonic extensions of Petri nets: Forward and backward search revisited. *ENTCS*, 2002.
16. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science (TCS)*, 2001.
17. C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking of Software (SPIN)*, 2003.
18. P. Ganty, C. Meuter, G. Delzanno, G. Kalyon, J.-F. Raskin, and L. Van Begin. Symbolic data structure for sets of k-uples. Technical report, Université Libre de Bruxelles, 2007.
19. P. Ganty, J.-F. Raskin, and L. V. Begin. A complete abstract interpretation framework for coverability properties of WSTS. In *VMCAI*, 2006.
20. P. Ganty, J.-F. Raskin, and L. V. Begin. From many places to few: Automatic abstraction refinement for Petri nets. In *ICATPN*, 2007.
21. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check: New algorithms for the coverability problem of WSTS. In *JCSS*, 2006.
22. G. Geeraerts, J.-F. Raskin, and L. V. Begin. On the efficient computation of the minimal coverability set for Petri nets. In *ATVA*, 2007.
23. S. German and P. Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 1992.
24. A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *Computer-Aided Verification (CAV)*, 2011.
25. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *Computer-Aided Verification (CAV)*, 2010.
26. R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
27. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
28. A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC*, 2006.
29. R. Meyer and T. Strazny. Petruchio: From dynamic networks to nets. In *Computer-Aided Verification (CAV)*, 2010.
30. R. Meyer and T. Strazny. An algorithmic framework for coverability in well-structured systems. In *Application of Concurrency to System Design (ACSD)*, 2012.
31. T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
32. C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science (TCS)*, 1978.
33. P. Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *MFCS*, 2010.
34. D. Zufferey, T. Wies, and T. A. Henzinger. Ideal abstractions for well-structured transition systems. In *VMCAI*, 2012.