A Widening Approach to Multithreaded Program Verification

ALEXANDER KAISER and DANIEL KROENING, University of Oxford, United Kingdom
THOMAS WAHL, Northeastern University, Boston, MA, United States

Pthread-style multithreaded programs feature rich thread communication mechanisms, such as shared variables, signals, and broadcasts. In this article, we consider the automated verification of such programs where an unknown number of threads execute a given finite-data procedure in parallel. Such procedures are typically obtained as predicate abstractions of recursion-free source code written in `C` or Java. Many safety problems over finite-data replicated multithreaded programs are decidable via a reduction to the *coverability problem* in certain types of well-ordered infinite-state transition systems. On the other hand, in full generality, this problem is Ackermann-hard, which seems to rule out efficient algorithmic treatment.

We present a novel, sound, and complete yet empirically efficient solution. Our approach is to judiciously *widen* the original set of coverability targets by configurations that involve fewer threads and are thus easier to decide, and whose exploration may well be sufficient: if they turn out uncoverable, so are the original targets. To soften the impact of "bad guesses"—configurations that turn out coverable—the exploration is accompanied by a parallel engine that generates coverable configurations; none of these is ever selected for widening. Its job being merely to prevent bad widening choices, such an engine need not be complete for coverability analysis, which enables a range of existing partial (e.g., nonterminating) techniques. We present extensive experiments on multithreaded `C` programs, including device driver code from FreeBSD, Solaris, and Linux distributions. Our approach outperforms existing coverability methods by orders of magnitude.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal Methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical Verification; D.3.2 [**Language Classifications**]: Concurrent, Distributed, and Parallel Languages; F.1.2 [**Models of Computation**]: Parallelism and Concurrency

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Multithreaded program verification, well-structured transition systems, coverability

ACM Reference Format:
Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2014. A widening approach to multithreaded program verification. ACM Trans. Program. Lang. Syst. 36, 4, Article 14 (October 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2629608

14

1. INTRODUCTION

In anticipation of the prominent role that concurrency is expected to play in future software, popular programming languages such as `C` and Java readily embrace concurrent programming, via their `pthread` and `thread` class APIs, respectively. Communication among threads is naturally enabled via shared (global-scope) variables and mutexes,

This research is supported by EPSRC project EP/G026254/1, ERC project 280053, and U.S. National Science Foundation grant no. 1253331.
Authors' addresses: A. Kaiser and D. Kroening, Wolfson Building, Parks Road, Oxford OX1 3QD, UK; emails: {alexander.kaiser, kroening}@cs.ox.ac.uk; T. Wahl, 360 Huntington Avenue, Boston/MA, 02115, USA; email: wahl@ccs.neu.edu.

DOI: http://dx.doi.org/10.1145/2629608

```
static u_int
akbd_read_char(keyboard_t *kbd, ...) {
  //...
  sc = (struct adb_kbd_softc *)(kbd);
  mtx_lock(&sc->sc_mutex);
  if (!sc->buffers && wait)
    cv_wait(&sc->sc_cv,&sc->sc_mutex);
  if (!sc->buffers) {
    mtx_unlock(&sc->sc_mutex);
    return (0); }
  adb_code = sc->buffer[0];
  for (i = 1; i < sc->buffers; i++)
    sc->buffer[i-1] = sc->buffer[i];
  sc->buffers--;
  key = adb_code;
  mtx_unlock(&sc->sc_mutex);
  return (key); }
```

```
static u_int
adb_kbd_receive_packet(device_t dev, ...) {
  struct adb_kbd_softc *sc;
  sc = device_get_softc(dev);
  if (command != ADB_COMMAND_TALK)
    return 0;
  if (reg != 0 || len != 2)
  {
    return (0);
  }
  mtx_lock(&sc->sc_mutex);
  kbd = kbd_get_keyboard(/*...*/);
  //...
  mtx_unlock(&sc->sc_mutex);
  cv_broadcast(&sc->sc_cv);
  //...
  return (0); }
```

Fig. 1. Code fragment of a recent implementation of an Apple Bus protocol in the FreeBSD operating system (file dev/adb/adb_kbd.c), involving kernel thread synchronization via mutexes (`mtx_lock`/`mtx_unlock`) and POSIX-style broadcasts (`cv_wait`/`cv_broadcast`).

as well as via nonblocking API constructs such as signals and broadcasts. Much existing system-level code such as device drivers make use of these mechanisms to protect shared data and to synchronize thread execution; operating systems such as many Unix derivatives and Mac OS X directly support them. An example of thread communication in action is shown in Figure 1.

The correct use of communication mechanisms is largely up to the user. Attempts to find and reproduce concurrency bugs through conventional program testing are time consuming, hit-and-miss, and usually result in frustration. In this article, we present an automated, model checking–based approach to detect failures such as violations of assertions or mutual-exclusion conditions, or to prove the absence of such failures. We consider *finite-state* models (such as obtained via predicate abstraction) of programs executed by an unknown number of threads, created either on program start or dynamically during execution. This scenario is most relevant in practice: it covers workload-aware programs, where the number of worker threads is determined at runtime.

The difficulty for search-based analysis methods is that such program models give rise to infinite state spaces. Nevertheless, the detection of the previously mentioned failure types for such programs has long been known to be decidable, such as by reduction to the *coverability problem* for *well–quasi-ordered systems* [Abdulla et al. 1996]. In program terms, coverability of a particular configuration of threads asks for the existence of a number n and the reachability of a global state g in the execution of the finite-state program by n threads such that g contains that thread configuration. A "thread configuration" can, for instance, be a thread in a particular local state or a pair of threads simultaneously occupying a location in a given critical code section. Coverability thus allows us to express precisely the types of assertion or synchronization failures that we are after.

Coverability for the class of well–quasi-ordered systems, which subsumes popular concurrency models such as various forms of Petri nets, is known to have an exponential-space lower complexity bound. For example, for plain Petri nets and vector addition systems, the problem was shown to be complete for exponential space [Cardoza et al. 1976]. Extensions such as *transfer transitions*, which allow several threads to change their local state simultaneously and are used to model broadcasts in concurrent programs (Figure 1) increase the complexity of the problem further [Schnoebelen 2010]. These daunting computational costs, and the significance of the

coverability problem in practical concurrent program verification, have led to a flurry of activity in crafting solutions viable in practice [Finkel et al. 2002; Geeraerts et al. 2006; Ganty et al. 2006].

In this article, we introduce a new solution to the coverability problem in well–quasi-ordered systems that shares soundness and completeness with existing methods but follows a fundamentally different search strategy: before a configuration is selected for expansion, the set of target elements is *widened* by its downward closure, thus adding to the target set many smaller (in the partial configuration order) elements whose coverability has not yet been decided. The motivation behind this approach is twofold:

(1) the coverability of smaller elements, which involve fewer threads, is often less costly to decide, and
(2) *un*coverability of smaller elements immediately implies uncoverability of any larger elements, including the original query target.

The strategy to first settle the coverability of smallest-possible elements, side-stepping the original query, not only accelerates the search but also makes the search structure more compact, as we will demonstrate.

"Bad guesses"—that is, elements in the widening that end up coverable, permit no conclusion about the coverability of the original query elements. In this case, we backtrack out of the widening: all elements found to be coverable are purged from the search structure, and the search continues with another smallest-possible and currently undecided element. Such backtracking can impair the algorithm's efficiency. As a countermeasure, the backward search can be assisted by an engine that generates coverable elements; none of these is ever selected for widening. All we require of such an engine is that it soundly report coverability of elements. It need not be complete, as its job is only to accelerate the backward search by flagging coverable configurations early. An ideal candidate is a Karp-Miller–like forward-directed search for Petri nets with transfer arcs, which generates (possibly incomplete) coverability information quickly.

Combined with finite-data abstractions, our algorithm is applicable to multithreaded concurrent programs featuring rich communication mechanisms, including nonblocking forms such as signals and broadcasts, widely used in operating system code. The efficiency of our method makes it possible to analyze synchronization skeletons of programs with many thousands of lines of code in a matter of a few seconds, providing huge benefits over existing coverability techniques.

2. PROBLEM FORMALIZATION

The focus of our discussion is multithreaded programs consisting of a main function that creates an unbounded number of threads, each of which executes some fixed procedure. The procedure may itself spawn threads dynamically that execute it. This model subsumes the case of several procedures, each executed by an unbounded number of threads: such procedures can be merged into one procedure that, in step 1, nondeterministically chooses which of the original procedures to execute. The model also subsumes the case that there are a small *constant* number of threads (say one server thread) that execute the same procedure (alongside an unbounded number of other threads): the combined memory of all of these threads is finite and can be stored in the shared state space.

We therefore present our program as an (unspecified) number n and a single code fragment $\mathbb{P}$. The intention is that $\mathbb{P}$ represents the code of the procedure whose name is passed to the thread creation primitive, which is invoked n times by the main thread on start of the program. We begin this section by sketching a programming language used to represent $\mathbb{P}$; the emphasis here is on the synchronization mechanisms.

$$
\begin{array}{lcl}
\langle \text{PROG} \rangle & ::= & [\langle \text{VARS} \rangle] \\
 & & [[\langle \text{LABEL} \rangle :] \langle \text{STMT} \rangle]^* \\
 & & \\
\langle \text{VARS} \rangle & ::= & [\texttt{shared}\ \langle \text{ID} \rangle := \mathit{init}\ [, \langle \text{ID} \rangle := \mathit{init}]^*;] \\
 & & [\texttt{local}\ \ \langle \text{ID} \rangle := \mathit{init}\ [, \langle \text{ID} \rangle := \mathit{init}]^*;] \\
 & & [\ \texttt{cond}\ \ \langle \text{ID} \rangle := \mathit{init}\ [, \langle \text{ID} \rangle := \mathit{init}]^*;] \\
 & & \\
\langle \text{STMT} \rangle & ::= & \texttt{fork}\ \langle \text{LABEL} \rangle; \\
 & | & \texttt{wait}\ c; \quad | \quad \texttt{signal}\ c; \quad | \quad \texttt{broadcast}\ c; \\
 & | & \texttt{atomic}\ \{\ \langle \text{STMT} \rangle^*\ \} \\
 & | & \langle \text{SEQSTMT} \rangle
\end{array}
$$

Fig. 2. Input language syntax (partial).

2.1. Input Language

The syntax of our language is given by the grammar in Figure 2. A code fragment $\mathbb{P}$ consists of (optional) variable declarations and a sequence of (possibly labeled) statements. Variables are of certain finite domains and come in three flavors; ⟨ID⟩ stands for a valid identifier. Condition variables are used in synchronization statements described next. Declared variables must be initialized: *init* returns a *set* (to permit nondeterminism) of compile-time computable values of the appropriate type; the semantics of := in initializations is "select a member of." Unspecified initial values can be emulated by choosing *init* to be the entire domain of the variable.

The significance of `shared` and `local` descriptors is explained in Sections 2.2 and 2.3, as is the precise semantics of the statements in our language; the intuition behind them is as follows:

thread creation: `fork` ⟨LABEL⟩ creates a new thread that starts executing at the statement pointed to by ⟨LABEL⟩;

synchronization via condition variables: `wait` c, for some variable c declared under the keyword `cond`, blocks a thread until some other thread executes either `signal` c (waking up exactly one thread waiting on c) or `broadcast` c (waking up all threads waiting on c). The `signal` and `broadcast` calls do not block the executing thread if no thread is currently waiting on c. We assume there are no "orphan" synchronization constructs: a program containing `wait` c must also contain `signal` c or `broadcast` c, and vice versa.

atomic sections: The statement sequence { ⟨STMT⟩* } following the keyword `atomic` is executed atomically: when a thread t enters an atomic section, all other threads are preempted until t leaves that section (typically by reaching the section's last statement, although also via jumps; we disallow jumps/forks *into* atomic sections).

These primitives allow us to encode the synchronization skeleton of the program shown in Figure 1. For example, mutexes such as `mtx_lock` and `mtx_unlock` are expressed using Boolean variables and suitable atomic code sections. The `cv_wait` statement is encoded using `wait`, by unlocking the mutex associated with the condition variable before waiting and reacquiring it afterward.

The nonterminal ⟨SEQSTMT⟩ in the grammar in Figure 2 represents classical sequential statements, such as assignments, conditionals, loops, and (nonrecursive) procedure calls. Their precise syntax and semantics are immaterial to this article.

2.2. Thread Transition Systems

We define the semantics of a code fragment $\mathbb{P}$ written in the preceding language using a form of abstract state machine referred to as the thread transition system (TTS). A TTS is defined over a set $T = S \times L$ of *thread states*, each a pair of a *shared* and a *local* state from finite sets S and L, respectively. Intuitively, the elements of S are valuations of the `shared` and `cond` variables of $\mathbb{P}$, and those of L are valuations of the `local` variables. The translation of $\mathbb{P}$ into a TTS is explained in detail in Section 2.3.

Definition 1. A *thread transition system (TTS)* is a pair (T, Δ) such that $\Delta \subseteq T \times T$ is a binary relation on T partitioned into $\mapsto \cup \twoheadrightarrow \cup \hookrightarrow \cup \rightsquigarrow$.

The four parts of the transition relation Δ represent *thread transitions*, *spawn transitions*, *signal transitions*, and *broadcast transitions*, formalized as follows. A TTS gives rise to a transition system over a set V of multithreaded *configurations*: for a positive integer n, let $V_n = S \times L^n$ and $V = \cup_{i=1}^{\infty} V_i$. We write configurations in the form $v = (s \mid \ell_1, \ldots, \ell_n) \in V_n$. Configuration v consists of a shared state s, and local state ℓ_i of thread i, for $i \in \{1, \ldots, n\}$. By $|v|$, we denote the *dimension* of a configuration $v \in V$—that is, the number of existing threads (n in the preceding case).

The transition system is then $M = (V, \rightarrowtail)$; we have $(s \mid \ell_1, \ldots, \ell_n) \rightarrowtail (s' \mid \ell'_1, \ldots, \ell'_{n'})$ exactly when one of the following transition conditions holds:

thread transition: $n' = n$ and there exist $(s, \ell) \mapsto (s', \ell') \in \Delta$ and i such that $\ell_i = \ell$, $\ell'_i = \ell'$, and for all $j \neq i$, $\ell_j = \ell'_j$.

spawn transition: $n' = n + 1$ and there exist $(s, \ell) \twoheadrightarrow (s', \ell') \in \Delta$ and i such that $\ell_i = \ell$, for all $j < n'$, $\ell_j = \ell'_j$, and $\ell'_{n'} = \ell'$.

signal transition: $n' = n$ and there exists $(s, \ell) \hookrightarrow (s', \ell') \in \Delta$ such that
(1) for every i, $\ell_i = \ell'_i \neq \ell$, *or*
(2) for some i, $\ell_i = \ell$, $\ell'_i = \ell'$, and for all $j \neq i$, $\ell_j = \ell'_j$.

broadcast transition: $n' = n$ and there exists $(s, \ell) \rightsquigarrow (s', \ell') \in \Delta$ such that for all i, the following holds: let $\mathcal{L} = \{k \in L : (s, \ell_i) \rightsquigarrow (s', k) \in \Delta\}$. If $\mathcal{L} \neq \emptyset$, then $\ell'_i \in \mathcal{L}$, otherwise $\ell'_i = \ell_i$.

The dimensions of configurations related by $\rightarrowtail$ differ by at most one. Thread and spawn transitions affect the local state of exactly one thread (in case of spawn: the new one). They are *blocking*—in other words, executable only if *enabled* by a thread occupying the transition's source thread state. A signal transition is the nonblocking version of a thread transition: it affects the local state of at most one thread; if no enabling thread exists, a signal may still "silently" change the shared state. Given a thread signal transition $(s, A) \hookrightarrow (s', B)$, examples of global signal transitions are $(s \mid B, C, D) \rightarrowtail (s' \mid B, C, D)$ and $(s \mid A, C, A) \rightarrowtail (s' \mid B, C, A)$, without and with enabled thread, respectively.

Broadcast transitions are also nonblocking. Moreover, broadcasts with source shared state s and target shared state s' can execute simultaneously: the shared state is updated to s', and the local state of thread i is updated nondeterministically according to one of the broadcast transitions of the form $(s, \ell_i) \rightsquigarrow (s', k)$ if any, unchanged if none. As an example, suppose that our program has two thread broadcast transitions $(s, A) \rightsquigarrow (s', B)$ and $(s, A) \rightsquigarrow (s', C)$, and no other transitions. An example of a valid global broadcast transition is $(s \mid B, C, D) \rightarrowtail (s' \mid B, C, D)$; this is similar in effect to a (silent) signal. Another valid example is $(s \mid A, C, A) \rightarrowtail (s' \mid B, C, C)$. Participation in a broadcast is mandatory for threads that are enabled in it: $(s \mid A, C, A) \rightarrowtail (s' \mid B, C, A)$ is not a valid transition.

For simplicity, we omit thread terminating transitions (which for our purposes can be simulated by directing such a thread to a dead-end local state).

```
shared s := 0; // shared; range {0, 1, 2, 3}
// implicit: instruction counter κ := 0; local; range {0, 1, 2}
0: atomic {                 1: atomic {                 2: atomic {
     if s ≠ 0                    if s ≠ 3                    if s = 2
       goto 0;                     goto 1; } // κ := 2         goto 2;
     else                                                    s := (s + 1) mod 4 ;
       s := 3 ; } // κ := 1                                  goto 0; }
```

Fig. 3. A program with three atomic sections (arranged horizontally and labeled 0, 1, 2, respectively). To simplify presentation, we use a non-Boolean but finite-domain shared variable s. Observe that each thread can perform only one execution sequence, namely from the initial configuration $(0, 0)$ to $(3, 1)$ to $(3, 2)$, and finally back to $(0, 0)$. Hence, only configurations with $s \in \{0, 3\}$ are reachable, and the jump instructions `goto` 0, `goto` 1, and `goto` 2 in locations $\kappa = 0$, $\kappa = 1$, and $\kappa = 2$, respectively, are never executed.

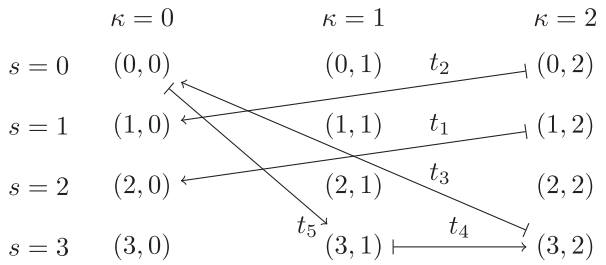

Fig. 4. The TTS induced by the program in Figure 3. Valuations of s and κ are shown as (s, κ). The initial thread state is $(0, 0)$; intermediate states of atomic sections and self-loops are omitted.

Transition system executions. Let T_I be a set of initial thread states. The set of initial configurations is defined as $I = \{(s \mid \ell_1, \ldots, \ell_n) \mid n \geq 0 \wedge (s, \ell_1), \ldots, (s, \ell_n) \in T_I\}$. As usual, an *execution* of transition system M is a (finite or infinite) sequence of configurations, beginning with an initial one, whose adjacent elements are related by $\rightarrowtail$. Let $\rightarrowtail^*$ denote the reflexive transitive closure of $\rightarrowtail$. A configuration v is *reachable* if there exists $i \in I$ such that $i \rightarrowtail^* v$.

2.3. From Programs to TTS and Infinite-State Transition Systems

From Programs to TTS. A program $\mathbb{P}$ written according to the grammar in Figure 2 can be encoded as a TTS; we sketch this process in this section. A complete example of a program and the corresponding TTS is given in Figures 3 and 4.

Let $\text{Vars} = \text{SVars} \cup \text{LVars} \cup \text{CVars}$ be the disjoint sets of variables declared in $\mathbb{P}$ under the keywords `shared`, `local`, and `cond`, respectively. The variables in $\text{SVars} \cup \text{LVars}$ are of type Boolean, whereas a condition variable in CVars takes values from $\{sig, bc, idle\}$, indicating that a signal has been received, or a broadcast has been received, or neither has been received. Let further $\kappa \notin \text{Vars}$ be a fresh symbol serving as the program counter of each thread. A shared state of the TTS is a mapping from SVars to $\{0, 1\}$ and from CVars to $\{sig, bc, idle\}$. A local state is a mapping from LVars to $\{0, 1\}$ and from $\{\kappa\}$ to $\{0, \ldots, |\mathbb{P}| - 1\}$, where $|\mathbb{P}|$ denotes the number of statements in $\mathbb{P}$. A thread state thus comprises the values of all variables of $\mathbb{P}$, including the implicit variable κ, and hence characterizes the state of a single thread executing $\mathbb{P}$.

In an execution of the transition system M derived from the TTS, the local variables of $\mathbb{P}$ are replicated for each thread, whereas the shared and condition variables are not. The initial thread state set T_I of the TTS is defined by the variable initializations according to Figure 2; in particular, these will satisfy $\kappa = 0$—that is, execution starts in the first program location.

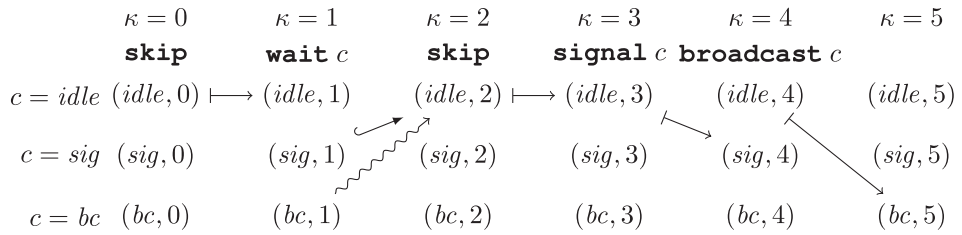

Fig. 5. Modeling statements **wait**, **signal**, and **broadcast** on a condition variable c. The program (not shown) encoded by this TTS consists of the five statements given in the second line; $\kappa = 5$ marks the end of the program. Sequential statements are abbreviated as **skip**. There are no local variables other than the program counter.

Sequential statements of $\mathbb{P}$ translate directly into standard thread transitions in M. A **fork** statement translates into a spawn transition, except a *spawn* transition does not change the local state of the spawning thread, whereas a *fork*ing thread moves to the next program location. The latter can be enforced in M by treating the spawn and the move of the spawning thread to the next program location as an atomic section. Such sections are implemented via fresh Boolean shared lock variables whose value marks a thread state as inside or outside an atomic section.

There is no transition type in a TTS that directly corresponds to a **wait** c statement. We model such as follows (Figure 5 provides an illustration). Thread states whose local state corresponds to a **wait** c statement in $\mathbb{P}$ (such as at $\kappa = 1$ in the figure) can only be departed via a transition from a shared state satisfying $c = sig$ or $c = bc$, thus effectively blocking such transitions until the shared state is set to *sig* or *bc*. This happens via normal thread transitions that correspond to **signal** c or **broadcast** c statements in $\mathbb{P}$, respectively. The fact that multiple threads can be released by a single broadcast is realized by making the transition out of shared state $c = bc$ itself a TTS broadcast transition: in Figure 5, any thread waiting in $(bc, 1)$ is released; if none, the condition variable c is reset to *idle* without any local state change. As with **fork**, the translations of **wait**/**signal** **broadcast** statements must be packed into appropriate atomic sections.

From TTS to Petri nets and well–quasi-ordered systems. The global transition system M induced by a TTS is naturally symmetric: all threads execute the same procedure, and symmetry-breaking constructs such as expressions involving the identity of a thread are not allowed. This symmetry permits representing M as a machine that counts the number of occurrences of each local state in a configuration (i.e., essentially a Petri net). This rewriting of the local thread state system representation into a counter representation appeared early in German and Sistla [1992] and has been referred to as *counter abstraction* since [Pnueli et al. 2002] (although in our context, it is—like its finite-state counterpart symmetry reduction—an "exact abstraction").

The types of synchronization constructs permitted in the TTS directly determine the "Petri net class" of M. More precisely, if we only allow standard thread and spawn transitions $\mapsto$ and $\mapsto\!\!\!\!\!+$ in Δ, system M can be expressed as a (plain) Petri net. If we allow signal/broadcast transitions $\rightsquigarrow/\hookrightarrow$ in Δ, then our model coincides in expressiveness with Petri nets with *asynchronous rendezvous/transfer* transitions, respectively [Ciardo 1994; Eliëns and de Vink 1992], which are strictly more expressive than plain Petri nets [Dufourd et al. 1998; Finkel et al. 2006]. We omit proofs of these equivalences in this article, as they are mostly straightforward.

More generally, the transition system M induced by any TTS is a *well–quasi-ordered system* [Abdulla et al. 1996; Finkel and Schnoebelen 2001]. For a transition system

$(V, \rightarrowtail)$ to be well quasi-ordering, two conditions need to be in place [Finkel and Schnoebelen 2001; Abdulla 2010; Abdulla et al. 1996]:

well quasi-ordering: There exists a relation $\preceq$ on V that is reflexive and transitive ($\preceq$ is a quasi order), and such that for every infinite sequence $v_0, v_1, v_2, \ldots \in V^\omega$ there exist i, j with $i < j$ and $v_i \preceq v_j$.

monotonicity: For all configurations u, u', r, if $u \rightarrowtail u'$ and $u \preceq r$, then there exists r' such that $r \rightarrowtail r'$ and $u' \preceq r'$.

Given a TTS and an induced transition system M with configurations V, let $\preceq$ be defined over V as follows: $v = (s \mid \ell_1, \ldots, \ell_n) \preceq v' = (s' \mid \ell'_1, \ldots, \ell'_{n'})$ whenever

(1) $s = s'$, and
(2) for each local state symbol $\ell \in L$, $|\{i : i \leq n \wedge \ell_i = \ell\}| \leq |\{i : i \leq n' \wedge \ell'_i = \ell\}|$.

That is, v and v' agree on the shared state, and the multiset of local state occurrences in v is a subset of the multiset of local state occurrences in v'. It is easy to see that $\preceq$ is a well quasi-ordering that satisfies the preceding monotonicity property. As usual,

(1) $v \succeq v'$ stands for $v' \preceq v$,
(2) $v \succ v'$ stands for $v \succeq v' \wedge v' \not\preceq v$, and
(3) $v \preceq\succeq v'$ stands for $v \preceq v' \wedge v \succeq v'$ ("equivalence").

TTS are a convenient and succinct intermediate notation between our input programming language, and infinite-state concurrency models such as Petri nets and well–quasi-ordered systems, over which the algorithms in this article are presented. We use TTS instead of Petri nets in illustrations and examples since the former largely separate the concerns of thread transitions and multithreading and thus compactly reflect the replicated nature of our programs: a given finite-state procedure is executed by an indeterminate number of threads. Our implementation, on the other hand, takes both TTS and Petri nets with transfer arcs as input: many of the benchmarks that we use in Section 4 are Petri nets of various flavors, and we compare the performance of our implementation against many Petri net tools.

2.4. Goal of This Work

Given a program $\mathbb{P}$ written according to the grammar in Figure 2, let M be the infinite-state transition system induced by the TTS that in turn is induced by $\mathbb{P}$, as described previously in this section. Further, let Reach be the set of reachable configurations of M—that is, $\mathsf{Reach} = \{v \in V \mid \exists i \in I : i \rightarrowtail^* v\}$. We write Cover for the set of coverable configurations of the well–quasi-ordered system $(M, \preceq)$—in other words, those "covered" by some reachable configuration:

$$\mathsf{Cover} = \{v \in V \mid \exists v' \in \mathsf{Reach} : v \preceq v'\}.$$

Cover is an overapproximation of the reachability set whose determination suffices to *exactly* decide classical safety properties like program location reachability, program assertions, mutual exclusion, and so forth. For instance, program location x is reachable if and only if there exists $s \in S$ and a local state ℓ such that $\ell(\kappa) = x$ and the one-thread configuration (s, ℓ) is coverable. Thus, in this article, we investigate the coverability problem: given a configuration q, determine whether q is coverable.

Decidability and complexity. The coverability problem is decidable for transfer and rendezvous Petri nets (e.g., Dufourd et al. [1998] and Abdulla et al. [1996]), and more generally for well–quasi-ordered systems provided some natural conditions on the computability of the well–quasi-ordering relation hold [Leuschel and Lehmann 2000]. The misery is in the complexity: coverability of transfer Petri nets was shown to be

Ackermann-complete [Schnoebelen 2010], which means that the complexity grows as fast as the Ackermann function. If the Petri net is hardwired into the algorithm (the input consists only of the query), then the complexity subsides gracefully to primitive recursive [Schnoebelen 2010; Figueira et al. 2011]. These daunting complexity measures have triggered much research on devising practically viable solutions to the coverability problem.

3. COVERABILITY ANALYSIS BY TARGET SET WIDENING

The method presented in this section pursues a strategy that may seem counterintuitive at first: instead of focusing on the original input configuration q, during its execution the algorithm builds a hierarchy of elements for which coverability is tested; configuration q itself may well never be directly investigated. The motivation is that uncoverability of smaller (in the sense of $\preceq$) elements, which suffices to prove the uncoverability of q, is likely to terminate more quickly, as it involves fewer threads. Our approach thus biases the search algorithm toward proving *un*coverability of elements. An external coverability generator, which will typically trade in termination in favor of efficiency, intervenes in case the search attempts to prove uncoverability of coverable configurations. In this section, we first briefly review the fundamentals of coverability analysis in well–quasi-ordered systems [Abdulla 2010], then we illustrate and formalize the idea sketched earlier, and finally describe our algorithm in detail.

3.1. Review: Backward Coverability Analysis

We write $\uparrow P$ for the *upward closure* $\{v' \mid \exists v \in P : v' \succeq v\}$ of a set $P \subseteq V$. An element v of P is *minimal* if there is no $r \in P$ such that $r \prec v$. Two minimal elements of a set are either incomparable or equivalent; the wqo properties ensure that the equivalence relation $\preceq\succeq$ has a finite index (otherwise, any selection of representatives would constitute an infinite sequence of incomparable elements). In general, an equivalence class can be infinite, whereas in our special case an equivalence class contains the finitely many *permutations* of a given local state tuple. We denote by $\min P$ a finite set of canonical representatives for each equivalence class of minimal elements of P. Specifically, for our case of local state tuples, we choose the *lexicographically least* among all equivalent minimal elements in the set as representative, such that $\min\{(0 \mid 0, 1), (0 \mid 1, 0)\} = \{(0 \mid 0, 1)\}$ and $\min\{(0 \mid 1, 0)\} = \{(0 \mid 1, 0)\}$. Set P is *upward closed* if $\uparrow P = P$; in that case, $\min P$ is a minimal subset M of P such that $\uparrow M = P$. Every upward-closed set P is representable as $\uparrow \min P$, for the finite set $\min P$. We abbreviate $\uparrow\{v\}$ by $\uparrow v$. The concept *downward closed* and the symbol $\downarrow P$ are defined analogously (although, in contrast, not every downward-closed set can be represented by a finite set of *maximal* elements).

Given a configuration $v \in V$ of a well–quasi-ordered system, the set of predecessors of elements in its upward closure $\uparrow v$ is again upward closed and can therefore be represented by its minimal elements. We call these minimal elements the *cover predecessors* of v and denote them by $\mathsf{C\text{-}Pre}(v)$:

$$\mathsf{C\text{-}Pre}(v) = \min \bigcup_{v' \succeq v} \{p \in V \mid p \rightarrowtail v'\}.$$

We write $p \hookrightarrow v$ for $p \in \mathsf{C\text{-}Pre}(v)$. Note that the dimensions of a configuration and its cover predecessors can differ: we will see many examples later where a thread can enter a particular thread state (s, ℓ), only if another thread "helps," by setting the shared state to s. Configuration $(s \mid \ell)$ (dimension 1) then has a cover predecessor of dimension 2.

Algorithm 1 shows a version of the classical backward search for well–quasi-ordered systems [Abdulla et al. 1996; Abdulla 2010]. Input is a set of initial configurations $I \subseteq V$

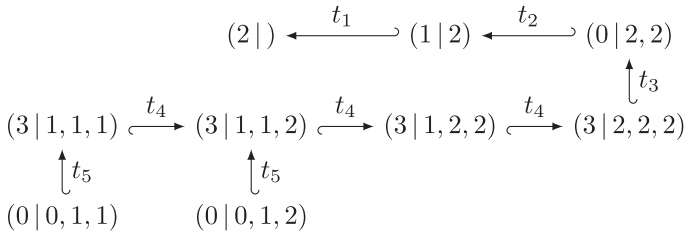

Fig. 6. Uncoverability proof of the classical backward search. Shown are the configurations in the final set U computed by Algorithm 1 for the TTS in Figure 4. The target set is $\{(2\,|\,)\}$—in other words, we wish to check whether shared state $s = 2$ is reachable in the program in Figure 3. Edges denote cover predecessor relations and are labeled by the inducing thread transition (from Figure 4).

ALGORITHM 1: $\text{BC}(I, q)$

Require: initial conf. set I, query $q \notin I$
1: $W := \{q\}$; $U := \{q\}$
2: **while** $\exists w \in W$ **do**
3: $W := W \setminus \{w\}$
4: **for all** $p \in \mathsf{C\text{-}Pre}(w) \setminus \uparrow U$ **do**
5: **if** $p \in I$ **then**
6: **return** "$q \in$ Cover"
7: $W := \min(W \cup \{p\})$
8: $U := \min(U \cup \{p\})$
9: **return** "$q \notin$ Cover"

and a target configuration $q \notin I$. The algorithm maintains a set $U \subseteq V$ of minimal encountered configurations and a *work set* $W \subseteq U$ of unprocessed configurations. It successively computes cover predecessors, starting from q, and terminates either by backward reaching an initial configuration (thus proving coverability of q), or when no unprocessed vertex remains (thus proving uncoverability; this will happen eventually since $\preceq$ is a well quasi-ordering).

3.2. Coverability Analysis by Target Set Widening: The Idea

In case Algorithm 1 terminates in line 9, set U contains the minimal configurations backward reachable from $\uparrow q$—that is, the minimal elements of the set $\mathsf{C\text{-}Pre}^*(q)$ of configurations that have a path to an element in $\uparrow q$. Figure 6 shows an example of minimal backward-reachable configurations computed by Algorithm 1. The uncoverability of q follows from the disjointness of $\mathsf{C\text{-}Pre}^*(q)$ and the initial configurations I; set $\mathsf{C\text{-}Pre}^*(q)$ thus serves as an *uncoverability proof* for the target configuration q.

Instead of computing this set, we can, however, also prove q uncoverable using an overapproximation of $\mathsf{C\text{-}Pre}^*(q)$ that is closed under C-Pre and does not intersect with the initial configurations. The potential benefit of proving uncoverability via such an overapproximation is that overapproximating $\mathsf{C\text{-}Pre}^*(q)$ by appropriately pushing down ("$\prec$") its minimal elements leads to more compact or, in the limit, *minimal* uncoverability proofs.

Definition 2. An *uncoverability proof* for q is an upward-closed set UCP of configurations that

(1) contains q;
(2) is closed under preimages, and hence, since upward closed, also under cover preimages; and
(3) is disjoint from the initial configuration set I.

Further, the uncoverability proof is *minimal* if

(4) for any $r \in \min \mathrm{UCP}$ and any $v \prec r$, v is coverable; and
(5) no proper subset of UCP satisfies the foregoing conditions (1)–(4).

For example, if q is uncoverable, then the upward closure of $\mathsf{C\text{-}Pre}^*(q)$ is an uncoverability proof, as is the set of all uncoverable configurations (i.e., the complement of the set Cover). Condition (4) of a *minimal* uncoverability proof UCP states that every minimal element of UCP is also a minimal element of the complement of Cover: $\min \mathrm{UCP} \subseteq \min(\neg\, \mathsf{Cover})$.

The notion of an uncoverability proof is sound: conditions (1)–(3) imply that all elements of UCP are uncoverable; in particular, the query configuration q is. The notion is also complete: our algorithm can always find a minimal uncoverability proof, provided q is uncoverable.

In contrast to the set $\mathsf{C\text{-}Pre}^*(q)$, which is unique for a given q, multiple minimal uncoverability proofs may exist. Moreover, the upward closure of $\mathsf{C\text{-}Pre}^*(q)$ and the complement of Cover do not in general represent minimal uncoverability proofs. We illustrate these claims with an example. Consider the well–quasi-ordered system with states $\{i, q, x_1, x_2\}$; i and q are the initial and query states, respectively. The ordering $\preceq$ is the reflexive closure of $\{(x_1, q), (x_2, q)\}$. Further, the system has no transitions (hence, condition (2) holds trivially). Query q is therefore uncoverable. There are four uncoverability proofs of this fact:

—$\uparrow\{q\} = \{q\}$ is the upward closure of $\mathsf{C\text{-}Pre}^*(q)$; it violates condition (4) for $v \in \{x_1, x_2\}$ and $r = q$ and hence is not minimal.
—$\uparrow\{x_1\} = \{x_1, q\}$ and $\uparrow\{x_2\} = \{x_2, q\}$ both satisfy conditions (4) (vacuously) and (5), and are therefore minimal uncoverability proofs for q.
—$\uparrow\{x_1, x_2\} = \{x_1, x_2, q\}$ is the complement of Cover; it satisfies (4) but violates (5) by the previous item.

Uncoverability proof construction. The idea underlying our widening approach is that "cover predecessors of smaller elements are smaller" (an observation that also appears in Abdulla [2010]).

LEMMA 3. *For configurations r, r', v', if r is a cover predecessor of r' and $v' \preceq r'$, then there exists a cover predecessor v of v' such that $v \preceq r$.*

PROOF. Let $C_1 = \{w \mid w \hookrightarrow v'\}$ and $C_2 = (\downarrow r) \cap C_1$. The latter set is nonempty, since $r \in C_2$. Therefore, let v be a minimal element of C_2. Then $v \in \downarrow r$, so $v \preceq r$. In addition, $v \in C_1$. It remains to be shown that v is a *minimal* element of C_1, since then $v \in \mathsf{C\text{-}Pre}(v')$.

To this end, let $w \in C_1$ be arbitrary. If $w \prec v$, then $w \preceq r$, hence $w \in \downarrow r$, hence $w \in C_2$. This is not possible, however, since v (supposedly $\succ w$) is a minimal element of C_2. Thus, $w \not\prec v$. Since $w \in C_1$ was arbitrary, it follows that v is minimal in C_1. □

The lemma suggests that before expanding an element, we first select smaller elements for expansion; the resulting cover predecessors will be smaller as well. Since smaller elements have fewer (cover) predecessors, this leads to earlier termination along the path and thus faster decisions.

We can observe the impact of these observations on the performance of Algorithm 1 by using the program in Figure 4. Let the query target be $(2\,|\,)$, as we want to determine whether shared state 2 is reachable. Figure 7 sketches the search process.

We start at target $q = (2\,|\,)$. Before expanding it into cover predecessors, we check whether a widening candidate exists. As this is not the case ($\downarrow q = \{q\}$), we proceed by

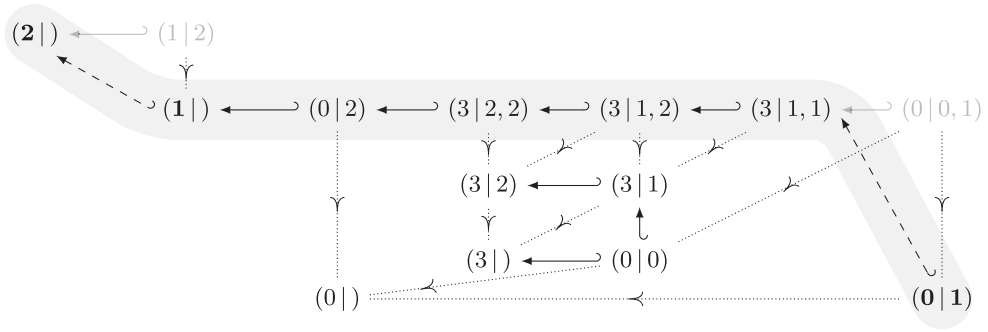

Fig. 7. Minimal uncoverability proof construction. The initial configuration set is given by $\{(0 \mid 0^i) : i \in \mathbb{N}\}$. We write $t \hookleftarrow\!\!\dashrightarrow v$ if $p \hookrightarrow v$ for some $p \succ t$. The (initially singleton) set of coverability targets is widened on-the-fly by so far undecided elements of the downward closure of encountered configurations (indicated via edges $\hookleftarrow\!\!\dashrightarrow$). If an element turns out coverable, we backtrack and mark it and other coverable configurations so that they are not reselected for widening.

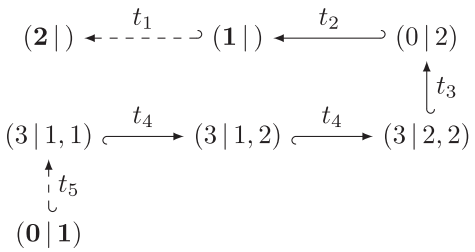

Fig. 8. Minimal uncoverability proof. Configurations in the widened node are set in bold.

obtaining cover predecessor $(1 \mid 2)$, which covers $(1 \mid)$, a widening candidate. From $(1 \mid)$, we obtain cover predecessor $(0 \mid 2)$, which covers $(0 \mid)$. The latter configuration is not considered for expansion, as it is initial and hence coverable. Thus, we expand $(0 \mid 2)$ to obtain $(3 \mid 2, 2)$, which covers $(3 \mid)$; the latter becomes a new candidate configuration and is expanded. We now find that $(3 \mid)$ is coverable, with initial cover predecessor $(0 \mid 0)$. We thus have to cancel the backward search from $(3 \mid)$ and mark it as coverable. Similarly, trying to add $(3 \mid 2)$ as a candidate fails, since its cover predecessor $(3 \mid 1)$ is coverable.

The algorithm thus resorts to expanding $(3 \mid 2, 2)$ to $(3 \mid 1, 2)$. All three configurations strictly covered by $(3 \mid 1, 2)$ have previously been shown coverable and are thus not added to the node set. The same is true for its predecessor $(3 \mid 1, 1)$, which is hence expanded to $(0 \mid 0, 1)$; the latter configuration strictly covers $(0 \mid 1)$. The only cover predecessor of $(0 \mid 1)$ is $(3 \mid 1, 2)$, which is not new, so the search terminates: the query is uncoverable.

Figure 8 shows the uncoverability proof that is actually generated for the example of Figure 4 and target $(2 \mid)$. Configuration $(2 \mid)$ is expanded to $(1 \mid 2)$ followed by widening to $(1 \mid)$. Comparing this proof with that in Figure 6, we observe reductions in the number of minimal configurations (9 vs. 7), in the longest traversed path (7 vs. 6), and in the maximum thread count (3 vs. 2). The uncoverability proof is even minimal: every proper subset is not an uncoverability proof, and every state that is strictly covered by one of the seven states in Figure 8 turns out to be coverable.

We can think of the cancellation of the backward search from a configuration that turns out coverable as *backtracking out of the widening*: the algorithm made a bad choice that needs to be rolled back. By contrast, if the termination condition is satisfied

for a configuration v obtained by widening p (such as for configuration $(0 \mid 1)$ earlier), $v \in \downarrow p$, we do not need to go back to the prewidening query configuration p: any cover predecessor of p is also a cover predecessor of v and would thus have been "caught" while expanding v.

In Section 4, we present experimental evidence showing the potential of compressing the proof size by target set widening in practice: for our concurrent `C` program benchmarks, we observed reductions in the numbers of proof nodes, in the longest traversed paths, and in the maximum thread counts across the proofs by 95%, 67%, and 50%, respectively.

3.3. Coverability Analysis by Target Set Widening: The Algorithm

We now present our algorithm in detail. It features all data structures used in Algorithm 1, namely a set $U \subseteq V$ with *vertices* that represent encountered configurations, and a work set $W \subseteq U$ of *unprocessed* vertices. In contrast to Algorithm 1, we organize the elements of U into a forest, with candidate vertices as roots, and nodes discovered as cover predecessors of each candidate forming the tree underneath the root. The idea is that whenever a node turns out coverable, so are all its cover successors, including the widening candidate at the root. This candidate therefore was a bad choice and should never have been expanded in the first place. Our algorithm thus prunes the entire tree rooted at that candidate. It identifies the tree to prune using a function ζ that maps each node to the root of its containing tree.

More precisely, the algorithm maintains

(1) a set E of directed edges between vertices: $E \subseteq U \times U$,
(2) a downward-closed set $D \subset U$ of configurations, and
(3) a mapping $\zeta : U \to U$.

We write $u \hookrightarrow_E r$ for $(u, r) \in E$ and $\hookrightarrow_E^*$ for the reflexive transitive closure of $\hookrightarrow_E$. Intuitively, E stores cover predecessor edges that were expanded: $\hookrightarrow_E \subseteq \hookrightarrow$.[1] The set D stores states that were shown to be coverable and hence is an underapproximation of the coverability set: $D \subseteq \mathsf{Cover}$.

For a vertex r, $\zeta(p) = r$ indicates that p was encountered in the wake of expanding r backward, and r is either the query q or a widening candidate and was thus added to U during target set widening. We have $\zeta(v) = v$ precisely for candidate vertices v; other vertices are called *predecessor vertices*. Graph structure (U, E) gives rise to paths from predecessor vertices to candidate vertices.

We illustrate these definitions using the graph in Figure 7. The set U contains nine elements: $q = (2 \mid)$ is the query candidate vertex, $(1 \mid)$ and $(0 \mid 1)$ are widening candidates, and $(1 \mid 2)$, $(0 \mid 2)$, $(3 \mid 2, 2)$, $(3 \mid 1, 2)$, $(3 \mid 1, 1)$, and $(0 \mid 0, 1)$ are predecessor vertices; $(1 \mid 2)$ and $(0 \mid 0, 1)$ are nonminimal in U. (All other vertices in the figure have turned out coverable and have been removed from U.) Solid harpoon arrows determine the edges in the set E. The mapping ζ induces three partitions, one for each of the candidate vertices $(2 \mid)$, $(1 \mid)$, and $(0 \mid 1)$ (with two, six and one element(s), respectively); for example, $(1 \mid 2)$ was encountered after backward expanding $(2 \mid)$, and hence $\zeta(1 \mid 2) = (2 \mid)$. The set D is $\downarrow\{(3 \mid 1), (3 \mid 2)\} \cup I$ with $I = \{(0 \mid 0^i) : i \in \mathbb{N}\}$, witnessing failed widening attempts, which slow down the algorithm. (We discuss in Section 3.5 how we remedy this problem.)

The algorithm takes a set of initial states I and a noninitial target $q \notin I$ as input and maintains the invariant that the subgraph of (U, E) over vertices from the same partition forms a tree, with the candidate vertex as root. Each tree represents an

[1] In figures, we omit the subscript E on harpoon arrows.

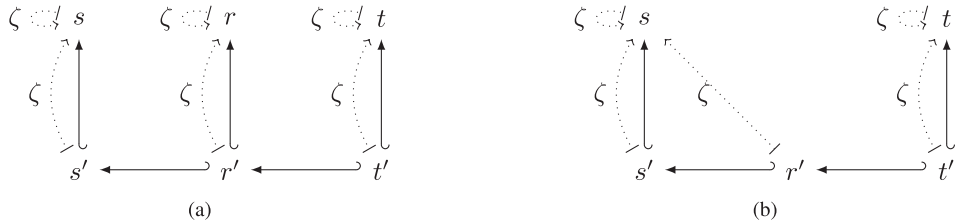

Fig. 9. Backtracking. (a) A partitioned graph structure with three candidate vertices, s, r, and t, each with a single cover predecessor (primed states) in its partition. The partition underneath r is to be pruned, but edge $r' \hookrightarrow_E s'$ is $\{r\}$-conflicting. (b) The structure obtained after calling Backtrack($\{r\}$). Node r' now belongs to s's partition, and r has been pruned.

attempt to prove the corresponding candidate uncoverable. The algorithm consists of three routines: Widen tries to add new candidate vertices, Backtrack prunes partitions whose candidate vertex proved coverable, and CAT is the main routine.

Widening. The Widen routine takes a vertex u and tries to widen the node set by an element in $\downarrow u$. More precisely, if the set $\mathcal{C}(u) = (\downarrow u) \setminus (\{u\} \cup D)$ of candidate elements is nonempty, we select a *minimal* element r from it; note that candidates must not come from the set D of elements known to be coverable. If r is new ($r \notin U$), it is inserted in the work and vertex sets; we set $\zeta(r) := r$. Otherwise, r's new role as candidate vertex and partition root must be acknowledged: the graph is *repartitioned* by modifying ζ for all vertices in the subtree with root r—that is, the set

$$\Lambda(r) = \{s \in U \mid s \hookrightarrow_E^* r \wedge \zeta(s) = \zeta(r)\}.$$

All vertices in r's subtree are removed from their old partition and form a new partition with r as root: we set $\zeta(s) := r$ for all $s \in \Lambda(r)$. For the example in Figure 7, the Widen routine is successfully called four times, with vertices $(1 \mid 2)$, $(3 \mid 2, 2)$, $(3 \mid 2, 2)$ again (D has changed in the meantime), and $(0 \mid 0, 1)$ as input.

Note that in the definition of the Widen routine, we assume that the downward closure of a finite set is finite (which ensures that the candidates set $\mathcal{C}(\cdot)$ is finite). This is guaranteed for the well–quasi-ordered systems induced by TTS according to Section 2 and generally for Petri nets, Broadcast protocols, and Lossy counter machines.

Backtracking. The Backtrack routine (Algorithm 2) prunes partitions represented by candidate vertices from a set P; this happens whenever some vertex in such a partition is found coverable. An obstacle is that some edges may point from the partition to be removed into another partition. Such edges (and the adjacent vertices) must be preserved, since otherwise paths generated by the other partition are destroyed.

Definition 4. Given a set P of candidate vertices, an edge $(r, s) \in E$ is called *P-conflicting* if $\zeta(r) \in P$ and $\zeta(s) \notin P$.

The Backtrack routine first resolves all conflicts (lines 1–3): for a conflicting edge $r \hookrightarrow_E s$, we reassociate vertices in $\Lambda(r)$ to $\zeta(s)$. Remaining vertices and edges of partitions in P can now be pruned (lines 4–7). It suffices to prune edges *ending* in $\zeta(r)$: edges starting from $\zeta(r)$ are pruned when their target vertices are processed; note that after resolving P-conflicts, those target vertices also have their roots in P. Figure 9 illustrates both steps. In the example in Figure 7, routine Backtrack is called on candidate vertices $(3 \mid)$ and $(3 \mid 2)$: the former is pruned alone, whereas the latter is pruned along with its cover predecessor $(3 \mid 1)$ (in both cases, no P-conflicting edges exist).

ALGORITHM 2: Backtrack(P)

Require: Set P with configurations to be removed, $P \subseteq \zeta(U)$
1: **for all** $(r, s) \in E$ such that (r, s) is P-conflicting **do**
2: **for all** $t \in \Lambda(r)$ **do**
3: $\zeta(t) := \zeta(s)$
4: **for all** $r \in U : \zeta(r) \in P$ **do**
5: $W := W \setminus \{r\}$; $U := U \setminus \{r\}$
6: **for all** $(t, r) \in E$ **do**
7: $E := E \setminus \{(t, r)\}$

Main routine. We introduce some terminology. A configuration v is *u-minimal* if it covers none of the vertices in u's partition, nor any immediate predecessor of such a vertex (the predecessors may lie outside of this partition).

Definition 5. Let $v \in V$ and $u \in \zeta(U)$. State v is *u-minimal* if $v \not\geq u$ and for all $s, s' \in U$ such that $s \hookrightarrow_E s'$ and $\zeta(s') = u$, we have $v \not\geq s$.

A set is *lower successor closed* if it is closed both under $\hookrightarrow_E$ successors and downward.

Definition 6. Let $X \subseteq V$. Set X is *lower successor closed* if for every $p \in X$, all vertices in $\downarrow p$ and all $\hookrightarrow_E$ successors of p belong to X.

We write $\lfloor v$ for the *least* lower–successor-closed set containing v. This set is obtained by closing $\{v\}$ downward and under $\hookrightarrow_E$ successors until fixed point. The point of this definition is that if v is coverable, so is every vertex in $\lfloor v$: the set Cover is lower successor closed.

Algorithm 3 shows the main routine CAT (for **C**overability **A**nalysis via **T**arget set widening). Input is a set I of initial configurations (downward closed by definition) and a noninitial target query $q \notin I$. At the outset, W and U contain one candidate vertex, the target q. The set D of elements found coverable contains the initial configurations, the set E of edges is empty, and ζ maps q to itself (line 1). Then we try to widen the target set by elements smaller than q.

The algorithm terminates with "q uncoverable" if no minimal unprocessed vertex remains. Otherwise, it selects and removes a minimal such vertex w. The **for** loop in line 6 now steps through all cover predecessors p of w that are $\zeta(w)$-minimal and processes them as follows:

Line 7 If p is in D, then p and all elements in $\lfloor p$ are known to be coverable. We will now distinguish.

Lines 8–9 If the query q is among the elements in $\lfloor p$, it is coverable; the algorithm terminates.

Lines 10–15 If q is not among the elements in $\lfloor p$, then the search must go on. We add all of $\lfloor p$ 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 candidates. We therefore create new minimal candidate vertices (lines 13 and 14). The **break** instruction then skips forward to the next iteration of the **while** loop (line 3). As a consequence of backtracking, unprocessed vertices that were previously not minimal may now be.

Lines 16–20 If p is not in D and hence not currently known to be coverable, then the graph is *expanded*. If p is new (line 18), then we add it to our work and

ALGORITHM 3: CAT(I, q): Coverability Analysis by Target Set Widening

Require: Set I of initial configurations, query target $q \notin I$
1: $W := \{q\}$; $U := \{q\}$; $D := I$; $E := \emptyset$; $\zeta : q \mapsto q$
2: Widen(q)
3: **while** $W \cap \min U \neq \emptyset$ **do**
4: **select** $w \in W \cap \min U$
5: $W := W \setminus \{w\}$
6: **for all** $p \in$ C-Pre(w): p is $\zeta(w)$minimal **do**
7: **if** $p \in D$ **then**
8: **if** $q \in \lfloor p$ **then**
9: **return** "q coverable"
10: **else**
11: $D := D \cup \lfloor p$
12: Backtrack$(\zeta(\lfloor p))$
13: **for all** $u \in \min U$ **do**
14: Widen(u)
15: **break** // exit the **for all** loop and go to next iteration of **while** in line 3
16: **else**
17: $E := E \cup \{(p, w)\}$
18: **if** $p \notin U$ **then**
19: $W := W \cup \{p\}$; $U := U \cup \{p\}$; $\zeta(p) := (w)$
20: Widen(p)
21: **return** "q uncoverable"

undecided lists and add it as predecessor vertex to w's partition. We also call the Widen routine to (try to) add smaller target elements.

3.4. Coverability Analysis by Target Set Widening: Correctness and Efficiency

We prove our target set widening algorithm correct, study its time complexity, and give a preview of its compact operation. We begin with a classical termination + partial correctness argument.

Algorithm 3 manipulates a node set U by adding elements to it in line 19 and during widening, but also by pruning elements from it during backtracking. As a result, set U does not grow monotonically. We will first prove termination of a variant of Algorithm 3 *without* backtracking.

Therefore, let Algorithm 3′ be the same as Algorithm 3, except that line 12 is replaced by **skip**.

THEOREM 7. *Algorithm 3′ terminates on all inputs.*

We first show the following property.

LEMMA 8. *Line 4 of Algorithm 3′ never selects an element twice during the run of the algorithm.*

PROOF. In this proof, we write "$\in_t$" for "element of, at time t".

Let w be an element selected in line 4, say at time t_0; we show that it will never be selected again. In line 5, w is removed from W. Since line 4 selects from $W \cap \min U$, element w needs to be added back into W, say at time $t_1 > t_0$, before it can be selected a second time.

Element w can be added to W in line 19, or during widening. In both cases, only elements not in U are added to W. Since $w \in_0 U$ (in fact, $w \in_0 \min U$), and—in Algorithm 3′—elements are never removed from U, we have $w \in_1 U$, so the addition of w to W at time t_1 is not possible. □

Now the proof of Theorem 7. The only loop that could cause nontermination is the **while** loop in line 3. We show that the set $\min U$ will eventually be depleted, terminating the loop.

Consider the sequence $p_1, p_2, \ldots$ of elements p added to U (keep in mind that in Algorithm 3′, these are never removed from U). Let $J = \{j \in \mathbb{N} | \neg(\exists i : i < j \wedge p_i \preceq p_j)\}$. The elements p_j with $j \in J$ constitute a "bad" sequence of configurations: one that never increases. Since $(V, \succeq)$ is a well quasi-ordering, J is finite, and since $1 \in J$, it is also nonempty.

Therefore, let $j = \max J$. After adding element p_j, only elements p_k $(k > j)$ that satisfy $p_k \succeq p_i$ for at least one previously added element p_i $(i < k)$ are added to U. We now distinguish:

(1) If there exists $i < k$ such that $p_k \succ p_i$, then $\min U$ is not changed by the addition of p_k: p_k is guaranteed not to be a minimal element of U.
(2) Otherwise, we have, for all $i < k$, $p_k \not\succ p_i$. By the definition of $\succ$, this means that for all $i < k$, $p_k \not\succeq p_i \vee p_i \succeq p_k$. Now pick $i < k$ such that $p_k \succeq p_i$ (existence guaranteed since $k > j$). For this i, we have $p_k \preceq\succeq p_i$. This in turn implies that $p_i \in \min U$: otherwise, there would exist $p_r \in U$ with $p_r \prec p_i \preceq p_k$, contradicting the condition leading to case (2). Element p_k is thus equivalent to $(\preceq\succeq)$ some minimal element (p_i) of U.

The addition of elements p_k to U thus either does not modify $\min U$ (1), or at most modifies $\min U$ by adding elements to $\min U$ whose local state vector is a permutation of the local state vector of some existing element of $\min U$ (2). The latter can happen only finitely many times, as there are only finitely many permutations of elements in $\min U$. Eventually, $\min U$ therefore stops changing. By Lemma 8, the **while** loop in line 3 of Algorithm 3′ eventually terminates. □

COROLLARY 9. *Algorithm 3 terminates on all inputs.*

PROOF. We need to determine the impact (on termination) of the backtracking. Algorithm 2 mostly removes elements that are also contained in D, namely all elements in $\downarrow p \subseteq D$, for some $p \in D$ (cf. line 12). Set D grows monotonically: elements are never removed from it. Further, elements in D are never added to U; this is guaranteed both in line 19 and in the calls to the widening routine. As a result, the elements in D that fall victim to pruning during backtracking will never be added into U again, so pruning these elements can only accelerate termination.

On the other hand, Algorithm 2 may also remove some element $s \notin D$, whose coverability has not been decided yet. Element s's root, in contrast, does belong to D: in line 12 of Algorithm 2, we have $\zeta(\downarrow p) \subseteq \downarrow p$, and all elements in $\downarrow p$ were added to D in line 11. Root $\zeta(s)$, which is removed in the same call to Algorithm 2, will thus never be added to U again. When and if s later reappears in U, it must therefore be associated with another root node (s may itself be a root at that time). Element s can therefore be removed from and reintroduced into U only finitely many times, namely at most whenever we add query elements during widening, which happens finitely often. □

Intuitively, pruning and reintroducing an element $s \notin D$ causes finite delay of the termination of the algorithm but keeps the search graph compact.

We continue by proving partial correctness.

THEOREM 10. *If control reaches line 9 in Algorithm 3, the query target q is coverable.*

PROOF. We show that $D \subseteq \mathsf{Cover}$ is an invariant of the algorithm. The theorem then follows: in line 9, we have $p \in D$ (hence, $p \in \mathsf{Cover}$) and $q \in \downarrow p \subseteq \mathsf{Cover}$ (since $p \in \mathsf{Cover}$, and coverability is closed under $\hookrightarrow_E$ successors and downward).

We prove $D \subseteq \mathsf{Cover}$ by induction over the number of modifications made to D in line 11. Before any such modifications, we have, as per line 1, $D = I \subseteq \mathsf{Cover}$. Now assume that $D \subseteq \mathsf{Cover}$. In line 11, we have $p \in D$, and hence $p \in \mathsf{Cover}$ by the induction hypothesis. Again, by the preceding closure property of coverability, we obtain $D' = D \cup \lfloor p \subseteq \mathsf{Cover}$, which completes the step. □

THEOREM 11. *If control reaches line 21 in Algorithm 3, the query target q is uncoverable.*

PROOF. We show that if control reaches line 21, the upward-closed set $\uparrow U_{fin}$ is an *uncoverability proof* for q (Definition 2) for the final value U_{fin} of variable U at line 21. The fact that q is uncoverable then follows immediately by that definition: all elements of U_{fin} are uncoverable.

We prove the first three conditions of Definition 2:

Condition (1): $\uparrow U_{fin}$ contains q, in fact $q \in U_{fin} \subseteq \uparrow U_{fin}$: q is added to U in line 1, and the call to backtrack (the only chance for q to be removed) is guarded by the condition $q \notin \lfloor p$ (line 10), which implies that $q = \zeta(q) \notin \zeta(\lfloor p)$, so q is never removed.

Condition (2): $\uparrow U_{fin}$ is closed under preimages:

(a) We show the following: cover predecessors of elements of $\min U_{fin}$ are in U_{fin}: $\mathsf{C\text{-}Pre}(\min U_{fin}) \subseteq U_{fin}$:
Suppose that $w \in \min U_{fin}$, hence $w \in U_{fin}$. At the time in which w was added to U, it was also added to W (this is true in line 19 and for all calls to the widening routine). When the algorithm terminates in line 21, we have $W \cap \min U_{fin} = \emptyset$. Hence, w was, at some point, removed from W but not from U, which only happens in line 5, following which w's cover predecessors p are processed.
If $p \notin D$ (and new to U), it is added to U. If $p \in D$ and $q \in \lfloor p$, then the algorithm terminates in line 9; this case does not apply to Theorem 11. If let $p \in D$ and $q \notin \lfloor p$ (line 10), then p is not added to U, but we backtrack: we also remove the successor w, so the closure property is preserved.

(b) Now let (i) $w \in \uparrow U_{fin}$ and (ii) $p \rightarrowtail w$; we show $p \in \uparrow U_{fin}$:
Due to (i), there exists $v \in U_{fin}$ such that $v \preceq w$. Further, let $t \in \min U_{fin}$ such that $t \preceq v$. Due to (ii), there exists $o \in \mathsf{C\text{-}Pre}(w)$ such that $o \preceq p$. Applying Lemma 3 to o, w, and $t(\preceq w)$ tells us that there exists $m \in \mathsf{C\text{-}Pre}(t)$ such that $m \preceq o$. By (a), we conclude $m \in U_{fin}$. Since $p \succeq m$, we also have $p \in \uparrow U_{fin}$. □

Condition (3): $\uparrow U_{fin}$ is disjoint from the initial configuration set I: we first show $U_{fin} \cap I = \emptyset$:

(a) By Algorithm 3′s precondition, $q \notin I$. Hence, $U_0 \cap I = \emptyset$, for the initial value $U_0 = \{q\}$ of U.

(b) Only elements not in D are ever added to U (this is true in line 19 and for all calls to the widening routine); since $I \subseteq D$ is an invariant of the algorithm, such additions exclude initial configurations.

Now let $y \in \uparrow U_{fin}$—that is, there exists $x \in U_{fin}$ with $y \succeq x$. Since $U_{fin} \cap I = \emptyset$, $x \notin I$. This implies $y \notin I$, since I is downward closed. As a result, $\uparrow U_{fin} \cap I = \emptyset$. □

Space efficiency. We give a preview of our algorithm's compact operation by comparing the widening and backtracking strategy to the classical backward exploration without it (a comprehensive empirical evaluation of the algorithm, including runtime

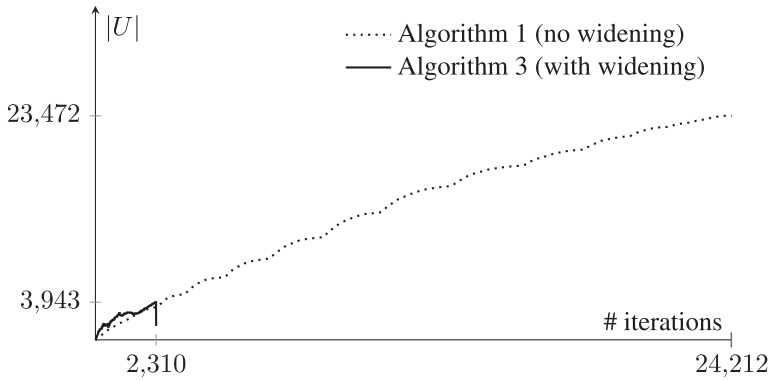

Fig. 10. Impact of widening on the number of iterations and graph vertices. The target set widening strategy reduces the iteration count for the Apple Bus protocol benchmark from 24,212 to 2,310, with a maximum of 3,943 explored vertices compared to 23,472

performance, is presented in Section 4). The example at hand concerns a mutual exclusion property for an implementation of an Apple Bus protocol in the FreeBSD operating system (benchmark FreeBSD-ak; a code fragment was shown earlier in Figure 1).

Figure 10 plots the size of U on the vertical axis against the iterations of the main **while** loop. The backtracking nature of the widening approach is evident from the fact that the curve occasionally drops, namely whenever candidates with large partitions in their backward expansion were found to be coverable and pruned.

Algorithm 3 computes uncoverability proofs that satisfy minimality requirement (4) of Definition 2: for $v \notin U$ and $r \in U$, if $v \prec r$ then v is coverable. To see this, note first that $r \in U$, and $v \prec r$ implies that at some point during the run of the algorithm, v was explored and therefore an element of U: the Widen routine is called eagerly on the entire downward closure of encountered configurations such as r. Since at the end $v \notin U$, configuration v was removed some time thereafter. This only happens in the Backtrack routine and only to elements found coverable.

Our current implementation does not satisfy requirement (5) of Definition 2, stating that the uncoverability proof be "least": no proper subset satisfies conditions (1)–(4) of that definition. It is easy to enforce this requirement by adding redundancy checks to the set U at certain points where the algorithm modifies it. This of course affects the runtime of the algorithm adversely; we found that the benefit in reduced proof size does not compensate for this effect.

3.5. Trimming the Node Set: Discarding Nonminimal Elements

Compared to the classical backward search Algorithm 1, our coverability algorithm has an apparent blemish: it regularly keeps nonminimal elements in the node set U, which is therefore not as compact as it could be. As an example, we revisit the proof construction shown in Figure 7. When configuration $(3 \mid 2, 2)$ is reached as a cover predecessor of another element, widening identifies configuration $(3 \mid)$ as a candidate and adds it to U. At this point, the node set contains the nonminimal element $(3 \mid 2, 2)$. This element is not discarded at this time (as is done in line 8 in Algorithm 1). The reason is that elements nonminimal at discovery time may become minimal later, when smaller elements have been identified as coverable and are pruned. That is exactly what happens in Figure 7: both widening candidates $(3 \mid)$ and (later) $(3 \mid 2)$ eventually turn out coverable, are pruned, and $(3 \mid 2, 2)$ is part of the final uncoverability proof as a minimal element (Figure 8).

An alternative to keeping nonminimal elements in U is to maintain a *list* L of the vertices in the order they were encountered. Knowledge of the processing order allows us to identify elements v that were encountered after an element r that is about to be pruned. If v was nonminimal at discovery time, it will have been discarded from U by an algorithm that restricts U to contain minimal elements. Given list L, we can now revive v, by adding it to the node and work sets U and W.

With this modification, we can ensure that U contains minimal elements only: whenever a new element p is to be added to U, we replace U by $\min(U \cup \{p\})$. In particular, this means that for each call Widen(u) that successfully identifies undecided candidates smaller than u, u is in fact removed from U, since it is no longer minimal. In agreement with the classical backward search Algorithm 1, this algorithm only keeps minimal vertices in U—that is, it maintains the invariant $\min U = U$. Note that the list L (containing many nonminimal configurations) will be depleted on termination, whereas U contains the (now minimal) nodes that constitute the uncoverability proof (if q is uncoverable). The number of stored nodes can therefore be expected to be smaller on termination than with Algorithm 3. We call this version the *lean* variant of Algorithm 3.

In contrast to Algorithm 1, the lean variant suffers multiply from bad widening choices: the bad candidates have to be pruned, along with all nodes encountered after expanding them. Moreover, unexpanded vertices encountered after vertices that have just been pruned need to be revived. The benefit of the lean variant over the original Algorithm 1 therefore decreases with an increasing number of bad candidate choices. The incentive to avoid candidates that are in fact coverable is thus particularly high for this variant.

External coverability results. The determination of what elements are coverable is, of course, the ultimate goal of our algorithm, so we cannot *rely* on such information when deciding whether to add a potential widening candidate. If, however, we happen to know that a particular element is coverable, we will not add it to the candidate set. Such incidental information can come from an external source. We call such a source a *coverability oracle*.

We require that a coverability oracle be *sound* in reporting coverable configurations. The oracle and Algorithm 3 run in parallel and synchronize via the set D: the oracle populates this set with coverable elements. Receiving such updates, Algorithm 3 terminates if the input query belongs to D or otherwise invokes the Backtrack routine on now known-to-be-coverable candidate vertices in regular intervals to restore disjointness of the sets U and D. Should the algorithm ever accidentally choose an unreported coverable element for expansion, it will eventually prune such element once correctly classified.

The fact that we only require sound reporting permits coverability oracles of varying degrees of incompleteness: nonterminating procedures are as eligible as partial algorithms that "overlook" some coverable elements. Typical oracles will perform a forward-directed search, such as a random or enumerative reachability analysis like BOOM [Basler et al. 2009], or generalizations of the Karp-Miller procedure [Karp and Miller 1969] to broadcast synchronization. In our experiments, we use a version of Karp-Miller that never accelerates across broadcast transitions: the backward traversal used to find opportunities for acceleration simply stops on encountering broadcasts. As a result, this version of the procedure is sound for broadcasts but may not terminate.

By contrast, the standard Karp-Miller procedure is *not* suitable as an oracle, since—in the presence of broadcasts—it may return an overapproximation of the set of coverable configurations. This can be observed for the TTS on the left of Figure 11;

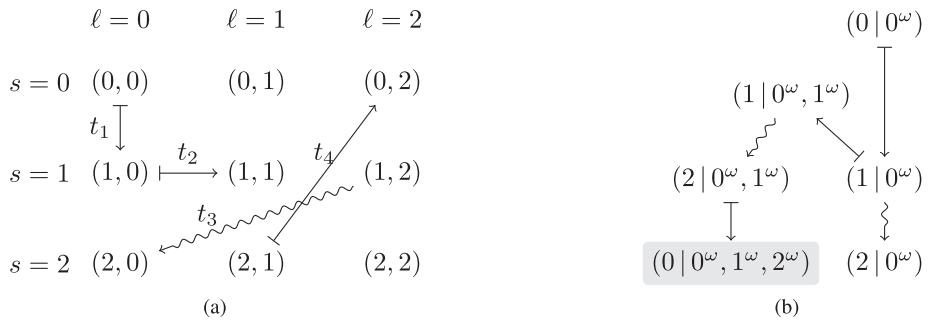

Fig. 11. Karp-Miller overapproximates for broadcasts. (a) TTS with initial thread state $(0, 0)$ and broadcast transition t_3. (b) The corresponding Karp-Miller tree, which contains omega configuration $(0 \mid 0^\omega, 1^\omega, 2^\omega)$ (gray) and thus falsely reports $(0 \mid 2, 2)$ as coverable.

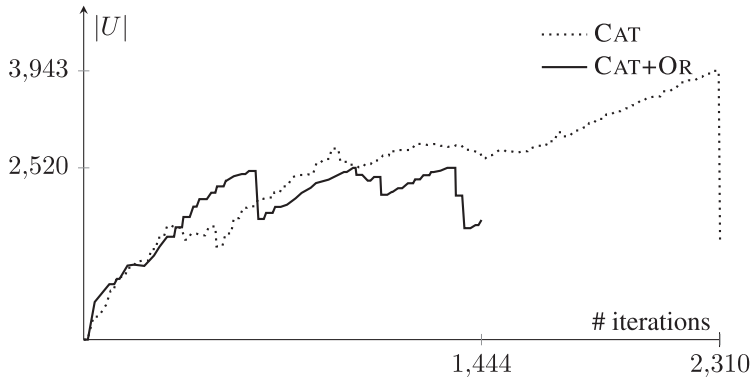

Fig. 12. Benefiting from external coverability results. Algorithm 3 without and with support by a generalization of the Karp-Miller procedure as oracle (cf. Figure 10). The oracle significantly reduces the work related to coverable candidates, both in terms of iteration count and maximum graph size.

the corresponding Karp-Miller tree is shown on the right. The acceleration that yields $(0 \mid 0^\omega, 1^\omega, 2^\omega)$ introduces imprecision: due to the broadcast transition, only one thread can reside in local state $\ell = 2$ at any time, which the algorithm misses. As a result, it falsely reports, for example, that configuration $(0 \mid 2, 2)$ is coverable. Receiving this false report, Algorithm 3 would remove this configuration from U and fail to expand it.

It is clear that a coverability oracle is beneficial whether or not the algorithm enforces minimality of elements in U (the oracle is just more direly needed if it does). In our experimental section, we therefore present results not only for Algorithm 3, called CAT, and its lean variant with oracle, called CATLEAN+OR, but also for Algorithm 3 equipped with an oracle, called CAT+OR. Figure 12 presents a preview of how Algorithm 3 benefits from an oracle, reusing the Apple Bus protocol benchmark. The plot reveals a significant reduction in work related to coverable candidates. The oracle reports roughly 90% of the coverable candidates ahead of our approach without oracle. These observations indicate that while forward directed search may not be complete for checking programs with arbitrary thread numbers, it is good at reporting coverable configurations rapidly, which in the context of our algorithm is all that is needed.

Table I. Benchmark Characteristics

id/Program	*S*	*L*	*LOC*	*Mtx*	*Cnd*	*Safe*	id/Program	*S*	*L*	*LOC*	*Mtx*	*Cnd*	*Safe*
1/PrngSimp-L	2	4	63	●	○	●	13/Peterson	4	0	37	○	○	●
2/PrngSimp-C	1	5	95	○	○	●	14/Szymanski	3	0	54	○	○	●
3/Stack-L	4	2	79	●	○	●	15/Dekker	4	0	50	○	○	●
4/Stack-C	3	3	89	○	○	●	16/RW-lock	4	0	58	○	○	●
5/BSD-ak	1	7	516	●	●	●	17/Timed-mutex	5	0	63	○	○	●
6/BSD-ra	2	21	413	●	●	●	18/BS-loop	0	6	24	○	○	○
7/NetBSD-sp	1	28	1045	●	●	●	19/Cond	1	3	56	●	○	●
8/Solaris-sm	1	56	616	●	●	●	20/S-Loop	5	0	60	●	○	●
9/Boop	5	2	89	○	○	○	21/Func-P	2	1	67	●	○	●
10/Double-1	3	0	70	●	○	●	22/Pthread	5	0	85	●	○	●
11/Double-2	3	0	73	●	○	●	23/MinUCP-ex	1	0	80	○	○	●
12/Double-3	3	0	66	●	○	●	24/Mozilla-vf	4	3	82	●	○	●
							25/Spin	2	0	37	●	○	●

Note: *S*, number of shared vars; *L*, number of local vars; *LOC*, number of lines of code. In columns *Mtx* and *Cnd*, a black disc indicates "has mutexes" or "has condition variables for broadcast synchronization," respectively.

4. EXPERIMENTAL EVALUATION

4.1. Benchmarks and Experimental Setup

In this section, we evaluate Algorithm 3 (Section 3.3) and its lean variant (Section 3.5) with and without widening on 25 concurrent `C` programs, in which threads synchronize via locks, or in a lock-free manner via atomic compare-and-swap instructions, and broadcasts. For each benchmark, we consider a safety property specified via an assertion. In total, the programs comprise roughly 4,000 lines of code, featuring three shared and six local variables on average; Table I enumerates statistical data for each program individually.

We provide a brief description of the programs (most are used in our prior work [Donaldson et al. 2011] or in Gupta et al. [2011b]):

1–4: Thread-safe algorithms. Concurrent pseudo–random number generators and stack data structures (both from Goetz et al. [2006]; the latter are adapted from IBM implementations). For each type, we consider a blocking version that uses mutexes and a lock-free variant with compare-and-swap primitives (indicated by the suffix **L** and **C**, respectively).

5–9: OS code. Implementation of an Apple Bus protocol in FreeBSD (see Figure 1), the operating system code related to the RDMA ZFS file system support and interface/system monitoring (obtained from svn.freebsd.org and src.opensolaris.org; all use Mesa style condition variables), and a Linux driver skeleton.

10–17: Mutex algorithms. Programs where multiple locks control access to a shared resource and classical mutex algorithms [Gupta et al. 2011b].

18–22: Pthread programs. Several programs that use the `C` POSIX Threads library.

23–25: Miscellaneous. The program in Figure 3 used to illustrate our minimal uncoverability proof construction, a program from Donaldson et al. [2012], a fix for a concurrency bug in Mozilla [Lu et al. 2008], and a program used in Flanagan and Qadeer [2003].

Almost all programs are parametric—that is, their procedures are executed by an arbitrary number of threads. Exceptions are programs 13 through 17 and 24, taken from Gupta et al. [2011a], which are designed for a fixed thread count (of two). These examples do not even exploit the strength of our unbounded-thread approach.

Implementation. We implemented Algorithm 3 and its lean variant for TTSs and transfer Petri nets in our tool BREACH, equipped with a generalization of the Karp-Miller procedure as coverability oracle. Our tool (v1.0 implements the original, and v2.0 the lean variant) is available at www.cprover.org/bfc. The backward search runs in parallel with the oracle, which reports coverability results to a shared data pool that the backward search taps into at regular intervals. To measure the impact of our approach, the oracle can be deactivated, turning BREACH into the refined version of the classical backward search (Algorithm 1). As a trade-off between efficiency and proof compaction, we do not add candidate vertices that involve two threads or more. 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 [Donaldson et al. 2011] and handles function calls (nonrecursive) by inlining. All experiments are performed on a 3GHz Intel Xeon machine with 20GB memory, running 64-bit Linux, with a timeout of 30 minutes.

Our evaluation is carried out in two steps: (1) we compare against several coverability checkers for (extended) Petri nets (Section 4.2), and (2) we compare against the recent software verifier CREAM [Gupta et al. 2011a, 2011b] (Section 4.3), which generates rely-guarantee and Owicki-Gries type proofs.[2]

4.2. Comparison with Coverability Checkers

The SATABS verifier requires 59 CEGAR iterations until the (TTS) abstractions for the 25 programs stabilize; for example, 3 iterations are necessary to prove the program in Figure 3 correct (MINUCP-EX). The obtained abstractions feature up to 34,880 thread states (for FUNC-P) and 746,770 transitions (for DOUBLE-2). Figure 13 plots the total model checking runtimes (scaled logarithmically) for all methods. The curves in the graph correspond to the following checkers ($*$ indicates that the tool supports transfer transitions):

- CAT$*$: Our Algorithm 3 with no coverability oracle (v1.0)
- CAT+OR$*$: Our Algorithm 3 equipped with the coverability oracle (v1.0)
- CATLEAN+OR$*$: Lean variant of Algorithm 3 (Section 3.5) with oracle (v2.0)
- PETR-BC$*$: Backward search with several heuristics (v0.1) [Meyer and Strazny 2010]
- BC$*$: Backward reachability analysis [Abdulla et al. 1996; Abdulla 2010] (Algorithm 1)
- EEC-AR: Forward analysis with enumerative refinement (v1.03) [Geeraerts et al. 2006]
- CSC-KM: A refined Karp-Miller procedure (v0.1) [Geeraerts et al. 2007]
- IIC: Incremental, inductive coverability algorithm [Kloos et al. 2013]
- TINA-KM: The classic Karp-Miller tree construction (v3.0) [Berthomieu and Vernadat 2009]
- IST-BC$*$: Standard backward reachability analysis (v1.03) [Ganty et al. 2007]
- TSI-AR: A variant of EEC-AR with backward refinement (v1.03) [Ganty et al. 2009]

Our methods based on widening achieve significantly better results compared to existing methods. The improvement of about two orders of magnitude over the best previous method, PETR-BC, which in turn is roughly two orders of magnitude faster than

[2]Available at software.imdea.org/~pierreganty; www.ulb.ac.be/di/verif/ggeeraer/CSC.html; petruchio.informatik.uni-oldenburg.de; http://projects.laas.fr/tina; and www.model.in.tum.de/~popeea/research/threader.html.

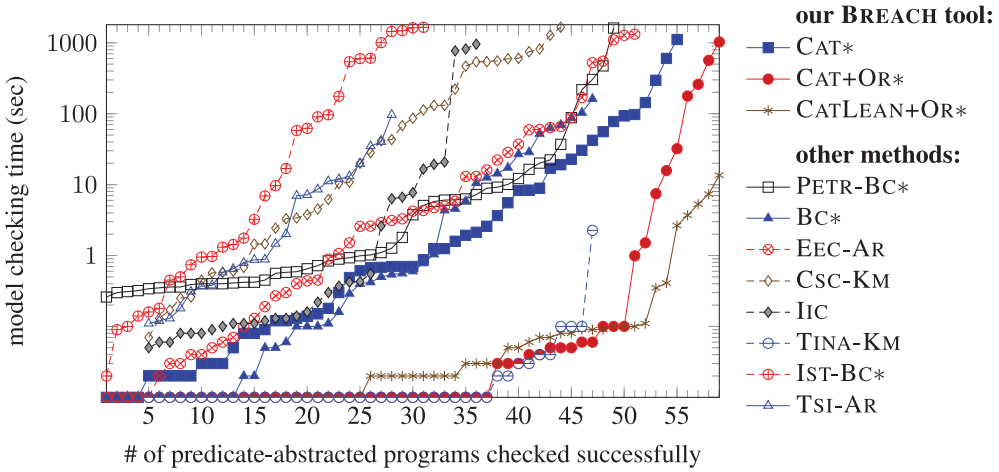

Fig. 13. Comparison for CEGAR abstractions. A Cactus plot compares our implementation of the target set widening algorithm in the BREACH tool with existing coverability methods. An entry of the form (k, t) for some curve shows the time t it took to solve the k easiest—for the method associated with that curve—predicate-abstracted C programs (the order thus varies across methods). Four benchmarks feature transfer transitions, namely those obtained from programs with condition variables (cf. Table I). Tools supporting transfers are marked with an asterisk (∗). The curves for the other tools start at $k = 5$, skipping the four transfer benchmarks.

the remaining methods, shows that our widening approach has far more impact than, for example, structural invariant heuristics. Particularly for program abstractions, we found that statically precomputed overapproximations tend to be irrelevant for the safety property or too imprecise, underpinning the claim made in Finkel et al. [2002]. On the other hand, the inferiority of the pure backward analysis BC compared to PETR-BC indicates that the observed improvements are a consequence of our widening technique.

To measure the difference between standard and minimal uncoverability proofs, we removed the self-imposed upper bound on the number of threads in candidate vertices. In this setup, we observed the following reductions (averaged): the length of the longest traversed path goes down from 28 to 14 (−50%), the thread count appearing in the proof from 6 to 2 (−67%), and the number of minimal configurations (= proof size) from 22,518 to 1,222 (−95%). Although the classical backward approach involves up to eight threads in a proof, our approach generates a minimal uncoverability proofs with three threads for MINUCP-EX (cf. Figure 7) and proofs with no more than two threads for the other 24 programs. The bound on the thread dimension in candidate vertices mentioned earlier diminishes these improvements somewhat but marginally (we enforced the bound for the results in Figure 13, as it results in much reduced runtimes).

4.3. Comparison with CREAM

We evaluate the scalability of our methods against the recent verifier CREAM, which exploits the compositional nature of many programs [Gupta et al. 2011a, 2011b]. Front-end capabilities of CREAM are similar to that of SATABS, facilitating comparison of both.[3] We therefore compare the performance of CREAM and SATABS on the 25 benchmark programs described in Section 4.1. We emphasize that CREAM is not optimized for

[3]A notable difference is that CREAM supports neither pointers nor broadcasts and approximates numeric C types via exact arithmetic. To apply CREAM to benchmarks 5 through 8, we use overapproximations that do not affect the verification outcome.

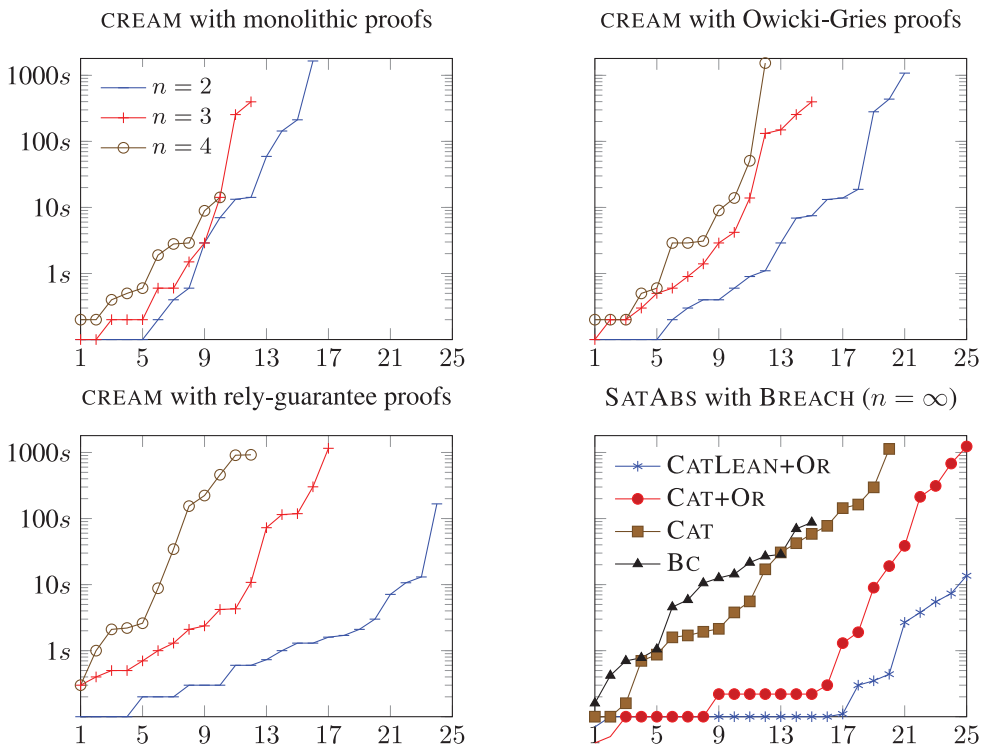

Fig. 14. Cactus plots comparing runtimes of SATABS with BREACH as model checker for the 25 programs and unbounded threads with CREAM for a given number of threads.

parametric systems but is still one of the few available verifiers operating directly on multithreaded C programs.

Figure 14 plots the fraction of programs checked successfully by different methods for different thread numbers. Again, the y-axes are on a logarithmic scale. The four subfigures correspond to CREAM with monolithic, Owicki-Gries [Owicki 1975], and rely-guarantee [Jones 1983] style reasoning, respectively, and SATABS with BREACH as model checker. Our approach (*unbounded* thread count) performs significantly better than the CREAM verifier (*fixed* thread count), for which time and space used for the proofs usually grow exponentially in the number of threads.

5. RELATED WORK

Algorithmic solutions to coverability analysis were first proposed for vector addition systems in a landmark paper by Karp and Miller [1969] (implemented in TINA-KM [Berthomieu and Vernadat 2009]). The solution constructs a pseudoreachability tree by forward exploration and replaces newly discovered configurations that are strictly greater than predecessors using an infinity measure. The approach has nonprimitive recursive worst-case complexity [Rackoff 1978]. Due to the undecidability of the place-boundedness problem [Dufourd et al. 1998], extensions to broadcast operations are inevitably incomplete. An example is the Covering Graph procedure from Emerson and Namjoshi [1998], which was shown to fail to terminate on certain systems [Esparza et al. 1999]. An improvement of the Karp-Miller procedure that computes minimal coverability sets is Geeraerts et al. [2007] (implemented in CSC-KM), and the Karp and Miller algorithm with pruning from Reynier and Servais [2011] (not available

online). An approach that constructs compact yet not necessarily minimal coverability sets is Valmari and Hansen [2012]. Their algorithm prioritizes unexplored states with larger thread numbers to speed up convergence (not available online). We experimented with various selection heuristics for the Karp-Miller–like coverability oracle. In our setup, treating configurations with *smallest* thread numbers performed best by far. One explanation is that dealing with such configurations is algorithmically easier, and that our widening algorithm does not benefit from large configurations being reported by the oracle. In our experiments, the largest thread count that appeared in a standard uncoverability proof is six.

To afford more flexibility in modeling parameterized programs, various algorithms were later proposed for well–quasi-ordered systems, originally in a pure backward fashion [Abdulla et al. 1996] (implemented in IST-BC and PETR-BC [Meyer and Strazny 2010]), later as forward exploration [Finkel and Goubault-Larrecq 2009; Zufferey et al. 2012], or as a backward and forward unfolding algorithm [Abdulla et al. 2004]. The paradigm presented in Geeraerts et al. [2006] is also a pure forward algorithm; it constructs abstractions of increasing precision (implemented in EEC-AR). The recent approach from Abdulla et al. [2013] is based on cutoffs and exploits the fact that analyzing a small number of threads is often sufficient to expose errors (not available online). It performs parameterized verification by inspecting a small set of instances of a system to show correctness. Such an approach has two drawbacks. First, it is likely to fail when systems require the inspection of large thread numbers. Second, performance degrades when many transitions do not affect the correctness of a property, such as those induced by detached processes that operated on a disjoint set of shared variables. As an example, their approach fails for the Kanban system from [Ganty et al. n.d.], whereas Karp-Miller–like approaches (including our coverability oracle) report the error almost instantaneously.

Solutions combining forward and backward exploration are rare; we are only aware of the methods described in Finkel et al. [2002] and Ganty et al. [2006], as well as the very recent approach from Kloos et al. [2013]. Finkel et al. [2002] propose to use a CSC-KM–like approach to compute overapproximations of the coverability set, which are then used in a *subsequent* backward exploration to prune the search space. Our experimental results reported in Section 4.2 demonstrate, however, that this approach cannot cope with programs of the sizes that we consider, simply because their computation is too expensive. Ganty et al. [2006] combine overapproximations computed in a forward fashion, which are refined by using backward underapproximations (implemented in TSI-AR). On an abstract level, our algorithm can be seen as the dual of this approach. Interestingly, performance-wise TSI-AR cannot benefit from this similarity; TSI-AR performed worst in our experimental comparison. Finally, the algorithm in Kloos et al. [2013] (implemented in IIC) computes an inductive invariant by maintaining a list of overapproximations of forward-reachable states, and strengthening them (in a backward manner) using counterexamples to inductiveness. Note that our uncoverability proofs introduced in Definition 2 are inductive proofs of the backward nonreachability of the initial states from q.

Other work that, like ours, takes parameterized system-level software as input includes earlier work on multithreaded Java programs [Delzanno et al. 2002], which in fact uses a set of communication primitives and derived semantics very similar to ours, and rewrites them into *multitransfer Petri nets* using a form of counterabstraction. Our earlier cutoff-based approach [Kaiser et al. 2010] combines *finite-state* forward exploration with infinite-state backward exploration. Recent work [Farzan and Kincaid 2012; Farzan et al. 2013] over dataflow graph representations of parameterized concurrent programs has been applied to safety property verification. These methods do not explore, in a model checking fashion, replicated finite-state procedures but instead

aim to find (possibly inductive) program invariants. Interestingly, Farzan et al. [2013] shows that their notion of inductive dataflow graphs can serve as succinct correctness proofs for safety properties, much like the minimal uncoverability proofs our algorithm tries to find. We will leave the precise relationship between this very recent work and ours for future investigation. We have compared with PETR-BC, EEC-AR, CSC-KM, TINA-KM, IST-BC, TSI-AR, and IIC in Section 4.2.

6. CONCLUSION

We introduced a new approach to the coverability problem in well–quasi-ordered systems. The novelty of our algorithm is the way in which it identifies and compactly represents the uncoverable elements backward reachable from a given query target by widening the target set by elements whose backward expansion can be expected to terminate quickly. Far from being a mere infinite state machine coverability checker, our algorithm can be used to check assertion failures, mutual exclusion violations, and many other properties for parameterized programs communicating via realistic primitives such as mutexes, shared variables, or 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 previous coverability approaches by orders of magnitude, enabling the verification of programs that to date were out of the scope of existing technology.

Although our concrete implementation of these ideas has proven to be very successful and efficient in solving real verification problems, the purpose of this article is also to propose our search organization and the cooperation between the backward and forward components of the algorithm as a new general paradigm for tackling coverability problems. Instances of this paradigm may independently combine well with other efficiency improvements, such as those based on structural net invariants.

ACKNOWLEDGMENTS

We would like to thank Michael Tautschnig for assistance with SATABS, as well as Pierre Ganty, Leopold Haller, Philipp Rümmer, and Emelie Vollmer for their insightful comments on earlier drafts of this work. We finally appreciate the careful reading and the numerous suggestions for corrections and improvements by the TOPLAS referees.

REFERENCES

P. A. Abdulla. 2010. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic* 16, 4, 457–515.

P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. 1996. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*. 313–321.

P. A. Abdulla, F. Haziza, and L. Holík. 2013. All for the price of few. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 7737. Springer, 476–495.

P. A. Abdulla, S. P. Iyer, and A. Nylén. 2004. SAT-solving the coverability problem for Petri nets. *Formal Methods in System Design* 24, 1, 25–43.

G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. 2009. Symbolic counter abstraction for concurrent software. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*. 64–78.

B. Berthomieu and F. Vernadat. 2009. The Tina Tool, Release 2.9.6, LAAS/CNRS. Retrieved September 4, 2014, from http://projects.laas.fr/tina.

E. Cardoza, R. J. Lipton, and A. R. Meyer. 1976. Exponential space complete problems for Petri nets and commutative semigroups: Preliminary report. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing (STOC'76)*. 50–54.

G. Ciardo. 1994. Petri nets with marking-dependent arc cardinality: Properties and analysis. In *Application and Theory of Petri Nets*. Lecture Notes in Computer Science, Vol. 815. 179–198.

G. Delzanno, J.-F. Raskin, and L. V. Begin. 2002. Towards the automated verification of multithreaded Java programs. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*. 173–187.

A. F. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl. 2012. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design* 41, 1, 25–44.

A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. 2011. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. 356–371.

C. Dufourd, A. Finkel, and P. Schnoebelen. 1998. Reset nets between decidability and undecidability. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*. 103–115.

A. Eliëns and E. P. de Vink. 1992. Asynchronous rendez-vous in distributed logic programming. In *Proceedings of the REX Workshop on Semantics: Foundations and Applications*. 174–203.

E. A. Emerson and K. S. Namjoshi. 1998. On model checking for non-deterministic infinite-state systems. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*. 70–80.

J. Esparza, A. Finkel, and R. Mayr. 1999. On the verification of broadcast protocols. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*. 352–359.

A. Farzan and Z. Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 297–308.

A. Farzan, Z. Kincaid, and A. Podelski. 2013. Inductive data flow graphs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. 129–142.

D. Figueira, S. Figueira, S. Schmitz, and P. Schnoebelen. 2011. Ackermannian and primitive-recursive bounds with Dickson's lemma. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS'11)*. 269–278.

A. Finkel, G. Geeraerts, J.-F. Raskin, and L. V. Begin. 2006. On the *omega*-language expressive power of extended Petri nets. *Theoretical Computer Science* 356, 3, 374–386.

A. Finkel and J. Goubault-Larrecq. 2009. Forward analysis for WSTS, part II: Complete WSTS. In *Automata, Languages, and Programming*. Lecture Notes in Computer Science, Vol. 2009. Springer, 188–199.

A. Finkel, J.-F. Raskin, M. Samuelides, and L. V. Begin. 2002. Monotonic extensions of Petri nets: Forward and backward search revisited. *Electronic Notes in Theoretical Computer Science* 68, 6, 85–106.

A. Finkel and P. Schnoebelen. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science* 256, 1–2, 63–92.

C. Flanagan and S. Qadeer. 2003. Thread-modular model checking. In *Proceedings of the 10th International Conference on Model Checking Software (SPIN'03)*. 213–224.

P. Ganty, L. V. Begin, and A. Piron. n.d. The MIST2 Tool (Version 0.1). Retrieved September 4, 2014, from https://github.com/pierreganty/mist.

P. Ganty, G. Geeraerts, J.-F. Raskin, and L. V. Begin. 2009. Le problème de couverture pour les réseaux de Petri: Résultats classiques et développements récents. *Technique et Science Informatiques* 28, 9, 1107–1142.

P. Ganty, C. Meuter, G. Delzanno, G. Kalyon, J.-F. Raskin, and L. Van Begin. 2007. Symbolic data structure for sets of *k*-uples. Technical Report. Université Libre de Bruxelles, Belgium.

P. Ganty, J.-F. Raskin, and L. V. Begin. 2006. A complete abstract interpretation framework for coverability properties of WSTS. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*. 49–64.

G. Geeraerts, J.-F. Raskin, and L. V. Begin. 2006. Expand, enlarge and check: New algorithms for the coverability problem of WSTS. *Journal of Computer and Systems Sciences* 72, 1, 180–203.

G. Geeraerts, J.-F. Raskin, and L. V. Begin. 2007. On the efficient computation of the minimal coverability set for Petri nets. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis (ATVA'07)*. 98–113.

S. M. German and A. P. Sistla. 1992. Reasoning about systems with many processes. *Journal of the ACM* 39, 3, 675–735.

B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.

A. Gupta, C. Popeea, and A. Rybalchenko. 2011a. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. 331–344.

A. Gupta, C. Popeea, and A. Rybalchenko. 2011b. Threader: A constraint-based verifier for multi-threaded programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. 412–417.

C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems* 5, 4, 596–619.

A. Kaiser, D. Kroening, and T. Wahl. 2010. Dynamic cutoff detection in parameterized concurrent programs. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. 645–659.

R. M. Karp and R. E. Miller. 1969. Parallel program schemata. *Journal of Computer and System Sciences* 3, 2, 147–195.

J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. 2013. Incremental, inductive coverability. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. 158–173.

M. Leuschel and H. Lehmann. 2000. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *Computational Logic—CL 2000*. Lecture Notes in Computer Science, Vol. 1861. Springer, 101–115.

S. Lu, S. Park, E. Seo, and Y. Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. 329–339.

R. Meyer and T. Strazny. 2010. Petruchio: From dynamic networks to nets. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. 175–179.

S. S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, NY.

A. Pnueli, J. Xu, and L. D. Zuck. 2002. Liveness with $(0, 1, \infty)$-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*. 107–122.

C. Rackoff. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science* 6, 223–231.

P.-A. Reynier and F. Servais. 2011. Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. In *Applications and Theory of Petri Nets*. Lecture Notes in Computer Science, Vol. 6709. Springer, 69–88.

P. Schnoebelen. 2010. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *Proceedings of the 35th International Conference on Mathematical Foundations of Computer Science (MFCS'10)*. 616–628.

A. Valmari and H. Hansen. 2012. Old and new algorithms for minimal coverability sets. In *Application and Theory of Petri Nets*. Lecture Notes in Computer Science, Vol. 7347. Springer, 208–227.

D. Zufferey, T. Wies, and T. A. Henzinger. 2012. Ideal abstractions for well-structured transition systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*. 445–460.

Received December 2012; revised December 2013; accepted May 2014