Dynamic Cutoff Detection in Parameterized Concurrent Programs*

Alexander Kaiser, Daniel Kroening, and Thomas Wahl

Oxford University Computing Laboratory, United Kingdom

Abstract. We consider the class of finite-state programs executed by an unbounded number of replicated threads communicating via shared variables. The *thread-state reachability* problem for this class is essential in software verification using predicate abstraction. While this problem is decidable via *Petri net coverability* analysis, techniques solely based on coverability suffer from the problem's exponential-space complexity. In this paper, we present an alternative method based on a *thread-state cutoff*: a number n of threads that suffice to generate all reachable thread states. We give a condition, verifiable dynamically during reachability analysis for increasing n, that is sufficient to conclude that n is a cutoff. We then make the method complete, via a coverability query that is of low cost in practice. We demonstrate the efficiency of the approach on Petri net encodings of communication protocols, as well as on non-recursive Boolean programs run by arbitrarily many parallel threads.

1 Introduction

Concurrent software is gaining tremendous importance due to the shift towards multi-core computing architectures. The software is executed by parallel threads, in an asynchronous interleaving fashion. The most prominent and flexible model of communication between the threads is the use of fully shared variables. This model is supported by well-known programming APIs, e.g. the POSIX pthread model and Windows' WIN32 API. Bugs in programs written for such environments tend to be subtle and hard to detect by means of testing, strongly motivating formal analysis techniques.

In this paper, we consider the case in which no a-priori bound on the number n of concurrent threads is known. This scenario is most relevant in practice; it applies, for example, to a server that spawns additional worker threads in response to a high work load. We focus here on *replicated finite-state* programs: the program itself only allows finitely many configurations, but is executed by an unknown number of threads, thus generating an unbounded state space. An important practical instance of this scenario is given by non-recursive concurrent Boolean programs. Boolean program verification is a bottleneck in the widely-used predicate abstraction-refinement framework.

* This research is supported by the EPSRC project EP/G026254/1.

We tackle in this paper the *thread-state reachability* problem. A thread state is defined as a valuation of the shared program variables, plus the local state of one thread. Thread-state reachability is routinely used to encode multi-index safety properties of systems, such as mutually exclusive access to some resource.

The thread-state reachability problem for replicated finite-state programs is equivalent in complexity to the *coverability problem* for Petri nets. The latter problem is decidable [18], but has an exponential lower space bound [7]. In order to not fall victim to this complexity, the approach presented in this paper takes advantage of widely accepted empirical evidence that often a small number of threads suffice to exhibit all relevant behavior that may lead to a bug. If this number is efficiently computable, the unbounded thread-state reachability problem reduces to a finite-state exploration problem, for which quite efficient engines have recently emerged [3].

To be more precise, for every finite-state program $\mathbb{P}$, there is a number c such that any thread state reachable for *some* number of threads running $\mathbb{P}$ can in fact be reached given c threads. We call such a number a *thread-state cutoff* of $\mathbb{P}$. Previous results on computing cutoffs of a program $\mathbb{P}$ tend to either restrict the communication scheme [12, 17], or yield cutoffs that are polynomial in the number of states of $\mathbb{P}$ [11]. Both types are inapplicable to Boolean program verification, since concurrent programming APIs rely on very liberal shared-variable communication, while a Boolean program $\mathbb{P}$ typically has millions of states, rendering even linear-size cutoffs useless.

In contrast to previous solutions, we give in this paper a condition on a number n whose satisfaction allows us to conclude that n is the cutoff of a program $\mathbb{P}$. To obtain such a condition, we first show that, if n is *not* the cutoff, then there exists a number $n' > n$ and a thread state reachable in the n'-thread system $\mathbb{P}_{n'}$ whose reachability requires a particular conducive constellation of several threads in $\mathbb{P}_n$. If the reachable states in $\mathbb{P}_n$ do not permit such a constellation, then n is indeed the cutoff of $\mathbb{P}$.

We then turn this idea into a complete and *tight* cutoff detection algorithm. Completeness is achieved using backward coverability analysis to rule out the reachability of the thread states identified as candidates for the constellation mentioned above. We argue that these candidate state are *benign*, in that backward coverability analysis on them is efficient and does not defeat the original purpose of avoiding such analysis. Minimality of the cutoff is ensured by applying the cutoff detection method iteratively to the values $n = 1, 2, \ldots$. Since our method uses reachability information, we speak of *dynamic* cutoff detection.

We experimentally investigate the cutoffs of a large number of Petri net and Boolean program examples, modeling concurrent systems of various types. We demonstrate the superiority of our cutoff method over several earlier algorithms based solely on Petri net coverability. Our experiments showcase the method as a very promising algorithmic solution to coverability problems for Petri nets, and as an efficient technique for thread-state reachability analysis in realistic, if non-recursive, Boolean programs run by arbitrarily many threads.

2 Basic Definitions

Let $\mathbb{P}$ be a program that permits only finitely many configurations. In particular, $\mathbb{P}$'s variables are of finite range, and the function call graph, if any, of $\mathbb{P}$ is acyclic. An instance of the class of programs $\mathbb{P}$ is given by non-recursive Boolean programs, which are obtained from `C` programs using predicate abstraction. The use of Boolean programs as abstractions of `C` programs was promoted by the success of the SLAM project [1]. We use concurrent Boolean programs in the experimental evaluation of our approach and refer the reader to [8] for a detailed description.

Program $\mathbb{P}$ gives rise to a family $(M_n)_{n=1}^{\infty}$ of *replicated finite-state system* models as follows. $\mathbb{P}$'s variables are declared to be either *shared* or *local.* A valuation of the shared variables is called a *shared state*, a valuation of the local variables is called a *local state.* M_n is a Kripke structure modeling an n-thread concurrent program. The states of M_n have the form $(s, l_1, \ldots, l_n)$, where s is a shared state and l_i is a local state; we say l_i is the local state *of thread i*. M_n's execution model is that of interleaved asynchrony. That is, the set of transitions of M_n is the set of pairs of the form

$$(\,(s, l_1, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_n) \quad , \quad (s', l_1, \ldots, l_{i-1}, l_i', l_{i+1}, \ldots, l_n)\,) \tag{1}$$

such that $(s, l_i) \rightarrow (s', l_i')$ corresponds to a statement in $\mathbb{P}$. Only one thread, i, can make a step at a time. A step may change the local state of that thread and the shared state; we call thread i *active* in the transition. The pair (s, l_i) is called the *thread state* of thread i in global state $(s, l_1, \ldots, l_n)$; a thread state summarizes the part of the global state that is accessible to a thread. A thread has neither read nor write access to local variables of other threads. Note that if a transition changes the shared state of M_n (i.e., $s \neq s'$), it changes the thread state of *every* thread of M_n. Such transitions capture thread communication.

In order to define the thread-state reachability problem considered in this paper, let T be the (finite) universe of thread states, i.e., pairs of shared and local variable valuations, **irrespective of** n. A state $(h, l_1, \ldots, l_n)$ of M_n *contains* thread state (s, l) if $h = s$ and, for some i, $l_i = l$. Thread state t is *reachable in* M_n if there exists a reachable global state of M_n that contains t; reachability of global states in M_n is defined with respect to some set of initial states as usual. We denote the set of thread states reachable in M_n by R_n, and the set $\cup_{n=1}^{\infty} R_n$ of thread states reachable for *some* number of threads by R. Note that, for any n, $R_n \subseteq R \subseteq T$; in particular, these reachability sets are finite. The *thread-state reachability problem* is now defined as follows: given $\mathbb{P}$, determine R.

Our model of replicated finite-state system families $(M_n)_{n=1}^{\infty}$ formalizes classical *parameterized* systems, where the number of running threads is fixed up-front but unknown. Our techniques apply equally to systems where the number of threads can change at runtime. It is quite easy to show that the two models are equivalent for reachability properties. Further, our techniques extend to the case of multiple program templates, as in a heterogeneous synchronization problem with arbitrarily many *readers* and *writers.* For simplicity, we focus in the rest of this paper on the single-template parameterized case formalized above.

3 Background: Decidability of Thread-State Reachability

The thread-state reachability problem as defined in the previous section is decidable, via a reduction to the *coverability problem* for *vector addition systems with states* (VASS), as follows. A VASS is a finite-state machine whose edges are labelled with integer vectors of some fixed dimension. A *configuration* of a VASS is a pair (q, x) where q is a state and x is a vector of **non-negative** integers. There is a transition $(q, x) \to (q', x')$ if there is an edge $q \xrightarrow{v} q'$ in the VASS such that $x' = x + v$; symbol $+$ denotes pointwise addition. Given an initial configuration (q_0, x_0), a configuration (q, x) is *reachable* if there exists a sequence of transitions starting at (q_0, x_0) and ending at (q, x). The *coverability problem* asks whether a given configuration (q, x) is *covered* by the reachable configurations of the VASS, i.e., whether a configuration (q, x') is reachable such that $x' \geq x$, where $\geq$ is defined pointwise.

Theorem 1 ([18]) *The coverability problem for VASS is decidable.*

The decision procedure by Karp and Miller [18] builds a rooted tree that represents the set of covered configurations of a vector addition system. Unfortunately, it operates not even in primitive-recursive space. In response to this daunting complexity, alternative algorithms exploring *well-structured transition systems* (WSTS), of which VASS are an example, have been developed [13, 15]. Their efficiency is handicapped by the EXPSPACE lower bound of the coverability problem, the proof of which is attributed to Cardoza, Lipton and Meyer [7].

Replicated finite-state systems as vector addition systems. Using the components of a vector to count the number of threads in each of the possible local states, a VASS can simulate a replicated finite-state system: a thread transition $(s, l) \to (s', l')$ is represented by a VASS edge $s \xrightarrow{v} s'$ such that the l-th component of v is -1, the l'-th component is 1, and all others are 0. A thread state (s, l) of the program is reachable in the program's concurrent execution exactly if there is a reachable VASS configuration (s, x) such that the l-th component of x is at least 1. By definition, this is the case exactly if the VASS configuration (s, x_0) is *covered*, where x_0 is all-zero except the entry at position l, which is 1. The latter problem is decidable by Theorem 1. We obtain:

Corollary 2 *The thread-state reachability problem for replicated finite-state programs is decidable.*

It can be shown that the VASS coverability problem is, conversely, reducible to the thread-state reachability problem, in a way that makes the thread state reachability problem EXPSPACE complete as well. We remark that all reduction results sketched in this section hold equivalently for Petri nets in place of vector addition systems. Since the former are of a somewhat greater practical appeal, we will use Petri nets and their tools as a reference point in the experimental Section 6 later in this paper.

4 Thread-State Reachability via Cutoffs

Our computational model, according to which the possible transitions of a thread are determined only by its local state and the shared state, guarantees the following monotonicity property:

Property 3 *Sequence $(R_n)_{n=1}^{\infty}$ is monotone in n: $n_1 \le n_2$ implies $R_{n_1} \subseteq R_{n_2}$.*

This property holds since every path in M_{n_1} can be extended to a path in M_{n_2} of the same length by adding $n_2 - n_1$ thread components to each state along the path and letting the new threads idle in their initial state.

Sequence $(R_n)_{n=1}^{\infty}$ thus never decreases. Since, on the other hand, the set R of reachable thread states is finite and $R_n \subseteq R$ for every n, the sequence can increase only a finite number of times. This implies that, for every finite-state program $\mathbb{P}$, there is a number c such that any reachable thread state can in fact be reached given c threads. Such a number is called a (thread-state) cutoff.

Definition 4 *A **thread-state cutoff** (or **cutoff** for short) for family $(M_n)_{n=1}^{\infty}$ is a number $c \in \mathbb{N}$ such that, for all $n \ge c$, $R_n = R_c$.*

In particular, we have $R_c = R$. Knowing the cutoff would therefore allow us to compute the set of reachable thread states using an efficient *finite-state model checker*. In order to turn this possibility into a viable alternative to coverability methods, we not only have to find means of computing the cutoff efficiently. We also need the minimum cutoff c_0 to be small enough that a model checker can compute R_{c_0} with reasonable resources.

The minimum cutoff of a finite-state program can in principle be arbitrarily large: given a number c, consider the following program with a shared variable $s \in \{0, \ldots, c\}$, initially 0.

```
0: s := s + 1 (mod c+1)
1: if s = c: error
```

This program has a minimum cutoff of c. There is, however, widely accepted (although, to our knowledge, rarely documented) empirical evidence that, in "typical" parameterized programs, a small number of threads suffice to exhibit all relevant behavior that may lead to a bug. We will be able to gauge the precision of this claim in the experimental Section 6 at the end of the paper. For now, we return to our main objective: determining cutoffs efficiently in practice.

5 Determining Thread-State Cutoffs

Emerson and Kahlon present several results for *statically* obtained cutoffs that are linear in the size of the program template (such as a Kripke model of a Boolean program) [11]. While valuable in establishing the decidability of certain fragments of the parameterized model checking problem, such cutoffs are unlikely to be of practical value in our context, since they are often not *tight* and in fact

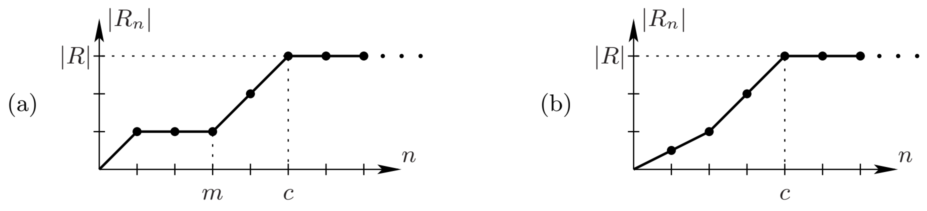


Fig. 1. (a) An intermediate plateau; (b) a strictly monotone thread state sequence

vastly overapproximate the minimum number of threads needed to reach all reachable thread states.

We propose in this paper a *dynamic* method to determine the cutoff. That is, instead of pre-computing the cutoff for the family, we *detect* it during the reachability analysis on the systems M_n, for increasing values of n. Our first contribution will be a condition that, based on certain observations on the reachability result obtained for M_n, allows us to conclude that we do not need to increase n further. Such a method has the potential of finding cutoffs that are orders of magnitude smaller than those computed by the static techniques.

5.1 Thread-State Sequences with Plateaus

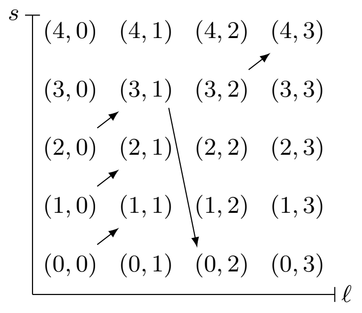
Consider the thread-state sequence $(R_n)_{n=1}^{\infty}$ and a value m at which the sequence *plateaus*, i.e. $R_m = R_{m-1}$. It is tempting to conclude that a cutoff has been found when this happens. This temptation is fallacious, however, as the sequence of reached thread states may resume growth for thread counts exceeding m, even after several steps of plateauing.

Definition 5 *Value m is a* ***plateau endpoint*** *of (R_n) if $R_{m-1} = R_m \subsetneq R_{m+1}$.*

This situation is depicted in Figure 1 (a). The fallacious argument mentioned above would only be valid if every thread-state sequence was *strictly monotone* up to the minimum cutoff c, as shown in Figure 1 (b). A system with an intermediate plateau is induced by the finite-state program given in Figure 2. It can be synthesized into a four-line Boolean program with three shared variables.

Let us investigate the somewhat unintuitive phenomenon of intermediate plateaus more closely. Recall that if a transition changes the shared state of the program, the thread state of *every* thread is affected. As a result, a thread that is not itself active in the transition may reach a new thread state. We say that such a thread state is reached *passively.*

This situation is shown in Figure 3 (a). Thread i is active and changes, in addition to its local state, the shared state from r to s (solid line). As a side effect, thread state (s, h_j) is reached passively (dashed line). Note that the local state of thread j remains at h_j. Figure 3 (b) is a special case of (a) where threads i and j happen to reside in the same local state $h_i = h_j$ before i executes.

(a) 

(b)

n	R_n
1	$\{(0,0),(1,1)\}$
2	$R_1 \cup \{(1,0),(2,1)\}$
3	$R_2 \cup \{(0,1),(0,2),(2,0),(3,1)\}$
4	$R_3 \cup \{(1,2),(3,0)\}$
5	$R_4 \cup \{(2,2)\}$
6	$R_5 \cup \{(3,2),(4,1),(4,3)\}$
7	$R_6 \cup \{(4,0)\}$
8	R_7 $\rightarrow$ **plateau endpoint**
9	$R_8 \cup \{(4,2)\}$ $\rightarrow$ **cutoff**

Fig. 2. (a) A finite-state program over variables (s, ℓ) with initial state $(0,0)$; (b) the thread-state sequence induced by (a), exhibiting a plateau of length 1.

Returning to the issue of intermediate plateaus: one can show that, if m is a plateau endpoint, there exists a thread state in $R_{m+1} \setminus R_m$ that is reached passively. We will see next that in fact a much stronger statement holds.

5.2 A Sufficient Cutoff Condition

Equipped with the considerations from Section 5.1, we can now derive a sufficient cutoff condition for thread-state reachability. Technically, we will establish instead a *necessary* condition for m *not being* a cutoff. The following lemma is the crucial insight.

Lemma 6 *Suppose m is **not** a cutoff for family $(M_n)_{n=1}^{\infty}$. Let $m' = \min\{n : R_n \supsetneq R_m\}$, and let t be a thread state in $R_{m'} \setminus R_m$ with minimum distance from the initial state set. Then t is reached **passively**.*

Proof. Let i be the thread active during the global transition of $M_{m'}$ when t is first reached. We have to show that t is *not* reached by thread i.

To this end, let $t_1 \rightarrow t_2$ be the thread transition executed by thread i that causes t to be reached by some thread; we prove $t \neq t_2$. Transition $t_1 \rightarrow t_2$ happens in $M_{m'}$, so $t_1 \in R_{m'}$. Since t_1 has shorter distance to the initial state

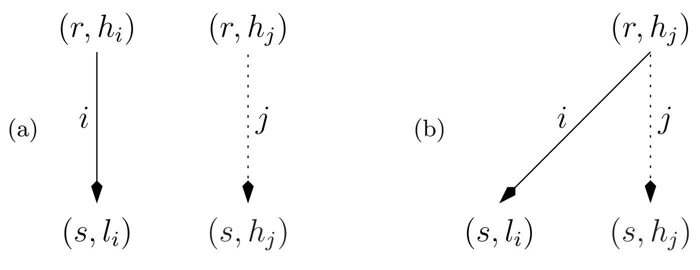


Fig. 3. (a) The general and (b) a special case of reaching thread state (s, h_j) passively

set than t_2 and thus than t, we conclude $t_1 \notin R_{m'} \setminus R_m$, thus $t_1 \in R_m$. This in turn implies $t_2 \in R_m$, since the set R_m is closed under thread transitions: any path in M_m to a state containing t_1 can be extended, via $t_1 \rightarrow t_2$, to a path in M_m to a state containing t_2. Since $t \notin R_m$, it follows $t_2 \neq t$. $\square$

We can exploit this lemma as follows to derive a necessary is-not-the-cutoff condition. If m is not the cutoff, then the first new thread state encountered for $n > m$ is in fact reached passively. The ability to reach a thread state passively requires a constellation of reachable thread states as shown in Figure 3 (a), where the new thread state is denoted (s, h_j). We now observe that the thread states (r, h_i), (r, h_j) and (s, l_i) mentioned in the figure are all members of the current reachability set R_m. To see this, note that (r, h_i) and (r, h_j) are reached *before* (s, h_j). Since (s, h_j) has minimum distance, among all new thread states, we conclude that (r, h_i) and (r, h_j) are not new and are thus elements of R_m. Thread state (s, l_i) is an element of R_m since it is a direct successor of (r, h_i). We summarize: if m is not the cutoff, there exist three thread states (r, h_i), (r, h_j) and (s, l_i) in R_m such that

- $(r, h_i) \rightarrow (s, l_i)$ is a valid thread transition according to $\mathbb{P}$, and (2)
- $(s, h_j) \notin R_m$. (3)

We call thread states (r, h_i), (r, h_j) and (s, l_i) in R_m with these properties a *candidate triple*. If no candidate triple can be found, no thread state can possibly be reached passively in the future. Together with Lemma 6, we obtain:

Corollary 7 *Suppose no candidate triple exists in R_m. Then m is a cutoff for family $(M_n)_{n=1}^{\infty}$.*

We refer to the check of absence of candidates as the *cutoff test*. Unlike Lemma 6, the test conditions depend only on the program $\mathbb{P}$ and on M_m. The downside is that the cutoff test is incomplete for cutoff detection. To see this, consider the finite-state program over the state space $\{0, 1, 2\} \times \{0, 1\}$ with initial state $(0, 0)$ and the two transitions

$$(0,0) \rightarrow (1,1) \qquad \text{and} \qquad (1,1) \rightarrow (2,0)\,.$$

This program induces a parameterized family $(M_n)_{n=1}^{\infty}$ where the cutoff test fails for every n: the candidate triple $(1, 1)$, $(1, 1)$, $(2, 0)$ never vanishes. (The triple happens to be of the special form of Figure 3 (b).) We will fix this problem in the following section.

5.3 Sound, Complete and Tight Cutoff Detection

The cutoff test ignores that, in order to give rise to the new thread state (s, h_j), the candidate triple must be *realizable*: there must exist an n and a global state reachable in M_n that contains **both** (r, h_i) and (r, h_j). In the example in Section 5.2, no two threads can simultaneously enter a state of the form $(-, 1)$. It turns out that realizability of candidate triples precisely characterizes cutoffs:

Theorem 8 *Thread count m is a cutoff for family $(M_n)_{n=1}^{\infty}$* ***exactly if*** *R_m contains no realizable candidate triples.*

Proof. (i) If m is a cutoff and the candidate thread states (r, h_i) and (r, h_j) are, for some $n \geq m$, simultaneously reachable, then thread state (s, l_j) becomes reachable when M_n is analyzed. Since $(s, l_j) \notin R_m$ by equation (3), (s, l_j) is new, contradicting the stipulation that m is a cutoff.

(ii) If m is not a cutoff, then, by Lemma 6, there exists a passive thread transition that reaches a thread state unreachable in M_m. As shown in the proof of that lemma, the three thread states participating in the reaching of the new thread state all belong to R_m and thus form a candidate triple. For the passive transition to actually happen (the lemma proves that it does), the thread states (r, h_i) and (r, h_j) must be simultaneously reachable. So there exists at least one realizable candidate triple. $\square$

Simultaneous reachability of (r, h_i) and (r, h_j) in the family $(M_n)_{n=1}^{\infty}$ cannot be checked by looking only at R_m. We will use *backward coverability* analysis for this step. The candidates represent a *minimal* set of thread states whose unrealizability guarantees the cutoff property. This minimality gives rise to the hope that candidates can be reachability-checked more efficiently than arbitrary thread states. We measure the cost of this check in detail in Section 6, using the Mist tool set [13] as the coverability engine.

Putting the cutoff test and this analysis together, we obtain Algorithm 1 for cutoff detection. The algorithm maintains the invariant that, at entry to the loop in Line 2, the reachability set R_n is guaranteed to have been computed, for the current value of n. In Line 2, the algorithm starts two computational threads in parallel. The first, **A**, computes the candidate triples for R_n. If any of them is realizable, which is checked using backward coverability analysis, we know by Theorem 8 that n is not a cutoff. The thread aborts, and control proceeds to Line 3. If no triple is realizable (or there are no candidates), we return that n is the cutoff; this terminates the algorithm.

The second thread, **B**, computes the next reachability set R_{n+1}, using a finite-state forward search. This is done in parallel with the candidate check since, as soon as we know that $R_{n+1} \supsetneq R_n$, we can abort the candidate check in thread **A**: we know that n is not the cutoff. If $R_{n+1} = R_n$, thread **B** terminates normally.

In Line 3, the main thread synchronizes the computation by waiting for the termination (or abortion) of **A** and **B**. This is crucial since the set R_{n+1} needs to be available in the next round. We then increase n and re-enter the loop. Note that if the backward analysis in round n reveals that some triple $\mathcal{T}$ is realizable, we do not know for which value $n' > n$ this will happen. As a result, intermediate plateaus of the sequence $(R_n)_{n=1}^{\infty}$ cannot be short-circuited.

Correctness. Termination of the algorithm follows immediately from Theorem 8: Suppose n is the cutoff. Then any candidate triples in R_n are not realizable, so thread **A** returns "cutoff n". Note that $R_{n+1} = R_n$, so thread **B** does not

Algorithm 1 Cutoff detection

```
Input:  system family $(M_n)_{n=1}^{\infty}$
1: $n := 1$; compute $R_1$ // finite-state
2:   A: | compute set $C_n$ of cand. triples        ||   B: | compute $R_{n+1}$ // finite-state
        | if $\exists\,\mathcal{T} \in C_n$: $\mathcal{T}$ realizable ||      | if $R_{n+1} \supsetneq R_n$
        |     abort A                               ||      |     abort A
        | return "cutoff $n$"                       ||
3: sync(A,B)
4: $n := n + 1$; goto 2
```

abort **A**. Theorem 8 similarly guarantees partial correctness. The combination of termination and partial correctness guarantees that Algorithm 1 returns in fact the *minimum* cutoff c_0: it does not terminate for $n < c_0$, by the partial correctness. It never reaches $n > c_0$, since it terminates for $n = c_0$.

Implementation. We illustrate how to compute candidate triples (first step of thread **A** in Algorithm 1). First note that conditions (2) and (3) on the candidates (r, h_i), (r, h_j) and (s, l_i) imply all of the following:

- $r \neq s$ (since $(r, h_j) \in R_m$, but $(s, h_j) \notin R_m$)
- $l_i \neq h_j$ (since $(s, l_i) \in R_m$, but $(s, h_j) \notin R_m$)
- (r, h_i), (r, h_j) are not simultaneously reachable in M_m (since otherwise $(s, h_j) \in R_m$, passively)

To compute the candidates, we iterate over pairs of thread states (r, h_i), (r, h_j) in R_m that are not simultaneously reachable in M_m (this information is taken from the reachable global states set of M_m), and select successor thread state (s, l_i) by consulting the program text under the additional constraints that $r \neq s$ and $l_i \neq h_j$. The remaining condition $(s, h_j) \notin R_m$ can be tested efficiently, say by storing R_m in a sorted container or a hash table.

6 Experimental Evaluation

We implemented two variants of Algorithm 1. The first is our Petri net coverability checker, ECUT, which we tested on 23 Petri nets examples from diverse programming domains. The second is our symbolic thread-state reachability checker for Boolean programs, SCUT, which we tested on 852 Boolean programs, generated from Linux device driver code. The Petri nets induce relatively small state spaces, but exhibit challenging concurrent behavior. In contrast, the Boolean programs induce huge state spaces, but exhibit rather simple concurrency. All experiments were performed on a 16GB/3GHz Intel Xeon machine running the 64-bit variant of Linux 2.6 with a 45min timeout.

Benchmark	S	L	T	$\sum fw$	$\sum bw$	eCUT	c	$\lvert R_c\rvert$	$\lvert C_c\rvert$	Result
Readwrite	24	14	33	0.01	0.2	0.2	9	198	29	safe
Mesh2x2	35	33	71	0.6	0.01	0.8	9	844	40	safe
Multip.	20	19	45	0.1	0.8	0.9	8	257	10	safe
Pncsa	37	32	73	1.4	0.1	1.5	7	860	122	unsafe
Fms	26	23	49	0.6	1.2	1.6	12	361	5	safe
Bh250	507	254	1,009	0.6	6.0	6.7	3	1,768	31,875	safe
Mesh3x2	55	53	115	277.4	1.2	278.6	13	2,228	67	safe
Kanban	29	17	49	–	–	–	–	–	–	mem-out

Table 1. Results of eCUT on Petri net benchmarks. S, L, T: # shared states, local states, thread transitions; $\sum fw$, $\sum bw$, eCUT: time for forward searches, backward searches, total eCUT runtime in seconds; c: cutoff (if unsafe: #threads until error); $|R_c|/|C_c|$: # reachable thread states/# candidate triples at bound c.

6.1 Petri net coverability

Our coverability checker eCUT forward-computes an explicit-state representation of the sets R_n, and uses the backward search engine of the Mist toolset to check candidates for reachability. We evaluate eCUT using 5 bounded and 18 unbounded Petri nets, ranging from concurrent production systems and communication protocols to broadcast protocols. Each net is transformed into a replicated finite-state system. Transitions are split into sequences of thread transitions using fresh intermediate shared states. Given p places and t transitions, this required $p + 1$ local states, $1.2t$ shared states and $2.2t$ thread transitions on average. The original coverability property translates into the reachability of a suitable thread state. All examples and correctness properties are from [13] and [4].

Within 5min or much less, eCUT succeeds on 22 examples (21 safe, 1 unsafe), and memory-outs on 1. Table 1 shows details of the analysis; we omit instances with runtimes below 0.2s and only show the most challenging from [4], namely Bh250. The *Kanban* example has a cutoff beyond 20; our implementation reaches the memory limit after 10min and more than $6 \cdot 10^7$ explored states in round $n = 15$. In these examples, neither the finite-state forward nor the backward search dominate the overall runtime, advocating the use of a combination of both.

Comparison with other algorithms. We compare our implementation with four algorithms implemented in the Petri net coverability tool set Mist [13]: a pure backward search (BW) (the same we use to check candidates), and three abstraction refinement schemes. The latter combine infinite-state forward and backward search, using abstractions that minimize the number of predicates used to encode places (TSI, EEC) or the dimensionality of the Petri net (IC4PN).

Figure 4 shows the number of instances the different algorithms can solve within 45min/16GB: ECUT performs best, solving 22 instances, followed by EEC (20), BW (17), TSI (15) and IC4PN (12). Besides solving most instances, ECUT does so fastest in most cases (one exception is *Kanban*, which only BW and TSI can handle), proving it generally more robust than the other tools.

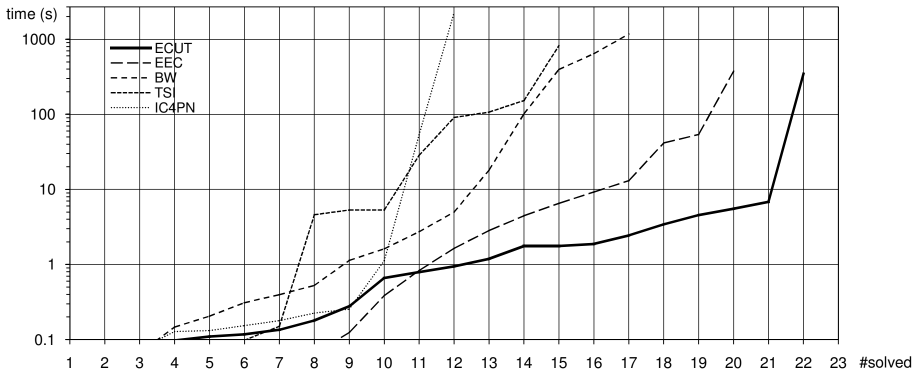


Fig. 4. Comparison of ECUT and the algorithms EEC, BW, TSI and IC4PN for the 23 Petri net benchmarks. Entry (n, t), for a number n and a time t, indicates that it took time t to solve the n easiest instances for the algorithm indicated by the curve.

6.2 Boolean Program Reachability

Our reachability checker SCUT computes the sets R_n using the symbolic model checker BOOM [3]. Since there is no symbolic backward search engine able to handle Boolean programs of the size we consider, we simplify Algorithm 1: thread **A** merely checks whether there are any candidates; if they don't (seem to) vanish for any n, we consider the run a timeout.

We evaluate our implementation of SCUT using 852 Boolean programs. The programs stem from Linux device driver code and were embedded into a concurrent test environment using the DDVERIFY tool [19]. The programs feature on average about 1000 program locations and 9 shared and 18 local variables. We are not aware of any other tool that can perform even finite-state reachability of concurrent Boolean programs of this size. We therefore only present results obtained with our tool.

Table 2 shows analysis results grouped by their cutoff. SCUT succeeds in 798 of the cases (94%); the remaining 54 examples time out (6%). In all safe examples, the cutoff test alone is sufficient: all candidates disappear eventually. We see that for a vast majority of the examples, the cutoffs are indeed very small and easily within the performance limits of BOOM.

#P	*Sh*	*Lcs*	*Loc*	sCUT	*c*	Safe	Unsafe	t/o
773	17	8	1170	0.1	1	407	366	0
17	21	22	1139	0.8	2	3	14	0
8	13	26	1131	72.3	3	8	0	0
54	18	31	1267	874.0	?	–	–	54

Table 2. Results of sCUT for the Boolean program benchmarks, grouped by cutoffs. #P: # of programs in group; *Sh*, *Lcs*, *Loc*: avg. # shared variables, local variables, program locations; sCUT: avg. sCUT runtime; *c*: cutoff/#threads until error (? = unknown); t/o: # timeouts.

7 Related Work

There is a vast amount of literature on tackling reachability analysis for concurrent software, with or without recursion. We focus on work related to the use of cutoffs, and work related to Petri nets. We believe our work to be the first to combine finite-state forward search, cutoff detection and infinite-state backward coverability analysis in a symbiotic manner.

Cutoffs: Much of the work on verifying concurrent programs using cutoffs restricts communication [6, 12]. For example, small constant-size cutoffs are known for ring networks communicating only by token passing [12], and for multi-threaded programs communicating only using locks [17]. These results fail to hold, however, with general shared-variable concurrency, as we consider it. On the other hand, [11] permits communication via guards over shared local variables, but gives rise to cutoffs that are linear in the number of states of the program $\mathbb{P}$ being replicated. Such cutoffs are unacceptable for us, as $\mathbb{P}$ may have millions of states.

Bingham presents a technique for coverability that seems closest to our work [4, 5]. Standard finite-state BDD techniques are used to compute, for an instance of size n and in a backward fashion, the set of states that have a path to some set U of "bad" states. If the initial state set is intersected, we have encountered an error. Otherwise, n is increased until some convergence criterion is met. Unfortunately, the method is applied to only one (parametric) Petri net. Also, Bingham does not disclose the experimental values of n at which his algorithm terminates, which might give a clue as to the general scalability of the approach — we have found the cutoff of Bingham's Petri net Bh250 to be very small (see Table 1).

Petri nets: Many data structures and algorithms have been proposed for their efficient analysis and coverability checking [15, 10]. Most of these algorithms suffer, however, from an intractable number of vector elements after the translation from (Boolean) programs: one per local program state. Recent work by Raskin et al. has attempted to address the dimensionality problem using an abstraction refinement loop [14], where abstract models of the Petri net under investigation are of lower dimension than the original.

Tools: There are several tools available for the analysis of Petri nets [16]. The Mist tool set [13] implements the *Expand, Enlarge and Check* algorithms due to Geeraerts et al. [15]. Furthermore, Petri net/VASS analysis has been applied to Java programs [9] and Boolean programs [2]. These tools compile their input into an explicit-state representation of the underlying program, which may result in a net with a high number of places. Our experiments indicate that, for the case of Boolean program verification, a symbolic representation is essential.

8 Conclusion

We set out to solve the thread-state reachability problem for replicated finite-state programs efficiently. Our proposal is to exploit the (guaranteed) existence of *thread-state cutoffs*, by analyzing the programs for increasing numbers of thread counts. We have presented a sufficient (but not necessary) condition under which the current thread count is a cutoff, so that no larger thread counts need to be considered. We have shown how to make the algorithm complete, using a backward coverability analysis to rule out the reachability of certain *candidate thread states* that were identified to potentially lead to new thread states. The algorithm returns the set of reachable thread state and the minimum cutoff of the given parameterized family.

We have empirically demonstrated, on a large selection of benchmarks, that cutoffs tend to be small enough in practice to allow our incremental technique to beat various methods based solely on coverability algorithms. Our technique is useful both for general Petri net coverability analysis, and specifically for thread-state reachability analysis in non-recursive Boolean programs run by arbitrarily many threads.

Our method can be seen as an opportunity to shift the burden in solving the parameterized verification problem from heavy-weight unbounded tools to lighter-weight *bounded* concurrency model checkers. This is of utmost value, since efficient bounded tools have recently become available, such as Boom, that can solve reachability queries for non-trivial thread counts.

Future work includes the application of our method to extended types of Petri nets, such as transfer nets, which allow richer inter-thread communication, such as broadcasts (an example is S. German's protocol used in [5]).

References

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85, 2006.
2. T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, pages 158–173, 2001.

3. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, pages 64–78, 2009.
4. J. D. Bingham. A new approach to upward-closed set backward reachability analysis. *Electr. Notes Theor. Comput. Sci.*, 138(3):37–48, 2005.
5. J. D. Bingham and A. J. Hu. Empirically efficient verification for a class of infinite-state systems. In *TACAS*, pages 77–92, 2005.
6. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, pages 473–487, 2005.
7. E. Cardoza, R. J. Lipton, and A. R. Meyer. Exponential space complete problems for Petri nets and commutative semigroups: Preliminary report. In *STOC*, pages 50–54, 1976.
8. B. Cook, D. Kroening, and N. Sharygina. Verification of Boolean programs with unbounded thread creation. *Theor. Comput. Sci.*, 388(1-3):227–242, 2007.
9. G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, pages 173–187, 2002.
10. G. Delzanno, J.-F. Raskin, and L. V. Begin. Covering sharing trees: a compact data structure for parameterized verification. *STTT*, 5(2-3):268–297, 2004.
11. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, pages 236–254, 2000.
12. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.
13. P. Ganty, L. V. Begin, G. Delzanno, and J.-F. Raskin. *The MIST2 tool, release 1.0, June 2009.* Université Libre de Bruxelles, `http://www.ulb.ac.be/di/ssd/pganty/software/software.html`.
14. P. Ganty, J.-F. Raskin, and L. V. Begin. From many places to few: Automatic abstraction refinement for Petri nets. *Fundam. Inform.*, 88(3):275–305, 2008.
15. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check… made efficient. In *CAV*, pages 394–407, 2005.
16. F. Heitmann and D. Moldt. Petri net tool database. Available from `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.
17. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.
18. R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
19. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *ASE*, pages 501–504, 2007.