Dynamic Cutoff Detection in Parameterized Concurrent Programs*

Alexander Kaiser, Daniel Kroening, and Thomas Wahl

Oxford University Computing Laboratory, United Kingdom

Abstract. The verification problem for parameterized concurrent programs is a grand challenge in computing. We consider the class of finite-state programs executed by an unbounded number of replicated threads, which is essential in concurrent software verification using predicate abstraction. While the reachability problem for this class is decidable, existing algorithms are of limited use in practice, due to an exponential-space lower bound. In this paper, we present an alternative method based on a *reachability cutoff*: a number n of threads that suffice to generate all reachable program locations. We give a sufficient condition, verifiable dynamically during the reachability analysis, that allows us to conclude that n is a cutoff. We then make the method complete, using a lean backward coverability analysis. 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., by the POSIX pthread model and Windows' WIN32 API. Bugs in programs written for such an environment 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 the 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 comprises the local state of one thread, plus a valuation of the shared program variables. Thread-state reachability is routinely used to encode multi-index safety properties of systems, such as mutually exclusive resource access.

Thread-state reachability for replicated finite-state programs can be reduced to the *coverability problem* for Petri nets, which in turn is known to be decidable [19, which equivalently considers vector addition systems]. In principle, this reduction can be used as an algorithm to decide the thread-state reachability problem. In practice, however, such algorithms and tools suffer from the *dimensionality problem* of Petri nets, and from the high exponential-space complexity of coverability procedures [21].

In this paper, we consider an alternative approach to thread-state reachability: For every finite-state program $\mathbb{P}$, there is a number c such that any thread state reachable given *some* (arbitrarily large) number of threads running $\mathbb{P}$ can in fact be reached given c threads. We call such a number a *reachability cutoff* of $\mathbb{P}$. Previous results on computing cutoffs of a program $\mathbb{P}$ can roughly be classified into two types. [10, 18] report very small, constant cutoffs, but restrict the communication scheme, to cyclic token-passing, or to locks, resp. On the other hand, [9] permits communication via guards over shared local variables, but gives rise to cutoffs that are linear in the number of states of $\mathbb{P}$. Both types are inapplicable to Boolean program verification, since concurrent programming APIs typically rely on most liberal shared-variable communication, *and* the Boolean program $\mathbb{P}$ typically has millions of states, rendering linear-size cutoffs useless.

In contrast to previous solutions, we give in this paper a criterion that allows us to *decide* whether a given value m is the cutoff of a program $\mathbb{P}$. To obtain such a criterion, we first show that, if m is *not* the cutoff, then there exists a number $m' > m$ and a thread state reachable in the m'-thread system $\mathbb{P}_{m'}$ whose reachability requires a particular conducive constellation of several threads in $\mathbb{P}_m$. If the set of states reachable in system $\mathbb{P}_m$ does not permit such a constellation, then m 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 (generally few) thread states identified as candidates for a constellation as mentioned above. Tightness 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.

The efficiency of our cutoff detection method relies on the observation that the cutoff tends to be reasonably small in realistic concurrency examples. We verify this claim on a large number of Petri nets and parallel Boolean programs, modeling concurrent systems of various types. We also 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 non-recursive but otherwise realistic 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 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 [6] for a detailed description.

To make $\mathbb{P}$ amenable to parallel execution, its variables are declared to be either *shared* or *local*. When $\mathbb{P}$ is executed by several threads, there is one copy of all shared variables of $\mathbb{P}$, and one copy *per thread* of all local variables of $\mathbb{P}$. A *shared state* is then given by a valuation of the shared variables, a *local state* by a valuation of (one copy of) the local variables. A *thread state* is a pair (s, l) where s is a shared state and l is a local state, summarizing the information accessible to a single thread. A thread has neither read nor write access to local variables of other threads.

Formally, program $\mathbb{P}$ gives rise to a family $(M_n)_{n=1}^{\infty}$ of *replicated finite-state systems*, as follows. 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 the local state of thread i. M_n's execution model is asynchronous. That is, every transition of M_n has the form

$$(s, l_1, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_n) \rightarrow (s', l_1, \ldots, l_{i-1}, l'_i, l_{i+1}, \ldots, l_n) . \qquad (1)$$

Only the local state of the *active* thread i and the shared state may change. Transition (1) is summarized by the *thread transition* $(s, l_i) \rightarrow (s', l'_i)$, which amounts to a change of the shared and local variables according to program $\mathbb{P}$. Note that if a transition changes the shared state of M_n, 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) set 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 *any* thread count 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) 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, as each can simulate the other. In the rest of this paper, we therefore focus on the 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. 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 $(q, x) \rightarrow (q', x')$ if there is an edge $q \xrightarrow{v} q'$ in the VASS such that $x' = x + v$. 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 VASS, i.e., whether a configuration (q, x') is reachable such that $x' \geq x$, where $\geq$ is defined pointwise.

Theorem 1 ([19]) *The coverability problem for VASS is decidable.*

The algorithm by Karp and Miller [19] builds a rooted tree that compactly represents the set of covered configurations of a vector addition system. The construction can be done in a forward or backward fashion [12].

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 $(s, l) \rightarrow (s', l')$ 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 that position l equals 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.*

We remark that these reduction results 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.

Converting a replicated finite-state program into a vector addition system tends to cause a blow-up in the vector dimension. On top of this blow-up is the daunting complexity of the Karp-Miller algorithm, which was shown by Rackoff to operate in EXPSPACE [21].

4 Thread-State Reachability via Cutoffs

Our computational model guarantees the following monotonicity property:

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

This property holds since every path in M_{n_1} can be "widened" 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 cutoff.

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

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. Reachability analysis via cutoffs is practicable only if the minimum cutoff c_0 is small enough that a model checker can compute R_{c_0} quickly.

Unfortunately, the minimum cutoff of a finite-state program can be arbitrarily large. Consider the following program with a shared variable $s \in \{0, \ldots, c\}$:

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

This program has a minimum cutoff of c. The good news is that, by widely accepted empirical evidence [20, for example], 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 Reachability 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) [9]. While valuable in establishing the decidability of certain instances 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 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, rather than pre-computing it for the family, we *detect* it during the reachability

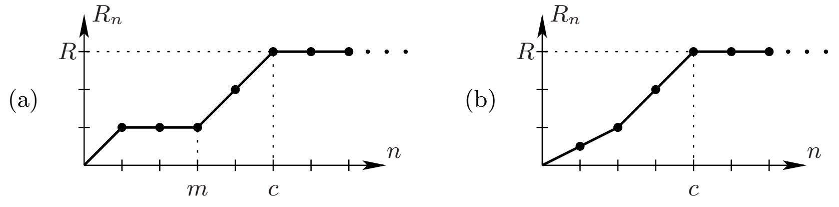


Fig. 1. (a) An intermediate plateau and (b) strictly monotone thread state sequence

analysis on the systems M_n, for increasing values of n. Our first contribution is a criterion 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 mentioned earlier.

5.1 Thread-State Sequences with Plateaus

A simple observation one can make about the thread-state sequence $(R_n)_{n=1}^{\infty}$ is that, for some value m, the set R_m has not increased in the last round, i.e., $R_{m-1} = R_m$. 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*.

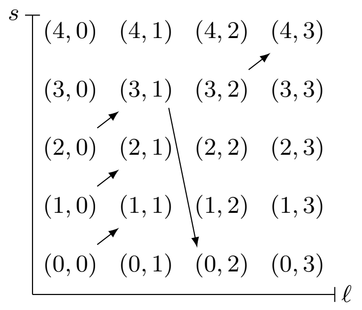
Definition 5 *Value m is a* ***plateau endpoint*** *of (R_n) if $R_{m-1} = R_m \subsetneq R_{m+1}$.*

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 cutoff c), as indicated in figure 1 (b). A system with an intermediate plateau is given in figure 2 as a thread transition diagram. It can be written as a Boolean program with three shared variables and four lines of code.

One possible strategy for detecting a cutoff is now to determine whether a plateau at m, i.e., we have $R_{m-1} = R_m$, is intermediate or infinite. To this end, we investigate the cause of intermediate plateauing. 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. As a result, thread state (s, h_j) is reached — passively. 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_j before i executes.

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.

(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	$R_7 \quad \rightarrow$ **plateau endpoint**
9	$R_8 \cup \{(4,2)\} \quad \rightarrow$ **cutoff**

Fig. 2. (a) A program $\mathbb{P}$ with initial state $(0,0)$ as a thread transition diagram; (b) the thread-state sequence induced by (a), with a plateau of length 1.

5.2 A Sufficient Cutoff Criterion

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

Lemma 6 *Suppose m is not a cutoff for family (M_n). Then any **first** thread state reached in any system $M_{>m}$ is reached **passively**.*

We clarify the terminology used in the lemma. A "first thread state t reached in any system $M_{>m}$" is defined as follows. Since m is not the cutoff, let $m' > m$ be minimum such that $R_{m'} \supsetneq R_m$. Then t is any thread state in $R_{m'} \setminus R_m$ with *minimum distance* from the initial state set. Intuitively, t is the first new thread state encountered during breadth-first search of the systems $M_{m+1}, M_{m+2}, \ldots$. The lemma says that t is reached by a non-active thread.

Proof of lemma 6. Let m' and t be as above, and let i be the thread active during the global transition of $M_{m'}$ when t is first reached. Suppose t is reached actively, i.e., thread i moves into thread state t, say via thread transition $s \rightarrow t$.

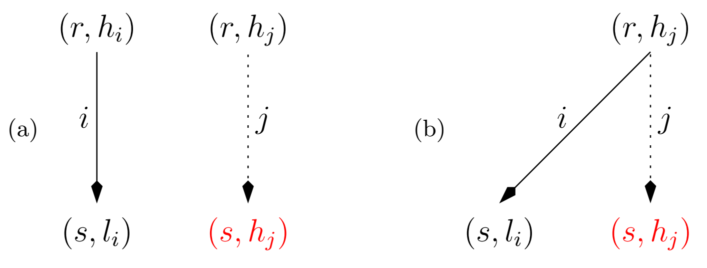


Fig. 3. (a) The general and (b) a special case how to reach a thread state (red) passively

Thread state s belongs to $R_{m'}$ as well. Since s has shorter distance to the initial state set than t, we conclude $s \notin R_{m'} \setminus R_m$, thus $s \in R_{m'} \setminus (R_{m'} \setminus R_m) = R_m$. This, however, is a contradiction: if s belongs to R_m, so does t: any path in M_m to a state containing s can be extended, via the thread transition $s \to t$, to a path in M_m to a state containing t. As a consequence, t is reached passively. □

We can exploit this lemma as follows to derive a necessary is-not-the-cutoff condition. If m is not the cutoff, i.e., there exists a thread state that becomes reachable only for some value $m' > m$, then the "first" such thread state 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) is the *first* new thread state, 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 that, if m is not the cutoff, then there exist three thread states (r, h_i), (r, h_j) and (s, l_i) in R_m such that

- $(r, h_i) \to (s, l_i)$ is a valid thread transition, 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 *candidate triples*. 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).*

We refer to the check of existence of candidates as the *cutoff test*. The test conditions are "local": they depend only on the program $\mathbb{P}$ and on system M_m. We will use this test in section 5.3 in a cutoff detection algorithm. The test can be implemented efficiently by selecting the candidates (r, h_i), (r, h_j) and (s, l_i) judiciously: It is easy to see that conditions (2) and (3) imply all of the following:

- $r \neq s$
- $l_i \neq h_j$
- (r, h_i), (r, h_j) are not *simultaneously* reachable in M_m.

"Simultaneously reachable" means "reachable in the same global state". To implement the test, we iterate over pairs of thread states (r, h_i), (r, h_j) in R_m that are not simultaneously reachable, and select successor thread state (s, l_i) by consulting the program text under the additional constraint that $s \neq r$ and $l_i \neq h_j$. What remains is to test the condition $(s, h_j) \notin R_m$, which can be done in $\mathcal{O}(\log |R_m|)$ time using binary search.

5.3 Sound, Complete and Tight Cutoff Detection

The cutoff test is not complete for cutoff detection because it ignores that the candidates for engendering a new thread state passively may become simultaneously reachable in the future. More precisely:

Observation 8 *A candidate triple found during a cutoff test at point m engenders a new thread state* ***exactly if****, for some $m' > m$, the candidates are simultaneously reachable in system $M_{m'}$, which is* ***exactly if*** *they are simultaneously reachable (in the family (M_n)).*

The test for simultaneous reachability in the family (M_n) can be efficiently performed using backward coverability analysis, as implemented for instance in the Mist tool set [13]. Putting the cutoff test and this analysis together, we obtain algorithm 1 for cutoff detection.

Algorithm 1 Cutoff detection

```
Input:  system family $(M_n)_{n=1}^{\infty}$
1: $n := 1$
2: compute $R_n$ // using finite-state model checker
3: for each candidate triple $\mathcal{T}$ do // triples computed using cutoff test
4:     if candidates in $\mathcal{T}$ are simultaneously reachable then
5:         $n := n + 1$; goto 2
6: return  "cutoff $n$"
```

Correctness. The algorithm terminates, which is seen as follows. Let n be a cutoff (existence is guaranteed). If any candidates exist in R_n, they are, by observation 8, not simultaneously reachable. Thus the condition in line 4 is false for every $\mathcal{T}$, and we terminate in line 6.

The algorithm is also partially correct, as follows. Suppose line 6 returns n. If there are no candidate triples in R_n, then n is a cutoff by corollary 7. If there is at least one triple, line 4 must evaluate to false for all of them, in order to reach line 6. So no candidates are simultaneously reachable. By observation 8, none can engender a new passively reached thread state. By lemma 6, there is no new thread state at all reachable at a later point. Thus n is a cutoff.

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 $n = c_0$.

Efficiency. In practice, the cutoff test may turn out to be complete, in that no candidate triples are found in line 3. In this case, the algorithm terminates immediately. As an optimization, one can use a plateau test as a "look-ahead": in parallel with the computation of the **for** loop in line 3, compute R_{n+1} (which may be needed anyway in round $n + 1$). If $R_{n+1} \supsetneq R_n$, we can terminate that loop and proceed immediately to round $n + 1$.

Petri net	Size			Runtime			State space				Result
	S	L	T	FW	BW	total	c	$\|R_c\|$	$\|\mathcal{T}_c\|$	$\|M_{c+1}\|$	
Mesh2x2	31	33	67	<1s	<1s	<1s	6	820	64	28,354	safe
Bh100	106	104	308	<1s	1s	1s	3	516	10,302	824	safe
Fms	23	23	46	1s	1s	2s	11	349	5	222,842	safe
Mesh3x2	49	53	109	1s	1s	2s	8	2170	113	165,044	safe
Multip.	16	19	41	1s	1s	2s	7	235	10	370,761	safe
Bh150	156	154	458	<1s	2s	2s	3	766	22,952	1224	safe
Bh250	256	254	758	<1s	6s	6s	3	1266	63,252	2024	safe
Pncsa	37	32	73	67s	<1s	68s	9	887	236	15,547,520	unsafe
Kanban	26	17	46	-	-	-	-	-	-	-	m/o

Table 1. Results of our explicit approach (eCUT) for the Petri net benchmarks (timeout 45min, memory limit 8GB). Columns show: shared/local states and transitions (S, L, T); forward-, backward- and total time (FW, BW, total); the minimum cutoff c, number of thread states $|R_c|$ and candidates $|\mathcal{T}_c|$; the number $|M_{c+1}|$ of states for the look-ahead; and the verification result (m/o = memory-out).

6 Experimental Evaluation

We implemented algorithm 1 with the plateau test, in two flavors: (i) eCUT, which features an explicit computation of the sets R_n ("forward search"), and a symbolic backward search [13], and (ii) sCUT, which is purely symbolic; the sets R_n are computed using the model checker Boom [3]. We evaluate the algorithm on two benchmark sets. The first includes 23 Petri nets from various domains, the second 1,087 Boolean programs, generated from `C` code using SatAbs and Slam. 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 3GHz Intel Xeon machine running the 64-bit variant of Linux 2.6.

Petri net benchmarks. The Petri nets we use for evaluation consist of concurrent production systems, communication protocols [13] and broadcast protocols [4]. The number of shared states ranges from 13 to 256, that of local states from 6 to 254, and that of transitions from 18 to 758. We consider the properties as distributed with the benchmarks, expressed as thread-state reachability problems.

Table 1 shows details of the analysis. To keep the representation legible, we omit instances with runtimes below 0.1s. All terminating examples finish in around 1min or much less. The time spent on the cutoff test is negligible and not shown. The *Kanban* example has a cutoff beyond 20; our implementation reaches a memory limit after 236s and more than $2 \cdot 10^7$ explored states in round $n = 15$.

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

Figure 4 shows the number of instances the different algorithms can solve within 45min/8GB. eCUT performs best, solving most instances in total and does so even as fastest on average (except for the last remaining instance *Kanban*). In addition, eCUT's performance is in a sense more robust than the others: the instances where BW, IC4PN, TSI and EEC time-out hardly overlap, making it difficult to assess which examples are harder than others.

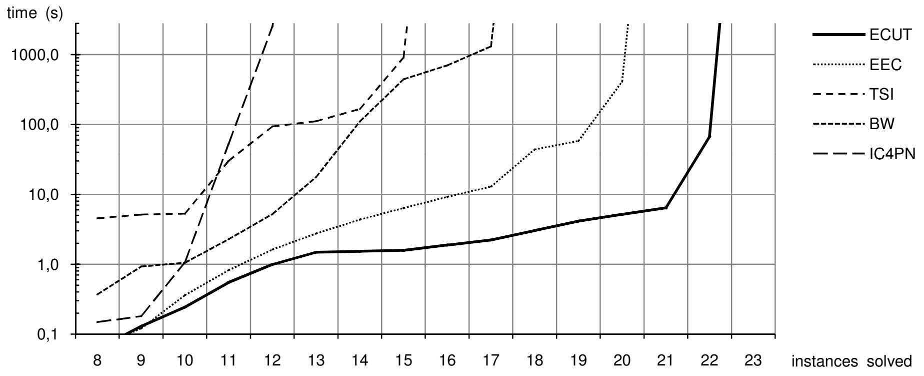


Fig. 4. Comparison of our explicit implementation (eCUT) and the algorithms BW, IC4PN, TSI and EEC for the Petri net benchmarks. The vertical axis gives the runtime (log-scale), the horizontal axis shows the numbers of instances that can be solved, starting from 8.

Boolean program benchmarks We evaluate our symbolic implementation sCUT using 1,087 Boolean programs. 852 of them were generated by SatAbs from two Linux device drivers (*ib700wdt* and *mixcomwd*), and 235 by Slam/Microsoft Research. The benchmarks feature, on average, 938 program locations, 15 local and 8 shared variables. We are not aware of any other tool that can deal with Boolean programs of this size and an unbounded number of threads. We therefore only present results obtained with our tool.

Table 2 shows details of the analysis. The tool terminates in 70% of the cases. In these examples, the cutoff test is complete, in that eventually all candidate triples disappear, and no backward search is necessary. The remaining 30% time out; this happens on average in round $n = 3.5$. Again, the time spent on computing candidate triples was negligible.

Benchmarks		Size (∅)			Time distr. (%)			Cutoff distr. (%)				State space
Name	Size	S	L	P	<.2s	>.2s	t/o	1	2	3	4	∅ $\|R_c\|$
ib700wdt	476	7	17	1,256	21	8	15	15	6	8	0	$2 \cdot 10^{28}$
micomwd	376	12	19	1,076	14	9	12	11	8	4	0	$3 \cdot 10^{37}$
slam	236	3	6	72	15	4	3	7	8	4	<1	$6 \cdot 10^{87}$
total	1,087	8	15	938	50	20	30	33	22	15	<1	$1 \cdot 10^{87}$

Table 2. Results of our symbolic approach (sCUT) for the Boolean program benchmarks (timeout 15min, memory limit 16GB). Columns show: benchmark name, and total number of programs; average shared/local variables and program locations (S, L, P); distribution of runtimes split at 0.2s (t/o = time-out); distribution of minimum cutoffs; and the average number of thread states R_c.

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.

Cutoffs: Much of the work on verifying concurrent programs using cutoffs restricts communication [5, 10]. For example, small constant-size cutoffs are known for ring networks communicating only by token passing [10], and for multi-threaded programs communicating only using locks [18]. These results do not hold, however, with general shared-variable concurrency, as we consider it. Cutoffs that are linear in the size of $\mathbb{P}$'s state space, such as in [9], are not acceptable for us, as $\mathbb{P}$ may have millions of states.

Bingham presents a cutoff-like technique for coverability [4]. The algorithm applies to parameterized finite-state systems. Beginning from an initial number of threads n, standard finite-state BDD techniques are used to compute the set of global states that are predecessors of the set of covering vectors. The analysis is repeated with increasing values of n until some necessary and sufficient convergence criterion is met. Unfortunately, Bingham does not discuss the experimental values of n at which his algorithm terminates. His technique seems to outperform standard Petri net covering techniques only in some cases.

Petri nets: Many data structures and algorithms have been proposed for their efficient analysis and coverability checking [15, 11, 8]. 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 many tools available for the analysis of Petri nets [17]. The PEP tool is a popular coverability checker [16] that takes as input a variety of languages such as $B(PN)^2$ and SDL. A more recent tool implements the *Expand, Enlarge and Check* algorithms due to Geeraerts et al. [15]. Furthermore, Petri net/VASS analysis has been applied to Java programs [7] 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. Moreover, 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 reachability cutoffs, by analyzing the programs for increasing numbers of thread counts. We have presented a 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 lean backward coverability analysis on candidate thread state pairs that have been identified to potentially lead to previously unseen thread states. The algorithm returns 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 algorithm to beat various methods based solely on Karp/Miller-like coverability tests. Our algorithm 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.

As with all cutoff-based approaches, our method can be seen as an opportunity to solve the (unbounded) parameterized model checking problem using only *bounded* tools. In our context of replicated finite-state programs, this is of utmost value, since there are efficient bounded tools (such as BOOM), but no generally efficient unbounded tools, particularly concerning symbolic implementations.

References

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
2. T. Ball, S. Chaki, and S. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, 2001.
3. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, 2009.
4. J. Bingham. A new approach to upward-closed set backward reachability analysis. *ENTCS*, 2005.
5. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.

6. B. Cook, D. Kroening, and N. Sharygina. Verification of Boolean programs with unbounded thread creation. *TCS*, 2007.
7. G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, 2002.
8. G. Delzanno, J.-F. Raskin, and L. V. Begin. Covering sharing trees: a compact data structure for parameterized verification. *STTT*, 2004.
9. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, 2000.
10. A. Emerson and K. Namjoshi. Reasoning about rings. In *POPL*, 1995.
11. A. Finkel. The minimal coverability graph for Petri nets. In *ATPN*, 1993.
12. A. Finkel, J.-F. Raskin, M. Samuelides, and L. V. Begin. Monotonic extensions of Petri nets: Forward and backward search revisited. *ENTCS*, 2002.
13. P. Ganty, L. V. Begin, G. Delzanno, and J.-F. Raskin. *The MIST2 tool, release 1.0, June 2009.* Université Libre de Bruxelles. Available from `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. Inf.*, 2008.
15. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check... made efficient. In *CAV*, 2005.
16. B. Grahlmann and E. Best. PEP – more than a Petri net tool. In *TACAS*, 1996.
17. F. Heitmann and D. Moldt. Petri net tool database. Available from `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.
18. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.
19. R. Karp and R. Miller. Parallel program schemata. *Computer and System Sciences*, 1969.
20. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.
21. C. Rackoff. The covering and boundedness problems for vector addition systems. *TCS*, 1978.