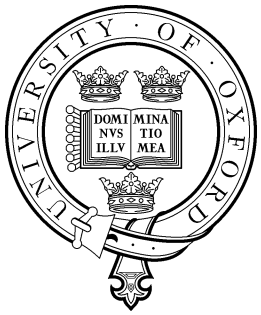
Using the Karp-Miller Tree Construction to Analyse Concurrent Finite-State Programs



Haoxian Zhao

Kellogg College

Submitted in partial fulfilment of the degree of

Master of Science in Computer Science

Supervised by Dr Daniel Kroening and Dr Thomas Wahl

Trinity 2009

Acknowledgements

I am grateful to Dr Daniel Kroening for introducing me to a most interesting and challenging research area and the discussions of the original ideas help me establish the framework of the topic.

I very appreciate Dr Thomas Wahl for sharing the knowledge and experience and the enlightening discussions about every edge of the ideas and the valuable suggestions facilitating the thesis writing.

I would like to take this opportunity to thank Xing Fu for giving me a tremendous amount of support to life and work in the UK. I am also thankful to my dear parents for their spiritual and financial support during my stay in Oxford.

In addition, I would like to thank Jiewen Huang sincerely for his helping and accompanying me during the whole year.

Abstract

The formal analysis of multi-threaded programs is among the grand challenges of software verification research. In this dissertation, we consider non-recursive multi-threaded Boolean programs, the principal ingredient in predicate abstraction. We introduced a exact and complete solution for thread-state reachability analysis of concurrent Boolean programs with *unbounded thread creation*. We present a novel implementation of the *Karp-Miller* procedure for vector addition systems with states, and evaluate its performance using a substantial set of Boolean program benchmarks. We then use our technique to prove that the benchmark programs have surprisingly small *reachability cutoffs*. We conclude that verifying a program for the fixed cutoff-number of threads tends to be cheaper than the Karp-Miller procedure. Our results mark a first step towards truly efficient strategies for analyzing multi-threaded programs.

Contents

List of Figures

Chapter 1

Introduction

Modern programming has been successfully applied to various fields in the society, attempting to obtain full or semi automatic solutions for particular problems in a specific domain, these applications range from programs for a vending machine to the flight control systems. People write programs for device drivers, safety-critical and embedded systems. The reliability of these systems is tending to be affecting everyone's life today especially in the safety-critical and economically vital applications. Diverging methods are available to achieve system reliability and satisfaction of requirements, such as testing, simulation and formal verification.

At the same time, concurrent software is gaining tremendous importance due to insatiable demands for computing power but the limited power of sole computing core and the widely use of distributed system architecture. Complex problems are managed to be divided into pieces and delegated to a set of computing entities which are performing in parallel. Such as a web server will employ multiple threads to handle the high load of serving, and multiple cores cooperate to process the same huge set of data at the same time. These can lead to a dramatic increase in efficiency. The most prominent and flexible model of communication between these parallel threads is the use of variables that are fully shared among the threads. This computational model is supported by a broad range of well-known programming APIs, e.g., by the POSIX pthread model and Windows' WIN32 API. However, the supports of communications between threads also complicates the behaviors of the systems, bugs in such a programming environment tend to be subtle and difficult to detect by means of testing [39], such as race condition, deadlock and etc.. To guarantee such com-

plex systems work properly would be a daunting challenge, thus strongly motivating efficient formal analysis techniques for such programs.

One of the most prominent techniques to guarantee the correctness of software and analyse the behaviours of software is Model Checking which verifies all reachable states from the system to see if some particular property holds, by performing a sophisticated search through the state space to provide full coverage. However, to apply such model check on concurrent software [27, 45, 16, 5] is extremely challenging since to search through the entire state space is the major obstacle that model checking concurrent software faces in practice: the *state explosion problem*, the number of reachable states will grow exponentially with the number of concurrent threads. The formal model that is needed in order to systematically explore the system is often much larger than the original system description. As a result, straightforward use of model checking on concurrent software can result unacceptable time and memory usage.

In this dissertation, we consider the case in which no a-priori bound on the number n of concurrent threads is known. This is the scenario most relevant in practice; Google would be a typical instance, instead of using sole thread to serve all the time, they spawn additional worker threads (can be many) to handle the huge loads of requests in parallel and simultaneously. The verification problem for such software is undecidable in general [44, 1], due to the infinite state space issued by these parallel programs. We focus on the special case of *replicated finite-state* programs: the program itself only allows finitely many different 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, where the threads communicate via shared program variables and have local storage. Boolean program verification is the bottleneck in the widely-used *predicate abstraction-refinement* framework.

The verification problem considered in this dissertation is that of *thread-state reachability*. A thread state comprises the local state of one thread, plus a valuation of the shared program variables. Thread states can be used to encode many common safety properties of systems, even if they involve several threads, such as mutually exclusive resource access.

The aim of this dissertation is therefore to develop efficient decision procedures for the thread-state reachability analysis for concurrent Boolean programs with unbounded replications.

Even the replicated finite-state programs with unknown number of threads will generate unbounded state space, the thread-state reachability problem for them can be shown to be decidable, by a reduction to the *coverability problem* for *vector addition systems with states* (VASS). This problem in turn is known to be decidable, due to early works by Karp and Miller [36] for decidable result on coverability of VAS and Hopcroft and Pansiot [33] for simulation of VASS with VAS. In principle, this reduction can be used as the starting point for an algorithm to decide the thread-state reachability problem. In attempting to do so, however, we face two practical obstacles:

1. concurrent Boolean programs are not VASSs: while the former can be converted to the latter using thread counters, doing so naively results in addition systems of prohibitively large dimension;

2. the worst-case complexity of known coverability of VAS decision procedures is intractably high, even with later improvements due to Rackoff [43].

The conversion of replicated finite-state programs into vector addition systems can be accomplished using a form of *counter abstraction*: we keep an (unbounded) counter per local state that records the number of threads in that local state. The problem with this conversion is that the number of counters thus introduced is exponential in the number of local variables per thread; in practice, this can amount to millions of counters. This problem has been identified in earlier work as the *local state space explosion* problem [5]. The same work presents a solution for the case of bounded replication.

In this dissertation, we extend this solution to programs with *unbounded dynamic thread creation*, which allows us to overcome the first of the two obstacles mentioned above. We then demonstrate, using substantial experimentation, that our *Karp-Miller procedure for Boolean programs* performs much better in practice than its worst-case complexity seems to suggest. Our benchmark harness is drawn from a large and diverse set of benchmark Boolean programs generated from Linux and

Windows kernel components. Our solution is, to our knowledge, the first practically useful and **exact** thread-state reachability implementation for realistic concurrent Boolean programs with arbitrarily many threads.

We compare this solution with an alternative approach to the thread-state reachability problem. For every finite-state program $\mathbb{P}$, there is a number n_0 such that any thread state reachable given *some* (arbitrarily large) number of threads can in fact be reached given n_0 threads. We call the smallest such number the *reachability cutoff* of $\mathbb{P}$. If this cutoff is known, the thread-state reachability problem can be solved without the Karp-Miller procedure.

Unfortunately, determining the precise cutoff is in general as expensive as the reachability problem itself. Moreover, there is no universal cutoff bound: it can in general be arbitrarily large. Using our implementation of the Karp-Miller procedure for concurrent Boolean programs, however, we can demonstrate empirically that, for a large benchmark collection, the cutoffs are almost universally very small, in the range of around 2 or 3. This observation has two consequences. First, assuming the cutoff is known, we demonstrate that thread-state reachability analysis via the cutoff method is much cheaper than via the Karp-Miller method. Even incomplete methods that can find tight cutoffs for *many* Boolean programs are therefore likely to be of high practical value. Second, if the cutoff is not known, an incomplete method that analyzes a finite-state program for an a-priori chosen fixed and small number of threads is likely to find all reachable thread states, and thus bugs, for *many* programs.

Contribution. In summary, this dissertation accomplishes the following.

1. We optimized Karp-Miller's algorithm for the coverability problem of VASS, making it efficient enough in practices, and thus, an efficient decision procedure for the thread-state reachability of replicated finite-state systems with unbounded number of threads.

2. We present a practical *and* exact method that: given a Boolean program, the algorithm computes the set of thread states reachable given an arbitrary number of parallel threads, whether instantiated up front or created dynamically. A

completed solution is also provided by implementing the above algorithms in a tool called STAR ("Scalable Thread-state reachability AnalyzeR ")[1].

3. We use this method to show that the *reachability cutoffs* of a large and representative set of concurrent Boolean programs are very small, giving rise to a cheap testing method with high coverage, that is at the same time cheaper than Karp-Miller based techniques.

These contributions have been submitted to The Eleventh International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010) under the title "Analyzing Multi-Threaded Programs With Unbounded Replication"

[1]The executable binary and the source are available in http://www.mrzhao.com

Chapter 2

Preliminaries

2.1 Computational Model

A *replicated finite-state system* is a tuple $\mathcal{D} = (S, L, \Delta, c_0)$, where S is a finite set of shared states and L is a finite set of local states. Δ is a relation containing thread transitions of the form $(s, l) \to (s', l')$ and $(s, l) \to (s', l', l'')$. System $\mathcal{D}$ gives rise to a Kripke structure M as follows. The state set of M is $S \times L^+$; in particular, c_0 is the initial global state. Whenever $(s, l) \to (s', l') \in \Delta$, we have a transition $(s, l_1, \ldots, l_n) \to (s', l'_1, \ldots, l'_n)$ in M where $l_k = l$ and $l'_k = l'$ for some k, and $l_j = l'_j$ for all $j \neq k$. Similarly, whenever $(s, l) \to (s', l', l'') \in \Delta$, we have a transition $(s, l_1, \ldots, l_n) \to (s', l'_1, \ldots, l'_n, l'')$ where $l_k = l$ and $l'_k = l'$ for some k, and $l_j = l'_j$ for all $j \neq k$. The second type of transition models the dynamic creation of a thread. Thread termination can be simulated by forcing a thread into a self-loop.

In practice, replicated finite-state systems are given in the form of a single program $\mathbb{P}$ that permits only finitely many configurations but is executed by some number (possible infinite) of parallel threads. In particular, $\mathbb{P}$'s variables are of finite range. Further, if the language in which $\mathbb{P}$ is written supports function calls, the call graph of $\mathbb{P}$ is acyclic. An instance of this scenario is given by non-recursive Boolean programs, which are obtained from `C` programs using predicate abstraction, and we refer the reader to [16] for a conservative way of dealing with recursion. We note that the reachability problem for concurrent programs with recursive procedures is undecidable [44].

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 $\mathbb{P}$. Further, there is one copy *per thread* of all local variables of $\mathbb{P}$. We here adopt some terminology from [16].

Definition 1 . *A **shared state** is a valuation of the shared variables. A **local state** is a valuation of (one copy of) the local variables. A **thread state** is a pair (s, l) such that s is a shared state and l is local state. A **system state** comprises a valuation of shared variables, plus tuples of valuations of local variables for every thread.*

A thread state completely describes the information accessible to a single thread, namely, the shared variables and its copy of the local variables. Contrast this with more restricted ways of sharing variables, such as locks [35, 34]. A thread has neither read nor write access to any other local variables.

Definition 2 ***Thread-state reachability** problem asks: for a given thread state s and template program $\mathbb{P}$, whether the given thread state s is reachable by the executing the program $\mathbb{P}$ with some number n (possibly infinite) threads in parallel.*

We remark that our model of replicated finite-state systems cover both classical *parameterized* systems, where the number of threads running is fixed up-front but unknown, and *dynamic* systems, where the number of threads can change at runtime. It is quite easy to show that the two types have equivalent expressive power, as each one can simulate the other. We here also assume a standard asynchronous execution model. That is, exactly one of the threads executes a step of $\mathbb{P}$ at a time.

2.2 Predicate Abstraction and Concurrent Boolean Program

Abstraction is one of the main techniques the model checkers rely on to verify systems with large or infinite state space, which maps the concrete and large set of states to a smaller set of states in a way that preserves the property of interest. Predicate

```
   for (i=0; i < apthr_per_child; i++) {
     int status = ap_scoreboard_image -> servers[child_argno
        ][i].status;
     if (status != SRV_GRACEFUL && status != SRV_DEAD)
        continue;
(b)  apr_status_t rv = apr_thread_create(&threads[i]),
   thread_attr,worker_thread, my_info, pchile);
     if (rv != APR_SUCCESS) {
       ap_log_error(APLOG_MARK, APLOG_ALERT, rv,
          ap_server_conf, "apthr_create: error");
       clean_child_exit(CHILDSICK);
     }
     threads_created++;
   }

   main() begin
     decl i_lt_apthr_per_child, status_eq_SRV_GRACEFUL,
        status_eq_SRV_DEAD, rv_eq_APR_SUCCESS;
     L1: goto L2, L5;
     L2: assume i_lt_apthr_per_child;
         status_eq_SRV_GRACEFUL, status_eq_SRV_DEAD := *,*;
         goto L3, L4;
(b)  L3: assume (!status_eq_SRV_GRACEFUL) && (!
   status_eq_SRV_DEAD);
         i_lt_apthr_per_child := *; goto L1;
     L4: rv_eq_APR_SUCCESS := true;
         skip;
         i_lt_apthr_per_child := *; goto L1;
     L5: assume !i_lt_apthr_per_child;
   end
```

Figure 2.1: possible translation from a C program segment (a) to a Boolean program (b); adopted from [5]

Abstraction is one of the most popular and widely applied method for systematic state-space reduction of programs [28], and promoted by the success of the SLAM project [3, 4].

In the Predicate Abstraction, a concrete model M is mapped to an abstract model $\hat{M}$ according to a given set of predicates. Data is abstracted by keeping track of certain predicated over the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an over-approximation of the original program. Data is abstracted by keeping track of certain predicates over the data. The data variable in the original program will be eliminated and replaced by a Boolean variable according to the predicates, the resulting abstract program is an over-approximation of the original program. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to rene the abstract program, and the process proceeds until no spurious error traces can be found[13]. The actual steps of the loop follow the abstract-verify-rene paradigm [37].

Concurrent Boolean programs [3] are obtained from `C` programs using Predicate Abstraction [28, 38, 14], Figure 2.1 shows an instance for these abstractions. In this type of programs, all variables are in type of Boolean and are declared to be either shared or local where shared variables can be fully accessed by all threads executing the same program while local variables are visible to (accessible by) one thread only. [16] provides an asynchronous semantics of concurrent Boolean programs. Refer to figure 2.2 for the syntax of concurrent Boolean programs.

2.3 VASS and Karp-Miller Tree

Some notations. We denote by $\mathcal{S}, \mathbb{Z}^m, \mathbb{N}^m$ respectively the set of states, the set of m-tuple of integers and the set of m-tuple of non-negative integers.

Vector Addition System with States (VASS) is an variant of the Vector Additon System (VAS), which was introduced by [33] and proved can be simulated by VAS in the same work. A m-dimensional VASS is a finite-state machine whose (directed) edges are labelled with some $v \in \mathbb{Z}^m$, it is denoted by $\mathscr{W} = (d, \delta)$ where $d \in \mathcal{S} \times \mathbb{N}^m$ is the initial configuration and $\delta : \mathcal{S} \to \mathcal{S} \times \mathbb{Z}^m$ is the transition mapping function.

Syntax			Description
prog	::=	*decl** *proc**	A program is a list of shared variable declarations followed by a list of predure definitions
decl	::=	**decl** *id*$^+$ **;**	Declaration of variables
id	::=	[**a-zA-Z_**][**a-zA-Z0-9_**]*	An identifier is a regular C-style identifier
proc	::=	*main*() **begin** *decl** *sseq* **end**	Procedure definition, and local variable declaration
sseq	::=	*lstmt*$^+$	Sequence of statements
lstmt	::=	*stmt*	
	|	*id***:**$^+$ *stmt*	Labeled statement
stmt	::=	**skip ;**	
	|	**goto** *id*$^+$ **;**	Nondeterministic goto
	|	**return** *id** **;**	
	|	*id*$^+$:= *expr*$^+$ **constrain** *expr* **;**	Parallel assignment
	|	**if** (*expr*) **then** *sseq* **else** *sseq* **fi**	Conditional statement
	|	**assert** (*expr*) **;**	Assert statement
	|	**assume** (*expr*) **;**	Assume statement
	|	**start_thread** *id* **;**	Thread creation
	|	**end_thread ;**	Thread termination
	|	**atomic_begin ;**	Beginning of atomic section
	|	**atomic_end ;**	Ending of atomic section
expr	::=	*expr binop expr*	
	|	**!** *expr*	Negation
	|	(*expr*)	
	|	*const*	
	|	*id*	Variable
binop	::=	'|' | '**&**' | '∧' | '**=**' | '**!=**' | ' $\Longrightarrow$ '	Logical connectives
const	::=	**0** / **1**	False / True

Figure 2.2: Syntax of concurrent Boolean Programs (adapted from [3] and extended to SatAbs' syntax).

There is a transition $(q, x) \longrightarrow (q', x')$ if $(q', v) \in \delta(q)$ and $x' = x + v$, where $+$ denotes pointwise addition. A Configuration (q, x) is *reachable* if there exists a sequence of transitions start at d and ending at (q, x). Figure 2.3 shows an example of VASS.

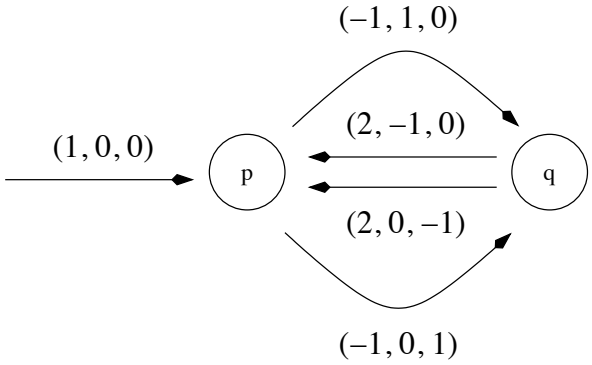


Figure 2.3: Example of VASS

Definition 3 ***VASS Coverability*** *problem asks, for a given q and x, whether a configuration (q, x') is reachable s.t. $x' \geq x$.*

Theorem 4 ([36]) *The coverability problem for VASS is decidable.*

The algorithm by Karp and Miller [36] builds a rooted tree (we call it the Karp-Miller Tree) $\mathscr{T}(\mathscr{W})$ with only finite number of nodes (i.e. guarantee termination) that compactly represents the set of covered configurations of a vector addition system. We here extend this classic algorithm to fit the VASSs naturally as followings:

1. For any two nodes ξ and η in the tree, if there is a path from ξ to η, we say $\xi \prec \eta$; if there is an edge from ξ to η, then η is a successor of ξ. A node without successors is called an ***end***.

2. Each node η in the tree is labelled with $l(\eta) = (q, x)$ where $q \in \mathcal{S}$ and x is an m-dimensition vector whose coordinates are elements of $\mathbb{N} \cup \{\omega\}$. For label $l(\xi) = (p, y)$ of any distinct node ξ, we say $l(\eta)$ ***covers*** $l(\xi)$ if $p = q$ and for each ith coordinates of vectors x and y, $x_i \geq y_i$.;

3. The root of the tree is labelled d;

4. For some node η and $l(\eta) = (q, x)$:

a. if, for some node $\xi \prec \eta$, $l(\xi) = l(\eta)$, then η is an end;

b. otherwise, the successors of η are in one-to-one correspondence with the elements $(q', v) \in \delta(q)$ s.t. $0 \leq x + v$. Let the successor of η corresponding to (q', v) be denoted by $\eta_{(q',v)}$. For each label $l(\eta_{(q',v)}) = (p, x')$, $p = q'$ and for each ith coordinates of the vector x' is determined as followings (We call it the ω procedure):

   i. if there exist $\xi \prec \eta$ and $l(\xi) = (s, y)$ s.t. η covers ξ and $y_i < (x + v)_i$ then $x'_i = \omega$. ω is a symbol s.t., for $z \in \mathbb{Z}, \omega > z$ and $\omega + z = \omega$;

   ii. if no such ξ exists, then $x'_i = (x + v)_i$;

These rules are recursively applied to any vertice with a *new* label. This procedure eventually reaches a fixpoint, as the set of labels that can be generated this way is finite [36]. A configuration (q, x) is then covered by the VASS exactly if there exists a label (q, v) in the tree whose vector component v satisfies $x \leq v$, with the obvious extension of $\leq$ to ω. Let's consider the example shown in Figure 2.3 , i.e. a VASS $\mathscr{W} = (d, \delta)$ where

$d = [p, (1, 0, 0)]$

$\delta(p) = \{$ [q, (-1, 1, 0)], [q, (-1, 0, 1)] $\}$

$\delta(q) = \{$ [p, (2, 0, -1)], [p, (2, -1, 0)] $\}$

Then we have a rooted tree $\mathscr{T}(\mathscr{W})$ as shown in Figure 2.4 by Karp-Miller's algorithm. The tree has only finite number of nodes, as shown, even $\mathscr{W}$ has an infinite state space.

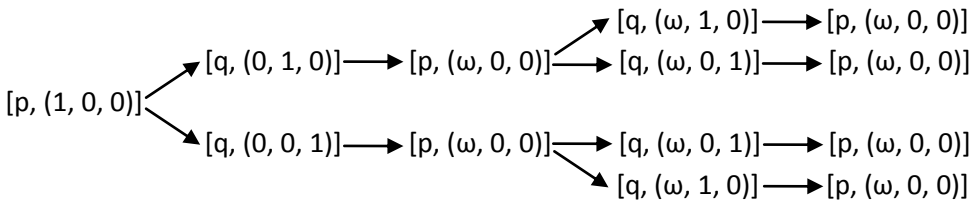


Figure 2.4: Karp-Miller Tree for VASS

2.4 Replicated Finite-state Systems as VASS

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. Dynamic thread creation $(s, l) \to (s', l', l'')$ is modeled by a VASS edge $s \xrightarrow{v} s'$ such that the l-th component of v is -1, the l'-th and the l''-th components are 1, and all others are 0. This reduction was used by German and Sistla in a related setting [27]. Similar reductions have been presented for models of `C` [2] and Java [18] as well as for other rewriting models [7]. A thread state (s, l) of the given 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 position l equals 1. The latter problem is decidable by Theorem 4. We obtain:

Corollary 5 *The thread-state reachability problem for replicated finite-state programs is decidable.*

The original Karp-Miller algorithm is of non-primitive recursive space complexity. Rackoff [43] later improved the algorithm to operate in exponential space, which is close to optimal. While daunting, these complexity measures are not terminal for our approach: vector addition systems that encode concurrent Boolean programs are of very simple structure, with most integer vectors having all-but-two zero entries.

Chapter 3

Karp-Miller Tree For Thread-state Reachability

In this section, we present some of the main contributions of this dissertation: *1.* a complete solution to encode the concurrent Boolean programs for *thread-state reachability* using Karp-Miller Tree; *2.* an optimized algorithm for coverability problem of VASS; *3.* and therefore an efficient decision procedure for the thread-state reachability of concurrent Boolean programs with unbounded replications through a reduction to the coverability of VASS and some novel ideas for resolving the *local state space explosion* problem inspired by the works in [5].

3.1 Concurrent Boolean Programs to VASSs

We convert the concurrent Boolean programs into VASSs using a form of *counter abstraction* [41] : instead of storing the system state of the concurrent Boolean program as a tuple of a shared state and a vector of local states that each thread residing in, we introduce a counter per local state, which records the number of threads currently in that particular local state. A thread transition from local state l_1 to local state l_2 is presented by a decrement of the counter for l_1 together with an increment of that for l_2. The state space which is originally exponential in the number of threads has been turned into the one polynomial in the number of threads. This abstraction is very useful in the context of the targeted programs, since the number of the local states

is always bounded while there are possiblly infinite many threads executing the the programs, and thus, compatible to the vectors inside VASS configurations.

Boolean programs are not VASSs, we generally need some intermediate transformations to make these programs compatible with the VASSs. *Thread-state transition diagram* (TTD) would be one instance, a simple extension of the *local-state transition diagram* [47, 20, 31]. In such diagrams, the behavior of a thread (a component of the vector) is given in terms of thread state changes. For example, refer to the TTD in Figure 3.1 (b), one of the transitions 0 0 → 0 1 represents the change from thread state (0,0) to (0,1). With this diagram, we can build up the addition set of VASS naturally, then finally the Boolean program can be encoded into VASS for thread-state reachability analysis. To illustrate how these conversion works, we take the examples shown in Figure 3.1 for a complete routine:

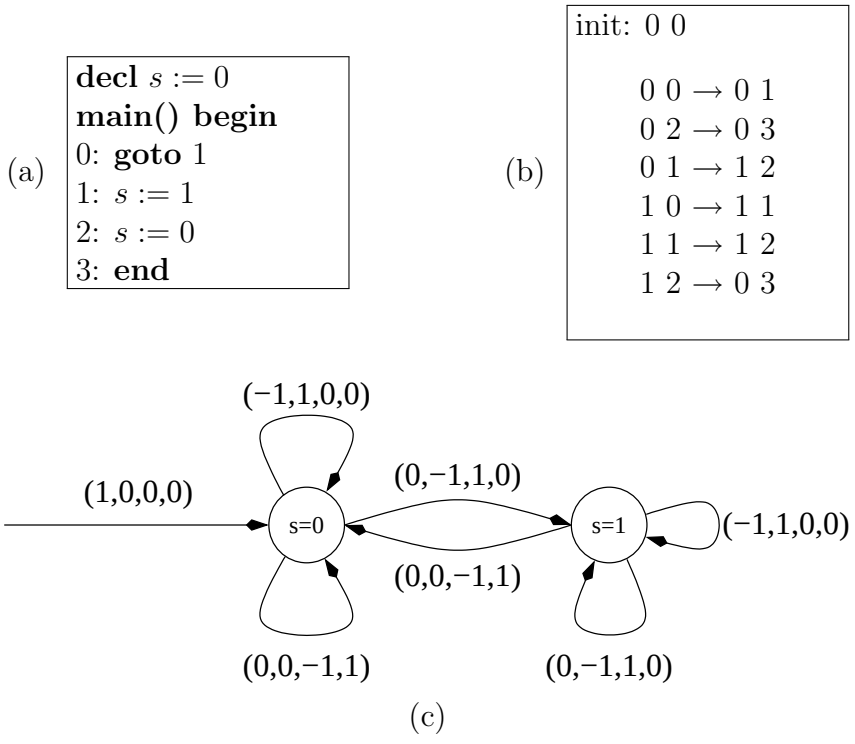


Figure 3.1: (a) Boolean program (b) TTD (c) VASS

Suppose we have a Boolean program as Figure 3.1 (a) shows, there is one shared variable s and program counter PC ranges from {0 - 3}, then we should have a set of transitions among $(2^2 * 2^4)$ thread states at most, and (0, 0) would be the initial thread state and presenting the valuation of ($s == 0$ and $PC == 0$). After the execution of the first line of the main code "0: **goto** 1", we have transitions from ($PC == 0$)

to ($PC == 1$) regardless of the valuations of other variables (i.e. the shared variable s in this example). Then we add transitions (0,0) to (0,1) and (1,0) to (1,1) to the resulting TTD. After the execution of the second line of the main code "1: $s := 1$", we have transitions from valuation ($PC == 1$) regardless of the valuation of other variables to the valuation of ($s == 1$ and $PC == 2$), i.e. we have two new transitions (1, 1) to (1, 2) and (0, 1) to (1, 2). Similarly, after the executions of all lines of the main code, we will have a complete thread-state transitions diagram as Figure 3.1 (b) shows, i.e. a *replicated finite-state system*. Then, as described in section 2.4, this replicated finite-state system can be simulated by a VASS likes Figure 3.1 (c).

However, this approach is inefficient: a static conversion from Boolean program to thread-state transition diagram invariably incurs a large blowup: the number of transitions generated for each statement in the Boolean program is exponential on the number of variables (both shared and local). For every transition between states, we have to enumerate items in this highly blowup data structure, i.e. the addition set in VASS, which leads to the inefficiency in searching.

Moreover, an naive implementation is problematic: i.e. the *local state explosion* problem, which was identified earlier by [5]: in concurrent Boolean programs, the number of local states is exponential in the program text size and unmanageably too large in general. Take the Boolean program segment in Figure 2.1 for instance, it has 4 local variables with Program Counter in range $\{1..12\}$, then there will be $2^4 * 12 = 192$ local states, and hence a 192-dimensions vector for echo configuration in the VASS. The state space of the VASS is of size doubly-exponential in the number v of local variables, namely $\Omega(n^{2^v})$. In realistic applications, concurrent Boolean program can have thousands of program lines, and dozens of variables.

To tackle the above problems, we extend the solution proposed for bounded counter program (the amount of the counters is bounded) by [5] to our unbounded context:

1. Instead of translating the Boolean programs into VASS through thread-state transition diagrams statically, we interleave the translation of individual program statements with the state space exploration phase, i.e. execute the program statements and update the system states *on the fly*. This has the advantage that our algorithm is **context-aware**: the local-state context in which

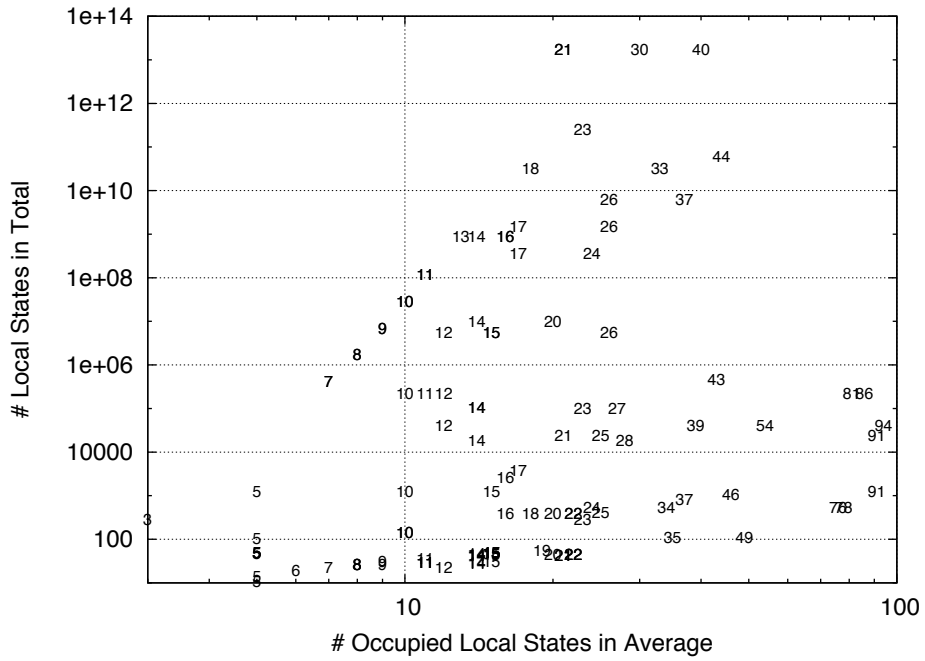


Figure 3.2: Numbers of Local States in Total and Occupied Local States in Average

the statement is executed is known; the context determines which local-state counters need to be updated.

2. In the system states, instead of manipulate counter values for all local states, we keep counters only for those local states that at least one thread resides in. In a bounded context, like [5], the number of "occupied" local states is obviously bounded by the number of threads in existence. Even in our unbounded context, from our empirical results shown in Figure 3.2 (we will explain how these results are obtained in section 3.4), this number remains tending to be a tiny fraction of all local states. The number of "occupied" local states goes only 21 in average and 94 at most even some of the programs can has $1 * 10^{14}$ local states in total. This illustrates drastically the difference to a method that keeps counters for *all* local states.

As a result, our implementation can be seen as a version of the Karp-Miller procedure that operates directly on a Boolean program, as opposed to on a VASS. We therefore completely avoid the blowup that an intermediate conversion to any type of addition systems (such as a counter program or a VASS) invariably entails.

3.2 Implementing The Karp-Miller Tree Construction

The Karp-Miller Tree Construction procedure is implemented in C++ combined with intensive uses of the Standard Template Library (STL) for high level of efficiency. For the ease of extending supports for new front-ends (e.g. Boolean programs and Thread-state Transitions Diagram), a set of techniques have been applied in this development, such as C++ template, polymorphism, design patterns and so on to make the construction procedure as generic as possible.

Combining the ideas presented in section 3.1 and the rules described in section 2.3, we could develop an explicit-state algorithm (algorithm 1) for thread-state reachability analysis of replicated finite-state system $\mathcal{D}$, with on-the-fly symmetry reduction on system state space implemented via the Karp-Miller Tree Construction. The algorithm takes a template program $\mathbb{P}$ as input and computes, in the variables Reach, the set of configuration (i.e. the symmetry-reduced system state space) reachable from a given initial configuration, then extract and output the the set of reached thread-state from Reached. Note, all rules mentioned in this section are from the Karp-Miller's coverability tree construction algorithm described in section 2.3.

Configurations are stored by the algorithm in form of a tuple

$$c := (s, L, f : L \longrightarrow \mathbb{N} \cup \{\omega\}) \tag{3.1}$$

(see line 4). Variable s is a valuation of the shared variables, L is a set of local states, and f is a function as declared in (3.1). Intuitively, given $c = (s, L, f)$, only local states mentioned in L occur in the system states of $\mathcal{D}$ represented by c. For such local states l, function f specifies the number of threads residing in l; observing that the range of f does not include 0. As a result, the system states represented by c form a *symmetry equivalence class*. Conversely, every such equivalence class can be represented in the form (3.1), making this representation sound and complete.

For the flexibility of the Karp-Miller Tree construction procedure, we have implemented two versions of the algorithm, one is based on *depth-first search* (DFS) order while the other is based on *breath-first search* (BFS) order. In addition, for the efficiency of the procedure, instead of proceeding the exploration in recursively, we

Algorithm 1 Karp-Miller Tree Construction

1: Unexplored := Reached := Initial
2: **while** Unexplored is not empty **do**
3: pop a config $c := (s, L, f : L \longrightarrow \mathbb{N} \cup \{\omega\})$ from the front of Unexplored
4: PREDS := PREDECESSORSOF(c) //included all predecessors till the root
5: **if** $c \notin$ PREDS **then**
6: **for all** $l \in L$ **do**
7: execute $\mathbb{P}$ on (s, l) to obtain *Successors*
8: **for all** $(s', l') \in Successors$ **do**
9: $L' := (\ f(l) = 1\ ?\ L \setminus \{l\}\ :\ L\ )\ \cup\ \{l'\}$
10: $$f'(x) = \begin{cases} f(x) - 1 & \text{if } x = l \\ f(x) + 1 & \text{if } x = l' \wedge l' \in L \\ 1 & \text{if } x = l' \wedge l' \notin L \\ f(x) & \text{otherwise} \end{cases}$$
11: $c' :=$ OMEGA $(s', L', f',$ PREDS) //introduce ω to the counters
12: **if** $order == DepthFirst$ **then**
13: append c' to the front of Unexplored
14: **else if** $order == BreathFirst$ **then**
15: append c' to the back of Unexplored
16: add c' to Reached;

implemented the algorithms in a non-recursive fashion (algorithm 1) which eliminates the costs from issuing (possible a large number of) function calls to the system stack.

While available, an unexplored state c is popped from the Unexplored queue (line 4). Before continues to the exploration phase, we compare c with all its predecessors till the root of the tree to see if c has been discovered previously in the same path. If so, we discard c and c is considered as an end of the tree (line 4, 5, for rule 4.a). Set L reveals which local states l occur in the system state encoded by c; these local states are paired with the s, the valuation of shared variable, and $\mathbb{P}$ is executed on the result directly (line 7). Note that l includes the value of the PC, revealing which line to start $\mathbb{P}$ from. The step of choosing a local state l in line 6 can be thought of as picking a *representative* thread out of those in local state l to make a step.

Executing $\mathbb{P}$ on the thread state (s, l) results in a new thread state (s', l') (line 7). In order to obtain a new state c' to add to either the back of the Unexplored queue for breath-first search order or the front of the Unexplored queue for depth-first search order, we combine the steps of constructing a new system state of $\mathcal{D}$ and converting that into counter notation (3.1), as follows. s' is copied into c' unchanged. If only 1

thread resided in local state l before, as indicated by $f(l) = 1$, we remove l from L; the new local state l' is added to L in any case, resulting in the set L' (line 9). The new function f' is identical to f except at l and l' (line 10). To ensure that the map f' contains only pairs with non-zero counter values, a cleanup step may remove the pair $(l, f'(l))$ (not shown in 1). Finally, the configuration c' is built by applying the components s', L' and f' to the OMEGA procedure (line 11, for rule 4.b.i) and added to the containers Reached and Unexplored appropriately (line 12 –15).

Algorithm 2 The Omega Procedure

1: **procedure** OMEGA (s, L, f, PREDS)
2: **for all** $p := (s_p, L_p, f_p) \in \text{PREDS}$ **do**
3: **if** COVERS $((s, L, f), (s_p, L_p, f_p)) == true$ **then**
4: **for all** $l \in L$ **do**
5: $$f'(l) = \begin{cases} \omega & \text{if } l \notin L_p) \\ \omega & \text{if } l \in L_p \wedge f(l) > f_p(l) \\ f(l) & \text{otherwise} \end{cases}$$
6: **return** (s, L, f')

7: **procedure** COVERS $((s, L, f), (s', L', f'))$ //the former covers the latter
8: **if** $\neg(L \sqsupseteq L')$ **then**
9: **return** $false$
10: **for all** $l' \in L'$ **do**
11: **if** $f'(l') > f(l')$ **then**
12: **return** $false$
13: **return** $true$

The OMEGA procedure (algorithm 2) which cooperates with the COVERS procedure (by rule 2), is for labeling the ω symbol to appropriate local state counters which can have infinite many threads resisting in. This procedure takes a configuration in form of (s, L, f) and a set of configurations which are the predecessors of the former as input. This procedure compares all elements of the set PREDS with the configuration (s, L, f) and updates the mapping of function f according to the rule 4.b.i. We pop a configuration $p := (s_p, L_p, f_p)$ from PREDS and compare with (s, L, f), the procedure proceeds only if the latter *covers* (procedures start from in line 7) the latter (line 3). Then we compare all local state counters $f(l)$ with $f'(l)$ and update the value of the former to ω if the former is larger than the latter(lines 5). Finally new configuration (s, L, f') with a updated mapping function f' is returned.

As mentioned in section 3.1, during the state exploration phase, we keep counters only for those local states which at least one thread resides in instead of manipulate

counter values for all local states. To facilitate the above procedure achieving this efficiently and making the algorithm generic, we implemented a template class *Config* for the system states, which consists of a copy of the valuation of the shared variables and a mapping from valuations of local variables to the associated counters which in range of $\mathbb{N} \cup \{\omega\}$. Set L and function f can be viewed as a mechanism to look up the counter value corresponding to some local state. This mechanism is efficient in time *and* memory: function f is implemented in algorithm 1 as an associative map (std::map), which has access time logarithmic in its size. By which, the procedure could manipulate the values of the local state counters effortlessly, and also generically due to the high level of flexibility provided by the STL data structures. Contrast this with a more straight-forward but naive implementation that stores the look-up table as a vector of constant width equal to the number of conceivable local states. To build the Karp-Miller tree and manipulate the nodes in the tree, we also implement a class *ConfigNode* which links the nodes with their predecessor. Besides, to make sure the procedure is generic so that it can be extended to support various type of template program input easily, we abstract the behaviors of executing template program $\mathbb{P}$ and resulting a set of successors (line 7 in algorithm 1) in the procedure to an abstract class called *AdditionSet*. By implementing *AdditionSet* interfaces, and specifying the types for local and shared states, we could accomplish the supports for new type of input.

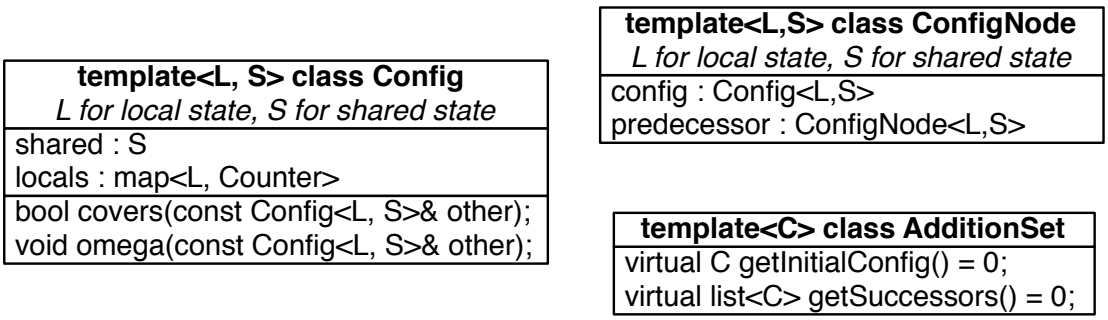


Figure 3.3: Class Diagrams for Configuration, Node of Configuration and AdditionSet

In the rest of this section, we focus on describing the implementations concerning the Karp-Miller procedure for Boolean programs.

To provide supports to the procedure for taking Boolean programs as inputs (the template programs), we implemented a parser for the Boolean programs according

to the syntax shown in figure 2.2 with the helps of FLEX[1] and BISON [2] which are a lexical analyser generator and a parser generator respectively. After the parsing phase, instead of building a single but big Abstract Syntax Tree (AST) for the whole Boolean program, we store the program statements into an *array* data structure which provides efficient random access and builds individual AST for each expression within the program statements for the valuations evaulation. For example, the parser will generate an AST as Figure 3.4 shown for the expression: $(!locked)$ & $((!queueSize - equ - 0) \mid bufferSize - le - MAX)$. These designs provide multiple benefits: (i) instead of traversing the whole AST to get access to a particular program statement, we could execute the program statement demanded instantly according the PCs of the statements i.e. the indices in the array data structure; (ii) With the associated ASTs which are generated during the parsing phase, and the use of visitor pattern, we could analyze the program expressions in a structured way which simplifies the complicated valuation evaluation procedures.

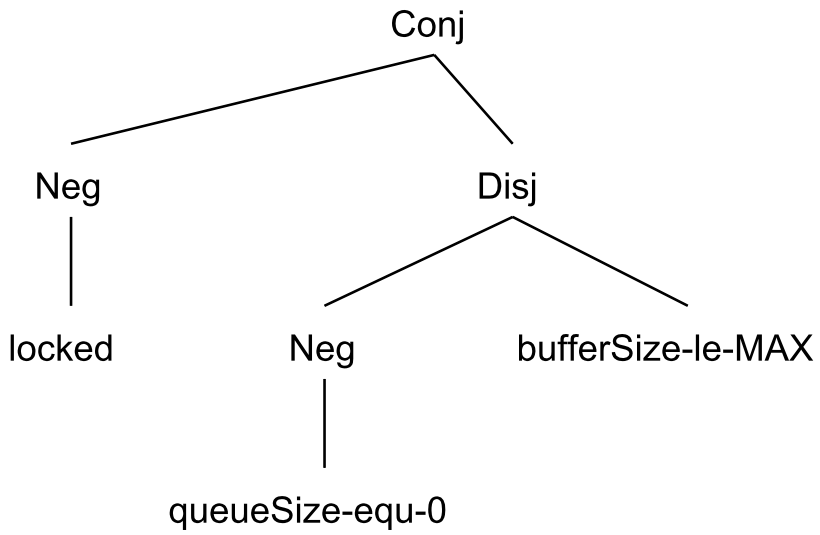


Figure 3.4: Abstract Syntax Tree for a possible expression in Boolean Program

In order to manipulate the valuations of variables in Boolean program efficiently, we encode the valuations of variables (both shared and local) into the configuration directly using bit-vectors (we use std::vector<bool> in our implementation), thus, we do not have to need any look-up mechanism to manipulate the valuation of the variables. Using bit-vector is space efficient: the memory used by a configuration with

[1]Flex is available for download on http://flex.sourceforge.net/

[2]Bison is under GNU and available for download on http://www.gnu.org/software/bison/

bit-vectors is bounded to the number v of variables in form of v bits + sizeof(PC). Contrast this with a more straight-forward but naive implementation that build and stores the look-up table as a vector of constant width equal to the number ls of conceivable local states which has been blowup highly (as section 3.1 described), thus we need memory in a constant size of ls bytes for indexing (we here assume the use of std::vector<T> where T is not bool) plus memory used for storing the valuations plus sizeof(PC) .

3.3 Optimizations on the Karp-Miller Tree Construction

The classic algorithm by Karp-Miller [36] described in section 2.3 caps the number of nodes in the tree to finite and accelerates the termination of the procedure by the use of rules that manipulate the symbol ω. But this procedure is impractical (either with DFS or BFS order) in most cases, even with the optimized implementation described in earlier sections, due to the intractably high worst-case complexity for this procedure, particularly for the Boolean programs which have thousands of transitions in general and we here note that the procedure using Karp-Miller's algorithm timed out (1 hour) on most of the cases that in our benchmark Boolean programs (we will describe these benchmarks in section 3.4). So in the rest of this section, we are going to introduce a set of algorithmic optimizations that make this procedure efficient enough in practice.

3.3.1 Depth-first Search and Breath-first Search.

The arguments between depth-first and breach-first algorithm have become a classic story, since neither would be dominant to the other for all cases. For example, breath-first search can provide the shortest path to the nodes with a particular label in the tree during the exploration phase, while depth-first search enjoys low level of memory usage. However, differ to other exploration algorithms which only require the top level of the tree to continue the exploration, we have to remember all nodes that have been discovered ever for the procedure introducing the ω symbol to the counter vector in Karp-Miller's algorithm.

From our empirical results, as Figure 3.2 shows, we know that the average number of "occupied" local states is 21, i.e. every nodes in the tree can have 21 child nodes in average if each has one successor in average. We note that we can have no but also many successors from the thread execution of a program statement in the realistic application, since our Boolean program syntax supports assume, dynamic thread creation, nondeterministic jumps and etc. So we can summarize that the number of nodes grows exponentially on the depth of the tree, in form of 21^{depth} in average, the generated tree would be extremely fat. For example, suppose we have a Boolean program which is typical in our benchmark collection (i.e. in the tree that is generated for this Boolean program by Karp-Miller's algorithm, each node has 21 child nodes); and suppose we have to explore the tree at least in depth 8 to terminate; then we need to explore $21^8 = 37,822,859,361$ nodes, even a lot of them are redundant in terms of having the same labels, there is still a large number of nodes to be explored if we explore the tree depth by depth i.e. in a Breath-first fashion. Moreover, in realistic Boolean programs, we generally have to explore much more than 8 depths (executing more than 8 program statements) to terminate the procedure. So, the optimizations we introduce below focuses on the Depth-first searching fashion which does not have to remember the high number of nodes during the exploration phase, making the procedure impractical in terms of memory usage.

3.3.2 Accelerate Termination

3.3.2.1 Determining the *end*

The main performance bottleneck of the Karp-Miller procedure, comes from the redundant explorations on the nodes that with labels providing the same information on coverability: some label would be discovered multiple times in nodes from different paths. To eliminate redundant exploration, we extend the criteria for determining terminal of the tree from existence of discovered node along the same path that has the same label with the *new* node, to the existence of discovered node anywhere in the tree that has label *covers* (note, the relation *covers* is true true for two identical labels in both sides) the label of the *new* node. Formally, we replace rule 4.a of the classic algorithm with the followings:

4.a Suppose $\mathcal{R}$ is the set of configurations that have been discovered so far (i.e. there are paths from d to these configurations), $\mathcal{C}$ is a subset of $\mathcal{R}$ and $\mathcal{C}$ includes only the necessary configurations that represent the same coverability as $\mathcal{R}$ does, i.e. for all $r \in \mathcal{R}$, there exists $c \in \mathcal{C}$ s.t. c covers r; and for all $c \in \mathcal{C}$, there exists no such $c' \in \mathcal{C}$ that c covers c'. Then the current node in the tree η is considered as an *end* if there exists $c \in \mathcal{C}$ s.t. c covers $l(\eta)$.

That means, when a new node η is found, our algorithm first checks whether some previously discovered node anywhere in the tree *covers* η; if so, η is discarded.

Conversely, the algorithm checks whether η covers some previously discovered node θ that is yet to be expanded; if so, θ can be discarded from the *Unexplored* list. The reason is a monotonicity [3] argument: adding threads to a global state, such as represented by θ, can at most increase the set of global states reachable from it. We can think of η *subsuming* θ. In both directions, we only need to compare θ against maximal candidates.

To make the search for redundant nodes efficient, our implementation keeps a separate copy of those discovered nodes that are *maximal* with respect to the *covers* relation as partial order (i.e. the set $\mathcal{C}$ mentioned in the new 4.a rule). New labels are compared against these maximal nodes only. In particular, this subset of the discovered nodes forms an anti-chain in the *covers* partial order.

Note that — in contrast with an approach by Finkel [23] — we discard unexplored nodes instead of entire subtrees. Our algorithm therefore does not suffer from the problems that make Finkel's method incomplete (see also [26]).

3.3.2.2 Exploring Strategy

After applying the optimization described previously, the procedure of determining the *end* of a particular path in the tree is not independent from other discovered paths anymore. This fact creates chance for us to construct strategies making the exploration procedure terminates earlier.

[3] Monotonicity in VASS means: for all configurations $c_1, c_2, c_3 \in \mathcal{S} \times \mathbb{N}^m$, there exists $c_4 \in \mathcal{S} \times \mathbb{N}^m$ reachable from c_3 s.t. c_4 covers c_2, if c_3 covers c_1.

By observing the generated tree for the VASSs, we found a lot of configurations (i.e. labels) that *covers* some configurations that have long subsequent paths, which means we could omit the exploration of the later if we could explore the former first.

So the strategy we use here is: we explore the successors that remain in the same state first, by which the exploration will enjoys a higher probability to reach a the configurations that have the most ability to cover other configurations (which has the same state) which can be reached by other paths.

This strategy can be accomplished simply by implementing the interface *getSuccessors* in the abstract class *AdditionSet* as: while an new configuration is available, we push it to the back of the resulting list (would be expanded first) if it has the same shared state with source configuration, otherwise, push it to the front of the resulting list (would be expanded last).

We note that, this optimization only can help in a *depth-first* implementation since the *breath-first* one expands some node's successors "at the same time", it makes no difference by specifying priority on the order of expanding.

3.3.3 Accelerate the ω procedure

The ω procedure is one of the performance bottleneck in the classic algorithm, the reason are that (i) we have to go backward along the path to determine if any coordinate of the vector should be replaced by the ω symbol and (ii) the paths in the generated Karp-Miller Tree often consist a lot of comparable configurations.

By observing the generated Karp-Miller Tree for the VASSs, we found that many of the paths in the tree alternate nodes with configurations (i.e. the labels) that are comparable (i.e. either can covers the other) and they go very deep generally. That means, in the original ω procedure, we also have to visit the redundant nodes in terms of labeling with configurations that have the same effects for introducing the ω symbol to the counter vector in the *new* node.

By the transitive property which holds in the *covers* relation: $c_1, c_2, c_3 \in \mathcal{S} \times \mathbb{N}^m$, c_3 covers c_1, if c_3 covers c_2 and c_2 covers c_1, here we manipulate a tracing acceleration set to make this procedure much more efficient.

Along the exploration, we manipulate a set $\mathcal{AS}$ that includes all configurations in the current path that do not *covers* any other configuration. i.e. suppose **ps** is the set that includes all configurations in the current path, then for all $c \in \mathcal{AS}$, there exists no $c_1 \in \mathcal{AS}$ or $c_2 \in (\mathbf{ps} - \mathcal{AS})$ s.t. c covers c_1 or c covers c_2. Then we will compare the *new* configuration with the configurations in $\mathcal{AS}$ instead of the labels of all nodes along the path to determinate if some coordinates of the counter vector should be updated with ω.

We note that, this optimization is practical in *depth-first* algorithm only since it trades space efficiency for time efficiency but the *breath-first* version of Karp-Miller procedure is inefficient in terms of space usage.

To implement this accelerating technique, we keeps a separate copy of those discovered nodes that are *minimum* with respect to the *covers* relation as partial order (i.e. the set $\mathcal{AS}$ mentioned above). New labels are compared against these minimum nodes only, for determining the introduction of ω symbol. In particular, this subset of the discovered nodes forms an anti-chain in the *covers* partial order. We note that, in *depth-first* algorithm, the search will backtrack, returning to the most recent node it hasn't finished exploring if it hits a node is an end. Thus, keeping only the set of *minimum* configuration does not work, we have to develop some mechanism that can manipulate these discovered configuration properly. A list nesting with lists of configurations would be one instance:

We keep a list of inner lists that ensures all configurations on the *front* of the inner lists are incomparable to each other, and for each inner list, the configurations are sort with respect to the partial order *covers* and the *minimum* configuration is put on the *front* of the list. While an new configuration is available, we compare it with all *front* of the inner lists, if there exists one which is comparable with it, it will be put into that list with respect to the partial order *covers*, and create a new inner list for the new configuration otherwise. When the searching procedure returns to the most recent discovered node that has not been expanded yet, we remove the configuration that the searching procedure returns from according to the iterator of the configuration (even in absence of the exact position of the inner list) directly, this works because the deletion of element in a list would not violate the structure of the list data since they are linked by pointer (STL's list implementation supports this functionality). Once the node in one of the list is deleted, the clean-up operation

would be needed, e.g. the list become empty, but we do not have to do it immediately, instead, we postpone this operation to the phase of inserting new configuration to these lists, and only if these lists are accessed by the procedure.

We end this section with an introduction of the optimized algorithm (algorithm 3) implemented in *depth-first* fashion.

Algorithm 3 Optimized Karp-Miller Tree Construction in DFS

1: Unexplored := Initial
2: Reached := $\emptyset$
3: **while** Unexplored is not empty **do**
4: pop a config $c := (s, L, f : L \longrightarrow \mathbb{N} \cup \{\omega\})$ from the front of Unexplored
5: **if** there exists no such $r \in$ Reached that r *covers* c **then**
6: **for all** $l \in L$ **do**
7: execute $\mathbb{P}$ on (s, l) to obtain $Successors$
8: **for all** $(s', l') \in Successors$ **do**
9: $L' := (\ f(l) = 1\ ?\ L \setminus \{l\}\ :\ L\ ) \cup \{l'\}$
10: $f'(x) = \begin{cases} f(x) - 1 & \text{if } x = l \\ f(x) + 1 & \text{if } x = l' \wedge l' \in L \\ 1 & \text{if } x = l' \wedge l' \notin L \\ f(x) & \text{otherwise} \end{cases}$
11: $c' :=$ Omega (s', L', f', AS) //introduce ω to the counters
12: append c' to the front of Unexplored

13: **for all** $list \in$ AS **do**
14: **if** c *covers* $list.front$ **then**
15: Comparable = true;
16: list.insert(c) and Break //sorted w.r.t partial order *covers*
17: **if** Comparable == false **then**
18: add c' to new-list;
19: add new-list to AS

20: **for all** $r \in$ Reached **do**
21: **if** c *covers* r **then**
22: remove r from Reached
23: **else if** c *covers* r **then**
24: Discarded = true
25: discard c and Break
26: **if** Discarded == false **then**
27: add c' to Reached;

3.4 Performance

We have implemented our optimized algorithm (algorithm 3) in a tool called STAR ("Scalable Thread-state reachability AnalyzeR"), in an explicit-state fashion. We have evaluated the implementation's performance on a diverse set of Boolean programs. We applied our tool to a large number of examples from two sources: Boolean programs generated by SATABS that abstract part of Linux kernel components, and Boolean programs abstracting Windows device drivers generated at Microsoft Research using SLAM.

Before discuss our experiments, we describe in detail how we obtained *concurrent* benchmarks from the Boolean program source. This step differs for the SLAM-generated programs and those generated with SATABS. For SLAM, we simply instantiate a sequential Boolean program once per thread; each thread executes the program's `main` procedure. Variables with global scope become shared variables of the concurrent programs, while variables with local scope become thread-local variables.

In contrast, the concurrent benchmarks produced by SATABS were generated using DDVERIFY [48], a harness for Linux device drivers. The resulting concurrency model supports synchronization primitives, such as semaphores and spinlocks, and memory-mapped IO-registers for communication with the underlying hardware. Parallel execution is handled as follows. An environment thread models the operating system threads and parallelism is caused by hardware events, e.g., interrupts. Interrupt service routines are themselves uninterruptable, that is, this concurrency model does not cover nested interrupts. The interaction between the driver and a client application is simulated in an infinite loop that nondeterministically calls the driver's functions. This loop is executed by multiple threads, since it accesses to a driver can be shared among multiple clients.

Our Boolean programs have between 17 and 3884 lines of code. The threads communicate using shared variables; the number of shared variables ranges from 1 to more than a thousand, resulting in a huge number of shared states. The experiments were performed on a 2.53 GHz, 4 GB Intel machine, running Linux.

Our experiments are based on a collection of 131 Boolean programs; the programs

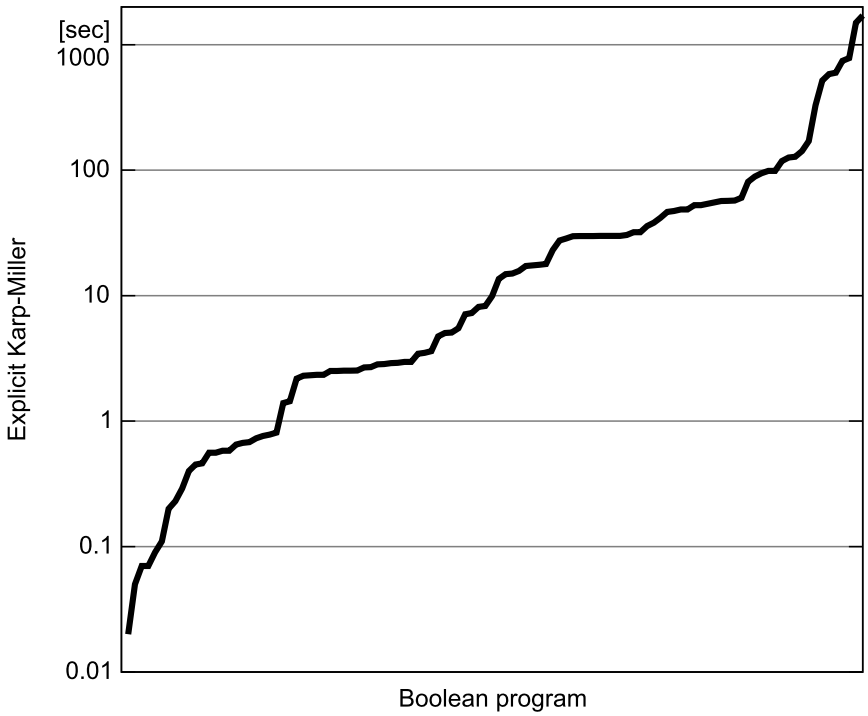


Figure 3.5: Performance of our optimized algorithm on Boolean programs

are listed in Figure 3.5 for increasing running time. We note that our exploration engines cannot cope effectively with the data nondeterminism, represented by the special value $\star$, that Boolean programs exhibit [15]. To make these experiments meaningful, especially for large programs, we determinized the Boolean programs, replaced $\star$ with 0 or 1 randomly, before feeding them into the tool.

In 20 cases, our Karp-Miller engine timed out; these cases are not shown in the Figure 3.5. The running times for the remaining 111 cases range from under 1 s to about 30 min. The number of shared variables in these cases ranges between 0 and 37. The programs without shared variables stem from very coarse Boolean abstractions; there are only 10 such examples (9 %).

Chapter 4

Thread-State Reachability Cutoffs

In the previous chapter, we have seen that the Karp-Miller procedure can effectively be used to decide, for a given Boolean program, the reachability of some thread state. We achieved much better performance than one could perhaps hope for, given the high worst-case complexity of the procedure. In this chapter, we present the contributions about trading completeness for efficiency: we present a method that (i) is more efficient than the Karp-Miller procedure, (ii) is not guaranteed to find all reachable thread states, but (iii) likely does find all reachable thread states for *many* programs. The method can therefore be seen as an efficient, incomplete but often exhaustive reachability analysis.

4.1 Existence of Cutoffs

We begin by observing that the set of reachable thread states is finite. When considering the same Boolean program for increasing numbers of created threads, this set can therefore grow only a finite number of times. This implies that, for every Boolean program $\mathbb{P}$, there is a number n such that any thread state reachable given *some* (arbitrarily large) number of threads can in fact be reached given n threads.

Remark. One can in fact show a slightly stronger result. Let $\mathcal{R}_n^d$ denote the set of thread states reachable up to exploration depth d in an n-thread system. For every program $\mathbb{P}$, there is a number N such that for all $n \geq N$ and for all d, $\mathcal{R}_n^d = \mathcal{R}_N^d$.

We call the smallest such number the *uniform cutoff* of $\mathbb{P}$ and denote it by N_0. It can be shown that $n_0 \leq N_0$, and that $n_0 < N_0$ for many examples. Given at least N_0 threads, the set of thread states reachable at any given exploration depth d is independent of the thread count. In other words, given two numbers $n_1, n_2 \geq N_0$, the two structures derived from replicating $\mathbb{P}$ n_1 or n_2 times respectively, are *bounded-reach equivalent*. This means that the structures satisfy the same formulas expressible in *bounded-reachability logic* [32]. This logic is similar to CTL, but allows only non-nested *reachability* and *bounded-reachability* modalities, $\mathsf{EF}\, P$ and $\mathsf{EF}^{\leq i}\, P$, respectively. *(End of Remark.)*

We call the smallest such number the *reachability cutoff* of $\mathbb{P}$ and denote it by n_0.

If the reachability cutoff of a Boolean program is known, the thread-state reachability problem can be solved by analyzing the program for the fixed cutoff-number of threads, without the Karp-Miller procedure. We show in the rest of this section why that might be useful, and why the observations made above can have a significant impact in practice even if the cutoff is *not* known.

4.2 Reachability Cutoffs in Practice

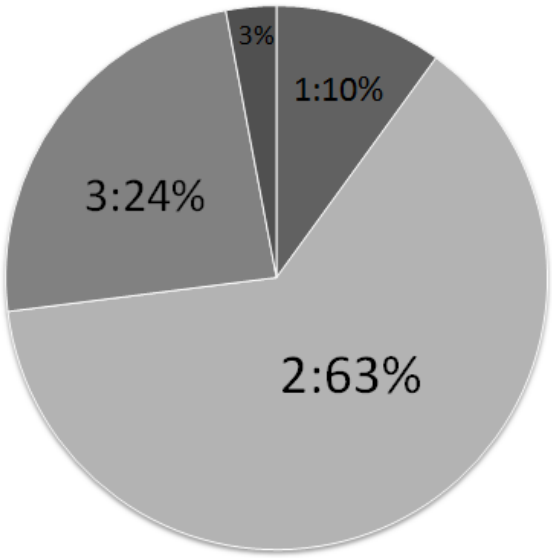


Figure 4.1: The Cutoffs in realistic Boolean programs

The cutoff of a concurrent program can be arbitrarily large. As an example,

given some number c, consider a program that first increments a shared variable s of range $\{0, \ldots, c\}$ and, in the next instruction, throws an exception if s equals c. This program has a reachability (and uniform) cutoff of exactly c.

Such behavior is, however, not typical. Figure 4.1 shows the distribution of the reachability cutoffs on our set of Boolean program benchmarks. The cutoff was computed using our tool STAR: we first determine the exact set of reachable thread states, and then, incrementally, how many threads are necessary to reach these thread states. The latter can be done with any concurrent Boolean program reachability engine requiring a *bound* on the number of threads; we used the model checker BOOM for this purpose [5]. We observe that the cutoff is below 4 in all but very few examples.

4.3 Reachability Cutoffs and the Karp-Miller Construction

In this section we demonstrate empirically that running a bounded-thread reachability analysis engine such as BOOM with the cutoff number of threads tends to be much cheaper than running the full Karp-Miller algorithm. Figures 4.2 compare, for our set of Boolean program benchmarks, the runtime performance of BOOM with the cutoff number of threads to the runtime performance of STAR on the same instance.

We see from the figures that the cutoff-based solution is always faster. The differences in the running times vary, they reach up to 25min. Note that the figures do not contain cases for which Karp-Miller timed out, since for those we cannot compute the exact cutoff. The memory consumption of the two procedures relates as follows: our implementation of the Karp-Miller procedure uses, on average, 3–5 times as much memory as used by the bounded one, and orders of magnitude more in some cases.

The reasons behind these result can be various: recall that for a concurrent system with a fixed number of threads, the number of non-zero counters in a system state is bounded by the number of threads. With dynamic thread creation, there is no such formal bound and we empirically observed an average of 21 and 94 at most, as Figure 3.2 shows. Although it remains to be a tiny fraction of the number of all local states, it is many times of the reachability cutoff n_0 in the realistic concurrent Boolean program, thus, the number of nodes in the Karp-Miller Tree grows much fast

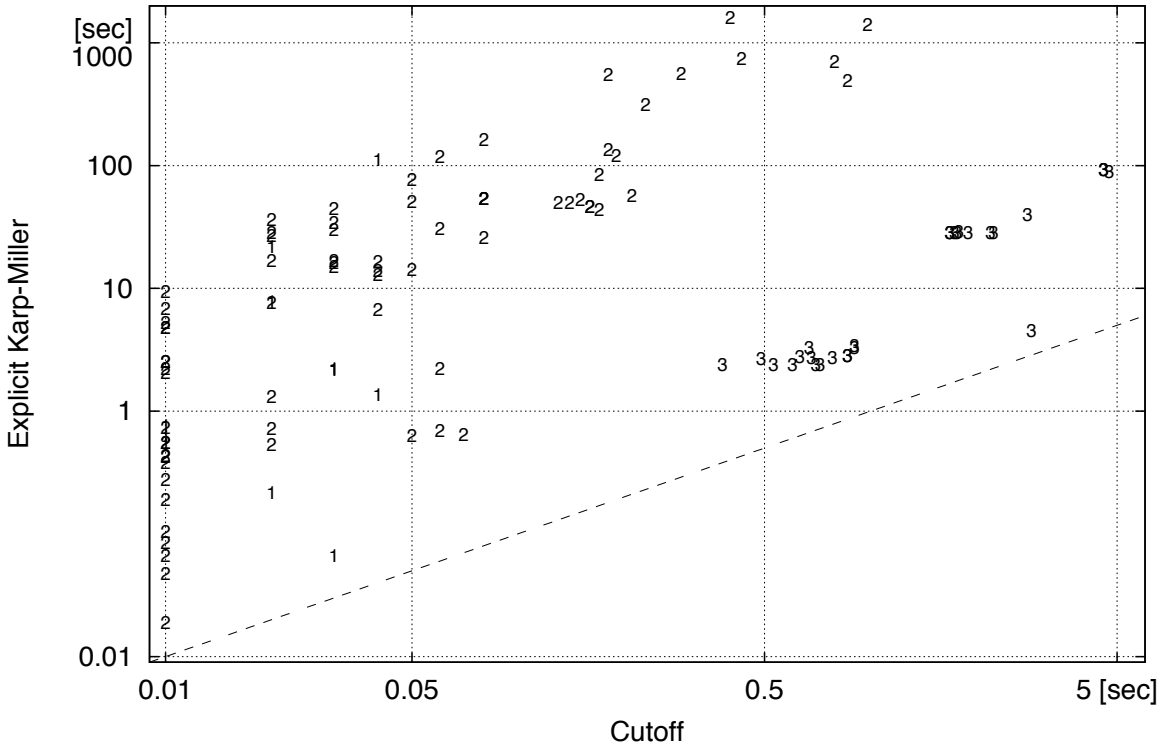


Figure 4.2: Comparing Karp-Miller and Cutoffs

than the nodes generated by the bounded-thread procedure. Moreover, most of the nodes generated by the former are redundant in term of leading to the same set of successors since we only need n_0 number of thread to discover all reachable thread states but the Karp-Miller procedure explores the program as expensive as using 21 in average threads to explore the same program. The additional cost of the Karp-Miller procedure, compared to one that explores states for a bounded number of threads, also comes from the need to check labels of new nodes for coverage against labels of previously discovered nodes although with the improvement we did in chapter 3.

The consequences of our findings are two-fold. First, if the cutoff can be pre-computed or at least tightly estimated, computing the reachable thread states until the cutoff bound is likely very efficient and gives the same precise result as the Karp-Miller procedure. Second, if the cutoff is not known, one can still capture a large fraction of the reachable thread states by analyzing the program for a fixed and small number of threads, say one that terminates before a reasonable time-out. The result is an efficient and highly effective bug-finding strategy.

Folklore tells us that often few threads suffice to exhibit all relevant concurrent

behavior that may lead to a bug. To quote from an early work by Clarke, Grumberg and Browne [12]: *"It is easy to contrive an example in which some pathological behavior only occurs when, say, 100 processes are connected together. ... Nevertheless, one has the feeling that in many cases this kind of intuitive reasoning (i.e., inferring correctness for all from correctness for a few) does lead to correct results."* Our work confirms that this "gut-feeling" achieves a high level of precision for selections of (Boolean) programs with a common designation, such as those abstracted from OS kernel components.

Chapter 5

Conclusion

5.1 Related Work

There is a vast amount of literature on tackling reachability analysis for concurrent software, with or without recursion, the entire coverage of it is far beyond the scope of this dissertation. We focus on work related to vector addition systems or the use of cutoffs. We remark in passing that many of the results for vector addition systems can be equivalently applied to, or have been discovered in the context of, *Petri nets*.

Vector Addition Systems: Many data structures and algorithms have been proposed for the efficient analysis and coverability checking of VASS [25, 23, 19]. Most of these algorithms, however, do not address the problem of introducing an intractable number of vector elements by the translation from (Boolean) programs. Thus, in BDD-based implementations, the encoding typically contains one variable per potential vector element.

Pastor and Cortadella provide a better technique that identifies, using linear programming, sets of elements such that only one element will be non-zero at any time [40]. This improvement is based on a-priori estimates on the set of reachable configurations. In contrast, our solution is dynamic in nature; the vector dimension varies from state to state, allowing for a much more aggressive compression. Recent work by Raskin *et al.* has attempted to address the dimensionality problem using an abstraction refinement loop [24], where abstract models of the Petri net under

investigation are of lower dimension than the original. When spurious results occur, the abstraction is refined, leading to increased dimensions. In contrast to our solution, this obviously suffers from the necessity to go through several iterations before finding a dimension that is "precise enough".

General techniques to reduce the time and memory requirements of Petri net analysis are based on different choices of data structures such as *multi-valued decision diagrams* [10] and *Data Decision Diagrams* [17], or based on algorithmic improvements such as symmetry reduction [46], deleting configurations that will not be seen again [9], or partial-order techniques [22].

Cutoffs: There is much of the work on verifying concurrent programs using cutoffs restricts communication [12, 21, 8, &c.]. For example, fixed cutoffs are known for ring networks communicating only by token passing [11]. Recently, multi-threaded programs communicating solely using locks were investigated by Kahlon, Gupta and Ivančić [35]. Such programs provably permit very small cutoffs, depending on the property to be checked. These results do not hold with general shared-variable concurrency, as we consider it.

Bingham presents a cutoff-like technique for coverability [6]. 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.

Tools: There are many tools available for the analysis of VASS [30]. The PEP tool is a popular coverability checker [29] 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. [25]. Furthermore, VASS analysis has been applied to Java programs [18] and Boolean programs [2]. These tools compile their input into an explicit-state representation of the underlying program, which may result in

very high-dimensional VASSs. They are therefore susceptible to the local state-space explosion problem.

Other. Similarly in spirit to our cutoff work, Qadeer and Rehof propose an algorithm that focuses on finding bugs, rather than proving correctness. Their algorithm limits the number of context switches between threads, thus omitting some interleavings [42, &c.]. The authors argue for the high coverage of their method in practice. In contrast to our work, the reason for introducing the context bound is to make the problem decidable, as they permit recursion. Due to the undecidability of the problem with an *unbounded* number of context-switches, the authors cannot automatically verify for a large number of samples that a certain context bound indeed catches all bugs.

5.2 Conclusion

In this dissertation, we have investigated the *thread-state reachability* problem for concurrent systems that originate from replicating a given finite-state program with *unbounded* number of threads. This problem is algorithmically decidable by a reduction to the coverability problem of VASS. We have presented a new application of the idea of this reduction — the construction of what has become known as the *Karp-Miller tree* — directly to Boolean programs. Our experimental results demonstrate the effectiveness of our implementation, which is much more efficient than one could perhaps hope for, given the high worst-case complexity in the procedure.

In order to explore a potentially yet more efficient solution to this problem, we also investigated the *reachability cutoff* of concurrent Boolean programs. We have demonstrated empirically that this cutoff tends to be *very small* in practice. In fact, it is often so small that running a fixed-thread model checker with the cutoff number of threads is more, sometimes much more, efficient than building the Karp-Miller tree. Even without knowing the exact cutoff, our findings therefore suggest a frequently exhaustive reachability method.

In future work we will investigate methods to statically predict, or tightly overapproximate, the cutoff for certain classes of programs. Alternatively, one may attempt

to *dynamically* recognize that it has been reached.

Bibliography

[1] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters (IPL)*, 1986.

[2] Thomas Ball, Sagar Chaki, and Sriram Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, 2001.

[3] Thomas Ball and Sriram Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking of Software (SPIN)*, 2000.

[4] Thomas Ball and Sriram Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, 2002.

[5] Gerard Basler, Daniel Kroening, Michele Mazzucchi, and Thomas Wahl. Symbolic counter abstraction for concurrent software. In *Computer-Aided Verification (CAV)*, 2009.

[6] Jesse Bingham. A new approach to upward-closed set backward reachability analysis. *Electronic Notes in Theoretical Computer Science*, 2005.

[7] Ahmed Bouajjani and Javier Esparza. Rewriting models of boolean programs. In *Term Rewriting and Applications (RTA)*, 2006.

[8] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. *CONCUR 2005 - Concurrency Theory*, 2005.

[9] Soren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In *TACAS*, 2001.

[10] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state space construction for asynchronous systems. In *Applications and Theory of Petri Nets (ATPN)*, 2000.

[11] Edmund Clarke and Orna Grumberg. Reasoning about rings. In *Principles of Programming Languages (POPL)*, 1995.

[12] Edmund Clarke, Orna Grumberg, and Michael Browne. Reasoning about networks with many identical finite-state processes. In *Principles of Distributed Computing (PODC)*, 1986.

[13] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification (CAV)*, 2000.

[14] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[15] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous Boolean programs. In *Model Checking of Software (SPIN)*, 2005.

[16] Byron Cook, Daniel Kroening, and Natasha Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Computer Science (TCS)*, 2007.

[17] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuelle Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for Petri net analysis. In *Applications and Theory of Petri Nets (ATPN)*, 2002.

[18] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, 2002.

[19] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Covering sharing trees: a compact data structure for parameterized verification. *Software Tools for Technology Transfer (STTT)*, 2004.

[20] Alastair Donaldson and Alice Miller. Symmetry reduction for probabilistic model checking using generic representatives. In *Automated Technology for Verification and Analysis (ATVA)*, 2006.

[21] Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Computer-Aided Design (CAD)*, 2000.

[22] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[23] Alain Finkel. The minimal coverability graph for Petri nets. In *Applications and Theory of Petri Nets (ATPN)*, 1993.

[24] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: Automatic abstraction refinement for Petri nets. *Fundam. Inf.*, 2008.

[25] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge and check... made efficient. In *Computer Aided Verification (CAV)*, 2005.

[26] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for Petri nets. In *Automated Technology for Verification and Analysis (ATVA)*, 2007.

[27] Steven German and Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 1992.

[28] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV)*, 1997.

[29] Bernd Grahlmann and Eike Best. PEP – more than a Petri net tool. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1996.

[30] Frank Heitmann and Daniel Moldt. Petri net tool database. Available from `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.

[31] Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Programming Language Design and Implementation (PLDI)*, 2004.

[32] Thomas Henzinger, Rupak Majumdar, and Jean-François Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Log.*, 2005.

[33] John E. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. Technical report, Ithaca, NY, USA, 1976.

[34] Vineet Kahlon and Aarti Gupta. On the analysis of interacting pushdown systems. In *Principles of Programming Languages (POPL)*, 2007.

[35] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *Computer-Aided Verification (CAV)*, 2005.

[36] Richard Karp and Raymond Miller. Parallel program schemata. *Computer and System Sciences*, 1969.

[37] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1995.

[38] Shuvendu K Lahiri, Randal Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Computer-Aided Verification (CAV)*, 2003.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[40] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of petri nets. In *Design, automation and test in Europe (DATE)*, 1998.

[41] Amir Pnueli, Jessie Xu, and Leonore Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Computer-Aided Verification (CAV)*, 2002.

[42] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[43] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science (TCS)*, 1978.

[44] Ganesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

[45] Martin Rinard. Analysis of multithreaded programs. In *Static Analysis Symposium (SAS)*, 2001.

[46] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.*, 1991.

[47] Ou Wei, Arie Gurfinkel, and Marsha Chechik. Identification and counter abstraction for full virtual symmetry. In *CHARME*, 2005.

[48] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *Automated Software Engineering (ASE)*, 2007.