

DISS. ETH Nr. 18870

Model Checking Boolean Programs

ABHANDLUNG
zur Erlangung des Titels

DOKTOR DER WISSENSCHAFTEN

der

ETH ZÜRICH

vorgelegt von

Gérard Charly Basler
Dipl. El.-Ing. ETH

geboren am 28. Oktober 1978

von Aadorf/TG

Angenommen auf Antrag von

Prof. Dr. Juraj Hromkovič,
Dr. Daniel Kröning,
Prof. Dr. Javier Esparza, and
Dr. Tayssir Touili

2010

© Gérard Charly Basler, 2010

ISBN 978-0-557-76719-9

Abstract

The reliability of software is crucial to the functioning of today's world, which heavily depends on computer systems. Given the ever increasing complexity of software, bugs are subtle and thus hard to find with manual inspection. A more promising approach is to use a formal method such as *model checking*, which employs exhaustive state-space search to ensure the correctness of the software. However, existing tools typically feature two flaws: either they have insufficient capacity to handle large programs or they report many false positives.

A successful approach to push the boundaries of model checking is *predicate abstraction (-refinement)*. With this method, an abstraction of a program in a high-level programming language is constructed using predicates and represented as a *Boolean program*. It is analyzed by a dedicated checker to determine if an error state is reachable. Boolean program verification remains, despite the reduced state space, the bottleneck within the automated abstraction-refinement loop.

This thesis introduces techniques for efficient reachability analysis of sequential and concurrent Boolean programs. We improve on known summarization algorithms for sequential Boolean programs and propose over-approximations of procedure calls. For non-recursive concurrent Boolean programs with *bounded* thread creation, we first introduce a transformation to a representation based on thread-state counters, which exploits the symmetry inherent in replicated programs. In a second step, we combine this representation with a method to locate thread creating cycles in order to allow an *unbounded* number of threads.

The algorithms are implemented in BOOM, a model checking platform for Boolean programs. It is the first tool which implements a summarizing reachability engine that entirely relies on a satisfiability solver for (quantified) Boolean formulæ. The concurrent variant of BOOM is implemented using binary decision diagrams and includes partial order reduction methods. BOOM has been used in combination with the SATABS software model checker to verify safety properties of Linux device drivers. BOOM is, to the best of our knowledge, the first efficient model checker for non-recursive concurrent Boolean programs.

Zusammenfassung

In der heutigen modernen Gesellschaft, die sich zu einem grossen Teil auf den Einsatz von Computern stützt, ist die Zuverlässigkeit der auf diesen Computern eingesetzten Software von kritischer Bedeutung. Im Widerspruch dazu führt die ständig wachsende Komplexität von Computersystemen zu immer subtileren Fehlern, deren Beseitigung mit manuellen Methoden nahezu aussichtslos ist. In vielen Bereichen ist jedoch eine fehlerfreie Software unabdingbar, z.B. medizinische Apparaturen, eingebettete Systeme in der Luft- und Raumfahrt, der Automobilindustrie usw.

Einen viel versprechenden alternativen Ansatz bieten formale Methoden wie das *Modelchecking*. Hierbei werden alle erreichbaren Systemzustände auf Fehler durchsucht, so dass die Korrektheit indirekt durch die Abwesenheit unerwünschter Zustände garantiert werden kann. Typischerweise sind die meisten verfügbaren Tools jedoch nicht in der Lage, grosse Programme zu untersuchen, oder leiden unter zu vielen Falschmeldungen.

Ein in den letzten Jahren sehr erfolgreicher Weg, die Kapazität des Modelchecking zu erweitern, ist *Predicate Abstraction*. Bei dieser Methode wird ein Programm aus einer höheren Programmiersprache mittels Prädikaten in die Form eines vereinfachten und approximierenden Bool'schen Programms überführt. Das Bool'sche Programm wird anschliessend von einem dezidierten Hilfsprogramm auf die Erreichbarkeit von fehlerhaften Zuständen untersucht. Diese Suche ist, trotz des reduzierten Zustandsraumes, der Engpass innerhalb der automatisierten Abstraction-Refinement Schleife.

Die vorliegende Arbeit stellt neue Techniken zur effizienten Erreichbarkeitsanalyse von sequentiellen und parallelen Bool'schen Programmen vor. Wir verbessern bekannte „Summarization“-Algorithmen für sequentielle Bool'sche Programme und präsentieren eine Überapproximation von Prozeduraufrufen. Für den Fall nichtrekursiver Bool'scher Programme mit *endlicher* dynamischer Erzeugung von Threads beschreiben wir eine Transformation in eine Symmetrie ausnützende Darstellung, welche auf einer Abzählung der Threadzustände basiert. Wir erweitern diese Darstellung so, dass Thread-erzeugende Schleifen erkannt und damit eine *unendliche* Anzahl von Threads behandelt werden können.

Alle vorgestellten Algorithmen sind implementiert in BOOM, einer Modelchecking-Plattform für Bool'sche Programme. BOOM ist das erste Tool, das Erreichbarkeitsanalyse auf Basis des Summarization-Algorithmus' ausschliesslich mit Hilfe von SAT/QBF-Solovern ermöglicht. Die parallele Variante von BOOM baut auf BDDs auf und beinhaltet sog. Partial-Order Reduktionsmethoden. BOOM wurde im Rahmen des SATABS-Frameworks zur Verifikation von Gerätetreibern unter LINUX verwendet. BOOM ist, unseres Wissens, der erste effiziente Modelchecker für nichtrekursive parallele Bool'sche Programme.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Daniel Kroening, for his guidance and support without which this dissertation would not have been possible. He gave me the opportunity to work on a fascinating topic after one and a half years routine work in industry. He was always there when I had questions and a source of new ideas whenever I hit a wall with mine.

I would also like to thank Juraj Hromkovič for volunteering on short notice to become my official supervisor. I am also grateful to my co-examiners Javier Esparza and Tayssir Touili for their valuable comments on my thesis.

I am deeply indebted to Thomas Wahl, who took on the role of an unofficial co-supervisor. I will never forget the countless and fruitful discussions we had — both scientific and personal. He was the one who pointed me to the idea of applying symmetry reduction to Boolean programs. His persistence and encouragement were essential to the success of this approach.

It was a privilege to work with Madanlal Musuvathi and Shaz Qadeer during my internship at Microsoft Research, Redmond. Madan's supervision and enthusiasm in particular made those three months an unforgettable experience.

I would also like to express my gratitude to Luke Ong, who enabled me to stay at Oxford University at the end of my PhD. I made contact with Oxford's academic spirit in all its aspects and was impressed by what I encountered. In addition to this, my visit proved a good opportunity to keep in touch with the Oxford half of my research group. Ultimately, the people I met made my Oxford days an amazing experience.

I would like to use this opportunity to thank individual members of my group. I was very fortunate to have Georg Weissenbacher on board — without him the first two publications and the first year of my PhD in general would have been a lot harder! I hope he will forgive me for not being able to enrich his repertoire of Swiss jokes about Austrians. Working together with Michele Mazzucchi on the CAV paper was a great pleasure; we not only had many interesting discussions, but also occasional diverging opinions — whenever I wanted to write one sentence he wanted to write five and remove half of the information. Fortunately, we always managed to reach a good Swiss-Italian compromise in the end. I would also like to thank Vijay D'Silva for all the time spent in answering my countless questions and giving me a push in the right direction whenever I needed one. A warm thank you also to Yury Chebiryak, my office mate, for his friendship and the good times we spent together on countless occasions. I am very grateful for the nice times I had with the other colleagues of my group among them were Christoph Wintersteiger, Mitra Purandare, Nicolas Blanc, Angelo Brillout, Leo Haller and Alexander Kaiser. Our group spirit certainly was one of a kind, and during the last few years many co-workers have become good friends.

One person I would certainly not like to forget is Cathy Zhuang. I would like to thank her for her heroic efforts made while proofreading my thesis.

Last but not least, I would like to thank my family for their love and constant support. I am forever indebted to my parents, who encouraged me to pursue my own way and never questioned my decisions — without them I would not be where I am now.¹

Zürich in September 2010.
Gérard Basler

¹It was my mother who kindled my undying interest in computers when she presented me with my first computer when I was eleven.

Contents

1	Introduction	1
1.1	Problem Definition	1
1.2	Contributions	2
1.3	Outline	3
2	Background	5
2.1	Model Checking	5
2.2	Binary Decision Diagrams	6
2.3	SAT	7
2.4	Bounded Model Checking	9
2.5	k-Induction	10
2.6	Predicate Abstraction	11
	Sequential Boolean Programs	15
3	Syntax and Semantics of Sequential Boolean Programs	17
3.1	Semantics of Sequential Boolean Programs	17
3.2	Symbolic Simulation	20
3.3	Pushdown Systems	22
4	Model Checking Sequential Boolean Programs	27
4.1	Summarization of Boolean Programs	28
4.2	Universal Summaries	31
4.3	Structural Abstraction with Summaries	35
4.4	Recursion	37
4.5	Experiments	39
4.6	Summary	39
4.7	Bibliographic Notes	40
	Concurrent Boolean Programs	43
5	Model Checking Concurrent Boolean Programs	45
5.1	Extensions with Concurrency Primitives	45
5.2	Semantics of the Concurrent Language Extensions	46
5.3	Thread-State Reachability	49
5.4	Overview of Model Checking Methods	49

6	Partial Order Reduction	51
6.1	Introduction	51
6.2	Ample Sets	52
6.3	Symbolic Computation of Persistent Sets	55
6.4	Cartesian Partial Order Reduction	57
7	Concurrent Boolean Programs with Bounded Replication	61
7.1	Introduction	61
7.2	Preliminaries	63
7.3	Classical Counter Abstraction – Merits and Problems	64
7.4	Symbolic Counter Abstraction	65
7.5	Experimental Evaluation	70
7.6	Summary	77
7.7	Bibliographic Notes	77
8	Concurrent Boolean Programs with Unbounded Replication	79
8.1	Introduction	79
8.2	Computational Model	80
8.3	The Karp-Miller Coverability Tree	81
8.4	Thread-State Reachability in Boolean Programs	83
8.5	Summary	88
8.6	Bibliographic Notes	88
9	Thread-State Reachability Cutoffs	91
9.1	Existence of Cutoffs	91
9.2	Reachability Cutoffs in Practice	92
9.3	Reachability Cutoffs and the Karp-Miller Construction	92
9.4	Summary	93
9.5	Bibliographic Notes	94
	Implementation	95
10	The Model Checker BOOM	97
10.1	Brief History of BOOM	97
10.2	Explicit vs. Symbolic Program Counters	98
10.3	State Representation	100
10.4	Path-Wise Unrolling	102
10.5	State Merging	103
10.6	Organization of Work-lists	104
10.7	Optimizations for the Karp-Miller Coverability Tree	105
10.8	Handling of Non-determinism in Expressions	106
10.9	Dead Variable Optimization	108
10.10A	GUI for Analyzing Boolean programs	110
11	Conclusions	113
11.1	Summary of Contributions	113
11.2	Future Work	114
	Appendices	119

A	Syntax of Boolean Programs and Boolean Program Examples	119
A.1	Syntax of Concurrent Boolean Programs	120
A.2	Original Bluetooth Driver Example	121
A.3	Converted Bluetooth Driver Example	123
B	Experimental Setup for Concurrent Benchmarks	125
C	Interprocedural Data-Flow Analysis	127
	Bibliography	129
	Index	139
	Curriculum Vitæ	141

The problem with engineers is that they cheat in order to get results. The problem with mathematicians is that they work on toy problems in order to get results. The problem with program verifiers is that they cheat on toy problems in order to get results.”
— Anonymous

1

Introduction

COMPUTERS are omnipresent in daily lives and control not only simple coffee machines but also airplane navigation systems, car brakes, and medical equipment. If such systems malfunction, then they can severely harm humans. Their behavior depends to a large extent on the software that they are running. The size and complexity of such software is ever increasing and manual code inspection or hand written tests can never assure the level of quality that is needed. As a consequence, methodologies for developing software of high quality are an important research area in computer science.

Formal verification applies mathematical reasoning to software in order to prove its correct behavior. The desired behavior of a program can be described by a formal specification, and formal methods can be used to prove if a program meets this specification. However, approaches for ensuring reliability of software must be fully automated, as tools that require a substantial manual effort are not accepted by programmers. In fact, many successful tools, e.g., SLAM [10], BLAST [23] or FINDBUGS [5], work by literally just “pushing a button”.

Unfortunately, it is impossible to build a tool that is able to determine if an arbitrary property holds for every program, since program verification is undecidable in general. Thus, these tools only answer simple questions. Since programmers also do not like to write specifications (which are error-prone as well), the properties that are checked, are often pre-defined by the tool creator. SATABS, for example, is limited to verifying the absence of “bad” actions that should never occur in a correct program, such as division by zero, wrong pointer manipulations, buffer overflows or violated assertions.

1.1 Problem Definition

Real-world programs have many artifacts such as stacks and heaps, which make direct verification of the original code very hard. Thus, a key concept is to extract an *abstract model* that enables tractable verification. The idea behind is to reduce the complexity of the original program by focusing on only

the relevant details needed to prove a property.

Predicate abstraction [80] is a widely-used framework for (automatically) constructing such abstract models from C programs. Promoted by the success of the SLAM project, Boolean programs are a frequently used model for software programs. A model checker analyzes the Boolean program and determines if it adheres to its specification by performing an exhaustive state-space search. Checking the Boolean program model is the bottleneck of this method, as it suffers from the state explosion problem, and that is the point where this thesis comes into play. The main contribution of this thesis are new algorithms and improvements on existing algorithms for assertion checking of Boolean programs. Regarding sequential recursive Boolean programs, there exists no model checker that is 1) optimized for the most common case, that is the assertion in the Boolean program is violated, and 2) makes use of modern SAT- and QBF-solver techniques. Microsoft's SLAM project has been applied successfully to sequential Windows driver code but it lacks support for concurrency. Hitherto BOPPO [96] is the only dedicated model checker for concurrent Boolean programs. However, BOPPO does not take advantage of symmetry in concurrent Boolean programs. There are also translators into other input languages available, e.g., CBP2GM [57] or reduction techniques to sequential Boolean programs such as [132].

At this point we would like to refer to the quote cited at the beginning of this chapter. Its reproach that formal methods deal only with toy examples is, at least for the case of Boolean programs, not true. The aim of this thesis is the search for methods and tools to tackle real-world problems. However, developing a verifier for programs written in a high-level language is very tough. Thus, we confine our algorithms to Boolean programs only but they are applicable to more powerful models as well. Furthermore, the Boolean programs, which we analyze with our implementation, are generated by SATABS or SLAM from real device driver code of the Linux (or Windows) operating system.

In summary, this thesis seeks to accomplish the following goal. *We develop practical and exact methods for assertion checking of sequential or concurrent Boolean programs.*

1.2 Contributions

BOOM Model Checking Framework The main deliverable of this dissertation is a tool called BOOM for reachability analysis of Boolean programs. It encompasses SAT- and BDD-based algorithms applicable for sequential as well as concurrent Boolean programs. BOOM has been integrated in the SATABS-framework and outperformed well-known tools such as SMV [106] or BOPPO in a study on LINUX driver verification [21]. BOOM is built in a modular way, and supports different reachability algorithms. Its modularity allows re-use of core components, such as partial-order reduction and trace generation.

Algorithms for Reachability Checking of Sequential Boolean Programs We improve the known algorithms for *summarizing* sequential recursive Boolean programs and explain how a purely SAT/QBF-based implementation is realized. Our techniques are tailored to checking automatically generated Boolean programs, which often feature many small functions: we introduce *universal summaries* that capture the side-effects of a procedure for any input. We propose a method based on universal summaries that marks the first *complete* algorithm for reachability checking of Boolean programs without the help of a QBF-solver or BDDs. In addition, we introduce *structural abstraction* for procedures — our method takes abstraction and refinement to the Boolean program level.

Algorithms for Reachability Checking of Concurrent Boolean Programs We show how to apply *counter abstraction* to concurrent non-recursive Boolean programs with a bounded number of threads

to factor out redundancy due to thread replication. The traditional global state representation as a vector of local states is replaced by a vector of thread counters, one per local state. In practice, however, straightforward implementations of this idea are unfavorably sensitive to the number of local states. We present a novel symbolic exploration algorithm that avoids this problem by carefully scheduling which counters to track at any moment during the search. In follow-up work, we present an improved algorithm that does not bound the number of threads in order to check *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. The result is an assertion checking engine for non-recursive concurrent Boolean programs with dynamic thread creation.

1.3 Outline

Chapter 2 provides background on the problems we address in this thesis, the utilities we use to solve them, and the general setting of the Boolean programs.

That subsequent chapters are broken into three parts:

Part I describes sequential Boolean programs. Their syntax and semantics are introduced in Chapter 3, and the algorithms for their analysis follow in Chapter 4.

Part II gives detailed account of concurrent Boolean programs. Chapter 5 explains the extended syntax and semantics of concurrent Boolean programs. Chapter 6 describes how partial-order reduction and symbolic reachability analysis of concurrent Boolean programs can be combined. Chapter 7 proposes an algorithm to tackle the local state explosion of non-recursive Boolean programs with a thread bounded. This bound is dropped in Chapter 8, where we extend the previous algorithm to handle unbounded non-recursive Boolean programs. Chapter 9 provides empirical evidence, that the behavior of concurrent Boolean programs can often be simulated by a small number of threads.

Part III contains the practical part of our work: Chapter 10 describes the implementation of the reachability checker BOOM and important details that were necessary in order to obtain an efficient tool.

Last but not least, Chapter 11 concludes the dissertation with a summary of the main contributions, and points out areas of further research.

2

Background

2.1 Model Checking

MODEL CHECKING [40, 6] has been a vibrant branch of research in formal methods for many years. It is an automated technique for verifying correctness specifications of finite or infinite-state models of systems. Given a system model, a *model checker* systematically examines whether a property in temporal logic holds for all reachable system states. Usually all reachable states are explored in a brute-force manner until a state is found that violates the property, or it holds for any reachable state. In the former case the model checker provides evidence in form of a *counterexample*. If the property under consideration is given in linear temporal logic (LTL), the counterexample describes an execution path¹ from initial state to a system state exhibiting the violation [45]. The ability to generate a counterexample is one of the main advantages of model checking, since it can be replayed in a debugger and facilitates locating the cause of a bug. Compared with testing, it can reveal subtle errors that would have remain undiscovered otherwise. Its limiting factor is the exponential growth of the state space, known as the *state explosion* problem, but — given sufficient resources — model checking is a *complete* method when applied to finite state models.

Finite state systems are well-suited for modeling hardware. Software, however, manipulates dynamic data structures, features procedures, which allow unbounded call stacks, and dynamically creates new processes. For this reason, finite state abstractions are often used to over- or under-approximate the behavior of software.

In practice, the latter method is often used as an efficient *bug finding* technique. For example, empirical studies show that many bugs emerge with a relatively small number of context switches [119].

Pre-and postconditions and invariants of high-level programming languages are properties that can be verified automatically by tools. These properties are also called *safety* properties. The transition systems that are considered in this thesis are generated by automated abstraction engines, that generate

¹Finite paths are also called *traces* in the literature.

error locations for violated safety properties. Thus, the task that a model checker for such abstract transition systems faces, is a question of *reachability* of system states. For the remainder of this thesis we will stick to *reachability* analysis.

Before we can formalize the model checking problem, that is being discussed in this thesis, we need to give our definition of a finite-state system:

Definition 2.1 (Finite-State Transition System) A Finite-State Transition System $\mathcal{M} = \langle S, T, I \rangle$ is defined by a finite set of states S , a transition relation $T \subseteq S \times S$ and a set of initial states $I \subseteq S$.

Now we are able to state formally the model checking problem, that is covered in this thesis: given a safety property φ and a model \mathcal{M} , decide if φ holds in \mathcal{M} , written as $\mathcal{M} \models \varphi$. It can be decided using a model checking procedure that starts from the initial states I and enumerates all reachable states in a brute-force manner until either an error state violating φ is found or no new states are encountered. This is called *forward* analysis. A forward state-space exploration engine utilizes an *image*-function to compute the next sets of states from a given current state:

Definition 2.2 (Image Operation) The Image Operation $\text{IMAGE}(S, T)$ calculates successors of a set of states S in one step under the transition relation T .

Algorithm 2.1 shows a straightforward approach to explore all reachable states of \mathcal{M} . We can check on-the-fly if a state, that violates φ , has been found.

Algorithm 2.1 Reachability analysis

<pre> 1: procedure REACHABLE(I, T) 2: $\mathcal{S}_C := \emptyset$ 3: $\mathcal{S}_N := I$ 4: while $\mathcal{S}_C \neq \mathcal{S}_N$ do 5: $\mathcal{S}_C = \mathcal{S}_N$ 6: $\mathcal{S}_N = \mathcal{S}_C \cup \text{IMAGE}(\mathcal{S}_C, T)$ 7: return \mathcal{S}_C </pre>	<pre> ▷ \mathcal{I}: set of initial states, T: transition relation ▷ \mathcal{S}_C: set of current states ▷ \mathcal{S}_N: set of next states </pre>
---	--

2.2 Binary Decision Diagrams

*Reduced ordered binary decision diagrams*² were introduced by Bryant [34] as a canonical and compact representation for Boolean formulae. An ROBDD can be obtained from a sequence of simplifications on similar structure called *ordered decision tree*. A decision tree maps a valuation of its support variables to a Boolean value. It can be computed by descending the tree from the root to a leaf as follows: the left sub-tree of a node corresponds to the sub-formula obtained if the value of the variable labeling the node is *false*, whereas the right sub-tree contains the sub-formula for the case if the variable is *true*. An *ordered* decision tree is a tree whose nodes are labeled in the same order on any path from root to leaf. Figure 2.1 shows the ordered binary decision tree for the formula $a \leftrightarrow b$.

An ROBDD is obtained by repeatedly applying the following two reduction rules on an ordered decision tree until no more simplification is possible:

²In the following chapters, we use the term “BDD” as a shorthand for reduced ordered binary decision diagrams.

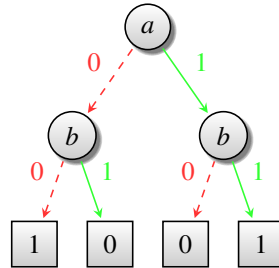


Figure 2.1: Ordered binary decision tree for the function $a \leftrightarrow b$

1. Merge any isomorphic sub-graphs.
2. Eliminate any node whose two children are isomorphic.

The size of the resulting graph is strongly dependent on the order of the variables. Consider, e.g., Figure 2.2, which shows two ROBDDs with different variable ordering for the same function: one is linear in the size of the formula whereas the other is exponential! The dependence of ROBDDs on the ordering of the variables is the major disadvantage of this data structure. Finding an optimal ordering, which yields a smallest BDD for a given function, was shown to be an NP-complete problem [28]. Therefore, heuristic algorithms are used in practice in the hope of reducing the BDD size. A heuristic that works well for many cases is to place dependent variables closely together. Note that there are some functions that result in ROBDDs that are always exponential in the number of variables – regardless of the variable ordering – e.g., the middle output of a combinational circuit of a multiplier.

In practice, the order of the variables is kept the same in any ROBDD existing in the program. As a consequence, the ROBDDs become *canonical*, i.e., any two equivalent formulae are represented by identical ROBDDs. Thus equivalence checking is linear in the size of the ROBDDs. It can be made to run in constant time, if the ROBDD library looks up any ROBDD in a hash table and eliminates copies of the same ROBDD. The interested reader is referred to [135] for further information on the subject of ROBDDs.

McMillan introduced ROBDDs into Model Checking: *symbolic model checking* uses ROBDDs to represent sets of states and their transition function respectively [107]. This technique combines a compact state representation with the ability to explore a set of successor states instead of individual states. Algorithm 2.2 depicts the pseudo-code of a rudimentary symbolic model checker for reachability. It is worth mentioning that BDD libraries (e.g., CUDD [126]) often offer a function that is optimized for image computation: it computes the set of next states as a function of the current state variables and eliminates the current state variables in a single operation (Line 6). The subsequent step renames the next state variables V' to the current state variables V (Line 7) and adds the new states to the already reached states.

2.3 SAT

Satisfiability (SAT) solving is one of the classic NP-complete problems — in fact it was the first decision problem proved to be NP-complete. Given a propositional formula in conjunctive normal form (CNF), a SAT-solver determines if there exists a variable assignment such that the formula evaluates to *true*. If there is no such assignment, the formula is said to be *unsatisfiable*. Propositional formulae that are not in CNF can be transformed into CNF in linear time by introducing auxiliary variables [133].

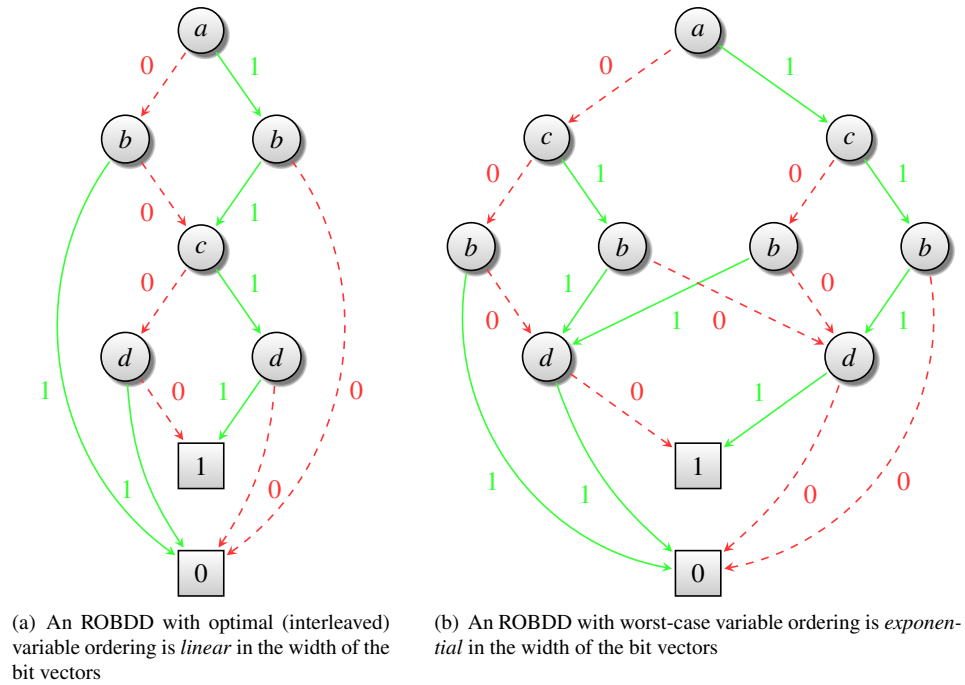


Figure 2.2: ROBDDs for equality of 2-bit vectors: $(a \leftrightarrow b) \wedge (c \leftrightarrow d)$

Algorithm 2.2 Symbolic reachability analysis

<pre> 1: procedure REACHABLE(\mathcal{I}, \mathcal{T}) 2: $\mathcal{S}_C := \text{ROBDD}(\text{false})$ 3: $\mathcal{S}_N := \mathcal{I}$ 4: while $\mathcal{S}_C \neq \mathcal{S}_N$ do 5: $\mathcal{S}_C = \mathcal{S}_N$ 6: $\mathcal{S}'_N = \text{ANDABSTRACT}(\mathcal{S}_C, \mathcal{T}, V)$ 7: $\mathcal{S}_N = \mathcal{S}_C \wedge \text{PERMUTE}(\mathcal{S}'_N, V \leftarrow V')$ return \mathcal{S}_C </pre>	<pre> $\triangleright \mathcal{I}$: ROBDD of initial states, \mathcal{T}: transition relation $\triangleright \text{ROBDD}(\text{false})$: ROBDD representing set of current states $\triangleright \text{ROBDD}$: ROBDD representing set of next states $\triangleright \text{ANDABSTRACT}$: existential quantification $\triangleright \text{PERMUTE}$: renaming </pre>
---	---

The classic approach to solve SAT formulæ comprises recursive backtracking and Boolean constraint propagation [54, 53]. In the late nineties, high-performance SAT-solvers became available (e.g., CHAFF [110]) and the use of SAT to formulate problems became popular. Modern SAT solvers enhance the original DPLL algorithm with a multitude of techniques like non-chronological backtracking, conflict-driven learning and improved decision heuristics. Furthermore, they are tailored to effectively use CPU caches and memory bandwidth. An excellent survey on SAT techniques can be found in [38].

In contrast to BDDs which suffer tremendously from the sensitivity to the ordering of the variables, SAT solvers are a lot less sensitive to problem size and variable ordering.

Quantified Boolean formulæ (QBFs) extend propositional formulæ with both existential quantifiers and universal quantifiers. Their application in Model Checking is twofold:

- BMC can be made *complete* using quantifiers, since the query to check if a Boolean formula represents a subset of the states that have been reached in a previous iteration corresponds to a QBF instance.
- SAT formulæ can be compressed [91] with the help of quantifiers. Thus large QBF formulæ can be held in memory, whereas with a pure SAT encoding the memory limit would be exceeded.

2.4 Bounded Model Checking

Bounded model checking was introduced by Biere [26] as a method for finding logical errors in finite-state transition systems. It is widely regarded as a complementary technique to symbolic BDD-based Model Checking, and frequently used in the hardware industry; see [25] for a survey of experiments conducted with BMC in industry.

Given a finite-state transition system \mathcal{M} , an invariant property φ specified in temporal logic, and a natural number k , a BMC procedure decides whether there exists a sequence of transitions of \mathcal{M} of length k that violates φ . Formally, we write $\mathcal{M} \models_k \varphi$, if all sequences of transitions up to length k satisfy φ . For the purpose of this thesis we shall consider only the linear temporal logic (LTL [62]) property $\mathbf{G}p$: at every state reachable from the initial state (\equiv “globally”) p holds. SAT-based BMC is performed by generating a propositional formula, that is satisfiable if and only if a path from the initial state to a state exhibiting the error exists. Such a formula corresponds to the given transition relation $T(s, s')$ being unwound k times, starting from initial states $I(s_0)$, and a violation of the property after k steps:

$$I(s_0) \wedge T(s_0, s_1) \wedge \dots \wedge T(s_{k-1}, s_k) \wedge \neg\varphi(s_k) \quad (2.1)$$

In practice, the application of BMC is typically restricted to the refutation of safety properties, and is conducted in an iterative manner: Starting with a small initial value of k , k is incremented until either 1) an error is found, or 2) the problem becomes intractable due to the complexity of solving the corresponding SAT instance.

Bounded Model Checking is *complete* iff k reaches a *completeness threshold* CT , which indicates that if there exists no path of length k in \mathcal{M} that violates φ , then there exists no path of any length in \mathcal{M} that violates φ .

Definition 2.3 (Completeness Threshold [97]) A completeness threshold of a transition system \mathcal{M} with respect to a property φ is any natural number CT such that, given that the property φ is not

violated by any path of length up to CT , then it cannot be violated at all, i.e.,

$$\mathcal{M} \models_{CT} \varphi \implies \mathcal{M} \models \varphi \quad (2.2)$$

holds.

Clearly, if $\mathcal{M} \models \varphi$, then the smallest CT is 0, and otherwise it is equal to the length of the shortest counterexample. This implies that finding the smallest CT is at least as hard as checking $\mathcal{M} \models \varphi$. Consequently, we concentrate on computing an over-approximation of the smallest CT .

The *Reachability Diameter* of a finite-state transition system \mathcal{M} is a completeness threshold for reachability properties of the form $\mathbf{G}p$:

Definition 2.4 (Initialized Reachability Diameter [97]) *Given a finite-state transition system \mathcal{M} , we define the Initialized Reachability Diameter $rd(\mathcal{M})$ of a \mathcal{M} as the minimal number of steps required for reaching all reachable states from the initial state.*

$$rd(\mathcal{M}) \stackrel{def}{=} \min\{k \mid \forall s'_0 \dots s'_{k+1}. I(s'_0) \wedge \bigwedge_{0 \leq i < k} T(s'_i, s'_{i+1}) \implies \exists s_0 \dots s_k. I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge s'_{k+1} = s_k\} \quad (2.3)$$

In the context of hardware designs, the Initialized Reachability Diameter is often called the *sequential depth*.

The *Reachability Recurrence Diameter* of a finite-state transition system \mathcal{M} characterizes a less tighter completeness threshold than the reachability diameter:

Definition 2.5 (Initialized Reachability Recurrence Diameter [97]) *The Initialized Reachability Recurrence Diameter with respect to a finite state transition system $\mathcal{M} = \langle S, T, s_0 \rangle$ is the longest loop-free path in \mathcal{M} starting from the initial states:*

$$rrd(\mathcal{M}) \stackrel{def}{=} \max\{k \mid \exists s_0 \dots s_k. I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge \bigwedge_{0 \leq j < i \leq k} s_j \neq s_i\} \quad (2.4)$$

2.5 k-Induction

One possibility to make BMC complete (i.e., to report that φ holds for any k), is to find an inductive argument. In other words, we prove that, under the assumption that the system satisfies φ for k steps, the property cannot be violated in $k + 1$ steps.

This idea is known under the name of k -induction [124] and works as follows. If the base condition in equation (2.5) is satisfiable, then a counterexample has been found:

$$base(\mathcal{M}, k) = \exists s_0 \dots s_k. I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge \bigvee_{0 \leq i \leq k} \neg \varphi(s_i) \quad (2.5)$$

Otherwise, the induction step is performed:

$$step(\mathcal{M}, k) = \exists s_0 \dots s_{k+1}. \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge \bigwedge_{0 \leq j < i \leq k} s_j \neq s_i \wedge \bigwedge_{0 \leq i \leq k} \neg \varphi(s_i) \wedge \varphi(s_{i+1}) \quad (2.6)$$

Intuitively one attempts to prove that φ holds in any state reachable from any state within $k + 1$ steps, assuming that the property holds in k consecutive states.

Unfortunately some properties are not k -inductive for any k , consider for example the following system:



The induction hypothesis cannot be discharged for an arbitrary number of steps k , because although the property cannot be violated for any number of steps from the initial state, it can be violated within one step from the state in the middle.

Thus in order to obtain a complete method, the authors restrict the feasible paths to paths without loops. The latter notion corresponds to the reachability recurrence diameter, which manifests itself in the middle term in equation (2.6).

2.6 Predicate Abstraction

Predicate abstraction [80, 100], pioneered by the model checking tool SLAM, is one of the most prominent approaches in software model checking. Predicate abstraction results in a *Boolean program* [13], which exclusively uses Boolean variables. These variables of the Boolean program \mathcal{B} correspond to a finite set of predicates over the variables of the original program \mathcal{P} . Each predicate p_i describes an observable “fact” about the program, and the evaluation of a variable b_i in an abstract state determines whether the corresponding fact holds or not. For each instruction of \mathcal{P} , the corresponding abstract transition is constructed independently of the surrounding instructions. Thus, the abstraction step preserves the control flow structure of the original program.

The Boolean program \mathcal{B} is created from the original program \mathcal{P} using *existential abstraction* [43]. This method guarantees that the transition relation of \mathcal{B} is a *conservative* over-approximation of \mathcal{P} . That is, if \mathcal{B} satisfies a temporal safety property φ , then \mathcal{P} satisfies φ as well but the reverse is not true. In other words, the Boolean program \mathcal{B} contains all execution traces of \mathcal{P} , and potentially more. A model checking engine that examines the Boolean program \mathcal{B} could report a counterexample that is feasible in \mathcal{B} but not in \mathcal{P} . This is usually called a *spurious counterexample*. This occurs whenever the abstraction is too coarse, i.e., the Boolean program allows too much behavior. Therefore, *refinement* is performed by adjusting the set of predicates in a way that eliminates the spurious counterexample.

The idea of automatically constructing and refining abstractions has first been presented by Kurshan [98] and has been made known to a wider audience by the success of the SLAM project [14, 52], and Clarke et al. [42], who coined the term counterexample-guided abstraction refinement (CEGAR). It comprises of four phases, namely *abstraction*, *model checking*, *simulation*, and *refinement* (see Figure 2.3). These four phases are executed repeatedly, until either a concrete counterexample is found, or the program is proven correct:

Abstraction Predicate abstraction is used to generate a Boolean program \mathcal{B} of the original program \mathcal{P} .

Model Checking A model checker examines the Boolean program \mathcal{B} . If it contains a path that viol-

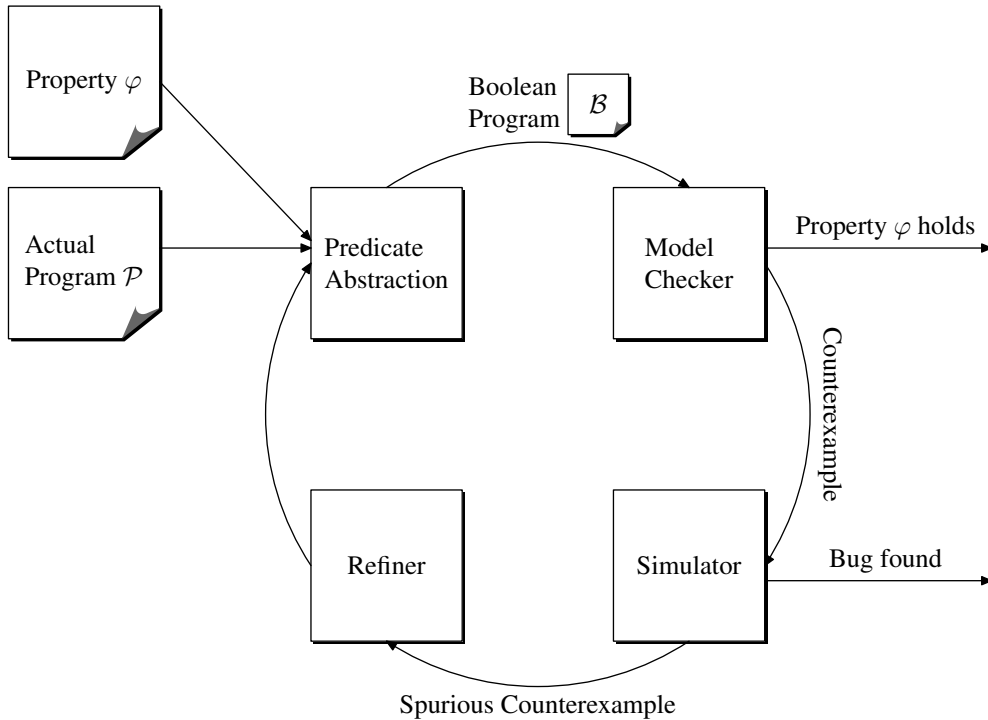


Figure 2.3: CEGAR loop

ates φ , then a counterexample is extracted and the third phase is entered³. Otherwise, φ holds in \mathcal{B} (and therefore also in \mathcal{P}) and the CEGAR loop terminates.

Simulation The feasibility of the counterexample is checked by symbolic simulation. The simulator replays the abstract counterexample on the concrete program. If it corresponds to an actual program behavior, then a bug is reported (φ is indeed violated by \mathcal{P}) and the CEGAR loop terminates. Otherwise, a fourth phase is entered.

Refinement The spurious counterexample is ruled out in the Boolean program by enriching the set of predicates and updating the transition relations. Given the more accurate Boolean program, a new iteration of the loop begins again with the first phase.

The CEGAR process can be fully automated: other than the property to be verified, no user interaction is required. In fact, verifiers for Boolean programs have been used successfully to increase the reliability of system-level software such as device drivers with more than 10,000 lines of code [7, 86, 137].

Example 2.1 *In order to get an impression of Boolean programs (a detailed description follows in the next chapter), a C program (see Listing 2.1) and a possible abstraction into a Boolean program (see Listing 2.2) are given. As their name already states, Boolean programs have only variables of Boolean type, e.g., no integer variables or pointers are allowed. The idea is to factor out data complexity such that a tractable model can be produced from a complex original program. Its control-flow is preserved by the abstraction.*

³A violation of φ corresponds to a failed assertion (SATABS) or the reachability of a specific label L (SLAM).

Listing 2.1: C program

```
int i;

i = 0;
while(i < 5) {
  assert(i < 10);
  i++;
};
```

Listing 2.2: Possible abstraction to a Boolean program

```
decl b1; // b1 = {i >= 10}

b1 := F;
while(!b1 ^ *) do
  assert(!b1);
  b1 = b1 ? T : *;
od
```

Example 2.2 *The mixcomwd Linux driver contains 303 lines of code (header files not included). DDVERIFY is a device driver harness that uses SATABS to verify the correct functionality of LINUX device drivers. The piece of code in Listing 2.3 illustrates the initialization routine of the driver.*

Listing 2.3: Initialization routine of the mixcomwd driver for LINUX

```
static int mixcomwd_ioports[] = { 0x180, 0x280, 0x380, 0x000 };
static int watchdog_port;

...

static int __init mixcomwd_init(void)
{
  int i;
  int ret;
  int found=0;

  for (i = 0; !found && mixcomwd_ioports[i] != 0; i++) {
    watchdog_port = mixcomwd_checkcard(mixcomwd_ioports[i]);
    if (watchdog_port) {
      found = 1;
    }
  }
  ...
}

...
```

This code iterates over the elements of a zero-terminated array. Therefore there is no check that ensures that the index variable stays within the boundaries of the array.

Claim 9 (of 171 reported by SATABS) SATABS verifies that no boundary violation can happen:

Claim 9:

```
file mixcomwd.c line 250 function mixcomwd_init
array 'mixcomwd_ioports' upper bound
!(_Bool)found => i < 4
```

*SATABS needs 11 refinement steps to prove that this claim holds. The loop is unrolled 4 times in the 12th iteration and the predicate $\{b57 \triangleq \text{mixcomwd_ioports}[3] == 0\}$ is added to prove that the **assert** statement that corresponds to this claim cannot be violated (see Listing 2.5).*

Listing 2.4: Iteration 11

```

decl b0_i_lt_4; // i < 4
decl b1_found_eq_0; // found == 0
...
decl b7_i_ge_0; // i >= 0
...

void main() begin
L23: b20, b21, b22, b23, b24 := *, 1, *, 1, *;
L24: b20, b22 := b21, b23;

// mixcomwd.c line 52
L33: b14, b15, b17, b18, b32, b33,
    b34, b35, b37, b38, b40, b41 :=
    *, *, *, *, *, *, *, *, *, *
    constrain ('b34) & ('b40) & ('b32);

// mixcomwd.c line 61
L35: b3_watchdog_port_eq_0 := 1;
...
L64: skip; // no predicates changed
L65: if !* then goto L70; fi;
L68: skip; // no predicates changed
L69: goto L65;
L70: skip; // no predicates changed
L71: if !* then goto L77; fi;
L75: skip; // no predicates changed
L76: goto L71;
L77: skip;
...

// mixcomwd.c line 250 mixcomwd_init
L84: b0_i_lt_4, b2, b7_i_ge_0, b8, b14,
    b15, b16, b17, b18, b19, b32, b33, b34,
    b35, b36, b37, b38, b39, b40, b41 :=
    1, 1, 1, 1, *,
    *, 1, *, *, 1, *, *, *,
    *, 1, *, *, 1, *, *;

...
L85: assume (!(b1_found_eq_0) | b7_i_ge_0);

// mixcomwd.c line 250 mixcomwd_init
L86: assert (!(b1_found_eq_0) | b0_i_lt_4);
...
L101: b6 := 0 constrain
    (b16 | b19) & (b40) & (b40);
L102: assume (!* & * & *);
...

// mixcomwd.c line 250 mixcomwd_init
L138: b0_i_lt_4, b2, b7_i_ge_0, b8, b14,
    b15, b16, b17, b18, b19, b32, b33, b34,
    b35, b36, b37, b38, b39, b40, b41 :=
    b2, b16, b8, b19, b18, b17, b36, b37,
    b38, b39, b34, b35, b40, b41, *,
    *, *, *, *, * constrain
    ('b16 | 'b19) & ('b16 | 'b19) &
    ('b40) & ('b40) & ('b36 | 'b39);

// mixcomwd.c line 250 mixcomwd_init
L139: goto L85;
114: return;

end

```

Listing 2.5: Iteration 12

```

decl b0_i_lt_4; // i < 4
decl b1_found_eq_0; // found == 0
...
decl b7_i_ge_0; // i >= 0
...
decl b57; // mixcomwd_ioports[3] == 0
...

void main() begin
L23: b20, b21, b22, b23, b24 := *, 1, *, 1, *;
L24: b20, b22 := b21, b23;

// mixcomwd.c line 52
L33: b14, b15, b17, b18, b32, b33,
    b34, b35, b37, b38, b40, b41,
    b46, b47, b49, b50, b51, b52, b53,
    b54, b55, b56, b57, b58, b59, b60 :=
    *, *, *, *, *,
    *, *, *, *, *,
    *, *, *, 0, 0, 0,
    0, 0, 0, 1, 0, 1, 0
    constrain ('b34) & ('b40) & ('b32);

// mixcomwd.c line 61
L35: b3_watchdog_port_eq_0 := 1;
...
L64: b61_i_lt_10 := 1;
L65: if !b61_i_lt_10 then goto L70; fi;
L68: b61_i_lt_10 := *;
L69: goto L65;
L70: b61_i_lt_10 := 1;
L71: if !b61_i_lt_10 then goto L77; fi;
L75: b61_i_lt_10 := *;
L76: goto L71;
L77: skip;
...

// mixcomwd.c line 250 mixcomwd_init
L84: b0_i_lt_4, b2, b7_i_ge_0, b8, b14,
    b15, b16, b17, b18, b19, b32, b33, b34,
    b35, b36, b37, b38, b39, b40, b41, b45,
    b46, b47, b48, b49, b50 :=
    1, 1, 1, 1, b52,
    b51, 1, b53, b54, 1, *, *, *,
    *, 1, b55, b56, 1, *, *, 0,
    b57, b58, 1, b59, b60;

...
L85: assume (!(b1_found_eq_0) | b7_i_ge_0);

// mixcomwd.c line 250 mixcomwd_init
L86: assert (!(b1_found_eq_0) | b0_i_lt_4);
...
L101: b6, b42, b43, b44 := 0, 1, *, * constrain
    (b16 | b19) & (b40) & (b40);
L102: assume (!b44 & b43 & b42);
...

// mixcomwd.c line 250 mixcomwd_init
L138: b0_i_lt_4, b2, b7_i_ge_0, b8, b14,
    b15, b16, b17, b18, b19, b32, b33, b34,
    b35, b36, b37, b38, b39, b40, b41,
    b45, b46, b47, b48, b49, b50 :=
    b2, b16, b8, b19, b18, b17, b36, b37,
    b38, b39, b34, b35, b40, b41, b45,
    b46, b47, b48, b49, b50,
    *, *, *, *, * constrain
    ('b16 | 'b19) & ('b16 | 'b19) &
    ('b40) & ('b40) & ('b36 | 'b39);

// mixcomwd.c line 250 mixcomwd_init
L139: goto L85;
114: return;

end

```

Figure 2.4: Difference of the Boolean program generated from the *machzwd* Linux driver.

Part I

Sequential Boolean Programs

3

Syntax and Semantics of Sequential Boolean Programs

THIS chapter presents the syntax and semantics of *Boolean programs* [14]. Boolean programs are frequently used as a representation for abstract models for software. They provide the usual control-flow constructs of an imperative language such as C, but variables are restricted to Boolean type. The use of Boolean programs as an abstract model has been promoted by the success of the SLAM project [14]. SLAM verifies control-flow dominated properties of Windows device drivers by abstracting an ANSI-C program into a Boolean program. An effective model checking strategy is obtained by combining *reachability analysis* with an automatic abstraction refinement mechanism. Predicate abstraction (see Section 2.6) is used to construct a *conservative* over-approximation of the original program. Predicate abstraction constructs the abstraction by tracking only certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. In order to preserve the soundness of the analysis, several sources of non-determinism are added to the abstraction: *control-flow*, *data*, and *thread scheduling* non-determinism.

The main advantage of Boolean programs over finite-state transition systems is that their stack allows a precise representation of the behavior of procedure calls, including procedure-local variables and (possibly unbounded) recursive calls. Nevertheless, reachability properties for Boolean programs are decidable [35]: Procedures can access and modify only the topmost element of the stack. Therefore, *summarizing* procedure calls prevents a re-evaluation of a call if the same calling context has already been considered before [123].

3.1 Semantics of Sequential Boolean Programs

We define Boolean programs and their sequential semantics in terms of the control-flow graph of a program. A Boolean program consists of a set of procedures, each of which is represented by its

control-flow graph (CFG). As usual, a control-flow graph is a directed graph with nodes corresponding to program locations. Without loss of generality, we assume that each procedure has exactly one entry node n_i and one exit node n_o .

The node in the CFG, also referred to as the program counter, and the valuation of the variables constitute the state of the program:

Definition 3.1 (Explicit State) *An explicit state of a Boolean program is a tuple $\langle n, \Omega \rangle$, where n identifies the node in the CFG and Ω is a valuation to the variables in scope, i.e., Ω is a function that associates every Boolean variable in scope with a Boolean value.*

Unlike the conventional notion of a program state, a state of a Boolean program \mathcal{B} that is analyzed by a model checker does not contain the contents of the call stack. We will return to this point later and explain it in more detail.

Each edge $\langle n, n' \rangle$ of the CFG corresponds to a transition $\langle n, \Omega \rangle \rightarrow \langle n', \Omega' \rangle$ that relates the values Ω of the variables in scope before the transition to those Ω' after the transition.

We use the following notation to describe transition functions:

- $\Omega(e)$ denotes the evaluation of the expression e according to the valuation Ω of the variables in e . Expressions and their evaluation are defined the usual way.
- We refer to the state before the execution of a transition function as *current* state, and to the state afterwards as *next* state. We use *primed* versions of the variables to distinguish variables that refer to the next state from the variables in the current state. We allow expressions to range over variables in two different states $\langle n, \Omega \rangle$ and $\langle n', \Omega' \rangle$. The evaluation of such an expression e is denoted by $(\Omega, \Omega')(e)$.
- Expressions may also contain non-deterministic choice denoted by the “ \star ” symbols.
- SLAM exclusively uses the **schoose**¹ expression to introduce data non-determinism. It is defined as follows:

$$\mathbf{schoose}(pos, neg) \equiv \begin{cases} true & \text{if } pos \\ false & \text{else if } neg \\ \star & \text{otherwise} \end{cases}$$

The key insight of the summarization idea is that only the topmost element of the stack has an immediate impact on the execution of a transition. Therefore, the outcome of a call to a procedure is exclusively determined by the values of the global variables and the actual parameters at the call site. Consequently, each actual call (i.e., terminating call) to a procedure **pr** can be *summarized* by a pair of states $\langle n_i, \Omega_i \rangle, \langle n_o, \Omega_o \rangle$, where n_i denotes the entry node of the control-flow graph of **pr**, and n_o denotes the corresponding exit node. We use $\Sigma(\mathbf{pr})$ to denote the set of these pairs for a procedure **pr**. Furthermore, we assume in this section that $\Sigma(\mathbf{pr})$ contains the entries for all reachable call contexts. Clearly, $\Sigma(\mathbf{pr})$ is finite for Boolean programs.

For a given entry state, the corresponding exit states² are determined by the transition functions of the control flow graph of **pr**. The transition functions are in turn determined by the statements corresponding to the nodes of the control-flow graph. We distinguish the following statements³:

¹ This function can be rewritten as the following expression: $\mathbf{schoose}(pos, neg) \equiv (\star \wedge \overline{neg}) \vee pos$

² The use of non-determinism may result in more than one exit valuation.

³ The syntax of Boolean programs can be found in appendix A.1.

- The **skip** statement does not modify the variables, but sets the program counter to the succeeding statement.
- The **goto** ℓ_1, \dots, ℓ_k statement non-deterministically changes the program counter to one of the program locations ℓ_1, \dots, ℓ_k provided as argument. The values of the variables do not change.
- The **assert** e statement specifies the property to be verified. The program terminates, if the expression e can be evaluated to *false* (e.g., `assert *`). If this is the case, the model checker provides a counterexample in the form of a trace from the initial state to the location of the assertion. Otherwise, the **assert** statement sets the program counter to the succeeding statement.
- The **assume** e statement sets the program counter to the succeeding statement, if the condition e evaluates to *true* in the current state. Otherwise, the **assume** statement has no successor states. In other words, the **assume** statement behaves like an **assert** statement that fails silently.
- The constrained assignment statement $x_1, \dots, x_k := e_1, \dots, e_k$ **constrain** e assigns the values of the expressions e_1, \dots, e_k to the variables x_1, \dots, x_k . The expressions are evaluated in the current state and may contain non-deterministic choice variables. The constraint e is a predicate that ranges over the variables of the current *and* the next state. It is evaluated in both states, and the statement has no successor state if e evaluates to *false*.
- The **return** statement corresponds to the exit node of the control-flow graph of **pr**. Whenever it is reached, the current state determines the exit valuation of the corresponding summary. We assume without loss of generality that all return values are passed to the caller via global variables, i.e., **return** has no parameters. Therefore, the variables are not modified. The program counter of the successor statement is determined by the *caller* of the corresponding procedure **pr**.
- The call $x_1, \dots, x_k := \mathbf{pr}(e_1, \dots, e_k)$ modifies the global and local variables according to an *applicable summary* $\langle n_i, \Omega_i \rangle, \langle n_o, \Omega_o \rangle$ in $\Sigma(\mathbf{pr})$. A summary is applicable if a) Ω_i agrees with the current state of the global variables, and b) the evaluation of e_1, \dots, e_k matches the corresponding actual parameters in Ω_i . (The calling context determines the entry valuation of a summary.) Then, the call $x_1, \dots, x_k := \mathbf{pr}(e_1, \dots, e_k)$ modifies the global and local variables according to Ω_o . If more than one summary is applicable, one summary is chosen non-deterministically (analogously to the **goto** statement).

In the case that an applicable summary exists, the call sets the program counter to the statement that succeeds the call. Otherwise, the statement succeeding the call is never reached.

SLAM generates two statements that are not known by SATABS but shall be mentioned here for the sake of a complete overview:

- The **dead** v_1, \dots, v_k statement indicates *dead* variables. A variable is said to be dead at a program location n , if along every possible path from n , the variable is either not used or redefined before it is used. In the setting of a BDD-based model checker, the dead statement corresponds to syntactic sugar for $v_1, \dots, v_k := *, \dots, *$.
- The **enforce** e statement can be used at the beginning of a procedure to express constraints (stemming from previous spurious counterexamples) over predicates that hold throughout the procedure [9].

The statements **skip**, **assume**, **assert**, the constrained assignment, and procedure calls have a single successor node in the control flow graph (according to the structure of the program). The **return** statement has no successor in the control-flow graph, since the program location that succeeds a return statement cannot be determined statically. Goto statements may have more than one successor. Conditional statements like `if-then-else` or `while` loops can be modeled using a combination of the **goto** and **assume** statements.

The recursive nature with respect to $\Sigma(\mathbf{pr})$ of the definition of the semantics indicates that the set of summaries $\Sigma(\mathbf{pr})$ of a Boolean program can be obtained by means of a fix-point computation. Several algorithms that compute the least fix-point of the set $\Sigma(\mathbf{pr})$ in order to determine the set of reachable states have been proposed [13, 73, 122, 131].

3.2 Symbolic Simulation

The valuation of a symbolic state is represented in terms of a Boolean formula over non-deterministic choice variables \star_1, \dots, \star_k , i.e., we use a parametric representation. We extend the usual definition of Boolean formulæ with non-deterministic choice variables:

Definition 3.2 (Symbolic Formula) *A symbolic formula is defined using the following syntax rules:*

1. *The Boolean constants true and false are formulæ.*
2. *The non-deterministic choice variables \star_1, \dots, \star_k are formulæ.*
3. *If f_1 and f_2 are formulæ, then $f_1 \wedge f_2$, $f_1 \vee f_2$, and $\neg f_1$ are formulæ. The set of such formulæ is denoted by \mathcal{F} .*

Given a particular valuation ι for the non-deterministic choices \star_i , we denote the value of a symbolic formula f as $\llbracket f \rrbracket_\omega$.

Example 3.1 *The pair of formulæ $\langle \neg \iota_1, \iota_1 \vee \iota_2 \rangle$ may evaluate to $\langle 1, 0 \rangle$, $\langle 1, 1 \rangle$, $\langle 0, 1 \rangle$ but not to $\langle 0, 0 \rangle$.*

These symbolic formulæ are used to represent sets of states in the following manner. Let N be the set of nodes in a CFG of a Boolean program, let V be the variables of that program.

Definition 3.3 (Symbolic State) *A symbolic state is a triple $\langle n, \gamma, \omega \rangle$, where $n \in N$ identifies the node in the CFG and is represented explicitly. The second component, called γ is a Boolean formula over the non-deterministic choice variables and represents the **guard** of the state symbolically, i.e., a constraint over these variables. The component ω maps the variables V to formulæ over \star_1, \dots, \star_k , representing a set of valuations for the variables in the symbolic state.*

Each symbolic state $\langle n, \gamma, \omega \rangle$ represents the set of explicit states

$$\{ \langle n, \Omega \rangle \mid \exists \iota. \llbracket \gamma \rrbracket_\iota \wedge \forall v \in V. \Omega(v) = \llbracket \omega(v) \rrbracket_\iota \}$$

where $\omega(e)$ denotes the evaluation of the expression e according to the mapping ω (analogously to $\Omega(e)$ in Section 3.1).

Instruction	γ'	ω'
skip	$\gamma' = \gamma$	$\omega' = \omega$
goto ℓ_1, \dots, ℓ_k	$\gamma' = \gamma$	$\omega' = \omega$
assume e	$\gamma' = (\gamma \wedge \omega(e))$	$\omega' = \omega$
$x_1, \dots, x_k := e_1, \dots, e_k$ constrain e	$\gamma' = (\gamma \wedge (\omega, \omega')(e))$	$\omega' = (\omega[x_1/\omega(e_1)] \dots [x_k/\omega(e_k)])$

Table 3.1: Conditions on the symbolic transitions $\langle n, \gamma, \omega \rangle \rightarrow \langle n', \gamma', \omega' \rangle$ for the statements **skip**, **goto**, **assume**, and the constrained assignment

Example 3.2 The symbolic state $\langle \ell_{10}, (\star_1 \vee \star_2), \{a \mapsto \star_1, b \mapsto (\neg \star_1 \wedge \star_2)\} \rangle$, for instance, represents the explicit states $\langle \ell_{10}, \{a \mapsto 0, b \mapsto 1\} \rangle$ and $\langle \ell_{10}, \{a \mapsto 1, b \mapsto 0\} \rangle$. The valuation $\{\star_1 \mapsto 0, \star_2 \mapsto 0\}$ is ruled out by the guard, and the valuations $\{\star_1 \mapsto 1, \star_2 \mapsto 0\}$ and $\{\star_1 \mapsto 1, \star_2 \mapsto 1\}$ yield the same explicit state.

An unsatisfiable guard indicates that there is no concrete state represented by $\langle n, \gamma, \omega \rangle$ [46].

In Table 3.1, we write $\omega[x/e]$ for the mapping that maps x to the formula e , while it agrees with the mapping ω on all other variables. We use γ and ω to refer to the components γ and ω of the current state, and γ' and ω' to refer to the next state. The program locations are omitted, since they change according to the rules presented in Section 3.1. The conditions in Table 3.1 are equivalent to those presented in [46]. According to this table, the components γ and ω are modified as follows:

- In case of a **skip**, **return**, or **goto** statement, γ as well as ω do not change.
- Conditions contributed by **assume** statements are instantiated according to ω and conjoined with the guard γ . The symbolic execution terminates if γ' is unsatisfiable.
- A constrained assignment updates the mapping ω' accordingly. If a constraining condition is present, it is instantiated using ω and ω' , and conjoined with γ .

Example 3.3 Continuing Example 3.2, we construct the symbolic successor for the state

$$\left\langle \ell_{10}, (\star_1 \vee \star_2), \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto (\neg \star_1 \wedge \star_2) \end{array} \right\} \right\rangle \quad (3.1)$$

and the following statement:

$$a := \neg a \text{ **constrain** } (\neg b \vee a')$$

According to Table 3.1, we obtain $\omega' = \{a \mapsto \neg \star_1, b \mapsto (\neg \star_1 \wedge \star_2)\}$ and $\gamma' = (\star_1 \vee \star_2) \wedge (\star_1 \vee \neg \star_2 \vee \neg \star_1) \equiv (\star_1 \vee \star_2)$. Thus, the symbolic successor state results in:

$$\left\langle \ell_{11}, (\star_1 \vee \star_2), \left\{ \begin{array}{l} a \mapsto \neg \star_1, \\ b \mapsto (\neg \star_1 \wedge \star_2) \end{array} \right\} \right\rangle \quad (3.2)$$

3.3 Pushdown Systems

Boolean programs are a natural formalization as abstractions of higher-level programs since the control-flow structure resembles the original program. The special treatment of the program counter alleviates optimizations in the implementation but might prohibit a compact representation of a model and complicates reasoning about algorithms.

This section introduces *pushdown systems* as a more general representation of abstractions of programs. Any Boolean program can be translated into a *pushdown system* and vice versa. Pushdown systems focus on the most complex part of an algorithm for analysis of abstractions: the program's stack.

Definition 3.4 (Pushdown Systems [122]) A pushdown system (PDS) is a transition system, specified as a quadruple $\mathcal{P} = \langle P, \Gamma, \Delta, s_0 \rangle$. The set of control locations P as well as the stack alphabet Γ is finite. A state s is a pair $\langle p, w \rangle$, where $p \in P$ denotes a control location, and $w \in \Gamma^*$ represents the stack content. The transition relation is defined in terms of a finite set of rules $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$. We use s_0 to denote the initial state of a PDS.

The *head* of a state comprises of the control location and the topmost element of the stack (see Figure 3.1(b)). The state space of a PDS may be infinitely large, since the stack height is not bounded. The number of heads is bounded by $|P| \cdot |\Gamma|$.

Figure 3.1(a) shows the state $\langle p, \gamma_1 \gamma_2 \gamma_3 \rangle$, where $p \in P$ and $\gamma_i \in \Gamma$.

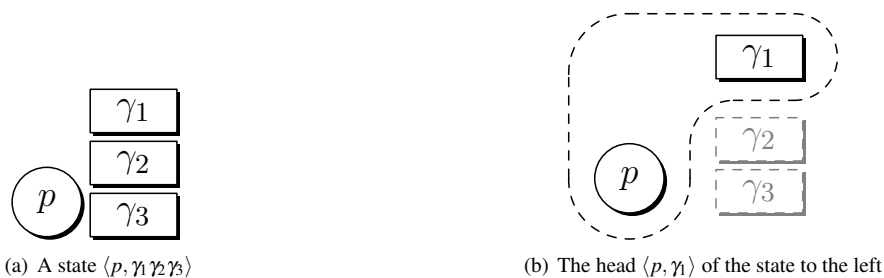


Figure 3.1: States and their heads

The rules determine the successors of a state $\langle p, \gamma w \rangle$ based on the head $\langle p, \gamma \rangle$ of this state. Each rule is of the form $\langle p, \gamma \rangle \hookrightarrow \langle q, w \rangle$, where $|w| \leq 2$ (⁴in particular, w may be ϵ , $|\epsilon| = 0$). Depending on the size of w , we distinguish between *expansion* rules, *neutration* rules, and *contraction* rules (see Figure 3.2).

We use \hookrightarrow^* to denote the reflexive transitive closure of Δ . A state s_i of a PDS is reachable iff $s_0 \hookrightarrow^* s_i$. The set of reachable states of a PDS can be represented by means of a finite automaton [35], which may be obtained using a saturation procedure [73, 69]. Intuitively, an expansion $\langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_2 \rangle$ followed by neutration that yields $\langle p_2, \gamma_3 \gamma_2 \rangle$, followed by a contraction $\langle p_2, \gamma_3 \gamma_2 \rangle \hookrightarrow \langle p_3, \gamma_2 \rangle x$ can be *summarized*⁵ by $\langle p_0, \gamma_0 \rangle \hookrightarrow^* \langle p_3, \gamma_2 \rangle$. Augmenting the transition relation with this summary may give rise to new summaries. The set of states reachable from s_0 is computed by repeatedly applying summarization until a fix-point is reached (i.e., no new summaries can be constructed). Since only the

⁴The restriction $|w| \leq 2$ allows a straightforward way of modeling Boolean programs as pushdown systems. However, any pushdown system can be converted into this form, as shown in [122].

⁵The idea of summarization was introduced by Sharir and Pnueli [123].

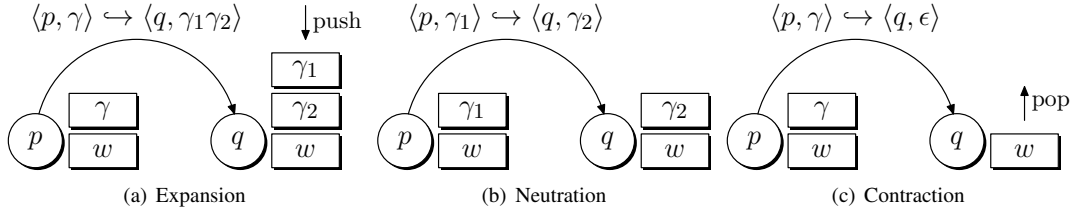


Figure 3.2: Transitions of a Pushdown System

head of a state has an immediate impact on the execution of a transition and the number of heads is bounded (see Definition 3.4), it is clear that there are only finitely many summaries.

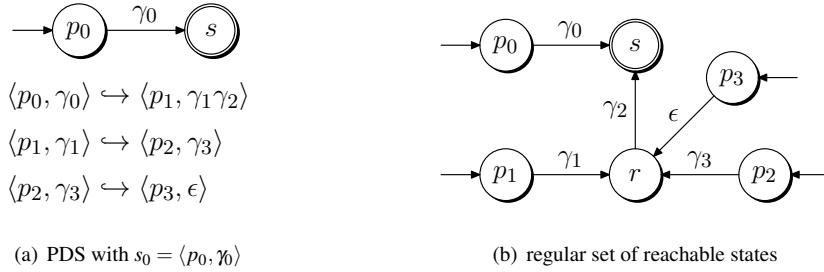


Figure 3.3: A PDS and the finite automaton accepting the reachable states

Example 3.4 Figure 3.3(a) shows a PDS $\langle P, \Gamma, \Delta, s_0 \rangle$ with three transition rules, four control locations $P = \{p_0, p_1, p_2, p_3\}$, the stack alphabet $\Gamma = \{\gamma_0, \gamma_1, \gamma_2, \gamma_3\}$ and the accepting state $s_0 = \langle p_0, \gamma_0 \rangle$. The finite automaton $\mathcal{A} = \langle Q, \Gamma, \delta, P \rangle$ in Figure 3.3(b) (where $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation of \mathcal{A} , Γ is the input alphabet, and $P \subseteq Q$ is the set of initial states) that accepts all reachable states of the PDS. The automaton \mathcal{A} , which we constructed using the algorithm presented in [69], uses the stack of a PDS state as input and the control locations of the PDS as initial states. The construction starts with the finite automaton in Figure 3.3(a), which accepts the initial state s_0 . As expected, the automaton in Figure 3.3(b) accepts the PDS state $\langle p_3, \gamma_2 \rangle$.

The efficiency of these saturation based algorithms can be significantly improved by using a symbolic BDD-based representation of the PDS. The notion of symbolic PDSs was introduced by Schwoon [122]:

Definition 3.5 (Symbolic Pushdown Systems) A Symbolic Pushdown System $\mathcal{P} = \langle P, \Gamma, \Delta_S, s_0 \rangle$ uses a propositional encoding for control locations $p \in P$ and stack elements $\gamma \in \Gamma$. A set of $\lceil \log_2(|P|) \rceil + \lceil \log_2(|\Gamma|) \rceil$ Boolean variables is sufficient to encode all heads of a PDS. The right hand side $\langle q, w \rangle$ (where $|w| \leq 2$) of a rule is represented using a separate set of Boolean variables. We use primed variables to denote elements of the latter set. The symbolic rules Δ_S are expressed in terms of a propositional relation over primed and unprimed variables. We use R^\nearrow , R^\rightarrow , and R^\searrow to refer to the relation for expansions, neutrations, and contractions respectively.

Assume, for instance, that P in Example 3.4 is represented using $\{a_0, a_1\}$, and Γ using $\{b_0, b_1\}$. One way to encode the neutrality rule $\langle p_1, \gamma_1 \rangle \leftrightarrow \langle p_2, \gamma_3 \rangle$ is

$$(\bar{a}_1 \cdot a_0) \cdot (\bar{b}_1 \cdot b_0) \cdot (a'_1 \cdot \bar{a}'_0) \cdot (b'_1 \cdot b'_0)$$

i.e., to use the variables to represent a binary encoding of the indices i of $p_i \in P$ and $\gamma_i \in \Gamma$ (where a_0 and b_0 represent the lower bits). The set of symbolic relations $\Delta_S = \{R^{\nearrow}, R^{\rightarrow}, R^{\searrow}\}$ is a disjunctive partitioning of symbolic relations corresponding to the union of transition rules.

The same technique is used to represent summaries. The symbolic representation of the summary $\langle p_0, \gamma_0 \rangle \hookrightarrow^* \langle p_3, \gamma_2 \rangle$ is $\bar{a}_1 \cdot \bar{a}_1 \cdot \bar{b}_1 \cdot \bar{b}_1 \cdot a'_1 \cdot a'_1 \cdot b'_1 \cdot \bar{b}'_1$ is presented in Figure 3.4(a). (We use the same graphical representation for symbolic transitions and explicit transitions.) The symbolic rep-

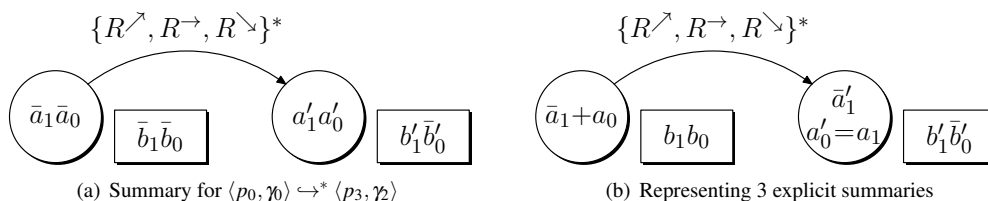


Figure 3.4: Symbolic representations for summaries

resentation of summaries (and states) is more succinct than the explicit representation used in [69] (see Example 3.4). For instance, the set of states $\{\langle p_0, \gamma_0 \rangle, \langle p_3, \gamma_2 \rangle\}$, which is a subset of the states represented by the automaton in Figure 3.3(b), has the symbolic representation $(a_1 = a_0) \cdot (\bar{b}_0)$. The symbolic summary

$$(\bar{a}_1 + a_0) \cdot (b_1 \cdot b_0) \cdot (a'_1 \cdot (a'_0 = a_1)) \cdot (b'_1 \cdot \bar{b}'_0)$$

(see Figure 3.4(b)) is a compact representation of the three explicit summaries

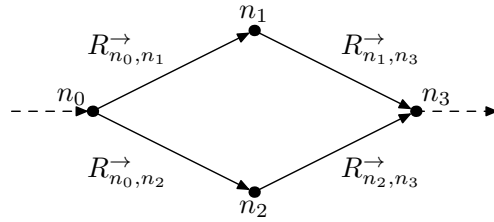
$$\langle p_0, \gamma_3 \rangle \hookrightarrow^* \langle p_2, \gamma_2 \rangle, \langle p_1, \gamma_3 \rangle \hookrightarrow^* \langle p_2, \gamma_2 \rangle, \text{ and } \langle p_3, \gamma_3 \rangle \hookrightarrow^* \langle p_3, \gamma_2 \rangle.$$

Symbolic PDSs are a widely-used formalism to represent abstractions of programs [14, 11]. In that case, it is convenient to maintain an explicit representation of program locations, i.e., to discriminate the rules in Δ_S by their source and target nodes of the control-flow graph (CFG).

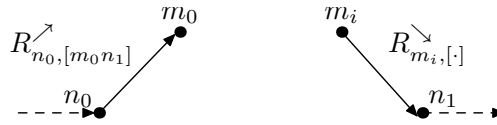
Figure 3.5 depicts examples for symbolic transitions that are annotated with explicit control flow information (n_i and m_j are control flow nodes in two different functions). From these pictures, it becomes clear, how pushdown rules are used to model programs: The neutralization rules in Figure 3.5(a) model a conditional statement, whereas the expansion and contraction rules in Figure 3.5(b) model a function call and a return statement. A finite number of global (Boolean) variables is modeled by means of the control locations, and the (also finite) local variables of functions are represented using the stack alphabet Γ . For instance, the summary $\langle p_0, \gamma_0 \rangle \hookrightarrow^* \langle p_3, \gamma_2 \rangle$ in Figure 3.4(a) modifies the control location (which represents the global variables of a program) and the topmost stack element. The return value of the corresponding function is passed on to the caller via the control location in this example. Throughout this paper, we use the notion of a *function* of a PDS to denote the transition rules associated with a function in the CFG. The correspondence indicates that PDSs are equally expressive as Boolean programs [14]:

Example 3.5 Recall the example in Figure 3.3(a). Listing 3.6(b) shows a translation into a corresponding Boolean program \mathcal{B} .

The control locations \mathcal{P} are translated into global variables and the stack alphabet Γ is modeled by the program counter and the local variables, respectively:



(a) Conditional instruction



(b) Function call

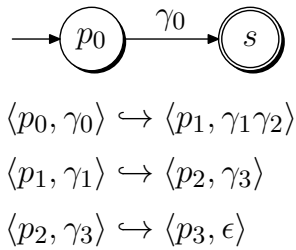
Figure 3.5: Symbolic transitions rules with explicit CFG node information

$$\begin{aligned}
 p_0 &\Longrightarrow \{g_{arg} \mapsto \star, g_{ret} \mapsto \star\} \\
 p_1 &\Longrightarrow \{g_{arg} \mapsto \text{true}, g_{ret} \mapsto \star\} \\
 p_2 &\Longrightarrow \{g_{arg} \mapsto \text{true}, g_{ret} \mapsto \star\} \\
 p_3 &\Longrightarrow \{g_{arg} \mapsto \text{true}, g_{ret} \mapsto \text{false}\}
 \end{aligned}$$

$$\begin{aligned}
 \gamma_0 &\Longrightarrow \langle L_0, \{\} \rangle \\
 \gamma_2 &\Longrightarrow \langle L_1, \{\} \rangle \\
 \gamma_1 &\Longrightarrow \langle L_2, \{a \mapsto g_{arg}, l \mapsto \star\} \rangle \\
 \gamma_3 &\Longrightarrow \langle L_3, \{a \mapsto g_{arg}, l \mapsto \neg g_{arg}\} \rangle
 \end{aligned}$$

In order to get a direct mapping from the PDS to the Boolean program \mathcal{B} , the function parameters and return values are modeled using the global variables g_{arg} and g_{ret} .

The transformation of any Boolean program into a PDS is described in [122] on page 12 ff.



(a) A pushdown system

```

void main()
begin
L0:   foo(T);
L1:   ...
end

bool foo(a)
    decl l;
begin
L2:   l := !a;
L3:   return l;
end

```

(b) Boolean program

Figure 3.6: A PDS and its corresponding Boolean program

4

Model Checking Sequential Boolean Programs

IN this chapter we present a novel algorithm for deciding reachability of particular error locations of sequential Boolean programs. We exploit the fact that most Boolean programs used in practice are *shallow*, and propose to use SAT-based Bounded Model Checking to search for counterexamples. We present a complete approach, that is based on a SAT solver and requires only few calls to a QBF solver for fix-point detection. The enabling technique for deciding the reachability of Boolean programs is the *summarization* of procedure calls. Most model checking tools for Boolean programs use BDDs to represent these summaries, allowing for efficient fix-point detection. However, BDDs are highly sensitive to the number of state variables. Furthermore, we improve the well-known algorithms for creating summaries of procedures with techniques for building *universal summaries* and over-approximations of summaries. The algorithms are described formally using pushdown systems.¹

We record two observations about such systems. First of all, most tools that implement an abstraction/refinement framework compute a *conservative abstraction* (see Section 2.6). If the property holds on the abstract model, it also holds on the original program. As a consequence, the abstraction/refinement loop terminates as soon as an abstraction is built in which the error location is unreachable. In all previous iterations, there exists a path that reaches an error location. It is reported in [10] that the verification of device drivers may require as much as twenty iterations, with an average of 5 iterations. This motivates the need for an algorithm that performs well on models with a counterexample.

Second, we exploit the fact that Boolean programs generated as abstractions of commodity software are typically very *shallow*. Formally, this means that any node of the Kripke structure that is reachable from an initial state is reachable within a few steps.

Bounded Model Checking (Section 2.4) is a perfect fit for this scenario. In BMC, a transition system is unwinded together with a property up to a given bound k to form a propositional formula.

¹ This chapter presents work that has been published at the 14th International SPIN Workshop [20] and the Haifa Verification Conference 2007 [19].

The formula is satisfiable if and only if there exists a path of length k that refutes the property. If the formula is unsatisfiable, BMC is inconclusive as deeper counterexamples might still exist. SAT-based BMC is therefore known as an effective method to discover shallow bugs.

In order to prove the absence of errors, we extend BMC with *procedure summarization* [123]. A procedure summary maps a configuration of a PDS at the entry of a procedure to the set of configurations observable upon exit. As there are only finitely many summaries, saturation can be used to compute the set of reachable states [73, 69].

4.1 Summarization of Boolean Programs

Sequential Boolean programs are introduced in Chapter 3. Recall that the control flow graph of a Boolean programs is a directed graph with nodes corresponding to program locations. Since Boolean programs allow recursion, a naïve algorithm for checking reachability might be stuck in infinite recursion.

As mentioned in Section 3.3, a Boolean program can be transformed into a symbolic pushdown system because they have equally expressive power.

The semantics of sequential Boolean programs is discussed in Section 3.1 - essentially all statements can be categorized into *expansion*, *contraction* or *neutration*. We write $\langle p, \gamma \rangle_n \hookrightarrow^* \langle p', \gamma' \rangle_{n'}$ to denote the relation of two heads at the program locations n and n' respectively.

In the given setting, only the head has an immediate impact on the execution of a transition. Whenever the algorithm encounters a contraction rule, it computes a summary that relates resulting head with the head of the corresponding expansion. A summary for a procedure \mathbf{pr} is a relation of the heads of two states with the same stack height: $\langle p_i, \gamma_i \rangle_i \hookrightarrow^* \langle p_e, \gamma_e \rangle_e$ where n_i denotes the entry node of the control flow graph of \mathbf{pr} , and n_e denotes the corresponding exit node. (Without loss of generality, we assume that each procedure has exactly one entry node n_i and one exit node n_e .)

We use $\Sigma(\mathbf{pr})$ to denote the set of these relations for a procedure \mathbf{pr} . Furthermore, we assume in this section that $\Sigma(\mathbf{pr})$ contains the entries for all reachable call contexts. Clearly, $\Sigma(\mathbf{pr})$ is finite for Boolean programs.

In the rest of this section we describe how we compute the least fix-point of $\Sigma(\mathbf{pr})$ using forward symbolic execution and QBF-based fix-point detection.

The representation of transitions by means of two symbolic states is not restricted to single transitions, but can be extended to sequences of transitions in the natural way. This representation enables the summarization of compound transitions, and is similar to the concept of *path edges* [13].

Definition 4.1 (Path Edge) A Path Edge $\Sigma_{\mathcal{P}}^n$ for a node n is a relation of a pair of symbolic heads $\Sigma_{\mathcal{P}}^n \subseteq (P \times \Gamma) \times (P \times \Gamma)$ such that

$$\Sigma_{\mathcal{P}}^n (\langle p, \gamma \rangle_i, \langle p', \gamma' \rangle_n) \stackrel{def}{=} \exists \langle p_1, \gamma_1 \rangle, \dots, \langle p_n, \gamma_n \rangle \in P \times \Gamma. \langle p, \gamma \rangle_i \hookrightarrow \langle p_1, \gamma_1 \rangle \hookrightarrow \dots \hookrightarrow \langle p_n, \gamma_n \rangle \hookrightarrow \langle p', \gamma' \rangle_n$$

This definition only allows *neutrations* to be applied and therefore there is no need to model the stack explicitly, hence path edges only refer to the top of the stack. In order to simulate a stack, the side-effects of an expansion and contractions pair have to be "summarized" into a neutration:

Definition 4.2 (Summary Edge) *A path edge that reaches the exit node of a function can be transformed into a Summary Edge $\Sigma_S \subseteq (P \times \Gamma) \times (P \times \Gamma)$ that captures the "side-effects" of an expansion and the corresponding contraction:*

$$\begin{aligned} \Sigma_{\mathcal{F}}(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) &\stackrel{\text{def}}{=} \\ R_f^{\nearrow}(\langle p, \gamma \rangle, \langle p_1, \gamma_1 \rangle_{n_i}) \wedge \Sigma_{\mathcal{D}}(\langle p_1, \gamma_1 \rangle_{n_i}, \langle p'_1, \gamma'_1 \rangle_{n_o}) \wedge R_f^{\searrow}(\langle p'_1, \gamma'_1 \rangle_{n_o}, \langle p', \varepsilon \rangle) \end{aligned} \quad (4.1)$$

After a summary edge has been computed, the transition system can be completed with the new neutralization (i.e., $\Delta' = \Delta \cup \Sigma_{\mathcal{F}}$).

As this formula suggests, the sequences of transitions that may form a summary are described by the grammar in Figure 4.1.

$$\begin{aligned} \text{Summary} &::= R^{\nearrow} \text{Transitions } R^{\searrow}; \\ \text{Transitions} &::= \text{Transitions } \text{Transition} \mid \varepsilon; \\ \text{Transition} &::= \text{Summary} \mid R^{\rightarrow}; \end{aligned}$$

Figure 4.1: A grammar for summaries

Each new summary is added to a set Σ , such that summaries can be reused whenever the algorithm encounters a head and an expansion which are already covered. Eventually, Σ converges, since there are at most $|P|^2 \cdot |\Gamma|^2$ summaries that are not logically equivalent. The result is the least fixpoint of the set of summaries for the PDS, i.e., Σ contains only summaries for which the corresponding heads are indeed reachable.

If a contraction is encountered, we try to construct a new summary edge according to equation (4.1) for each matching expansion that has been visited before. A new summary edge $\Sigma_{\mathcal{F}}$ is compatible with an expansion of the current path edge $\Sigma_{\mathcal{D}}^n$, if there is a path edge $\Sigma_{\mathcal{D}}$, such that the head of the procedure entry ($\langle p_2, \gamma_2 \rangle_{i_2}$) is a superset of the head after the expansion ($\langle p''', \gamma''' \rangle_{i_2}$), and there is a matching contraction. The application of $\Sigma_{\mathcal{F}}^n$ results in a new path edge $\Sigma_{\mathcal{D}}^n$:

$$\begin{aligned} \Sigma_{\mathcal{D}}^n(\langle p, \gamma \rangle_{i_1}, \langle p', \gamma' \rangle_n) &\iff \\ \Sigma_{\mathcal{D}}^n(\langle p, \gamma \rangle_{i_1}, \langle p'', \gamma'' \rangle_n) \wedge \\ \Sigma_{\mathcal{F}} \left\{ \begin{array}{l} (\langle p'', \gamma'' \rangle_n \hookrightarrow \langle p''', \gamma''' \rangle_{i_2}) \wedge \\ \forall \langle p''', \gamma''' \rangle_{i_2} \exists \langle p_2, \gamma_2 \rangle_{i_2}. (\langle p''', \gamma''' \rangle_{i_2} = \langle p_2, \gamma_2 \rangle_{i_2}) \Sigma_{\mathcal{D}}(\langle p_2, \gamma_2 \rangle_{i_2}, \langle p'_2, \gamma'_2 \rangle_{o_2}) \wedge \\ (\langle p'_2, \gamma'_2 \rangle_{o_2} \hookrightarrow \langle p', \varepsilon \rangle_{n'}) \end{array} \right. \end{aligned} \quad (4.2)$$

Equation (4.2) is illustrated in Figure 4.2.

The universal quantification requires resorting to a QBF instance to decide applicability. Alternatively, a summary edge could also be constructed if the head of the entry node of the path edge and the head after the expansion have some evaluations in common. In this setting, the QBF check (equation (4.2)) could be replaced with a SAT instance that is satisfiable, if there are any evaluations in common (the algorithm presented in [13] works this way). However, this would be wasteful in a non-BDD based environment, since it is not possible to compute a summary for the missing states only.

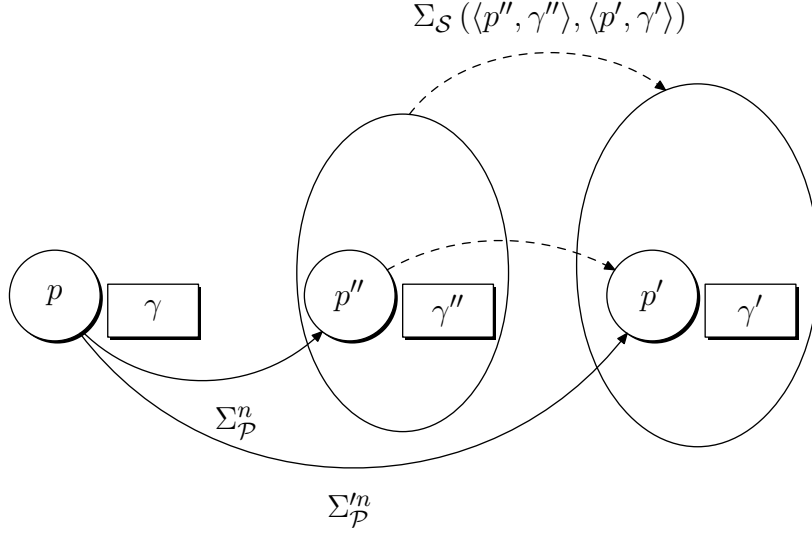


Figure 4.2: Application of a summary.

Assuming that equation (4.2) holds, a BDD-based model checker would existentially quantify the state variables at the procedure entry, however a SAT implementation has no means to eliminate these variables and must add constraints to force the variables to match the values of the entry node of the path edge $\Sigma_{\mathcal{P}}$.

Now consider the case that equation (4.2) does not hold, i.e., no summary can be added to $\Sigma_s(\mathbf{pr})$. In that case, a new path edge must be computed first that starts at the entry node of the callee. For this purpose, we construct a new symbolic state $\langle p''', \gamma''' \rangle_i$ that results from applying the expansion rule $(\langle p'', \gamma'' \rangle_n \hookrightarrow \langle p''', \gamma''' \rangle_i)$. The symbolic state $\langle p_i, \gamma_i \rangle$ serves as entry node for a new path edge $(\langle p''', \gamma''' \rangle_i, \langle p''', \gamma''' \rangle_i)$, and may eventually yield a new summary edge.

Fix-point Detection

In order to determine the least fix-point of $\Sigma_s(\mathbf{pr})$, our reachability checking algorithm performs symbolic simulation of a PDS. The algorithm maintains a set \mathcal{P} of path edges and summary edges $\Sigma_s(\mathbf{pr})$ that have been constructed so far. Our algorithm is similar to the BDD-based Model Checking algorithm presented in [13]. However, unlike a BDD-based representation of path edges, our representation is not canonical. The price we pay for being able to apply transition functions efficiently is that we need to solve a QBF instance in order to determine whether a path edge is already an element of \mathcal{P} .

Given two path edges $(\langle p_i, \gamma_i \rangle_{n_i}, \langle p'_{o_1}, \gamma'_{o_1} \rangle_{n_i})$ and $(\langle p_{i_2}, \gamma_{i_2} \rangle_{n_i}, \langle p'_{o_2}, \gamma'_{o_2} \rangle_{n_i})$, the latter is *at least as general as* the former iff

$$\begin{aligned}
 \forall (\langle p_1, \gamma_1 \rangle, \langle p'_1, \gamma'_1 \rangle) \in (\langle p_{i_1}, \gamma_{i_1} \rangle_{n_i}, \langle p'_{o_1}, \gamma'_{o_1} \rangle_{n_i}) \\
 \exists (\langle p_2, \gamma_2 \rangle, \langle p'_2, \gamma'_2 \rangle) \in (\langle p_{i_2}, \gamma_{i_2} \rangle_{n_i}, \langle p'_{o_2}, \gamma'_{o_2} \rangle_{n_i}) \\
 (\langle p_1, \gamma_1 \rangle, \langle p'_1, \gamma'_1 \rangle) = (\langle p_2, \gamma_2 \rangle, \langle p'_2, \gamma'_2 \rangle)
 \end{aligned} \tag{4.3}$$

Equation (4.3) holds iff the set of pairs of concrete states represented by the first path edge is a

subset of the corresponding set represented by the second. In that case, a further expansion of the path edge $(\langle p_i, \gamma_i \rangle_{n_i}, \langle p'_{o_1}, \gamma'_{o_1} \rangle_n)$ does not yield any states that are not discovered by expanding the more general path edge.

The pseudo code of our QBF-based algorithm is presented in Figure 4.3. It resembles the Model Checking algorithm presented in [13], but uses SAT and QBF instead of BDDs.

We use $\text{APPLICABLE}(\pi, \sigma)$ to denote the condition in equation (4.2), where $\pi = (\langle p, \gamma \rangle_{n_i}, \langle p', \gamma' \rangle_n)$ is a path edge that provides the calling context, and σ is a summary edge. Furthermore, $\text{APPLY}(\pi, \sigma)$ denotes the path edge that we obtain by applying the summary according to equation (4.2). The condition in equation (4.3) is expressed by $\pi_1 \subseteq \pi_2$ and holds if the path edge π_1 is subsumed by π_2 .

The algorithm maintains a work-list \mathcal{W} in which all path-edges that are currently explored are stored. Each path edge of this work-list is expanded as often as possible or until the resulting path edge is already in \mathcal{P} . For convenience, we define a procedure $\text{INSERT}(\pi)$, which we use to insert a path edge into the work-list, unless it is already contained in \mathcal{P} .

Procedure calls are handled in line 14. Matching summary edges are applied immediately. However, if there is no applicable summary edge, we construct an entry state for the called procedure and add a corresponding path edge to the work-list \mathcal{W} . Furthermore, we store the current path edge σ in \mathcal{W}' , which is examined whenever we add a new summary to $\Sigma(\mathbf{pr})$. Thus, we guarantee that any summary of \mathbf{pr} that is eventually generated is applied also to σ (see line 28).

4.2 Universal Summaries

An alternative to computing the set of reachable summaries Σ is to use *universal summaries* instead [20]:

Definition 4.3 (Universal Summary) A Universal Summary $\Sigma_{\mathcal{U}}$ for a PDS $\langle P, \Gamma, \Delta, s_0 \rangle$ is a relation $\Sigma_{\mathcal{U}} \subseteq (P \times \Gamma) \times (P \times \Gamma)$ such that

$$\Sigma_{\mathcal{U}}(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) \stackrel{\text{def}}{=} (\exists \langle p_1, w_1 \rangle, \dots, \langle p_n, w_n \rangle \in P \times \Gamma^*. \\ \langle p, \gamma \rangle \hookrightarrow \langle p_1, w_1 \rangle \hookrightarrow \dots \hookrightarrow \langle p_n, w_n \rangle \hookrightarrow \langle p', \gamma' \rangle \wedge \forall i \in \{1..n\}. |w_i| \geq 2)$$

holds.

Intuitively, for any head $\langle p, \gamma \rangle$, a universal summary subsumes *all* paths that “traverse a function” of the PDS, no matter whether there exists a reachable state $\langle p, \gamma w \rangle$ or not. The restriction $|w_i| \geq 2$ guarantees that $\Sigma_{\mathcal{U}}$ relates expansions to their matching contractions. Note that this definition does not rule out *nested* summaries. (A summary is nested if it is entirely contained in another summary according to the grammar in Figure 4.1.) In particular, it does not exclude *recursion*, i.e., a summary $\langle p_0, \gamma_0 \rangle \hookrightarrow^* \langle p'_0, \gamma'_0 \rangle$ may stem from a sequence of transitions that contains a nested summary $\langle p_1, \gamma_1 \rangle \hookrightarrow^* \langle p'_1, \gamma'_1 \rangle$ such that $p_0 = p_1$, $\gamma_0 = \gamma_1$, $p'_0 = p'_1$, and $\gamma'_0 = \gamma'_1$.

In the following section, we discuss how symbolic summaries are computed. Based on this, we present an algorithm that computes symbolic universal summaries for recursion-free PDSs in Section 4.2.

```

1: procedure INSERT( $\pi$ ) ▷ add new path edge
2:   if  $\pi \notin \mathcal{P}$  then
3:     insert  $\pi$  into  $\mathcal{P}$ 
4:     insert  $\pi$  into  $\mathcal{W}$ 

5: Initialize  $\mathcal{P}$  to  $\emptyset$ ;
6: for all  $\mathbf{pr}$  do Initialize  $\Sigma_s(\mathbf{pr})$  to  $\emptyset$ ;
7:  $\mathcal{W} := \{(\langle p_0, \gamma_0 \rangle_0, \langle p_0, \gamma_0 \rangle_0)\}$ ; ▷  $\langle p_0, \gamma_0 \rangle_0$  is initial state
8:  $\mathcal{W}' := \emptyset$ ; ▷ remembers postponed expansions
9: while  $\mathcal{W} \neq \emptyset$  do
10:  remove  $\pi = (\langle p, \gamma \rangle, \langle p', \gamma' \rangle)$  from  $\mathcal{W}$ 
11:  if  $\exists T(\langle p', \gamma' \rangle, \langle p'', \gamma'' \rangle) \in \Delta_S$  then ▷ neutrality
12:     $\pi' := (\langle p, \gamma \rangle, \langle p', \gamma' \rangle \wedge T)$ ;
13:    INSERT( $\pi'$ );
14:  else if  $\exists T(\langle p', \gamma' \rangle, \langle p'', \gamma''_1 \gamma''_2 \rangle) \in \Delta_S$  then ▷ expansion
15:    for all  $\sigma_{\mathbf{pr}} \in \Sigma(\mathbf{pr})$  do
16:      if APPLICABLE( $\pi, \sigma_{\mathbf{pr}}$ ) then
17:         $\pi' := \text{APPLY}(\pi, \sigma_{\mathbf{pr}})$ ;
18:        INSERT( $\pi'$ );
19:      if  $\{\sigma_{\mathbf{pr}} \in \Sigma(\mathbf{pr}) \mid \text{APPLICABLE}(\pi, \sigma_{\mathbf{pr}})\} = \emptyset$  then
20:        construct entry state  $\langle p_i, \gamma_i \rangle_i$  for  $\mathbf{pr}$ ;
21:         $\pi' := \{\langle p_i, \gamma_i \rangle_i, \langle p_i, \gamma_i \rangle_i\}$ ;
22:        INSERT( $\pi'$ );
23:        insert  $\pi$  into  $\mathcal{W}'$  ▷ postpone expansion
24:  else if  $\exists T(\langle p', \gamma' \rangle, \langle p'', \varepsilon \rangle) \in \Delta_S$  then ▷ contraction
25:    if  $\pi \notin \Sigma_s(\mathbf{pr} \text{ of } n_i)$  then
26:       $\sigma := \pi$ ; ▷ encountered new summary edge
27:      insert  $\sigma$  into  $\Sigma_s(\mathbf{pr})$ ;
28:      for all  $\pi_c \in \mathcal{W}'$  s.t. APPLICABLE( $\pi_c, \sigma$ ) do
29:         $\pi' := \text{APPLY}(\pi_c, \sigma)$ ; ▷ perform postponed expansion
30:        INSERT( $\pi'$ );

```

Figure 4.3: The SAT based Model Checking algorithm

```

1: procedure INSERT( $\pi$ ) ▷ add new path edge
2:   if  $\pi \not\subseteq \mathcal{P}$  then
3:     insert  $\pi$  into  $\mathcal{P}$ 
4:     insert  $\pi$  into  $\mathcal{W}$ 
5: procedure UNROLL(pr)
6:    $\mathcal{W} := \{\langle n_i, true, \omega_* \rangle, \langle n_i, true, \omega_* \rangle\}$ ; ▷  $\langle n_i, true, \omega_* \rangle$  is initial state
7:   for all nodes  $n \in \text{CFG}(\mathbf{pr})$  do
8:      $\mathcal{P}(n) := \emptyset$ ;
9:   assign priorities no nodes: the closer to a return statement, the lower;
10:  while  $\mathcal{W} \neq \emptyset$  do
11:    choose  $n_o$  with highest priority s.t.  $\exists \langle n_i, \gamma_i, \omega_i \rangle, \langle n_o, \gamma_o, \omega_o \rangle \in \mathcal{W}$ ;
12:     $\mathcal{W}' := \{\langle n'_i, \gamma'_i, \omega'_i \rangle, \langle n'_o, \gamma'_o, \omega'_o \rangle \in \mathcal{W} \mid n'_o = n_o\}$ ;
13:     $\mathcal{W} := \mathcal{W} \setminus \mathcal{W}'$ ;
14:     $\pi' := \text{MERGE}(\mathcal{W}')$ ;
15:    EXPAND( $\pi', n_o$ ); ▷ expands  $\pi'$  and adds result to  $\mathcal{W}$ 
16:  assert (statement of  $n_o$  is return);
17:  return  $\pi'$ ;

```

Figure 4.4: Expanding and merging path edges at every join node

4.2.1 Computing Symbolic Summaries

A symbolic Model Checking algorithms for PDSs represents a sequence of transitions by a propositional formula that is only satisfiable if the corresponding path is feasible. For instance, the path

$$\underbrace{\langle p_0, \gamma_0 \rangle}_{s_0} \hookrightarrow \underbrace{\langle p_1, \gamma_1 \gamma_2 \rangle}_{s_1} \hookrightarrow \underbrace{\langle p_2, \gamma_3 \gamma_2 \rangle}_{s_2} \hookrightarrow \underbrace{\langle p_3, \gamma_2 \rangle}_{s_3}$$

is represented by following *path formula*:

$$(\bar{a}_1 \cdot \bar{a}_0) \cdot (\bar{b}_1 \cdot \bar{b}_0) \cdot (a'_1 \cdot a'_0) \cdot (b'_1 \cdot b'_0) \cdot (a''_1 \cdot a''_0) \cdot (b''_1 \cdot b''_0) \cdot (a'''_1 \cdot a'''_0) \cdot (b'''_1 \cdot b'''_0)$$

where the variables $\{a_0, a_1, b_0, b_1\}$ are used for the representation of s_0 , the variables $\{a'_0, a'_1, b'_0, b'_1\}$ for s_1 , and so on. The parts of the formula that refer to s_1 and s_2 constrain only the topmost element of the stack. The content of the bottom element of the stack (γ_2) is determined by the expansion rule $\langle p_0, \gamma_0 \rangle \hookrightarrow \langle p_1, \gamma_1 \gamma_2 \rangle$, but the neutralization rule cannot access this element. It only becomes “visible” to subsequently applied transition rules after the contraction $\langle p_2, \gamma_3 \rangle \hookrightarrow \langle p_3, \varepsilon \rangle$.

As discussed in Section 3.3, this sequence of transitions gives rise to the symbolic summary $\bar{a}_1 \cdot \bar{a}_1 \cdot \bar{b}_1 \cdot \bar{b}_1 \cdot a'_1 \cdot a'_1 \cdot b'_1 \cdot b'_1$. The summary is obtained by existentially quantifying the variables that represent the intermediate heads (i.e., $\{a'_0, a'_1, b'_0, b'_1\}$ and $\{a''_0, a''_1, b''_0, b''_1\}$ in our example). Each symbolic summary is a propositional relation over a set of primed and unprimed variables.

Merging Paths.

Whenever the algorithm encounters a branch (as illustrated in Figure 3.5(a)) it splits the path and constructs a new formula for each branch. To avoid an exponential blowup, path formulas are *merged* (by means of a disjunction) when they agree on their initial and final CFG nodes (see, for instance, [103]). A detailed description of an algorithm that performs aggressive merging by delaying transitions until merging becomes possible is given in [20].

4.2.2 Using BMC to Compute Universal Summaries

The symbolic algorithms discussed compute the least fix-point of reachable summaries [13, 20]. This fix point detection is implemented using either BDDs or a QBF solver. Unfortunately, neither of these approaches scales for a large number of variables. Bounded Model Checking (BMC) addresses this issue by avoiding fix point detection altogether: The transition system is simply unrolled up to a bounded path length k . This idea is illustrated in Figure 4.5(b) for the cyclic transition system shown in Figure 4.5(a) (instead of unrolling each path separately, we merge paths as discussed in Section 4.2.1). The satisfiability of the resulting path formula can be decided using an efficient SAT-solver (e.g., MINISAT [59]).

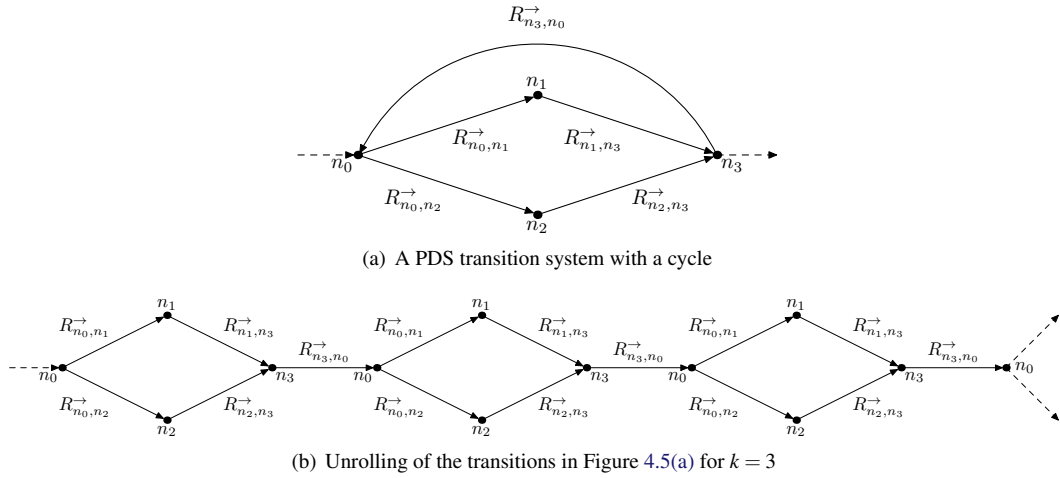


Figure 4.5: Bounded Model Checking for Pushdown Systems

BMC is complete with respect to reachability if (and only if) k is large enough to guarantee that all reachable states of the transition system are considered (the smallest k that has this property is called the *reachability diameter* of a transition system [97]). For a PDS with an infinite state space, there is no such finite k .

However, a function containing only neutrations is a finite state transition system. For a set of neutrations that are represented by $\xrightarrow{R^-}$ and a symbolic representation $I(s_0)$ of the initial state(s), the constant

$$k = \max\{i \in \mathbb{N} \mid \exists s_0, \dots, s_i \in P \times \Gamma^*. I(s_0) \wedge \bigwedge_{j=0}^{i-1} s_j \xrightarrow{R^-} s_{j+1} \wedge \bigwedge_{0 \leq j < l \leq i} s_j \neq s_l\}$$

is the length of the longest *loop-free* path that contains only neutrations (this corresponds to the *reachability recurrence diameter* of a finite state transition system [97]). Let $\langle p, \gamma \rangle \xrightarrow{R^-}^i \langle p', \gamma' \rangle$ denote the path formula obtained by means of unrolling $\xrightarrow{R^-}$ exactly i times. Then, the relation

$$R^{\rightarrow*}(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) \stackrel{\text{def}}{=} \bigvee_{i=1}^{\infty} \langle p, \gamma \rangle \xrightarrow{R^-}^i \langle p', \gamma' \rangle$$

is sufficient to determine all heads $\langle p', \gamma' \rangle$ that are reachable from $\langle p, \gamma \rangle$ by means of neutrations. Using this technique, we can compute a path formula that represents all heads reachable from an initial

state $\langle p, \gamma \rangle \in I(s_0)$ by the loop in Figure 4.5(a). In particular, if $I(s_0) = \text{true}$, then the relation $R^{\rightarrow*}$ determines the states reachable from an arbitrary initial state.

Given an explicit representation of the CFG (as suggested in Section 3.1), it is possible to determine the *innermost* function f that contains no expansion/contraction rules (at least as long as f is not a recursive function). Let R_f^{\rightarrow} denote the neutrations of this function, and let R_f^{\nearrow} and R_f^{\searrow} the initial expansion and the final contraction, respectively. Furthermore, let $R_f^{\rightarrow*}$ denote the unrolled path formula for the neutrations R_f^{\rightarrow} and an arbitrary initial state $I(s_0) = \text{true}$. Then, we obtain a universal summary for f by composing $R_f^{\rightarrow*}$ with R_f^{\nearrow} and R_f^{\searrow} :

$$\Sigma_{\mathcal{U}}^f(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) \stackrel{\text{def}}{=} R_f^{\nearrow}(\langle p, \gamma \rangle, \langle p_1, \gamma_1 \rangle) \wedge R_f^{\rightarrow*}(\langle p_1, \gamma_1 \rangle, \langle p_2, \gamma_2 \rangle) \wedge R_f^{\searrow}(\langle p_2, \gamma_2 \rangle, \langle p', \varepsilon \rangle)$$

We proceed by finding a function g that calls only f and compute the universal summary for R_g^{\rightarrow} using the universal summary for f . Thus, a universal summary for the whole PDS can be obtained in a top-down manner (assuming that the CFG representation of the PDS does not contain recursive function calls).

Corollary 4.4 *Let k be the length of the longest loop-free path through a recursion-free function f of a PDS. Then, the summary that subsumes all loop-free paths through f up to length k is a universal summary for the function f .*

4.3 Structural Abstraction with Summaries

The technique discussed in the previous section (and presented in [20]) applies universal summaries in an *eager* manner: Whenever the search algorithm encounters an expansion rule and an appropriate (universal) summary is available, the summary replaces the expansion transition. If we are only interested in the reachability of a given head (or a program location), this approach is wasteful: A subsequence of a path may be sufficient to show that a head is not reachable. In that case, computing and applying the universal summaries for the nested functions in a top-down manner does not contribute to the infeasibility of the resulting path formula.

Therefore, we propose to compute universal summaries for functions *on demand*, and to apply them in a bottom-up manner. Given the transition rules of a function g of the PDS, we obtain an *over-approximation* of the corresponding universal summary $\Sigma_{\mathcal{U}}^g$ by replacing all occurrences of f in g by a non-deterministic summary Σ_{\star}^f :

$$\Sigma_{\star}^f(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) \stackrel{\text{def}}{=} \exists \langle p_1, \gamma_1 \rangle, \langle p_2, \gamma_2 \rangle. R_f^{\nearrow}(\langle p, \gamma \rangle, \langle p_1, \gamma_1 \rangle) \wedge R_f^{\searrow}(\langle p_2, \gamma_2 \rangle, \langle p', \varepsilon \rangle)$$

The summary Σ_{\star}^f is a conservative over-approximation of $\Sigma_{\mathcal{U}}^f$, since

$$\Sigma_{\mathcal{U}}^f(\langle p, \gamma \rangle, \langle p', \gamma' \rangle) \implies \Sigma_{\star}^f(\langle p, \gamma \rangle, \langle p', \gamma' \rangle)$$

always holds. A head $\langle p', \gamma' \rangle$ that is *not* reachable via the over-approximated summary for g is also not reachable using $\Sigma_{\mathcal{U}}^g$. The reverse does not hold.

Example 4.1 *Consider the following transition rules for a function f :*

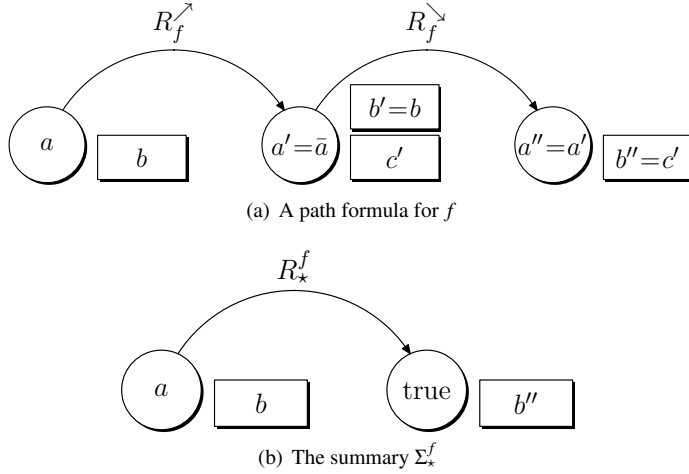


Figure 4.6: Over-approximation of a universal summary

$$\begin{aligned} R_f^{\rightarrow}(a, b, a', b', c') &= (a \cdot b) \cdot (a' = \bar{a}) \cdot (b' = b) \cdot c' \\ R_f^{\leftarrow}(a, b, a') &= (a' = a) \end{aligned}$$

The composition of R_f^{\rightarrow} and R_f^{\leftarrow} yields the path formula in Figure 4.6(a). The corresponding over-approximation of the summary for f is

$$\Sigma_*^f(a, b, a'', b'') = \exists a' b' a_* . (a \cdot b) \cdot (a' = \bar{a}) \cdot (b' = b) \cdot b'' \cdot (a'' = a_*)$$

The summary Σ_*^f does not constrain the value of a'' (see Figure 4.6(b)), even though the control state represented by $a'' = 1$ on return contradicts the path formula in Figure 4.6(a).

Now consider the transitions of a function g shown in Figure 4.7, and assume that the rules R_g^{\rightarrow} require that transitions from n_2 to n_3 are only feasible if $a'' = 1$ holds at n_2 (e.g., $R_g^{\rightarrow}(a, b, a', b') = a \cdot a' \cdot (b' = b)$ for the transition from n_2 to n_3). If we over-approximate the function f as indicated in Figure 4.7(a), then there exists a valuation (with $a'' = 1$ at n_2) to the variables of the corresponding path formula that represents a path through n_0, n_1, n_2 and n_3 .

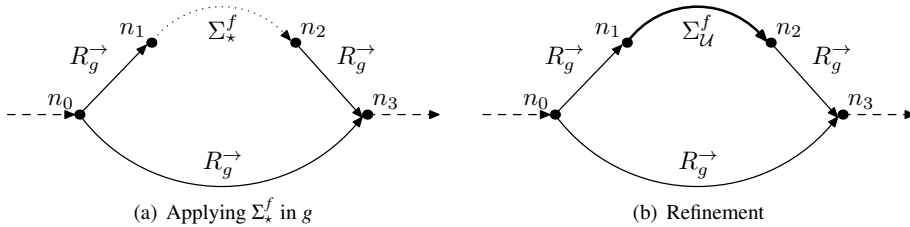


Figure 4.7: Refining an over-approximated universal summary

Unfortunately, this path is spurious, i.e., there is no feasible corresponding path in the original PDS. Therefore, we have to eliminate it from our over-approximated transition system. This can be

achieved by computing $\Sigma_{\mathcal{Q}}^f$ and using it to constrain the transition from n_1 to n_2 (as illustrated in Figure 4.7(b)).

The reachability of a head $\langle p, \gamma \rangle$ at a node n can be determined by repeatedly refining the transition system until either a feasible path is found, or the head becomes unreachable. Figure 4.8 shows the pseudo code for this algorithm. Unfortunately, this algorithm does not work if the CFG contains recursive function calls. Therefore, our implementation still uses the QBF-based approach presented in [20] to compute summaries for recursive functions. Universal summaries and the refinement technique are orthogonal to this approach and can be integrated easily.

```

1: procedure ISREACHABLE(PDS  $\langle P, \Gamma, \Delta, s_0 \rangle$ , head  $\langle p, \gamma \rangle$ , node  $n$ )
2:   for all functions  $f \in \text{CFG of PDS } \langle P, \Gamma, \Delta, s_0 \rangle$  do
3:      $\Sigma(f) := \Sigma_{*}^f$ ;
4:   while true do
5:     if  $n$  contained in function  $f$  s.t.  $\Sigma(f) = \Sigma_{*}^f$  then
6:        $n' := \text{exit node of } f$ ;
7:     else
8:        $n' := n$ ;
9:     Use BMC and  $\Sigma$  to construct a path formula  $\varphi$  ending at  $n'$ ;
10:    if  $\varphi(s_0, \langle p, \gamma \rangle)$  is satisfiable then
11:      if path does not traverse a function  $f$  with  $\Sigma(f) = \Sigma_{*}^f$  then
12:        return reachable;
13:      else
14:         $\Sigma(f) := \Sigma(f)_{\mathcal{Q}}^f$ ; ▷ Use  $\Sigma$  for function calls in  $f$ 
15:      else
16:        return unreachable;

```

Figure 4.8: Abstraction/Refinement algorithm for PDS

4.4 Recursion

In this section, we generalize corollary 4.4 and extend the algorithm in Figure 4.8 in order to enable the construction of universal summaries for recursive functions.

Given a recursive function f , we can compute an over-approximation Σ_{*1}^f of $\Sigma_{\mathcal{Q}}^f$ by replacing all recursive calls to f with Σ_{*1}^f . A refined over-approximation Σ_{*2}^f can then be obtained by applying Σ_{*1}^f for all calls to f . Unfortunately, this technique may fail to eliminate all spurious paths, since any Σ_{*i}^f contains a nested summary Σ_{*}^f (i.e., nested according to the grammar in Figure 4.1).

We can eliminate these spurious paths by “blocking” all paths in Σ_{*i}^f that traverse Σ_{*}^f , i.e., we can replace the corresponding expansion by *false*. Unfortunately, this approach may also eliminate feasible paths. The following Theorem states that it is sufficient to consider only paths up to a certain nesting depth:

Theorem 4.5 (Universal Summaries with Recursion) *Let r be the largest natural number for which both of the following conditions hold:*

1. There is a feasible path which contains r nested summaries, and
2. this path contains no pair of nested summaries $\langle p_0, \gamma_0 \rangle \leftrightarrow^* \langle p'_0, \gamma'_0 \rangle$ and $\langle p_1, \gamma_1 \rangle \leftrightarrow^* \langle p'_1, \gamma'_1 \rangle$ for which $\langle p_0, \gamma_0 \rangle = \langle p_1, \gamma_1 \rangle$ and $\langle p'_0, \gamma'_0 \rangle = \langle p'_1, \gamma'_1 \rangle$ holds.

We claim that

- a) such a finite r always exists and can be computed, and
- b) a summary that subsumes all loop-free paths with at most r nested summary applications is a universal summary $\Sigma_{\mathcal{P}}$.

Proof The proof of claim a) is simple: Given a summary $\langle p, \gamma \rangle \leftrightarrow^* \langle p', \gamma' \rangle$, we call $\langle p, \gamma \rangle$ the *entry-head* and $\langle p', \gamma' \rangle$ the *exit-head*. There are at most $|P|^2 \cdot |\Gamma|^2$ different combinations of entry- and exit-heads. After a path reaches a certain recursion depth $r \leq |P|^2 \cdot |\Gamma|^2$, the pairs of heads inevitably start to repeat. Furthermore, if this path exceeds a length l of at most $|P| \cdot \sum_{i=1}^r |\Gamma|^r$, then there is a state $\langle p, w \rangle$ that is visited twice. Thus, r can be computed by examining all paths up to depth $|P|^2 \cdot |\Gamma|^2$ and length $|P| \cdot \sum_{i=1}^r |\Gamma|^r$.

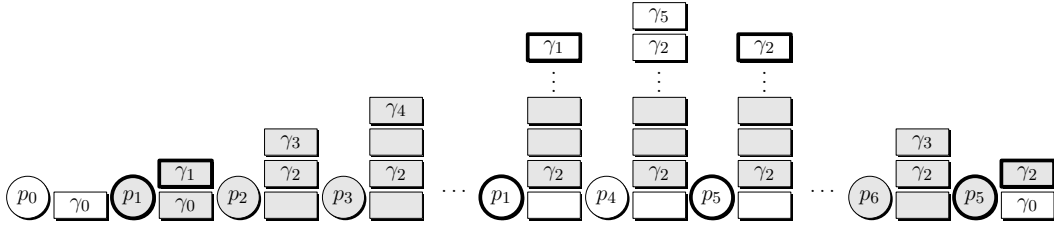


Figure 4.9: Nested summaries with repeating entry- and exit-heads

Claim b) follows from the observations made above: Figure 4.9 shows a path that contains two nested summaries for which the entry-heads and exit-heads are equal. By eliminating the control states and stack elements that are colored gray in Figure 4.9, we obtain a new path with a smaller number of nested summaries. This truncation has no impact on the states reachable from the final state $\langle p_5, \gamma_2 \gamma_0 \rangle$ of the path. Formally, let

$$\begin{aligned} \langle p_0, \gamma_0 \rangle \leftrightarrow \langle p_1, \gamma_1 w_1 \rangle \leftrightarrow \langle p_2, \gamma_2 w_2 \rangle \leftrightarrow \dots \leftrightarrow \\ \langle p_i, \gamma_i w_i \rangle \leftrightarrow \langle p_{i+1}, \gamma_{i+1} w_{i+1} \rangle \leftrightarrow \dots \leftrightarrow \langle p_{j-1}, \gamma_{j-1} w_{j-1} \rangle \leftrightarrow \langle p_j, \gamma_j w_j \rangle \\ \leftrightarrow \dots \leftrightarrow \langle p_{k-1}, \gamma_{k-1} w_{k-1} \rangle \leftrightarrow \langle p_k, \gamma_k w_k \rangle \leftrightarrow \langle p_{k+1}, \gamma_{k+1} \rangle \end{aligned}$$

be a summary of the PDS, where $|w_1| = |w_k|$, $|w_i| = |w_j|$, $|w_1| < |w_2| < \dots < |w_i| < |w_{i+1}|$ and $|w_{j-1}| > |w_j| > \dots > |w_{k-1}| > |w_k|$, i.e., the summaries $\langle p_1, \gamma_1 \rangle \leftrightarrow^* \langle p_k, \gamma_k \rangle$ and $\langle p_i, \gamma_i \rangle \leftrightarrow^* \langle p_j, \gamma_j \rangle$ are nested. Furthermore, let $p_1 = p_i$, $\gamma_1 = \gamma_i$, $p_k = p_j$, and $\gamma_k = \gamma_j$. Then, there exists a summary

$$\begin{aligned} \langle p_0, \gamma_0 \rangle \leftrightarrow \langle p_i, \gamma_i w_1 \rangle \leftrightarrow \langle p_{i+1}, \gamma_{i+1} w_r \rangle \leftrightarrow \dots \leftrightarrow \\ \langle p_{j-1}, \gamma_{j-1} w_i \rangle \leftrightarrow \langle p_j, \gamma_j w_k \rangle \leftrightarrow \langle p_{k+1}, \gamma_{k+1} \rangle \end{aligned}$$

that also covers $\langle p_0, \gamma_0 \rangle \leftrightarrow^* \langle p_{k+1}, \gamma_{k+1} \rangle$, and the nesting depth of this summary is smaller than the nesting depth of the original summary. Thus, any path with a pair of nested summaries can be truncated such that condition 2 holds without changing the set of states reachable via this path. A similar argument can be made for paths that contain a certain state twice (i.e., for paths that are not loop-free).

■

The same proof technique has been used by Richard Büchi to show that the set of reachable states of a PDS can be expressed using a finite automaton [35].

There is an obvious similarity between the reachability recurrence diameter presented in Section 4.2.2 and the bound for the nesting depth introduced in theorem 4.5. The reachability recurrence diameter of a PDS is two-dimensional and comprises of a sufficiently large nesting depth r and the length l of the longest loop-free path with a nesting depth at most r . This nesting depth can be computed symbolically using the same SAT-based unrolling technique: The longest loop-free path that contains no nested summaries is of length $|P| \cdot |\Gamma|$. If we increase the depth of the nestings to one, this length increases to $|P| \cdot (|\Gamma| + |\Gamma|^2)$, since each summary in this path may be of length $|P| \cdot |\Gamma|$. For any nesting depth greater than 1, we perform a pairwise comparison of the nested summaries. This can be achieved by means of a SAT-formula that is of quadratic size in the number of nested summaries. By repeatedly increasing the nesting depth r , we can determine the largest r for which the properties in theorem 4.5 hold. Note, that if the second condition of theorem 4.5 fails for all paths represented by a symbolic path that contains the summary Σ_{*i}^f , then it still fails if we replace Σ_{*i}^f with $\Sigma_{*(i+1)}^f$ (assuming that the first condition still holds after this transformation). In that case, it is not necessary to unroll the summary to the worst case depth $|P|^2 \cdot |\Gamma|^2$.

4.5 Experiments

We implemented the algorithm in Figure 4.8 and evaluated it using 40 PDSs generated by SLAM. In the scatter graphs in Figure 4.10 we distinguish between PDSs with reachable error locations (indicated by \times) and with unreachable error locations (indicated by \square). Figure 4.10(a) shows the scatter graph of our comparison of BEBOP and the version of BOOM (see Chapter 10) that applies universal summaries eagerly. We conclude from these results that our algorithm that applies universal summaries eagerly (BOOM) tends to perform better than the BDD based model checker BEBOP when the location in question is reachable in the PDS. It performs worse than BEBOP when it has to examine the whole state space. Figure 4.10(b) compares the effect of the lazy abstraction approach (called BOOM*) that we presented in Section 4.3 to the eager algorithm (BOOM*). The situation is less obvious than in Figure 4.10(a). To some extent, the algorithm improves the performance for model checking PDSs with reachable error locations. Unfortunately, this cannot be generalized. We observed that about a third of the non-deterministic summaries are not replaced by refined summaries by the algorithm. We believe that we can still improve on these results, but despite our efforts, the QBF solver remains the bottleneck of our approach.

4.6 Summary

BMC is an efficient technique for finding bugs. We showed how it can be applied to PDSs. Our algorithm uses BMC to compute *universal summaries* that relate arbitrary input states to their respective return states according to the transition relation of the function. Universal summaries can be applied in any calling context. We implemented an algorithm that uses SAT to compute universal summaries for functions without recursive calls, and QBF to compute summaries in the presence of recursion. Our benchmarks show that this approach performs better than BDD based algorithms when it comes to detecting bugs, but is less efficient for proving the unreachability of error states. This is a very useful result, since CEGAR-based model checkers generate PDSs with reachable error locations in all but the last iteration. Furthermore, we describe an extension to our algorithm that generates compute universal summaries for functions with recursion.

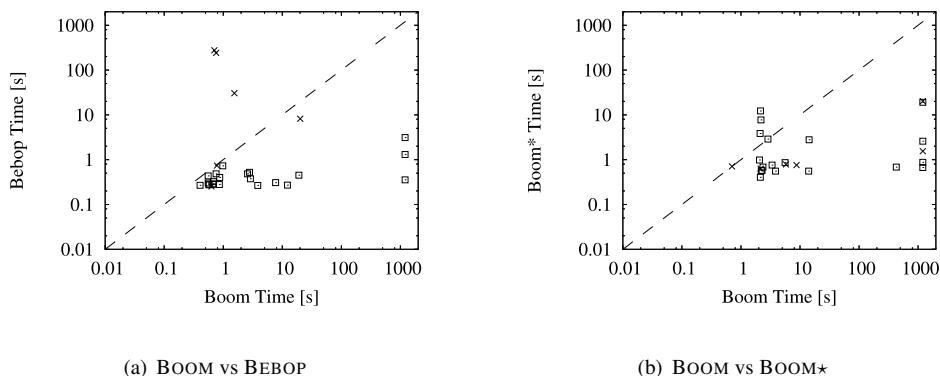


Figure 4.10: Comparison of BEBOP, BOOM, and BOOM*

4.7 Bibliographic Notes

The decidability of reachability properties of PDSs was shown by Büchi more than 40 years ago [35]. Efficient automata-based algorithms to construct the regular set of reachable states are presented by Finkel et al. [73] and Esparza et al. [69]. Schwoon improved the latter approach using a BDD-based symbolic representation of PDSs [122]. A saturation-based technique for similar models, namely recursive state machines, is presented in [1].

Summarization was introduced by Sharir and Pnueli as part of a dataflow analysis algorithm based on iterative fix-point detection [123]. Ball and Rajamani’s Model Checker BEBOP is based on this work and uses BDDs to represent states symbolically [13]. An implementation of their algorithm based on satisfiability solvers for propositional logic (SAT) and quantified Boolean formulas (QBF) is presented in [20] (see Section 4.2.1). Universal summaries are introduced in [20] (see Section 4.2). Kroening’s Model Checker BOPPO uses SAT-based symbolic simulation and QBF-based fix-point detection, but does not use summarization [46]. BOPPO requires that all function calls can be inlined. Leino combines BDDs and SAT-based techniques in his Model Checker DIZZY. He does not use summarization and reports that his benchmarks suggest that the approach is not scalable [103].

Several attempts have been made to extend the formalism of PDSs with concurrency. In that case, the reachability problem becomes undecidable. Various verification techniques for concurrent PDSs have been proposed, but are either unsound or incomplete: For instance, bounding the number of context switches [118] or bounding communication [125] may miss feasible paths, while over-approximating the set of reachable states [47, 30] may report spurious paths. We do not discuss these techniques here, since our approach is inappropriate for asynchronous systems: In general, there is no sufficiently large but finite bound for the sequential depth of concurrent PDSs.

Lal and Reps present a graph-theoretic approach for Model Checking *weighted* PDSs [101]. Their approach is incomparable to our algorithm, since we do not support weighted PDSs and their approach is not based on satisfiability solving techniques. Boujjani and Esparza survey approaches that use rewriting to solve the reachability problem for sequential as well as for concurrent pushdown systems [29].

BMC and the recurrence diameter [97] for finite state transition systems is discussed in Section 4.2.2. The SATURN verification tool uses SAT and summarization to detect errors in C pro-

grams [138], but handles loops in an unsound manner.

A recent publication shows that reachability of recursive programs with variables over finite domains can be decided by describing the analysis algorithm as a formula in a fixed-point calculus [131]. The authors provide an algorithm for analyzing Boolean programs using a model checker for μ -calculus formulae.

Part II

Concurrent Boolean Programs

5

Model Checking Concurrent Boolean Programs

WITH the advent of multi-core systems, a paradigm shift in the software industry happened. Nowadays most software systems are multi-threaded in order to utilize the parallel hardware. Unexpected interactions between threads generate subtle errors, which are very hard to find. Thus, *concurrent* software became an important target of computer-aided verification. Concurrent software verification means — in the scope of this thesis — model checking of concurrent Boolean programs, which will be discussed for the remaining sections.

In this chapter we will extend sequential Boolean programs with instructions to support dynamic thread creation and synchronization. Chapter 6 demonstrates how existing partial order reduction techniques can be combined with Boolean programs. Chapter 7 discusses efficient algorithms for model checking concurrent Boolean programs with bounded thread creation. We show how to apply *counter abstraction* to concurrent Boolean programs to factor out redundancy due to thread replication. Chapter 8 introduces an extended algorithm that allows unbounded dynamic thread creation.

5.1 Extensions with Concurrency Primitives

We extend the syntax of sequential Boolean programs (see Section 3.1) with four statements¹ to support concurrency. Dynamic thread creation is supported by the two following statements:

- The **start_thread goto** ℓ instruction creates a new thread that starts execution at the program location ℓ . It gets a copy of the local variables of the current thread, which continues execution at the proceeding statement.
- The **end_thread** statement terminates the actual thread, i.e., the actual thread is not executable

¹The syntax of Boolean programs can be found in appendix A.1.

anymore.

Atomic blocks are modeled using the keywords **atomic_begin** and **atomic_end**:

- The **atomic_begin** statement prevents the scheduler from making any context switches. That is, no thread is allowed to run other than the currently executing thread.
- The **atomic_end** instruction allows any thread to be executed.

The literature knows two classes of atomicity:

Weak Atomicity Only one thread is allowed to reside in a single atomic block. This model permits multiple threads to be in different atomic blocks.

Strong Atomicity Just one thread is permitted within an atomic block.

We support only the strong atomicity model, since it is anticipated by SATABS. An implementation of a scheduler that complies to this model could non-deterministically pick the next thread to execute from the set of runnable threads, if no thread is running atomically. If a thread has executed the **atomic_begin** statement, then the scheduler would schedule this thread only, until an **atomic_end** instruction is encountered. Note that such a scheduling policy does not need **atomic_begin** and **atomic_end** to form nested blocks.

Communication among threads is performed via *shared* variables. There exists no explicit locking statement, since locks can be simulated using shared variables (Listing 5.1).

Listing 5.1: Locking and unlocking simulated via access to a global variable

```

decl g_lock;      // (0 = free , 1 = taken)

void lock() begin
  g_lock := T constrain !g_lock;
end

void unlock() begin
  g_lock := F;
end

```

The support for concurrency is orthogonal to the methods used in the CEGAR loop, thus, a concurrent original program results in a concurrent Boolean program. An example of a translation of a fragment of the Apache web server into a Boolean program is given by Listing 5.2 and Listing 5.3.

5.2 Semantics of the Concurrent Language Extensions

With the extended syntax in mind, we are now able to describe the concurrent Boolean program semantics. Table 5.1 defines the extended statements and their semantics. The symbol *pc* represents the program counter, *V* the set of program variables. A global state has the atomic proposition *atomic* associated with it, which is set to *true* if context switching is prohibited, otherwise it is set to *false*. The symbol *id* denotes the thread identifier of the thread that is being executed. Analogously, the symbol

Listing 5.2: C program

```

while (1) {
  for (i = 0; i < ap_threads_per_child; i++) {

    int status = ap_scoreboard_image->servers[child_num_arg][i].status;
    if (status != SERVER.GRACEFUL && status != SERVER.DEAD)
      continue;

    apr_status_t rv = apr_thread_create(&threads[i], thread_attr,
                                       worker_thread, my_info, pchild);
    if (rv != APR.SUCCESS) {
      ap_log_error(APLOG_MARK, APLOG_ALERT, rv, ap_server_conf,
                  "apr_thread_create: cannot create worker thread");
      clean_child_exit(APEXIT_CHILDSICK);
    }
    threads_created++;
  }
  // handle service requests
}

```

Listing 5.3: Possible translation into a Boolean program

```

main() begin
  decl i_lt_ap_threads_per_child, status_eq_SRV_GRACEFUL,
        status_eq_SRV_DEAD, rv_eq_APR_SUCCESS;

  // predicate for "i < ap_threads_per_child":
  L0: i_lt_ap_threads_per_child := true;
  L1: goto L2, L9; // for-loop
  L2: assume (i_lt_ap_threads_per_child); // for-loop
      status_eq_SRV_GRACEFUL, status_eq_SRV_DEAD := *, *;
      goto L3, L4;
  L3: assume (status_eq_GRACEFUL || status_eq_SRV_DEAD);
      goto L8; // "continue"
  L4: assume (!status_eq_GRACEFUL && !status_eq_SRV_DEAD);
      start_thread Li; // Li: start location for thread
  L5: rv_eq_APR_SUCCESS := *;
      goto L6, L7; // "if (rv != APR_SUCCESS)"
  L6: assume (!rv_eq_APR_SUCCESS); // "if (rv != APR_SUCCESS)"
      rv_eq_APR_SUCCESS := *; // possible side effect of ap_log_error(...)
      goto L1; // end for
  L7: assume (rv_eq_APR_SUCCESS);
  L8: i_lt_ap_threads_per_child := *; // "i++"
      goto L1; // end for
  L9: assume (!i_lt_ap_threads_per_child);
      // handle service requests
      ...
      goto L0; // end while

```

Table 5.1: Semantics of fundamental Boolean program statements

Syntax	Semantics
$v_1, \dots, v_z := \text{expr}_1, \dots, \text{expr}_z$ constrain expr_c	$(\text{expr}_c \Rightarrow pc' = pc + 1 \wedge \forall i \in \{1, \dots, z\}, v'_i = \text{expr}_i \wedge$ $\text{same}(V \setminus \{v_1, \dots, v_z\})) \wedge (\neg \text{expr}_c \Rightarrow \perp) \wedge$ $(\text{atomic}' = \text{atomic}) \wedge (\text{atomic} \Rightarrow (id = id_{last}))$
goto l_1, \dots, l_z	$\bigvee_{l \in \{l_1, \dots, l_z\}} pc' = l \wedge \text{same}(V) \wedge$ $(\text{atomic}' = \text{atomic}) \wedge (\text{atomic} \Rightarrow (id = id_{last}))$
start_thread P	$pc' = pc + 1 \wedge \text{same}(V) \wedge$ (see main text) $(\text{atomic}' = \text{atomic}) \wedge (\text{atomic} \Rightarrow (id = id_{last}))$
end_thread	$\text{true} \wedge$ (see main text) $(\text{atomic}' = \text{atomic}) \wedge (\text{atomic} \Rightarrow (id = id_{last}))$
atomic_begin	$\text{atomic}' \wedge (\text{atomic} \Rightarrow (id = id_{last}))$ (see main text)
atomic_end	$\neg \text{atomic}' \wedge (\text{atomic} \Rightarrow (id = id_{last}))$ (see main text)

id_{last} stores the thread identifier of the last thread executed. Primes represent the next-state value of variables, and $\text{same}(Z)$ abbreviates $\bigwedge_{v \in Z} v' = v$, for some set of variables Z . The set of well-formed expressions is the Boolean closure of constants 0, 1 and \star (representing either value), and variable identifiers. The constructs `assume expr` and `skip` are shorthands for $v := v$ `constrain expr` and `assume 1`, respectively. We do not define semantics for branching statements such as `if` or `while`, because their behavior can be emulated using a combination of non-deterministic `gotos` and `assumes`. The `start_thread` and `end_thread` commands are used in Boolean programs concurrently executed by multiple threads and discussed in the next section. The table only shows their effects on the executing thread; see the next paragraph for side effects.

We sketch how a Boolean program \mathbb{P} induces a *concurrent system* \mathbb{P}^{\parallel} ; a full formalization is given by Cook, Kroening and Sharygina [48]. The set V of program variables is partitioned into two subsets V_s and V_l of *shared* and *thread-local* variables, respectively. A (global) state τ of \mathbb{P}^{\parallel} has the form (n, PC, Ω) , where n is the number of threads running in state τ , function $PC: \{1, \dots, n\} \rightarrow \{1, \dots, pc_{\max}\}$ maps each thread identifier to the PC the corresponding thread is located at, and $\Omega: (V_s \cup (\{1, \dots, n\} \times V_l)) \rightarrow \mathbb{B} \cup \{\star\}$ is the valuation of the program variables.

The execution model of \mathbb{P}^{\parallel} is asynchronous. That is, a step of \mathbb{P}^{\parallel} is performed by a single thread, say with identifier $i \in \{1, \dots, n\}$, executing the statement of \mathbb{P} at location $PC(i)$. Changes to the values of pc and the variables in V are reflected in updates to the state components PC and Ω , as indicated by symbolic constraints in Table 5.1. The value of n changes exactly in two circumstances:

- thread i executes a `start_thread P` command. In this case, the state is updated as follows. Let N be the bound on the number of threads that may be created. If $n < N$, then $n' = n + 1$, $PC'(i) = PC(i) + 1$, $PC'(n') = P$, and for each thread-local variable $v_l \in V_l$, $\Omega((n', v_l)) = \Omega((i, v_l))$, i.e., thread i is *cloned*. All other values are unchanged. If $n = N$, then $n' = n$, $PC'(i) = PC(i) + 1$, and all other values are unchanged. That is, if the number of dynamically created threads is exhausted, `start_thread` behaves like `skip` for the executing thread, and is free of side-effects.
- thread i executes an `end_thread` command. In this case, $n' = n - 1$, and PC and Ω are unchanged.

Finally, let n_0 be a natural number with $1 \leq n_0 \leq N$, the initial number of threads. The set of initial states of \mathbb{P}^{\parallel} is given by $n = n_0$, $PC(i) = 0$, and $\Omega((i, v_l)) = \star$, for each i in $\{1, \dots, n_0\}$ and each thread-local variable v_l . A classical concurrent system of a fixed number of threads is an instance of this formalization with $n_0 = n = N$ and a Boolean program without `start.thread` or `end.thread` commands.

When reasoning about the concurrent program \mathbb{P}^{\parallel} , the notation of a state can be simplified by considering the PC simply as a thread-local variable. In that case, a state of \mathbb{P}^{\parallel} can be described in the form (s, l_1, \dots, l_n) , where vector s is a valuation of the shared variables V_s , and l_i stands for the *local state* of thread i , comprising the value of the program counter $PC(i)$ and the value $\Omega((i, v_l))$, for each thread-local variable $v_l \in V_l$. The translation between the notations (n, PC, Ω) and (s, l_1, \dots, l_n) is straightforward. The *thread state* of thread i is the pair (s, l_i) . Intuitively, for $i \in \{1, \dots, n\}$, thread i has full access to the shared variables and to the i -th copy of the thread-local variables. It has neither read nor write access to any other thread-local variables.

5.3 Thread-State Reachability

The concurrent verification problem considered in this thesis 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.

5.4 Overview of Model Checking Methods

The model checking problem for reachability properties of sequential Boolean programs is, as we have seen in Chapter 4, decidable even in the presence of unbounded stacks. In contrast, recursion renders the concurrent verification problem undecidable, even for Boolean programs [121].

There are several approaches known in literature to get decidability for the concurrent model checking problem. We will — without making a claim to be complete — present a short survey of possible approaches.

No stacks. Decidability can be retained, if recursion is prohibited. Thread-state reachability properties are decidable for replicated finite-state systems, even in the presence of dynamic thread creation. This model has the advantage, that it is still exact, whereas other approaches described in this section lose precision. Because Boolean programs are already abstractions of higher-level programs, it is desirable to obtain exact results for this case. Therefore this is the approach taken within this thesis and will be discussed in the following chapters.

Bounded context switching. Quadeer and Wu [119] show that many bugs in concurrent software emerge with a small number of context switches. They coin the term *context-bounded reachability* and prove that the bounded context-switching reachability problem is decidable, even for recursive programs [118].

In [102], Lal and Reps propose an *eager* context-bounded translation of concurrent Boolean programs into sequential ones in order to reduce the problem to sequential reachability. Their approach consists of two parts. First, they specify program locations where context switches are allowed to happen and set the shared variables to non-deterministic values at exactly these points, then each thread is

checked individually. In a second step these over-approximated runs are stitched together to produce the full trace and rule out infeasible paths. This approach avoids creating a BDD that contains the global state, instead only thread-states are built.

Subsequently, this work is amended with a *lazy* approach by Torre et al. [132]. Their state representation is based on thread-states as well but manages to avoid the over-approximation that is happening in the first part of the eager algorithm. The key idea is to restart the thread that is supposed to take control after a context switch, until a point is reached, where the shared variables match with the current state. It is worth mentioning that in the context of a summarizing sequential model checker as the underlying reachability engine, the re-running step corresponds to a mere look-up of a summary that has been computed in a previous context.

The same authors also show that an even less restricted system is decidable: it suffices to bound the number of phases, where in each phase all processes can enter procedures but only one is allowed to return from the procedures [130].

Thread-modular model checking. One option to cope with recursion is to use an over-approximation, as done by Cook, Kroening, and Sharygina in [48]. The authors propose an algorithm that simulates an infinite number of threads by introducing new transitions that apply any observable transition of any thread to any state in the history.

Summarization. The summarization idea for sequential model checking is extended to concurrent software in [117]. Although the problem remains undecidable in general, the authors present an algorithm is able to terminate for some programs.

Others. Rewriting-based reachability analysis of concurrent pushdown system is covered by [30] and [33]. Boujjani and Esparza survey approaches that use rewriting to solve the reachability problem for sequential as well as for concurrent pushdown systems [29].

6

Partial Order Reduction

WHEN computing the successors of a concurrent symbolic state, we usually have to consider the possibility that any of the enabled threads can make a transition. The problem is that the number of states explored can grow dramatically with the number of threads. In contrast to that, a sequential program only requires as many symbolic states as there are execution steps. The different orderings of the interleaved executions of the threads cause state space explosion. A well-known remedy to limit this is *partial-order reduction*. It exploits the fact that the effect of concurrent executions is often independent of their ordering. In this chapter we present several approaches to integrate well-known partial order reduction techniques into a symbolic model checker for Boolean programs.

6.1 Introduction

The purpose of *partial-order reduction* (POR) [88] is to explore only a representative subset of all possible program traces. The key idea is to identify *commutable* transitions, and pick only a representative schedule of those transitions. Deciding what constitutes a representative schedule, however, may be as hard as the Model Checking problem itself. Instead, static analysis techniques can be used to approximate the dependencies between transitions, resulting in suboptimal but affordable POR methods.

The traditional literature classifies these methods into two following approaches:

The “Sleep sets” [79] technique avoids executing all transitions but still explores the whole state space. “Persistent sets” [79] reduce the number of states that have to be explored by checking if a transition potentially collides with an *infinite* future of another thread.

Several approaches have been presented to combine partial-order reduction with symbolic Model Checking. One possibility is an up-front transformation of the system [99] such that redundant interleavings are prohibited. The idea is to restrict the transition relations of a thread depending on the local state of other threads. Unfortunately this method could result in utterly complex transition relations

and can not be combined in a straightforward way with our symbolic counter abstraction algorithm. A recent method [93] that is specially tailored for a SAT-based Model Checker encodes the scheduler within the formula passed to the SAT-solver and adds additional constraints in order to give the solver hints where the search space can be pruned.

Since we use a *hybrid* approach to represent states, we are able to use classic partial-order reduction techniques without the additional overhead that symbolic methods usually bring along.

6.2 Ample Sets

The approach we take is related to what many explicit state model checkers implement. We look for a thread t that makes an *invisible* transition: one that is independent of *any possible future transitions* made by *any other thread* $t' \neq t$ [46]. This is known as *ample set* in the literature [40]. In order to give a formal definition of an ample set, we need to introduce some notation. Recall that an abstract global state is a sequence of tuples consisting of local state and counter: $\sigma = \langle S, (L_1, n_1), \dots, (L_k, n_k) \rangle$. We use $\sigma \xrightarrow{\alpha} \sigma'$ to denote that an α -labeled transition from state σ to state σ' is possible.

Definition 6.1 (Ample Set) *The literature [40] defines four constraints that an **ample set** must fulfill:*

At-least-one-successor condition (C0): $Ample(\sigma) = \emptyset$ if and only if $Enabled(\sigma) = \emptyset$. A transition α is called *enabled* in a state σ , if there exists a state σ' such that $\sigma \xrightarrow{\alpha} \sigma'$, and we write $\alpha \in Enabled(\sigma)$ to denote this fact.

Dependent-transition condition (C1): A long every execution sequence of transitions in a Boolean program \mathcal{P} that starts at σ , a transition that is dependent on any transition in $Ample(\sigma)$ cannot be executed without a transition from $Ample(\sigma)$ occurring first.

Visibility condition (C2): If $Ample(\sigma) \neq Enabled(\sigma)$, then every transition $\alpha \in Ample(\sigma)$ is *invisible*. This condition also holds by construction since we only seek after invisible transitions.

Cycle condition (C3) [113]: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $Ample(\sigma)$ for any state σ on the cycle.

6.2.1 Symbolic Computation of Ample Sets

Our method of computing ample sets is explained in Algorithm 6.1.

Algorithm 6.1 Computing an ample set $Ample(\sigma)$ for state σ

```

1: procedure GET_AMPLE( $\sigma$ )
2:   for all  $\alpha_t \in Enabled(\sigma)$  do
3:     if  $\alpha$  is goto statement with a back edge then continue
4:     if  $W_t(\alpha) \cap (\bigcup_{i \neq t} R_i^\infty(n) \cup W_i^\infty(\alpha)) = \emptyset$  then  $\triangleright \nexists \beta_i$  that is dependent on  $\alpha_t$ 
5:       if  $R_t(\alpha) \cap \bigcup_{i \neq t} W_i^\infty(\alpha) = \emptyset$  then
6:         return  $\{\alpha_t\}$ 
7:   return  $Enabled(\sigma)$ 

```

The algorithm ensures each condition mentioned in Definition 6.1 as follows.

- C0:** This condition guarantees that if at least one transition is enabled, then the reduced set of transitions also contains a transition. This condition is fulfilled by construction of the ample set because we execute all thread interleavings if no invisible transition can be found. Any thread that has not yet terminated is enabled in our setting.
- C1:** This condition ensures that all transitions of any thread $t' \neq t$ are independent of the transition $\alpha_t \in \text{Ample}(\sigma)$.

As [40] points out, the condition **C1** is the most difficult among all constraints. A computation of the exact dependencies among transitions can be as hard as the model checking problem itself. Therefore we use static analysis techniques to over-approximate the dependencies between transitions.

Our strategy is based on the set W_t of variables written by thread t , and the set R_t of variables read by t , in the current state. Analogously, let R_t^∞ and W_t^∞ denote the set of variables read or written at some time in the future (present included). As common in static analysis, these sets are computed using data-flow equations based on conservative assumptions of what constitutes a read or write. In particular, the set W_t contains all variables that appear in an instruction that could restrict the state space. Such an instruction may disable some instructions of other threads and must thus be considered a write. Specific to Boolean programs, the **assume** statement and constrained assignments belong to this category; see Example 6.1 below.

We use these sets in our POR algorithm as follows. If a thread t is found satisfying $W_t \cap (\bigcup_{i \neq t} R_i^\infty \cup W_i^\infty) = \emptyset$ (no variable written by t is ever used by another thread) and $R_t \cap \bigcup_{i \neq t} W_i^\infty = \emptyset$ (no variable read by t is ever written by another thread, Lines 4-5), we only explore the successors generated by executing t , but not by any other thread. Intuitively, t does not communicate with other threads during this transition. All other transitions are discarded at the current state. We illustrate this technique with a few examples.

- C2:** This condition also holds by construction since we only seek invisible transitions.
- C3** Without this condition there is a possibility that some transitions will be postponed indefinitely because of a cycle in the reduced model.

The cycle condition can be ensured as follows: if the instruction to be executed is not a **goto** statement with a *back edge*, then the normal ample set is computed (Line 3). Otherwise all interleavings are considered. This check is very favorable, since the program counter is stored explicitly. A similar approach to avoid delaying a transition forever has been proposed in [2]: the authors define an ample function with respect to the current history. That is, it insists on exploring all enabled transitions unless a state is visited, which is not in the current history. Thus, if a thread is looping in a cycle, eventually any succeeding state has been seen before and the postponed transitions are explored. However, in this setting, all interleavings may be explored more frequently than with our back edge detection approach.

6.2.2 Context-Sensitive Over-Approximation of Dependencies

Let a Boolean program \mathcal{P} be given as a control-flow graph (CFG) consisting of a set of nodes $\mathcal{N}_{\mathcal{P}}$ and their successor function $\text{Succ}_{\mathcal{P}} : \mathcal{N}_{\mathcal{P}} \mapsto 2^{\mathcal{N}_{\mathcal{P}}}$.

The sets of variables that are read or written in the infinite future by a thread executing node n can be over-approximated using equations that are similar to equations known from a classical live-variable analysis (see Table 6.1).

$$\begin{aligned}
R(n) &= \text{variables read by } n \\
W(n) &= \text{variables written by } n \\
R_{in}^\infty(n) &= R(n) \cup R_{out}^\infty(n) \\
R_{out}^\infty(n) &= \bigcup_{s \in Succ(n)} R_{in}^\infty(s) \\
R_{in}^{atomic}(n) &= \begin{cases} \emptyset & \text{if } n \equiv \text{'atomic_end'} \\ R(n) \cup R_{out}^{atomic}(n) & \text{else} \end{cases} \\
R_{out}^{atomic}(n) &\equiv \bigcup_{s \in Succ(n)} R_{in}^{atomic}(s) \\
W_{in}^\infty(n) &= W(n) \cup W_{out}^\infty(n) \\
W_{out}^\infty(n) &= \bigcup_{s \in Succ(n)} W_{in}^\infty(s) \\
W_{in}^{atomic}(n) &= \begin{cases} \emptyset & \text{if } n \equiv \text{'atomic_end'} \\ W(n) \cup W_{out}^{atomic}(n) & \text{else} \end{cases} \\
W_{out}^{atomic}(n) &\equiv \bigcup_{s \in Succ(n)} W_{in}^{atomic}(s)
\end{aligned}$$

Table 6.1: Data-flow equations to over-approximate the sets of variables that might be read or written by a thread at node n .

Sets of variables that are read or written by an instruction can be computed using a syntactic analysis over the expressions that are used (see Table 6.2.2). It is somewhat surprising that the effect that an **assume** statement has on the shared variables must be considered as a write operation. This stems from the fact a thread that executes an **assume** instruction might restrict the symbolic state and thus *disable* transitions of other threads.

Example 6.1 Consider the Boolean program in Listing 6.1. Suppose the **assume** instruction counts as a read access to the variables in its expression (only s in this case). Then, after the second thread has been created, both threads are about to execute invisible statements only. If POR picks the thread that continues execution at $P1$ and runs it until the end, the second thread cannot reach the **assert** instruction anymore, and the path that violates the property will remain undiscovered.

Example 6.2 Consider the Boolean program in Listing 6.2, and a state where the threads are at location $P1$ and $P2$, respectively. One thread is about to execute the invisible statement $\perp := \top$. The other thread is reading from s and thus does not interfere with the instruction of the first thread. One might therefore be tempted to regard the **assert** statement as invisible, and omit other possible interleavings at this point. The consequence would again be that the violation of the assertion in the other thread goes undetected.

Atomic Sections All instructions within an **atomic_begin** / **atomic_end** block need to be considered as a single instruction and therefore the sets of variables that can be read and written within such a block must be considered. These sets are over-approximated as R^{atomic} and W^{atomic} in the data-flow equations (Table 6.1).

Listing 6.1: An **assume** stmt. disabling other threads

```

decl s;

void main() begin
  s := *;

  start_thread P2;

P1: assume(s);
  end_thread;

P2: assume(!s);
  assert(F);

end

```

Listing 6.2: Effect of future transitions on POR

```

decl s;

void main() begin
  decl l;

  s := T;
  start_thread P2;

P1: l := T;
  s := F;
  end_thread;

P2: assert(s);

end

```

Instruction	R	W
skip, goto ℓ_1, \dots, ℓ_k	\emptyset	\emptyset
assume e	$\omega(e)$	$\omega(e)$
assert e	$\omega(e)$	\emptyset
$x_1, \dots, x_k := e_1, \dots, e_k$ constrain e	$\omega(e_1) \cup \dots \cup \omega(e_k) \cup \omega(e)$	$\{x_1, \dots, x_k\} \cup \omega(e)$
atomic_begin	R^{atomic}	W^{atomic}
atomic_end	\emptyset	\emptyset

Table 6.2: Sets of variables read and written.

6.2.3 Benchmarks

We measured the effect of the symbolic partial-order reduction using ample sets with the plain symbolic implementation on the benchmarks described in appendix B. The performance improvements are illustrated in Figure 6.1; on average a speedup of 4.2 has been measured and 85% of for all benchmarks were accelerated.

6.3 Symbolic Computation of Persistent Sets

If we cannot find an ample set at a state, then we might still be able to compute a persistent set [79]. We aim at splitting the set of threads into subsets such that every thread only communicates with threads among its subset. More formally a sets of threads $P(\sigma_0) \in \text{Enabled}(\sigma_0)$ is *persistent* in σ_0 , if and only if for all $\beta \in P(\sigma_0)$ and all sub-traces $\sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \dots \sigma_n \xrightarrow{\alpha_l} \sigma_{n+1}$ obtained from transitions $\alpha_i \notin P(\sigma_0)$, β and α_i are independent in σ_i .

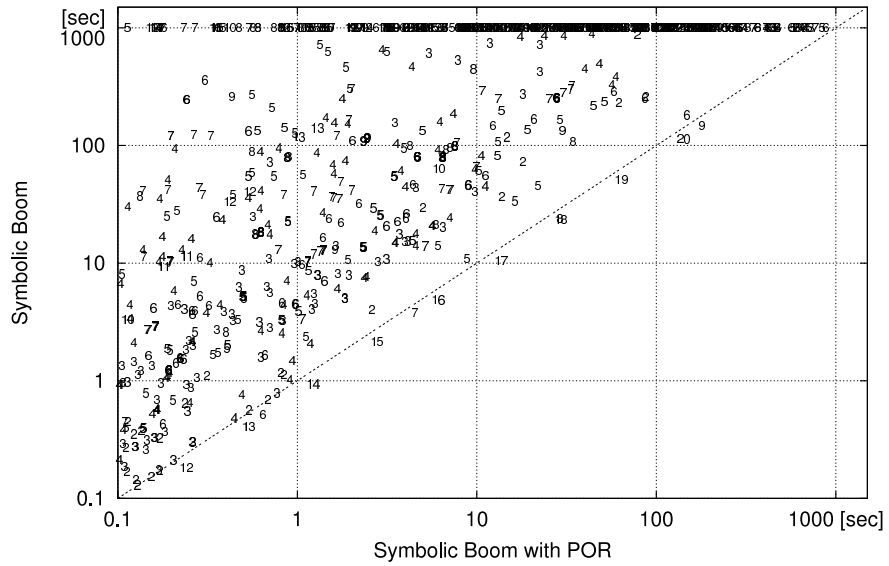


Figure 6.1: Running time of (symbolic) BOOM with partial-order reduction vs. (symbolic) BOOM, for various thread counts

Algorithm 6.2 describes our approach to compute persistent sets. It resembles the ample set algorithm (Algorithm 6.1) but the notion of independence is slightly different: a thread t_i belongs to a different persistent set than thread t_j , if every of t_i 's *future* transitions is independent of any *future* transition of t_j .

Algorithm 6.2 Computing persistent sets $Persistent(\sigma)$ at state σ

```

1: procedure GET_PERSISTENT_SETS( $\sigma$ )
2:    $P(\sigma) := \{\{t_1\}, \dots, \{t_k\}\}, t_{1,\dots,k} \in Enabled(\sigma)$ 
3:   for all  $P_i(\sigma) \in P(\sigma)$  do ▷ Merge dependent sets
4:     for all  $P_j(\sigma) \in P(\sigma)$  do
5:       if  $i > j$  then continue
6:       if ARE_SETS_DEPENDENT( $P_i(\sigma), P_j(\sigma)$ ) then
7:          $P_i(s) := P_i(s) \cup P_j(s)$ 
8:          $P(s) := P(s) - P_j(s)$ 
9:   return  $P(\sigma)$ 

10: procedure ARE_SETS_DEPENDENT( $P(\sigma), P'(\sigma)$ )
11:   return  $\exists t \in P, \exists t' \in P'. (W_t^\infty \cap (R_{t'}^\infty \cup W_{t'}^\infty) \neq \emptyset \text{ or } R_t^\infty \cap W_{t'}^\infty \neq \emptyset)$ 

```

Note that the cycle condition has been omitted as well. Instead we make use of the following observation: if a persistent set $P(\sigma)$ has been found, i.e., $P(\sigma) \neq Enabled(\sigma)$, then any persistent set of a successor state σ' of σ will be equal or a subset of $Persistent(\sigma)$.

More formally: $\exists P(\sigma') \in Persistent(\sigma'), \exists P(\sigma) \in Persistent(\sigma). P(\sigma') \subseteq P(\sigma)$.

If persistent sets have been found, we create a copy of the symbolic state for each persistent set and disable any thread not in that set (see Algorithm 6.3). Thus instead of postponing transitions of threads not in the persistent set, we partition the search space up-front such that unnecessary interleavings are not allowed to happen anymore. In fact, this corresponds to *Cartesian Semantics* (see Section 6.4) and provides an explanation why Cartesian partial order reduction saves at least as much states as persistent sets.

Algorithm 6.3 Partitioning a symbolic state σ according to its persistent sets

```

1: procedure PARTITION_STATE( $\sigma$ )
2:    $\mathcal{S} := \emptyset$ 
3:    $Persistent(\sigma) = GET\_PERSISTENT\_SETS(\sigma)$ 
4:   for all  $P(\sigma) \in Persistent(\sigma)$  do
5:      $\hat{\sigma} := \sigma$  such that  $Enabled(\hat{\sigma}) = P(\sigma)$ 
6:     add  $\hat{\sigma}$  to  $\mathcal{S}$ 
7:   return  $\mathcal{S}$ 

```

6.4 Cartesian Partial Order Reduction

Cartesian partial order reduction (see [82]) delays context switches as known from standard partial order reduction techniques, but in contrast, only considers a *finite* future. If no thread is about to execute an invisible transition (i.e., a transition that has no influence on the infinite future of any other thread), then a reduction might still be possible, if the current transition of some thread has no influence

on a finite future of any other thread. This results in an effective reduction technique, if one or more threads are executing loops in a way such that there is a circular dependency between these threads.

This is illustrated in Listing 6.3¹. The main thread creates two threads: t_1 increments a shared 3-Bit counter and then checks an assertion on a shared flag variable. The thread t_2 just flips the shared variable once and subsequently checks an assertion on the counter. Starting from a state where t_1 and t_2 are created, Cartesian partial-order reduction leads to exploration of two paths (also called vectors): one path where always t_1 is executed until it hits the statement **assert (g)** and one path where t_2 flips the global variable. Only after this point, the threads are interleaved. Thus interleavings between statements within the loop of t_1 and the global variable flip of t_2 are discarded. Using ample sets or persistent sets would not help to remove the redundant interleavings because there is a mutual dependency between the two threads.

Listing 6.3: Example where Cartesian partial-order reduction is able to reduce the number of interleavings

```

decl a,b,c;
decl g;

void t1() begin
  while (!a|!b|!c) do
    if (!a) then
      a := 1;
    elsif (!b) then
      a,b := 0,1;
    elsif (!c) then
      a,b,c := 0,0,1;
    fi
  od
  assert(g);
end

void t2() begin
  g := !g;
  assert(!(a&b&c));
end

void main() begin
  a,b,c,g := 0,0,0,0;
  atomic_begin;
  ASYNC.1: t1();
  ASYNC.2: t2();
  atomic_end;
end

```

Listing 6.4: Example where persistent sets are more optimal than cartesian vectors

```

decl x,y;

void t1() begin
  goto L1, L2;
L1:   x := !y;
      x := !x;
      x := !x;
      x := !x;
L2:   skip;
end

void t2() begin
  goto L1, L2;
L1:   y := !x;
      y := !y;
      y := !y;
      y := !y;
L2:   skip;
end

void main() begin
  atomic_begin;
  ASYNC.1: t1();
  ASYNC.2: t1();
  ASYNC.3: t2();
  ASYNC.4: t2();
  atomic_end;
end

```

Cartesian semantics do not necessarily outmatch persistent sets, e.g., consider Listing 6.4: there are two groups of threads that communicate among each other. If every spawned thread has executed its first instruction, then it is sufficient to interleave only the threads that execute t_1 or t_2 respectively. Thus this fact is recognized by the persistent sets algorithm. A model checker that relies on Cartesian partial order will not be able to extend its Cartesian vectors with more than one statement at a time,

¹We are using **while** and **if** statements to make the example more readable. BOOM (see Chapter 10) is able to parse this example and rewrites these keywords internally into **assume** and **goto** statements.

since there is a dependency between two threads.

A comparison of the symbolic algorithm with an implementation that uses Cartesian partial order reduction is presented in Figure 6.2. On average a speedup of 160 resulted and 91% of the benchmarks were solved quicker².

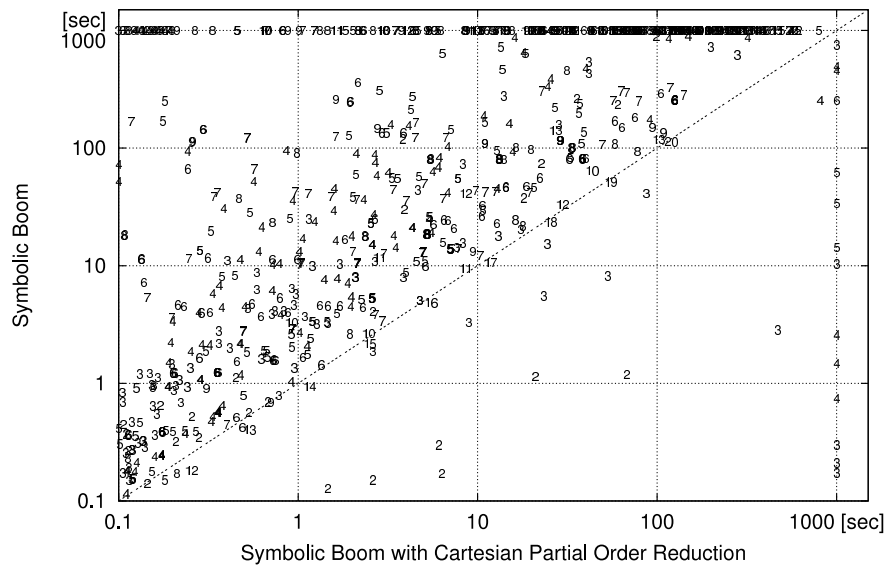


Figure 6.2: Running time of symbolic BOOM vs symbolic BOOM with Cartesian partial order reduction

Figure 6.3 illustrates that a combination of ample sets and Cartesian partial order reduction is beneficial in most of the cases. Almost 80% of the benchmarks show an improvement (which is a factor of 167 on average).

²Appendix B explains the experimental setup.

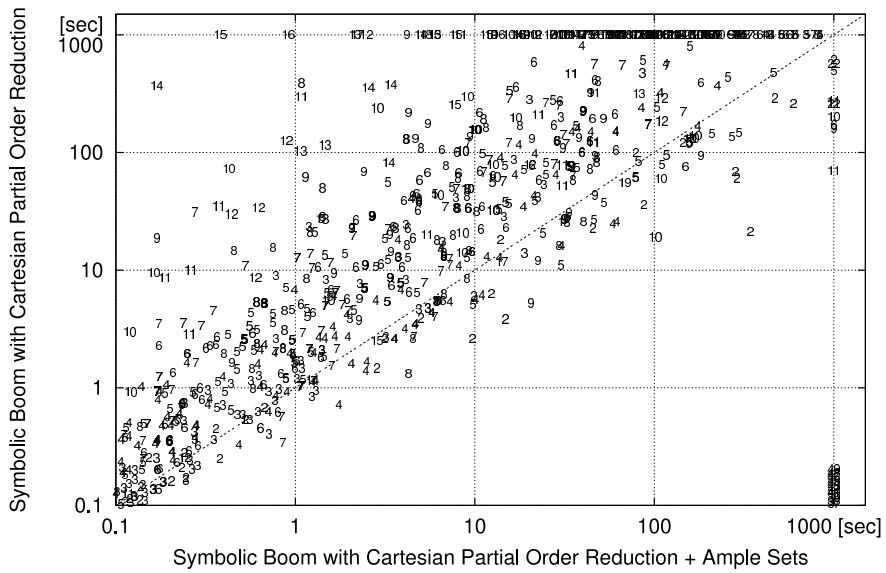


Figure 6.3: Running time of symbolic BOOM with Cartesian partial Order reduction vs symbolic BOOM with Cartesian partial Order reduction and ample sets

7

Concurrent Boolean Programs with Bounded Replication

IN this chapter we show how to apply *counter abstraction* to real-world concurrent programs to factor out redundancy due to thread replication. The traditional global state representation as a vector of local states is replaced by a vector of thread counters, one per local state. In practice, straightforward implementations of this idea are unfavorably sensitive to the number of local states. We present a novel symbolic exploration algorithm that avoids this problem by carefully scheduling which counters to track at any moment during the search.¹

7.1 Introduction

Recently, there have been attempts to extend *Predicate abstraction* (see Section 2.6) to the verification of *concurrent* software [137]. The resulting abstractions face the classical state space explosion problem: the number of reachable program states grows exponentially with the number of concurrent threads, which renders naive exploration impractical. The authors of [137] conclude that none of the currently available tools is able to handle device drivers of realistic size in the presence of many threads.

One observation that comes to the rescue is that concurrent components of multi-threaded software are often simply *replications* of a template program describing the behavior of a component. The ensuing regularity in the induced system model can be exploited to reduce the verification complexity. One technique towards this goal is *counter abstraction*. The idea is to record the global state of a system as a vector of counters, one per local state, tracking how many of the n components currently reside in the local state. This technique turns a formal model of size exponential in n into one of size polynomial in n , promising a serious stab at state space explosion.

¹This chapter presents and extends our work that has been published at the International Conference on Computer Aided Verification in the year 2009 [21].

Emerson and Trefler proposed counter abstraction as a way of achieving *symmetry reduction* for fixed-size systems [67]. In their approach, the template program \mathbb{P} is converted into a local-state transition diagram, by identifying a set of local states a component can be in, and translating the program statements into local state changes. Such a conversion is straightforward if there are only few component configurations, such as with certain high-level communication protocols [58]. For concurrent software, however, \mathbb{P} is given in a C-like language, with assignments to variables, branches, loops, etc. A local state is then defined as a valuation of all thread-local variables of a thread. As a result, there are exponentially many local states, measured in the number of thread-local variables. Introducing a counter variable for each local state is impractical and only possible for tiny programs.

In this chapter, we present a strategy to solve these complexity problems. Our solution is two-fold. First, we interleave the translation of individual program statements with the model checking phase. This has the advantage that our algorithm is **context-aware**: the local-state context in which the statement is executed is known; the context determines which local-state counters need to be updated. If the translation is performed up-front, one has to embed each statement into all contexts where the statement is enabled, which is infeasible for realistic programs. As a side-effect of the on-the-fly translation, only counters for *reachable* local states are ever introduced. Second, in a global state we keep counters only for those local states that at least one thread resides in. This idea leverages a simple counting argument: given n threads with l conceivable local states each, at most n of the corresponding local state counters are non-zero at any time during execution. Since n is typically much smaller than l , omitting the zero-valued counters results in huge savings: the sensitivity of counter abstraction to the *local state space explosion problem* mentioned in the previous paragraph is reduced from exponential in l to exponential in $\min\{n, l\}$.

Contributions. We present an efficient algorithm for BDD-based symbolic state space exploration of Boolean programs executed by a bounded number of possibly dynamically created parallel threads. This generalizes the traditional setting for symmetry reduction of systems with a fixed number of components known at modeling time. The algorithm’s primary accomplishment is to curb the local state space explosion problem, the classical bottleneck in implementations of counter abstraction. We demonstrate the effectiveness of our approach on a substantial set of Boolean program benchmarks, generated by two very different CEGAR-based toolkits, SATABS [44] and SLAM [15]. Since symmetry reduction, of which bounded counter abstraction is an instance, has so far been implemented more successfully in explicit-state model checkers, we also include an experimental comparison of an explicit-state version of our method against explicit-state symmetry reduction, using the well-known MUR ϕ model checker [109]. Finally, this chapter provides an extensive comparison of our counter abstraction method with *partial-order reduction*, an alternative technique to curb the verification complexity for programs with interleaved concurrent threads.

We believe our algorithm marks a major step towards the solution of an exigent problem in verification today, namely that of model checking concurrent software. While the concepts underlying our solution are relatively straightforward, exploiting them in symbolic model checking is not. The succinctness of state space representations that BDDs often permit is paid for by rather rigid data manipulation mechanisms. To the best of our knowledge, our implementation is the first *scalable* approach to counter abstraction in symbolic verification of concurrent software with replicated threads.

7.2 Preliminaries

7.2.1 Execution Model

Concurrent Boolean programs are described in Section 5.1. In this chapter we use an execution model that bounds the number of threads that are allowed to run concurrently. That is, we modify the semantics as follows. Let thread i execute a `start_thread P` command. Let N be the bound on the number of threads that may be created. If $n < N$, then thread i is *cloned*: $n' = n + 1$, $PC'(i) = PC(i) + 1$, $PC'(n') = P$, and for each thread-local variable $v_l \in V_l$, $\Omega((n', v_l)) = \Omega((i, v_l))$. All other values are unchanged. If $n = N$, then $n' = n$, $PC'(i) = PC(i) + 1$, and all other values are unchanged. That is, if the number of dynamically created threads is exhausted, `start_thread` behaves like `skip` for the executing thread, and is free of side-effects.

Let finally n_0 be a natural number with $1 \leq n_0 \leq N$, the initial number of threads. The set of initial states of the concurrent system \mathbb{P}^{\parallel} is given by $n = n_0$, $PC(i) = 0$, and $\Omega((i, v_l)) = \star$, for each i in $\{1, \dots, n_0\}$ and each thread-local variable v_l . A classical concurrent system of a fixed number of threads is an instance of this formalization with $n_0 = n = N$ and a Boolean program without `start_thread` or `end_thread` commands.

The code is assumed to be free of recursion. In a preprocessing step, any function calls are inlined.

7.2.2 Symmetry Reduction

Full symmetry is the property of a Kripke model of concurrent components to be invariant under permutations of these components. This invariance is traditionally formalized using permutations. A permutation π on $\{1, \dots, n\}$ is defined to act on a state $\sigma = (s, l_1, \dots, l_n)$ by acting on the thread indices, i.e. $\pi(\sigma) = (s, l_{\pi(1)}, \dots, l_{\pi(n)})$. We extend π to act on a transition (σ, τ) by acting point-wise on σ and τ .

Definition 7.1 *Structure M with transition relation R is (fully) symmetric if for all $r \in R$ and all permutations π on $\{1, \dots, n\}$, $\pi(r) \in R$.*

We observe that a concurrent Boolean program built by *replicating* a template written in the syntax given in Section 7.2.1 is (trivially) symmetric: the syntax does not allow thread identifiers in the program text, which could potentially break symmetry.

From a symmetric structure M , a reduced *quotient structure* \overline{M} can be constructed using standard existential abstraction. The quotient is based on the *orbit relation* on states, defined as $\sigma \equiv \tau$ if there exists π such that $\pi(\sigma) = \tau$.

Theorem 7.2 ([39, 60]) *Let f be a μ -calculus formula with atomic propositions that are invariant under thread index permutations. Let further σ be state of M and $\overline{\sigma}$ be the equivalence class of σ under \equiv .*

$$M, \sigma \models f \quad \text{iff} \quad \overline{M}, \overline{\sigma} \models f.$$

Thus, verification over M can be replaced by verification over \overline{M} , without loss of precision. This theorem can be proved by a bi-simulation argument; the bi-simulation relation between M and \overline{M} relates states of M to their orbit equivalence classes. In addition, \overline{M} is roughly exponentially smaller than M : the equivalence classes of \equiv collapse up to $n!$ many states of M . Symmetry reduction thus combines two often antagonistic features of abstractions – precision and compression.

Symmetry reduction in the above formalization has been tremendously successful as an abstraction technique in model checkers based on explicit state enumeration; see Section 7.7 for examples. It has enjoyed much less popularity in BDD-based symbolic model checking. The reason is that the state canonization that is required by symmetry reduction can be expensive to perform using BDDs. In particular, it was shown in [39] that the orbit relation has no succinct BDD representation, for any variable order. Emerson and Wahl present a symbolic symmetry reduction technique that avoids the orbit relation but still relies on state canonization [68].

Counter abstraction, the topic of this chapter, can be viewed as a form of symmetry reduction where state canonization is an automatic by-product of the state representation and thus does not have to be performed explicitly. The new representation, if used naively, has to be paid for with a blow-up of the program text, however, as we demonstrate in the next section.

7.3 Classical Counter Abstraction – Merits and Problems

Counter abstraction is an alternative formalization of symmetry reduction, namely using process counters. The idea is that two global states are identical up to permutations of the local states of the components exactly if, for every local state L , the same number of components reside in L . To implement this idea, we introduce a counter for each existing local state and translate a transition from local state A to local state B as a decrement of the counter for A and an increment of that for B . With some effort, this translation can actually be performed statically on the text of a symmetric program \mathbb{P} , *before* building a Kripke model. The resulting counter-abstracted program $\widehat{\mathbb{P}}$ gives rise to a Kripke structure \widehat{M} whose reachable part is *isomorphic* to that of the traditional quotient \overline{M} and that can be model-checked without further symmetry considerations.

Counter abstraction can be viewed as a translation that turns a state space of potential size l^n (n local states over $\{1, \dots, l\}$) to one of potential size $(n+1)^l$ (l counters over $\{0, \dots, n\}$). The abstraction therefore reduces a problem of size *exponential* in n to one of size *polynomial* in n . Since, for any given Boolean program, l is a constant, we appear to have solved the state-space explosion problem.

Let us look at an example. Classical counter abstraction assumes that the behavior of a single process is given as a local state transition diagram, as the one in Figure 7.1 (left). This abstraction level is often used in descriptions of communication and cache-coherence protocols. The result of counter-abstracting this program is shown in the same figure on the right, in a guarded-command notation. We see that the reduction happens completely statically, i.e., on the program text. The new program is single-threaded; thus there are no notions of shared and thread-local variables. The Kripke structure corresponding to the program has shrunk from exponential size $\mathcal{O}(3^n)$ to low-degree polynomial size $\mathcal{O}(n^3)$. The reduced structure can be model-checked for hundreds if not thousands of processes.

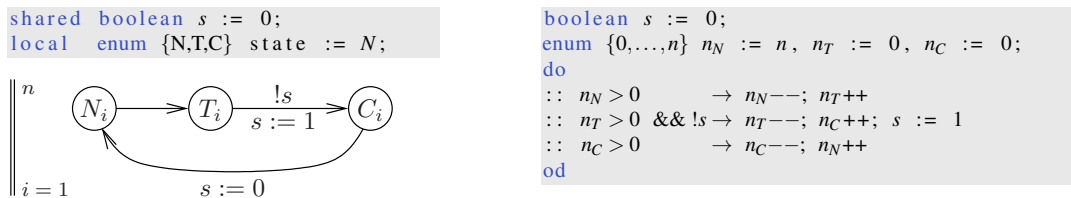


Figure 7.1: A model \mathbb{P} of a semaphore-based Mutex algorithm (left); its counter-abstracted version $\widehat{\mathbb{P}}$ (right)

This view does not, however, withstand a practical evaluation for concurrent software, where thread behavior is given in the form of a program that manipulates thread-local variables. The straightforward definition of a local state as a valuation of all thread-local variables is incompatible in practice with the idea of counter abstraction: the number of local states generated is simply too large. The Boolean program in Listing 5.3 declares only four thread-local Boolean variables and the PC with range $\{1, \dots, 12\}$, but gives rise to already $2^4 * 12 = 192$ local states. In applications of the magnitude we consider, concurrent Boolean programs routinely have several dozens of thread-local variables and many dozens of program lines (even after optimizations), resulting in many millions of local states. As a result of this *local state explosion* problem, the state space of the counter program is of size $\Omega(n^{2^{|V_l|}})$, **doubly-exponential** in the number of thread-local variables.

Let us apply these observations to the complexity analysis of counter abstraction. As seen, the abstract state space has size high-degree polynomial in n . This means that only for very large values of n , the classical counter abstraction approach will offer benefits over a model checking strategy that ignores symmetry and stores the local state for each thread.² The goal of this chapter is to demonstrate how we can reap the benefits of counter abstraction even for thread counts that are small compared to the number of local states. Our approach is two-fold:

1. Instead of statically translating each statement s of the input program into counter updates (which would require enumerating the many possible local states in which s is enabled), we make the algorithm **context-aware**, by triggering the translation *on the fly*. This way we have to execute s only in the narrow context of a given (and, thus, reachable) local state.
2. Instead of storing the counter values for all local states in a global state, store only the *non-zero* counters. This (obvious) idea exploits the observation that, if $l \gg n$, in every system state most counters are zero.

As a result, the worst-case size of the Kripke structure of the counter-abstracted program is reduced from proportional to n^l , to proportional to $n^{\min\{n, l\}}$, completely **eliminating** the sensitivity to the local state space explosion problem. In the rest of this chapter, we describe the symbolic state space exploration algorithm that implements this approach.

7.4 Symbolic Counter Abstraction

In this section, we present the contribution of this chapter, a symbolic algorithm for state space exploration of concurrent Boolean programs that achieves efficiency through counter abstraction. We first describe a data structure used to store system states compactly, and then present the algorithm.

7.4.1 A Compact Symbolic Representation

Resulting from predicate abstractions of C code, Boolean programs make heavy use of data-non-determinism, in the form of the non-deterministic Boolean value \star . Enumerating all possible values, an expression involving \star can stand for is infeasible in practice. A better approach is to interpret the value \star symbolically, as the set $\{0, 1\}$. This interpretation is not only compatible with encodings of Boolean programs using BDDs, but can also be combined well with counter abstraction.

²Even if model checkers of the future are able to explore systems with huge thread counts, it seems hard to conceive a scenario in which one would care to verify the behavior of a system with, say, 2 million threads.

Our approach to counter abstraction is to count *sets* of local states represented by propositional formulas, rather than individual local states. Consider 3 threads executing a Boolean program with a single thread-local Boolean variable x , and the global state $\tau = (\star, \star, \star)$: all threads satisfy $x = \star$. Defining the local state set $\mathbb{B} := \{0, 1\}$, we can represent τ compactly as the single abstract state characterized by $n_{\mathbb{B}} = 3$, indicating that there are 3 threads whose value for x belongs to \mathbb{B} (all other counters are zero).

To formalize our state representation, let L be the set of conceivable local states, i.e., $|L| = l$. An abstract global state takes the form of a set S of valuations of the shared variables, followed by a list of pairs of a local state set and a counter:

$$\langle S, (L_1, n_1), \dots, (L_k, n_k) \rangle. \quad (7.1)$$

In this notation, $L_i \subseteq L$ and $n_i \in \mathbb{N}$. We further maintain the invariant $n_i \geq 1$. The semantics of this representation is given by the set of concrete states that expression (7.1) represents, namely the states of the form (s, l_1, \dots, l_n) such that

- (a) $s \in S$,
 - (b) $n = \sum_{i=1}^k n_i$, and
 - (c) there exists a partition $\{I_1, \dots, I_k\}$ of $\{1, \dots, n\}$ such that for all $i \in \{1, \dots, k\}$, $|I_i| = n_i$, and for all $j \in I_i$, $l_j \in L_i$.
- (7.2)

That is, an abstract state of the form (7.1) represents precisely the concrete states in the Cartesian product of valuations of the shared variables in S , and valuations of the thread-local variables satisfying the constraint (7.2) (c). Intuitively, the number of threads active in the current global state is $\sum_{i=1}^k n_i$; this quantity is represented by n in the concretization (7.2). Further, each pair (L_i, n_i) represents the n_i threads such that the *most precise* information on their local states is that they belong to L_i . For instance, the abstract global state $\langle (\{x = 0\}, 3), (\{x = \star\}, 4) \rangle$ represents those concrete states where 3 threads satisfy $x = 0$, whereas we have no information on x for the remaining 4 threads. This example also shows that we do not require the sets L_i to be disjoint: forcing the symbolic local state $x = 0$ to be merged into the symbolic local state $x = \star$ would imply a loss of information, as the constraint $x = 0$ is more precise.

Traditional approaches that statically counter-abstract the entire input program often use a data structure that can be seen as a special case of (7.1), namely with $k = l = |L|$. Such implementations do not enforce the invariant $n_i \geq 1$ and thus suffer from the potential redundancy of n_i being 0 for most i .

On the other hand, representation (7.1) is marred by two caveats. The first is that constraints between shared and thread-local variables, such as introduced by an assignment of the form *shared* := *local*, are inexpressible. The reason is that neither S nor the sets L_i are defined by expressions over *both* shared and thread-local variables. Clearly, however, Boolean programs can introduce such constraints. Section 7.4.2 describes how our algorithm addresses this problem.

The second caveat is an artifact of not associating with each local state a fixed position in the counter vector. This opens up redundancy due to different orders of the tuples (L_i, n_i) in (7.1), which seems to require a call to a sorting routine. This is somewhat peculiar, since the very purpose of counter abstraction is to remove the requirement of sorting to canonize states.

As an example, the abstract global states $\langle (\{A\}, 1), (\{B\}, 2) \rangle$ and $\langle (\{B\}, 2), (\{A\}, 1) \rangle$ both legitimately represent one thread in local state A and two threads in local state B . To avoid redundancy, we require the local-state sets L_i to obey some total order. The choice of order is irrelevant; one can for instance simply base it on the numerical pointer value of the BDD representing L_i . This canonicity requirement is cheaply enforced in practice by sorting emerging (L_i, n_i) pairs into the existing representation (7.1); the cost is logarithmic in the size k of the global state.

7.4.2 Symbolic State Space Exploration

We present a symbolic algorithm for reachability analysis of symmetric Boolean programs with on-the-fly counter abstraction that employs the state representation described in Section 7.4.1. The input consists of a template program \mathbb{P} and the initial number n_0 of concurrent threads; the algorithm computes the counter-abstacted set of states reachable from a given set of initial states.

Algorithm 7.1 Symbolic Counter Abstraction

```

1:  $\mathcal{R} := \{\langle S_0, (L_0, n_0) \rangle\}$ ; insert  $\langle S_0, (L_0, n_0) \rangle$  into  $\mathcal{W}$  ▷  $n_0$  threads with local state in  $L_0$ 
2: while  $\mathcal{W} \neq \emptyset$  do
3:   remove  $\tau = \langle S, F \rangle$ , with  $F = \{(L_1, n_1), \dots, (L_k, n_k)\}$ , from  $\mathcal{W}$ 
4:   for  $i \in \{1, \dots, k\}$  do
5:      $T := \langle S, L_i \rangle$  ▷ extract  $i$ -th thread state from  $\tau$ 
6:     for  $v \in \text{valuations of } \textit{SpliceVariables}(T)$  do
7:        $T' = \langle S', L' \rangle := \text{Image}(T|_v)$  ▷ compute one image cofactor of  $T$ 
8:        $\tau' := \langle S', \text{UPDATECOUNTERS}(F, i, L') \rangle$  ▷ build new system state  $\tau'$  from  $T'$ 
9:       if  $\tau' \notin \mathcal{R}$  then
10:         $\mathcal{R} := \mathcal{R} \cup \tau'$  ▷ if new, store  $\tau'$  as reachable
11:        insert  $\tau'$  into  $\mathcal{W}$ 

12: procedure UPDATECOUNTERS( $F, i, L'$ )
13:   let  $(L_i, n_i)$  be the  $i$ -th pair in  $F$ 
14:    $F' := F \setminus \{(L_i, n_i)\} \cup (n_i > 1 ? \{(L_i, n_i - 1)\} : \emptyset)$  ▷ update or eliminate pair  $(L_i, n_i)$ 
15:   if  $\exists j : (L', n_j) \in F$  then ▷ update or add pair for  $L'$ 
16:      $F' := F' \setminus \{(L', n_j)\} \cup \{(L', n_j + 1)\}$ 
17:   else
18:      $F' := F' \cup \{(L', 1)\}$ 
19:   return  $F'$ 

```

Algorithm 7.1 expands unexplored system states from a work list \mathcal{W} , initialized to contain the symbolic state that constrains all threads to location 0. The loop in line 4 iterates over all pairs (L_i, n_i) contained in the popped state τ . To expand an individual pair, the algorithm first projects it to the i -th symbolic thread state.

The next, and crucial, step is to compute the successor thread states induced by the Boolean program (lines 6–7). Recall from the previous section that our Cartesian state representation does not permit constraints between shared and thread-local variables, which can, however, be introduced by the program.

Consider a shared variable s and a thread-local variable l , and the statement $l := s$. A naïve way of computing the successor state of the symbolic system state $\langle \{s = \star\}, (\{l = 0\}, 2) \rangle$ is the following. Since the symbolic transition function ranges over shared and local variables and they are stored as separate sets, a symbolic thread state $\langle \{s = \star \wedge l = 0\} \rangle$ must be computed first. In the second step, the image function is applied, resulting in the symbolic state $\langle l = s = \star \rangle$. Its transformation back into a Cartesian thread state representation is $\langle \{s = \star\}, \{l = \star\} \rangle$. That thread state is, however, an *over-approximation* of the exact set of successor thread states $\{\langle \{s = 0\}, \{l = 0\} \rangle, \langle \{s = 1\}, \{l = 1\} \rangle\}$. In order to make this constraint precisely expressible, we treat certain assignments and related statements specially.

Definition 7.3 A *splice state* is a symbolic thread state given as a predicate f over the variables in $V_s \cup V_l \cup \{pc\}$ such that

$$(\exists V_s . f) \wedge (\exists V_l \exists pc . f) \not\equiv f .$$

A *splice statement* is a statement s such that there exists a thread state u whose PC points to s and that, when executed on u , results in a splice state. A *splice variable* is a shared variable dependent on $\exists V_s . f$.

A splice statement marks a point where a thread communicates data via the shared variables, in a way that constrains its local state with the values of some splice variables. Fortunately, statements with the *potential* to induce such communication can be identified syntactically:

- assignments whose left-hand side is a shared variable and the right-hand side expression refers to thread-local variables, or vice versa,
- assignments with a `constrain` clause whose expression refers to both shared and thread-local variables, and
- `assume` statements whose expression refers to both shared and thread-local variables.

Before executing a splice statement, the current thread state is *split* using Shannon decomposition. Executing the statement on the separate co-factors yields a symbolic successor that can be represented precisely in the form (7.1). That is, if variable v is the splice variable of the statement in T , denoted by $\text{SpliceVariables}(T) = \{v\}$, we decompose $\text{Image}(T)$ as follows:

$$\text{Image}(T) = \text{Image}(T|_{v=0}) \vee \text{Image}(T|_{v=1}) .$$

The price of this expansion is an explosion worst-case exponential in the number of splice variables. However, as we observed in our experiments (see Section 7.5),

1. the percentage of splice statements is relatively small,
2. even within a splice statement, the number of splice variables involved is usually very small (1 or 2),
3. a significant fraction of Shannon co-factors encountered during the exploration is actually *unsatisfiable* and does not contribute new states.

As a result, the potential combinatorial explosion never materialized in our experiments.

We note in passing that the goal of splitting a state is similar to the goal of the focus operator presented in [12]. Its authors use the focus operator to increase the precision of Cartesian abstraction applied to image operators. Our motivation is different: we want to rewrite constraints that cannot be expressed in our representation, in order to avoid any imprecision.

After the image has been computed for each co-factor, the algorithm constructs the respective system state for it (line 8). The `UPDATECOUNTERS` function uses the local state part L' of the newly computed thread state to determine the new set of (state, counter) pairs F' . If no more threads reside in the departed state L_i , the i -th pair is eliminated (line 14). If the new local state L' was already present in the system state, its counter n_j is incremented, otherwise the state is inserted with counter value 1 (lines 15–18).

Finally, the algorithm adds the states encountered for the first time to the set of reachable states, and to the work list of states to expand (lines 9–11).

Theorem 7.4 *Let \mathcal{R} be as computed by Algorithm 7.1 on termination, and let γ be the concretization function for abstract states defined in Equation (7.2). The set $\gamma(\mathcal{R}) = \{\gamma(r) \mid r \in \mathcal{R}\}$ is the set of reachable states of the concurrent system induced by the Boolean program \mathbb{P} .*

Proof[sketch]: The proof of termination of Algorithm 7.1 follows since the explored state space is finite: the imposed bound N on the number of threads that may be created, and of which the `start_thread` command is aware, ensures that there is only a finite number of global states, whether they are represented in the concrete or abstract form. Algorithm 7.1 performs a standard search over this finite state space and thus terminates. The correctness argument of the algorithm follows from (i) the equivalent theorems for classical state space exploration under symmetry using canonical state representatives, and (ii) the isomorphism of the structures over such representatives and the counter representation. \square

7.4.3 Error Detection and Counterexample Generation

Errors are program locations containing violated assertions, say of the form `assert(Y)`. The predicate Y expresses a condition over the current thread state. The violation of this condition is checked in Algorithm 7.1 in line 7, by testing the new thread state $T' = \langle S', L' \rangle$ against the condition $\neg Y$: if the BDD for $S' \wedge L' \wedge \neg Y$ is non-empty, T' violates the assertion Y .

We now need to obtain a concrete path from an initial global state to a global state that contains thread state T' . Such a path can be presented in the form of a sequence of global states over Boolean program variables, including the PC. We have omitted from our data structure the standard back edges, and from the algorithm a description of the standard mechanisms to trace back a reached state to the initial state. Non-standard is the shape of the resulting path, namely a sequence U of states of the form $\langle S, (L_1, n_1), \dots, (L_k, n_k) \rangle$, ending in τ' (line line 8). This abstract trace can be mapped to a concrete trace over the concurrent Boolean program as in Algorithm 7.2.

Algorithm 7.2 Computing a concrete error trace

Require: abstract global state τ' , set-of-states list U

- 1: $e :=$ some concrete global state represented by τ' , according to Equation (7.2)
 - 2: $p := (e)$ $\triangleright p$ is the path to be constructed
 - 3: **for** all sets Z in U in reverse order **do**
 - 4: $I := \text{Prelm}_{\mathbb{P}}(e)$ \triangleright concrete concurrent pre-image operator
 - 5: $e :=$ some element of $\alpha(I) \cap Z$
 - 6: push e to the front of p
 - 7: **Return** p
-

In the algorithm, operator `Prelm` computes the concrete pre-image of state e under the *concurrent* Boolean program, i.e., for each thread in turn. The resulting set I is *canonized* in line 5 using the abstraction function α , which converts each state in I into the form (7.1). The pre-image applied in line 4 may lead out of the abstract error trace U given as input. The algorithm therefore intersects the abstraction of I with the current set Z of sequence U and selects a new element e from the intersection (still line 5), which is guaranteed to be non-empty. The new element is a suitable predecessor along the error path that is being constructed.

7.5 Experimental Evaluation

We have implemented the algorithm presented in this section in BOOM. While our main goal is the symbolic analysis of Boolean programs (Section 7.5.1), we have also built an *explicit-state* version of our tool. The reason is that symmetry reduction has so far proven to be more successful in explicit-state than symbolic model checking, which begs a comparison against explicit-state symmetry reduction. As a competitor, we chose the well-known MUR ϕ model checker [109] (Section 7.5.2).

Before we discuss our experiments, we would like to refer the reader to appendix B, where we describe in detail the experimental setup and how the benchmarks were generated.

7.5.1 Symbolic Experiments

Since other symbolic model checkers do not scale to interesting thread counts (including the few tools with built-in support for symmetry, see Section 7.7), we compare the symbolic algorithm to a “plain” symbolic reference implementation in BOOM that ignores the symmetry. On sequential programs, the performance of the reference implementation is similar to that of the model checker that ships with SLAM.

BOOM uses the CUDD BDD library by Fabio Somenzi ([126, version 2.4.1]) as the decision diagram package. Our implementation stores the sets S and L_i of shared and thread-local variable valuations as separate BDDs; the conjunction of S and L_i forms the thread-visible state T_i . As in most symbolic model checkers for software, the program counters are stored in explicit form: this permits partitioning the transition relation and ensures a minimum number of splice tests.

Figure 7.2 summarizes the running times of the symbolic counter abstraction implementation in BOOM and the plain symbolic exploration algorithm. The uniform distribution in the upper triangle signifies the improvement in scalability due to counter abstraction. The runs where traditional model checking is faster contain a small number of threads; in fact, our algorithm can verify many instances for 7 or more threads. Overall, BOOM is faster on 83 % of all tests, and on 96 % of those running three or more threads. Among those, the speed-up is five orders of magnitude and more.

Splice statements amount to less than 12 % of all statements. Where they occur, they do not cause a blow-up, in any of the benchmarks. In fact, the average number of splice variables they involve are small (in our benchmarks, mean 2.1, median 1), and the average number of cases with two valid cofactors encountered for each variable is less than 10 %. Each such variable produces two satisfiable co-factors in only 10 % of the cases.

State merging. Since states are not represented by a single BDD, but rather a set of pairs, we run the risk of obtaining many symbolic states that could be represented in a more consolidated way. Consider the case in which $\langle(\{A\}, 1), (\{B\}, 2)\rangle$ is encountered and in a subsequent step the state $\langle(\{B\}, 2), (\{C\}, 1)\rangle$ is found. Both of the states can be represented more compactly by the state $\langle(\{A, C\}, 1), (\{B\}, 2)\rangle$. Thus our implementation of the last step of Algorithm 7.1 (lines 9–10) uses state merging, an important optimization to compress sets of symbolic states. Two distinct symbolic states can be merged if they are identical except for (i) the valuation of the shared variables, **or** (ii) the local state of exactly *one* thread. This check is linear in the number of local-state pairs because they are totally ordered (see Section 7.4.1). The application of these merging rules provides an average speed-up of 83 % over exploration without merging.

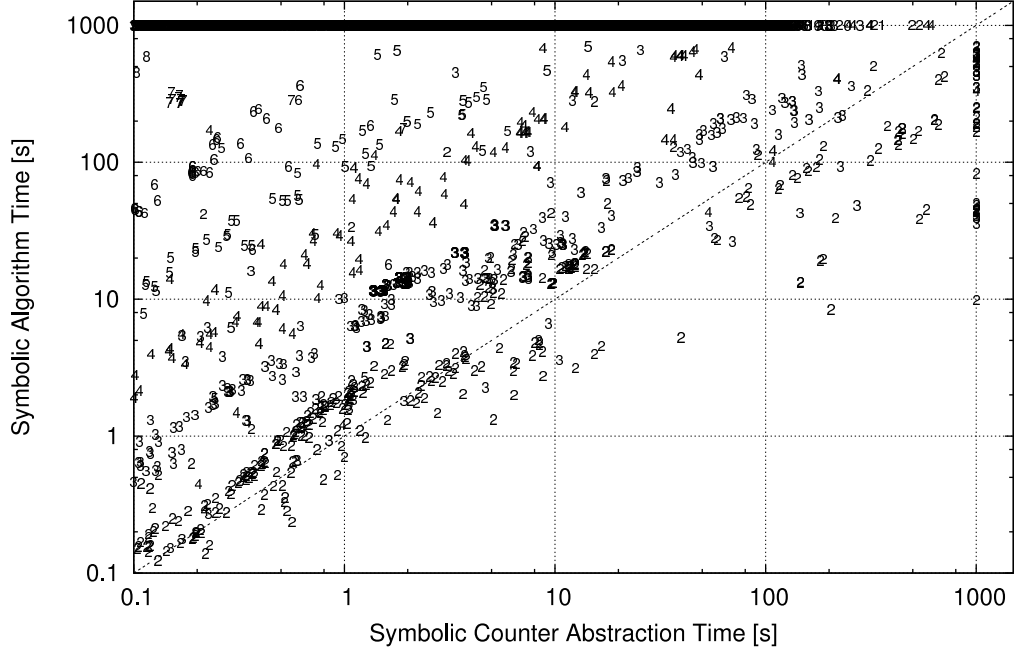


Figure 7.2: Running time of symbolic BOOM vs plain exploration, for various thread counts

State subsumption. A state σ is said to *subsume* a state σ' , written $\sigma' \subseteq \sigma$, if σ contains every state that is present in σ' . A consequence of our state representation is that checking if σ subsumes σ' , is not trivial (Algorithm 7.1, lines 9–11). Consider for example the state $\langle (\{A\}, 1), (\{B\}, 2) \rangle$, which is subsumed by $\langle (\{A, B\}, 3) \rangle$. An algorithm that detects this fact needs to be able to map a single local state pair of a state σ to a set of local state pairs of another state σ' . Since this partial mapping can become very expensive, we confine our notion of subsumption to the following approximation.

Definition 7.5 A state $\sigma = \langle S, (L_1, n_1), \dots, (L_k, n_k) \rangle$ *subsumes* a state $\sigma' = \langle S', (L'_1, n'_1), \dots, (L'_k, n'_k) \rangle$ iff

1. $S' \subseteq S$, and
2. there is a bijection $f : \{1, \dots, k\} \rightarrow \{1, \dots, k\} : \forall i, 1 \leq i \leq k. L_i \subseteq L_{f(i)}$ and $n_i = n_{f(i)}$.

Note that there is no total order on the BDDs that respects the set order. Thus, in the worst case we need to enumerate any permutations of local state pairs. There is, however, a simple argument why that is still efficient in practice. Since the program location and the local state counter are stored explicitly, it suffices to look at permutations of the local state pairs that have the same program locations and counter values but different valuations of local variables. This case turned out to occur rather rarely in our experiments. We ascribe this fact 1) to the non-determinism inherent in Boolean programs, that allows many variables to have either truth value and 2) to the symmetric nature of the benchmarks. We also investigated why the threads, which reside at the same program location, often have exactly the same variables. The analysis revealed that, at small exploration depths, the threads actually have different variables but in about 75% of the cases those threads that feature only a subset of the variable valuations of others, were able to take the same transitions at a later stage of the state space exploration.

7.5.2 Explicit-State Experiments

We compare our explicit-state implementation to $MUR\phi$ [109], a mature and popular model checker with long-standing support for symmetry reduction. Since $MUR\phi$ does not allow data non-determinism, we replace every occurrence of \star in the input programs randomly by 0 or 1. The resulting programs are converted into $MUR\phi$'s input language using one $MUR\phi$ rule per statement, guarded by the program counter value. In the explicit-state experiments, we compare the performance of explicit-state BOOM on each determinized Boolean program against $MUR\phi$ on the guarded-rule version of the same program.

Figure 7.3 is a scatter plot of the running times of BOOM and of $MUR\phi$ with symmetry reduction. BOOM is faster than $MUR\phi$ on 94 % of the tests; on 23 %, the improvement is better than one order of magnitude. It completes successfully on a significant number of problems where $MUR\phi$ times out (19%). In seven cases (1.2%), our tool runs out of memory. Note that removing the data non-determinism simplifies the programs, which is why the explicit-state explorations can often handle larger thread counts than the symbolic ones, reported in the previous section.

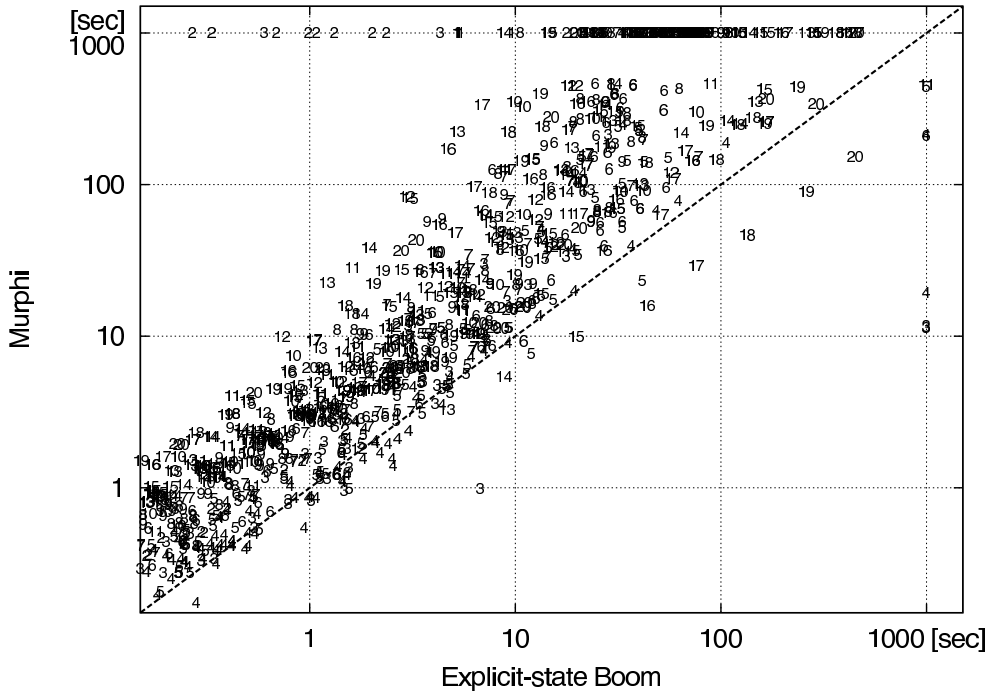


Figure 7.3: Running time of explicit-state BOOM vs $MUR\phi$, for various thread counts

7.5.3 Comparison with Partial-Order Methods

As shown in Section 6.2.3, the effect of our POR on the plain symbolic algorithm (**without** using counter abstraction) is an average speedup of 4.2 and combined runtime improvements of 85% for all benchmarks. We now compare our POR method to counter abstraction in various ways. We use the same sets of benchmarks and timing constraints as before.

Figure 7.4 shows the additional improvement of counter abstraction on an implementation that employs POR. The combination of both techniques is on average 156 times faster than POR alone. This indicates that counter abstraction is by no means “subsumed” by POR.

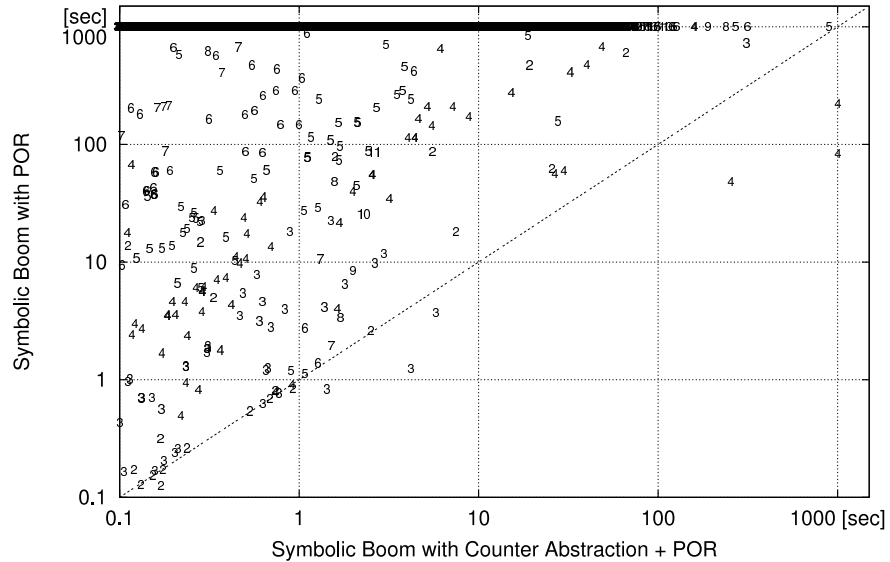


Figure 7.4: Running time of symbolic BOOM with partial-order reduction vs. symbolic BOOM with counter abstraction and partial-order reduction, for various thread counts

The scatter plot in Figure 7.5 depicts the speedup of employing POR and counter abstraction versus counter abstraction alone. In total, an average speedup of 140 could be measured. This indicates inversely that POR is not subsumed by counter abstraction either. Figures 7.4 and 7.5 support the general observation that symmetry based and partial-order based methods are combinable for yet better compression.

Finally, Figure 7.6 compares the methods directly against each other: counter abstraction alone outperforms partial-order reduction alone; the former gave an average speedup of 4.2.

7.5.4 Comparison with Other Methods

Table 7.1 compares the symbolic version of BOOM with counter abstraction against the *lazy* version of GETAFIX (see Section 5.4). We opted for the lazy variant, because it has better performance than the eager one. GETAFIX targets *recursive* Boolean programs with a bounded number of context-switches. To compare with BOOM, we chose the non-recursive driver example provided on the GETAFIX website. This concurrent Boolean program (printed in appendix A.2) models a Windows NT Bluetooth driver. It consists of two types of threads: adders and stoppers. An adder thread calls an I/O procedure in the driver, whereas a stopper thread calls a procedure to halt the driver. An error state is reached, if both kinds of threads enter a race and the driver stops, although a new request is added.

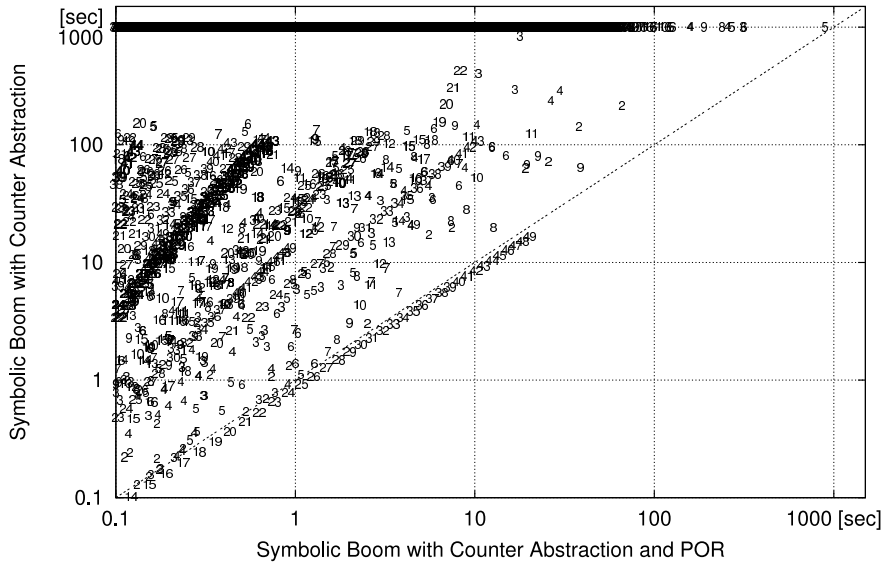


Figure 7.5: Running time of symbolic BOOM with counter abstraction vs. symbolic BOOM with counter abstraction and partial-order reduction, for various thread counts

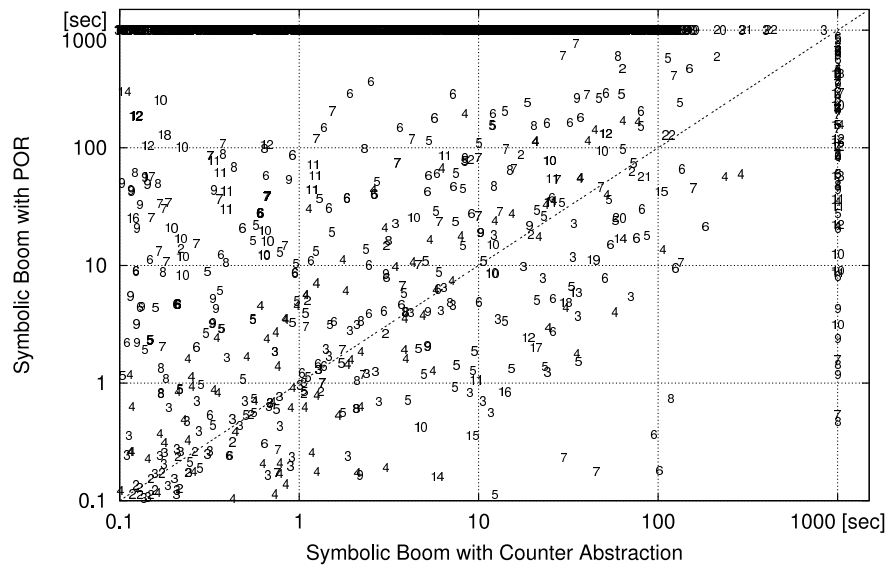


Figure 7.6: Running time of symbolic BOOM with partial-order reduction vs. symbolic BOOM with counter abstraction, for various thread counts

The time for GETAFIX to convert the example into a sequential program is negligible and has been omitted. Since GETAFIX creates a separate thread for each function in the Boolean program, we need to replicate each thread by hand, e.g., see Listing A.2 for the case of 2 adders and 2 stoppers³. Since BOOM uses either dynamic thread creation or replicates the main method internally, we had to add the code snippet illustrated in Listing A.4 that 1) initializes the global variables and 2) eventually non-deterministically jumps in in the adder or stopper procedure (see the last instruction). Thus, in the comparison we check any combination of adder and stopper threads for a fixed number of threads. Table 7.1 illustrates the time to explore the sequentialized program using MOPED-1, for different context-switch bounds. Note that BOOM, in contrast, explores any interleaving and any combination of adder / stopper threads.

n	BOOM [sec]	GETAFIX/cont. bd. [sec]					
		1	2	3	4	5	6
2	< 0.1	0.1	0.4	2.0	8.7	41	139
3	0.1	0.1	1.0	0.6	4.8	30	187
4	1.2	0.1	1.9	1.2	12.2	146	1318
5	12.1	0.14	2.8	2.3	30.6	426	—
6	88.8	0.2	3.9	3.1	51.7	901	—

Table 7.1: GETAFIX & MOPED-1 vs. BOOM benchmarks on Intel 3GHz, with timeout 60 mins, memory-out 4 GB.

7.5.5 Alternative Global State Representations

We have also considered global state representations alternative to equation (7.1). In one implementation, we use a *monolithic* BDD to represent the shared variables and all thread states, along with their counters. In another, we keep the counters explicit, but use a monolithic BDD for all other variables. Both implementations allow us to retain the inter-thread constraints introduced by splice statements, and thus render the decomposition step unnecessary. The first implementation has the additional advantage of not requiring state merging techniques (Section 7.5.1): given a single BDD, merging happens automatically when adding new frontier states to the BDD.

A technical challenge with these alternative representations is that they require more complex manipulations for computing successor states, especially in order to update the counters. These representations are also less compact than non-monolithic ones, because the system state is not comprised of the Cartesian product of the local states. The more serious downside, however, is efficiency, as is often the case with monolithic symbolic data structures: the resulting BDD for the set of reachable states is complex, foiling the scalability advantage inherent in counter abstraction. In fact, the separation of a global state into thread states and associated counters suggests a natural way of partitioning the BDD for the reachable states set, which should not be given up lightly. On our benchmarks, the algorithm proposed in Section 7.4 is at least 30% faster than all alternatives.

A variant of our proposed algorithm keeps a copy of the global variables within the BDD of each local-state pair and thus allows the image function to be applied directly, i.e., without prior building the conjunction of the BDDs representing the global and local variables. The price paid for omitting this

³We used the combinations 1 adder / 1 stopper, 1 adder / 2 stoppers, 2 adders / 2 stoppers, 2 adders / 3 stoppers, and 3 adders / 3 stoppers.

operation is that after any change to one pair of the global variables, any other pair has to be updated, too. For this reason, this alternative algorithm is at least 20 % slower than the proposed solution.

7.6 Summary

We have presented an algorithm for BDD-based symbolic state space exploration of concurrent Boolean programs, a significant branch of the pressing problem of concurrent software verification. The algorithm draws its efficiency from counter abstraction as a reduction technique, without resorting to approximation at any time. It is specifically designed to cope with large numbers of local states and thus addresses a classical bottleneck in implementations of counter abstraction. We have shown how to avoid the *local state space explosion* problem using a combination of two techniques: 1) achieving context-awareness by interleaving the translation with the state space exploration, and 2) ensuring that only non-zero counters and their corresponding local states are kept in memory.

We have presented experimental results both for an explicit-state and, more importantly, a symbolic implementation. While standard symmetry reduction is employed in tools like MUR ϕ and RULEBASE, we are not aware of a prior implementation of counter abstraction that is efficient on programs other than abstract protocols with very few control states. We believe our model checker to be the first with a true potential for scalability in concurrent software verification, thanks to its polynomial dependence on the thread count n , while incurring little verification time overhead.

We have also investigated in detail the relationship between our implementation of counter abstraction and partial-order methods. Our experiments seem to confirm the folk wisdom that symmetry and partial-order reduction are, although not independent, certainly complementary and can be combined for yet more effective compression (see, for instance, the work by Emerson, Jha, and Peled [64]).

Traditional symmetry reduction, no matter of what flavor, is limited in scope in that it considers only systems of a number of concurrent components that is a design-time constant. We have, in this chapter, extended the technique to software with bounded dynamic thread creation. An obvious extension is to consider unbounded dynamic thread creation, or the parametrized version of the concurrent reachability problem. Both topics are addressed in the next chapter.

7.7 Bibliographic Notes

The principal idea of using process counters already appeared in early work by Lubachevsky [104], *generic representatives* were suggested by Emerson and Trefler [67] as a means of addressing the complexity of symmetry-reducing symbolically represented systems. However, the term *counter abstraction* was actually coined by Pnueli, Xu, and Zuck, in the context of parametrized verification of liveness properties [114]. The counters are cut off at some value c , indicating that *at least* c components currently reside in the corresponding local state. We emphasize that, in this chapter, we use the term *counter abstraction* in the sense of **exact** counters. The method we propose can be seen as an “exact abstraction”, a notion that is common in symmetry reduction and other bisimulation-preserving reduction methods.

Local state-space explosion was identified by Emerson and Wahl as the major obstacle to using generic representatives with non-trivial symmetric programs [61]. The paper ameliorates this problem using a static live-variable analysis, and using an approximate but inexpensive local state reachability test. Being heuristic in nature, this work cannot guarantee a reduced complexity of the abstract program.

We are aware of a few significant works that resulted in tools using counter abstraction in symbolic model checking: by Wei, Gurfinkel, and Chechik [136], in the context of *virtual symmetry* [63], and by Donaldson and Miller [58], for probabilistic models. While valuable in their respective domains, both approaches suffer from a limitation that makes them unsuitable for general software: they are based on a system model (such as the *GSST* of [136]) that describes the process behavior by local state changes and thus require an up-front translation from whatever input language is used. The examples in [136, 58] include communication and mutual-exclusion protocols with at most a few dozen local states. The BEACON model checker [8] has been applied to a multi-threaded memory management system with 256 local states. In our benchmarks, threads have millions of local states (see Section 7.5).

Henzinger, Jhala, and Majumdar apply $0-1-\infty$ counter abstraction to predicate-abstracted concurrent C programs for race detection [85]. The counters monitor the states of *context threads*. To avoid local state space explosion, each context thread is simplified to an *abstract control flow automaton* (ACFA). According to the authors, the ACFA has at most a few dozen vertices and can thus be explicitly constructed. In contrast, our goal is a general solution for arbitrary predicate abstractions, where we cannot rely on a small number of predicates and, thus, local states. Consequently, our work does not require first building a local state transition diagram.

Compared to canonization-based symmetry reduction approaches such as in MUR ϕ [109] and ZING [3] (explicit-state) or SVISS [134] and RULEBASE [16] (symbolic), the model checking overhead that counter abstraction incurs reduces to translating the program statements into local state counter updates. Sorting local state sequences, or other representative mapping techniques, are implicit in the translation.

Finally, the general problem of symbolically verifying multi-threaded programs has been tackled in many recent publications [46, 129, and others]. None of these address the symmetry that concurrent Boolean programs exhibit, although some investigate partial-order based methods [74]. This chapter includes an extensive experimental comparison of our proposed algorithm to partial-order reduction techniques; see Section 7.5.3.

8

Concurrent Boolean Programs with Unbounded Replication

IN this chapter, we consider non-recursive multi-threaded Boolean programs, the principal ingredient in predicate abstraction. In the previous chapter, we introduced a scalable method for thread-state reachability analysis of concurrent Boolean programs with a bounded number of threads. In this chapter, we extend this method to programs with *unbounded thread creation*. We present a novel implementation of the *Karp-Miller* procedure for vector addition systems, and evaluate its performance using a substantial set of Boolean program benchmarks.

8.1 Introduction

In chapter, 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; it applies, for example, to a server that spawns additional worker threads in response to a high work load. The verification problem for such software is undecidable in general [4]. We focus on the special case of *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, 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.

Thread-state reachability (see Section 5.3) for replicated finite-state programs can be shown to be decidable — even in the absence of a bound on the number of threads that can be spawned — by a reduction to the *coverability problem* for *vector addition systems* (VASS). This problem in turn has been known to be decidable since the work by Karp and Miller [95]. 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. replicated finite-state programs are not vector addition systems: while the former can be converted to the latter using thread counters, doing so naively results in an unmanageably large number of counters and, thus, addition systems of prohibitively large dimension;
2. the worst-case complexity of the known VASS decision procedures is intractably high, even with later improvements due to Rackoff [120].

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 phenomenon was identified as the *local state explosion problem* in [61]. In the previous chapter, we presented a solution for the case of a bounded number of components.

In this chapter, 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.

8.2 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 $(s, l) \rightarrow (s', l')$ and $(s, l) \rightarrow (s', l', l'')$. The 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 $(s, l) \rightarrow (s', l') \in \Delta$, we have a transition $(s, l_1, \dots, l_n) \rightarrow (s', l'_1, \dots, l'_n)$ 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 $(s, l) \rightarrow (s', l', l'') \in \Delta$, we have a transition $(s, l_1, \dots, l_n) \rightarrow (s', l'_1, \dots, l'_n, l'')$ 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. 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.

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} . 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. A thread state completely describes the information accessible to a single thread, namely, the shared variables and the thread's copy of the local variables. A thread has neither read nor write access to any other local variables.

We remark that our model of replicated finite-state systems covers 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 can simulate the other.

8.3 The Karp-Miller Coverability Tree

We review the basics of vector addition systems with states (VASS) and the Karp-Miller procedure since they are both fundamental to this chapter.

A *Vector Addition System with States* (VASS) is a finite-state machine whose (directed) edges are labeled with m -tuples of integers. A *configuration* of a VASS is a pair (q, x) where q is a state and x is an m -tuple of non-negative integers. There is a transition $(q, x) \rightarrow (q', x')$ if $q \xrightarrow{v} q'$ is an edge in the VASS, and $x' = x + v$, where $+$ denotes point-wise addition. Figure 8.1 illustrates an example of a VASS. 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 $x' \geq x$.

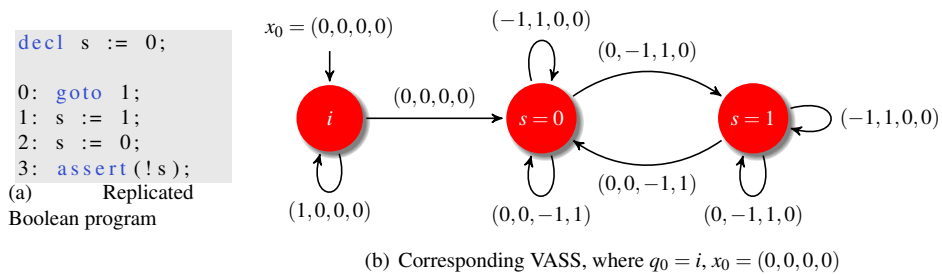


Figure 8.1: Example of a replicated Boolean program and its corresponding VASS

Theorem 8.1 ([95]) *The coverability problem for VASS is decidable.*

The algorithm by Karp and Miller [95] builds a rooted tree T that compactly represents the set of covered configurations of a vector addition system. Each node of T is labeled with a pair consisting of a state and a vector over $\mathbb{N} \cup \{\omega\}$ (ω means ‘any natural number’). The root is labeled with the initial configuration of the VASS. Each node η , say with label (q, x) , is incrementally expanded: there is a successor η' for each edge $q \xrightarrow{v} q'$ of the finite-state machine such that $x + v \geq 0$. The state component of η' 's label is q' . The i -th coordinate of the vector component of η' 's label is $x_i + v_i$, except in the following situation: there exists a node ξ along the path from the root to η with label (q', y) for some y such that $y \leq x + v$ and $y_i < x_i + v_i$. In that case, the i -th coordinate of the vector component of η' 's label is ω .

These rules are recursively applied to any node with a *new* label. This procedure eventually reaches a fix-point, as the set of labels that can be generated this way is finite [95]. 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 .

Figure 8.2 depicts the *Coverability Tree* of the example in Figure 8.1.

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 system¹: a thread transition $(s, l) \rightarrow (s', l')$ is represented by a VASS edge $s \xrightarrow{v} s'$ such that

¹Similar reductions can be found in [78, 55, 8, 29].

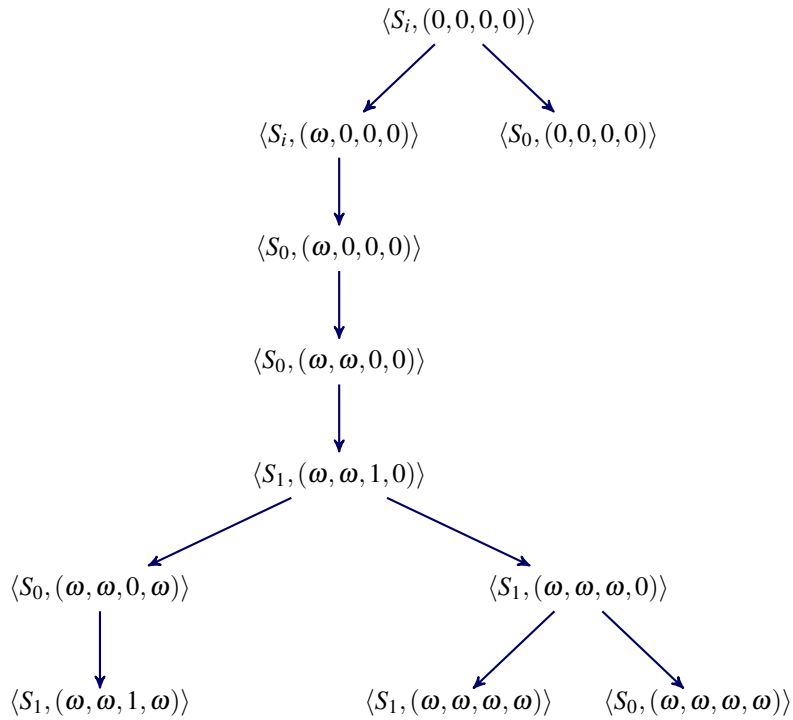


Figure 8.2: Corresponding Karp-Miller tree for Figure 8.1

the l -th component of v is -1 , the l' -th component is 1 , and all others are 0 . Dynamic thread creation $(s, l) \rightarrow (s', l', l'')$ 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 . 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 8.1. We obtain:

Corollary 8.2 *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 later improved this to operate in exponential space, which is close to optimal [120]. While daunting, these complexity measures are not terminal for our approach: vector addition systems encoding concurrent Boolean programs are of very simple structure, with most integer vectors having all-but-two zero entries.

We conclude this section by observing that the coverability problem for a VASS can inversely be reduced to a thread-state reachability problem. For this reduction, given vectors of m components, we introduce m local states. The number of threads in state i gives the value of the i -th component of the VASS configuration vector. As a result, the thread-state reachability problem for replicated finite-state systems and the coverability problem for vector addition systems are polynomial-time equivalent.

8.4 Thread-State Reachability in Boolean Programs

The proof of decidability of the thread-state reachability problem relies on the reduction of this problem to VASS coverability, and the solution of the latter problem using the Karp-Miller construction. While this proof is constructive, both parts of it translate into very inefficient algorithms if done naively. In this section, we describe how to curb the complexity, and our experiences with the performance of the resulting implementation in practice.

8.4.1 The Karp-Miller Graph Construction for Boolean Programs

The reduction of thread-state reachability to VASS coverability can be viewed as a form of *counter abstraction* [114]: instead of storing the system state of the concurrent Boolean program as a tuple of a shared state and a vector of thread-local states, we introduce a counter per local state. The counter program can be directly encoded as a VASS.

In this chapter, we extend these ideas to, and implement them for, the case of arbitrarily many threads. The counters are unbounded; in particular, dynamic thread creation is allowed in any program state and leads to an increase in the total number of threads. We dynamically determine the effect of each Boolean program statement on the counters. The aforementioned local state explosion problem directly translates into what is known as the *dimensionality problem* of VASS and Petri nets [75]. Our solution is to use vectors of variable dimension — the counters are *volatile*: they disappear if they ever become zero. Counters for emerging local states are created as needed. See Section 8.6 for a comparison with other approaches to the dimensionality problem.

The additional cost of the Karp-Miller procedure, compared to one that explores states for a bounded number of threads, comes from the need to check labels of new nodes for coverage against labels of previously discovered nodes. This information is used both in the process of introducing the

ω symbol into a node label, as well as to discard nodes with labels that are covered by previously discovered nodes on *different* paths. 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 covering relation as partial order. New labels are compared against these maximal nodes only. In particular, this subset of the discovered nodes forms an anti-chain in the covering partial order.

More precisely, when a new node η is found, our algorithm first checks whether some previously discovered node covers η ; if so, η is discarded. To detect this, it is sufficient to compare η against the maximal elements of each chain in the covering partial order. 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 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.

Algorithm 8.1 illustrates a symbolic Karp-Miller procedure that is based on the counter abstraction algorithm presented in Chapter 7. It differs from Algorithm 7.1 in two locations: the call to UPDATECOUNTERS needs to supply the values of the shared variables and a call to a function UPDATECOVER has been added.

Algorithm 8.1 Symbolic Coverability Tree

```

1:  $\mathcal{R} := \{\langle S_0, (L_0, n_0) \rangle\}$ ; insert  $\langle S_0, (L_0, n_0) \rangle$  into  $\mathcal{W}$   $\triangleright n_0$  threads with local state in  $L_0$ 
2: while  $\mathcal{W} \neq \emptyset$  do
3:   remove  $\tau = \langle S, F \rangle$ , with  $F = \{(L_1, n_1), \dots, (L_k, n_k)\}$ , from  $\mathcal{W}$ 
4:   for  $i \in \{1, \dots, k\}$  do
5:      $T := \langle S, L_i \rangle$   $\triangleright$  extract  $i$ -th thread state from  $\tau$ 
6:     for  $v$   $\in$  valuations of SpliceVariables( $T$ ) do
7:        $T' = \langle S', L' \rangle := \text{Image}(T|_v)$   $\triangleright$  compute one image cofactor of  $T$ 
8:        $\tau' := \text{UPDATECOUNTERS}(S, S', F, i, L')$   $\triangleright$  build new system state  $\tau'$  from  $T'$ 
9:        $\tau'' := \text{UPDATECOVER}(\tau')$   $\triangleright$  build covered system state
10:      if  $\tau'' \notin \mathcal{R}$  then
11:         $\mathcal{R} := \mathcal{R} \cup \tau'$   $\triangleright$  if new, store  $\tau'$  as reachable
12:      insert  $\tau'$  into  $\mathcal{W}$ 

```

We impose minor modifications to the function UPDATECOUNTERS (see Algorithm 8.2) in order to ensure that a pair with $n = \omega$ is processed correctly. We describe this explicitly because this results in a more efficient algorithm in practice. A pair whose counter value is ω is never updated (line 4), since we have $\omega - 1 = \omega$. Similarly, if $n_j = \omega$ (the value of the counter of the new thread state L'), then it is not incremented (line 6), because we have $\omega + 1 = \omega$. A simple but effective optimization that can often be applied in practice, is applied in line 8 and line 12. If the shared variables are not affected by a the last transition taken and an unbounded number of threads still reside in that state (i.e., $n_i = \omega$), we can conclude that any number of threads are able to execute the same transition and thus $n_j = \omega$. This optimization seems to be negligible at first glance but it is motivated by the following observation. A considerable fraction of the explored states have $n = \omega$ for any thread state counter and thus, if we check upfront, we can omit walking the coverability tree back to the root for any non- ω counter values.

The procedure UPDATECOVER (see Algorithm 8.3) introduces the value ω for thread states that can be reached by an unbounded number of threads. The idea is to look for cycles in the execution path that spawn new threads infinitely often. That is, we walk backwards towards the initial state (line 4) and check if any preceding state is subsumed by the actual state τ . Note that in addition the values

Algorithm 8.2 Symbolic Coverability Tree

```

1: procedure UPDATECOUNTERS( $S, S', F, i, L'$ )
2:   let  $(L_i, n_i)$  be the  $i$ -th pair in  $F$ 
3:   if  $n_i \neq \omega$  then
4:      $F' := F \setminus \{(L_i, n_i)\} \cup (n_i > 1 ? \{(L_i, n_i - 1)\} : \emptyset)$            ▷ update or eliminate pair
5:   if  $\exists j : (L', n_j) \in F$  then                                           ▷ update or add pair for  $L'$ 
6:     if  $n_j \neq \omega$  then
7:       if  $n_i = \omega$  and  $S' = S$  then
8:          $F' := F' \setminus \{(L', n_j)\} \cup \{(L', \omega)\}$            ▷ add with  $\omega$  if transition can be re-taken
9:       else
10:         $F' := F' \setminus \{(L', n_j)\} \cup \{(L', n_j + 1)\}$ 
11:     else if  $n_i = \omega$  and  $S' = S$  then
12:        $F' := F' \cup \{(L', \omega)\}$            ▷ add with  $\omega$  if transition can be re-taken
13:     else
14:        $F' := F' \cup \{(L', 1)\}$ 
15:   return  $\langle S', F' \rangle$ 

```

of the global variables have to be exactly the same. If such a state τ' has been found (line 5), any pair (L_i, n_i) that occurs in the new state τ with a higher counter value than in the preceding state τ' is replaced by the pair (L_i, ω) in lines 6 - 8.

Algorithm 8.3 Covering States

```

1: procedure UPDATECOVER( $\tau$ )
2:    $\sigma = \tau$ 
3:   let  $(L_i, n_i)$  be the  $i$ -th pair in  $\sigma = \langle S, (L_1, n_1), \dots, (L_k, n_k) \rangle$ 
4:   for all  $\tau' \in \text{Pred}(\tau)$  do
5:     if  $\tau' \subseteq \tau$  and  $S' = S$  then                                           ▷  $\tau$  subsumes  $\tau'$  and globals match
6:       let  $(L'_j, n'_j)$  be the  $j$ -th pair in  $\tau' = \langle S', (L'_1, n'_1), \dots, (L'_m, n'_m) \rangle$ 
7:        $I := \{i \mid L'_j = L_i, n'_j < n_i\}$            ▷ get partial index mapping from subsumption
8:        $\sigma := \sigma[(L_i, n_i)/(L_i, \omega), i \in I]$            ▷ 'Accelerate' counters
9:   return  $\sigma$ 

```

Note that — in contrast to an approach by Finkel [72] — we discard unexplored nodes instead of entire sub-trees. Our algorithm therefore does not suffer from the problems that make Finkel's method incomplete (see also [77]). 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. Bypassing the VASS gives us the opportunity to avoid the blowup that a static translation into any type of addition system invariably entails.

8.4.2 Practical Evaluation

We have implemented our algorithm both in an explicit-state and a symbolic version (see two subsections below). We have evaluated the implementation's performance on the same set of Boolean programs as the counter abstraction implementation (see Section 7.5 for a description).

Explicit-State Experiments

The explicit-state experiments are based on a collection of 131 Boolean programs; the programs are listed in Figure 8.3 for increasing running time. We note that explicit-state exploration engines cannot cope effectively with the data non-determinism, represented by the special value \star , that Boolean programs exhibit [46]. To make the explicit-state experiments meaningful, especially for large programs, we determinized the Boolean programs before feeding them into the tool. The transformed 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.

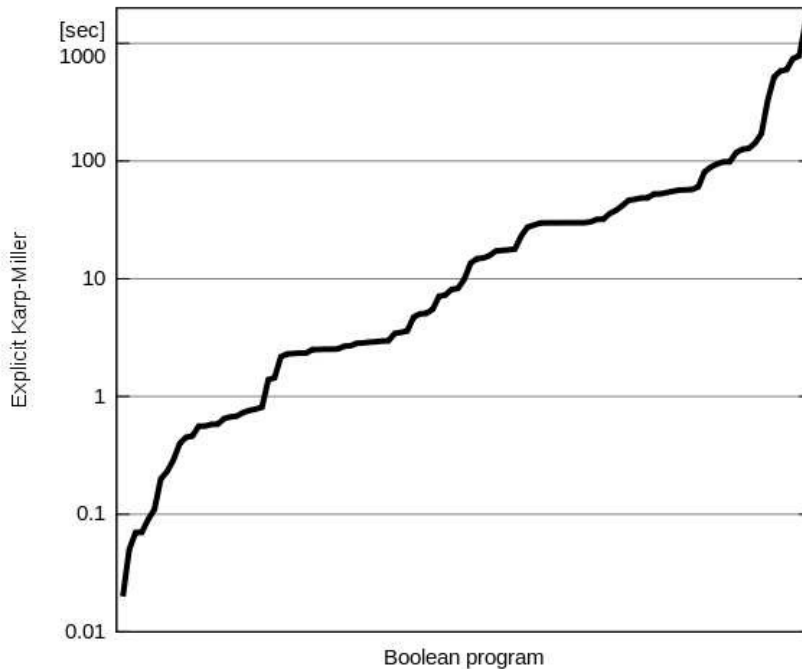


Figure 8.3: Performance of our explicit-state implementation on Boolean programs

In 20 cases, our Karp-Miller engine timed out; these cases are not shown in the figure. 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 %).

Symbolic Experiments

Our symbolic experiments are based on a collection of 270 Boolean programs for which our Karp-Miller implementation terminates; we chose a timeout of 60 min.

Recall that for a concurrent system with a fixed number of threads, the number of non-zero counters in a state is bounded by the number of threads. With dynamic thread creation, there is no such formal bound. Empirically, however, we observed no more than 43 non-zero counters, and the average being

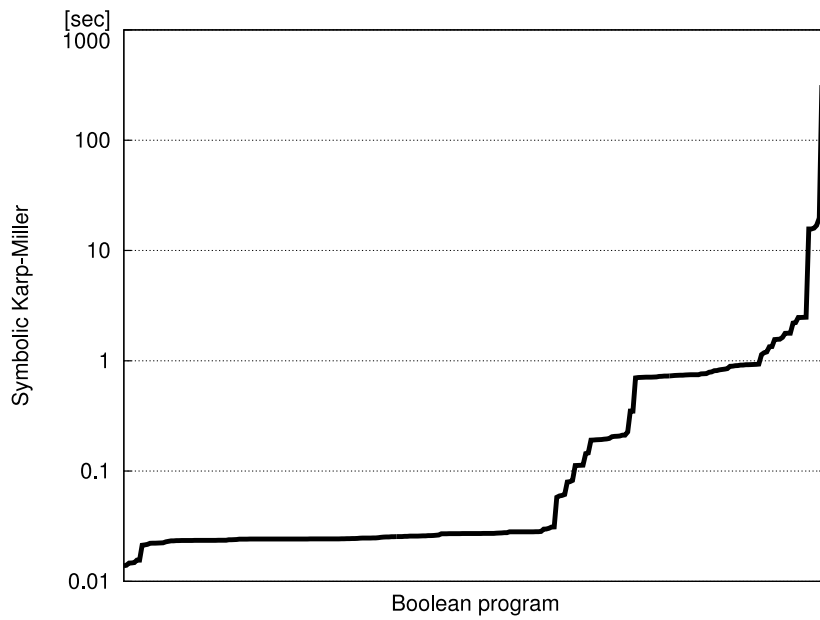


Figure 8.4: Performance of our Symbolic implementation on Boolean programs

12. This illustrates drastically the difference to a method that keeps counters for *all* local states, which can be millions.

Our experiments disclosed that a state subsumes its predecessor in 20% of the cases. Not surprisingly, recently discovered states have a higher likelihood to be subsumed by new states. Therefore, every expanded state is compared to the list of history states in reverse order. When we started working on the implementation, a profiler revealed that over 40% of the running time were spent in maintaining the history of maximal states. This motivated the search for more optimized data structures, which are described in the last chapter.

We also observe that a symbolic implementation is enabling for the verification of Boolean programs; the performance of the symbolic implementation is about one order of magnitude better than that of the explicit-state implementation, despite the fact that the benchmarks given to the explicit-state implementation have been simplified.

8.5 Summary

In this chapter, we have investigated the *thread-state reachability* problem for concurrent systems that originate from replicating a given finite-state program. This problem is algorithmically decidable by a reduction to VASS coverability. 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.

8.6 Bibliographic Notes

There is a vast amount of literature on tackling reachability analysis for concurrent software, with or without recursion, but the entire coverage of it is far beyond the scope of this thesis. 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 [76, 72, 56]. Most of these algorithms, however, do not address the problem of an intractable number of vector elements in 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 [112]. This improvement is based on static 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 [75], 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* [37] and *Data Decision*

Diagrams [50], or based on algorithmic improvements such as symmetry reduction [128], deleting configurations that will not be seen again [36], or partial-order techniques [70].

Tools: There are many tools available for the analysis of VASS [84]. The PEP tool is a popular coverability checker [81] 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. [76]. Furthermore, VASS analysis has been applied to Java programs [55] and Boolean programs [8]. 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; our experimental experiments indicate that a symbolic representation is mandatory for the verification of Boolean programs.

9

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 section, we trade 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.

9.1 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.¹ We call the smallest such number the *reachability cutoff* of \mathbb{P} and denote it by n_0 .

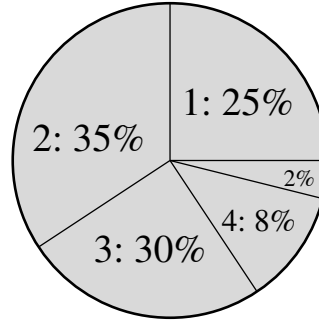
If the reachability cutoff of a Boolean program is known, the thread-state reachability problem can be solved by analyzing the program for the fixed cutoff-number of threads, without the Karp-Miller procedure. We show in the rest of this section why that might be useful, and why the observations made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

¹ 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. Put 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* [87]. This logic is similar to CTL, but allows only non-nested *reachability* and *bounded-reachability* modalities, $EF P$ and $EF^{\leq i} P$, respectively.

9.2 Reachability Cutoffs in Practice

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 $\{0, \dots, c\}$ 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. The chart on the right 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 calculated 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 (see Chapter 10). We observe that the cutoff is below 5 in all but very few examples.



9.3 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 9.1 (explicit) and 9.2 (symbolic) 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.

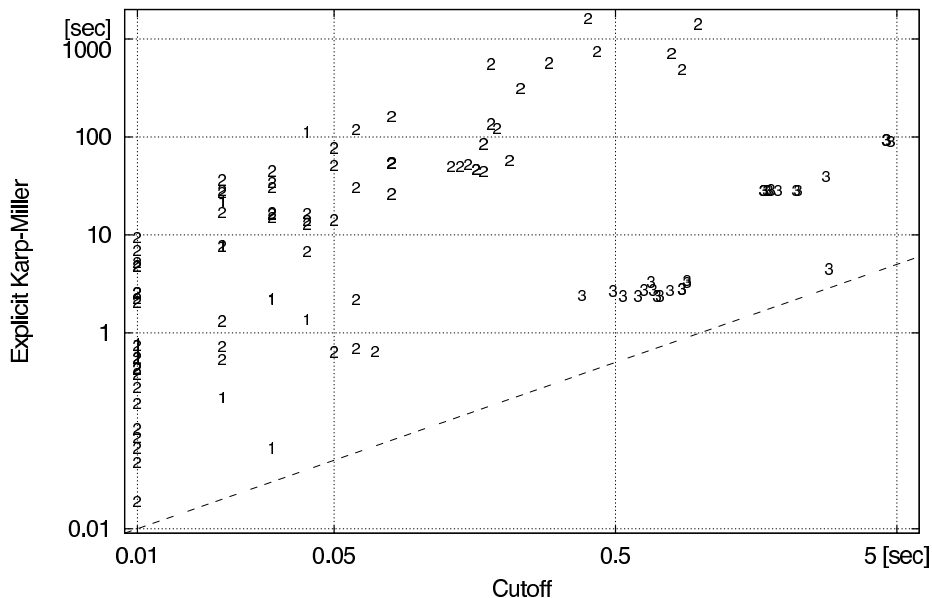


Figure 9.1: Comparing Karp-Miller and Cutoffs (explicit state)

We see from the figures that the cutoff-based solution is always faster, except in very few cases for which BOOM takes a tiny amount of time. 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.

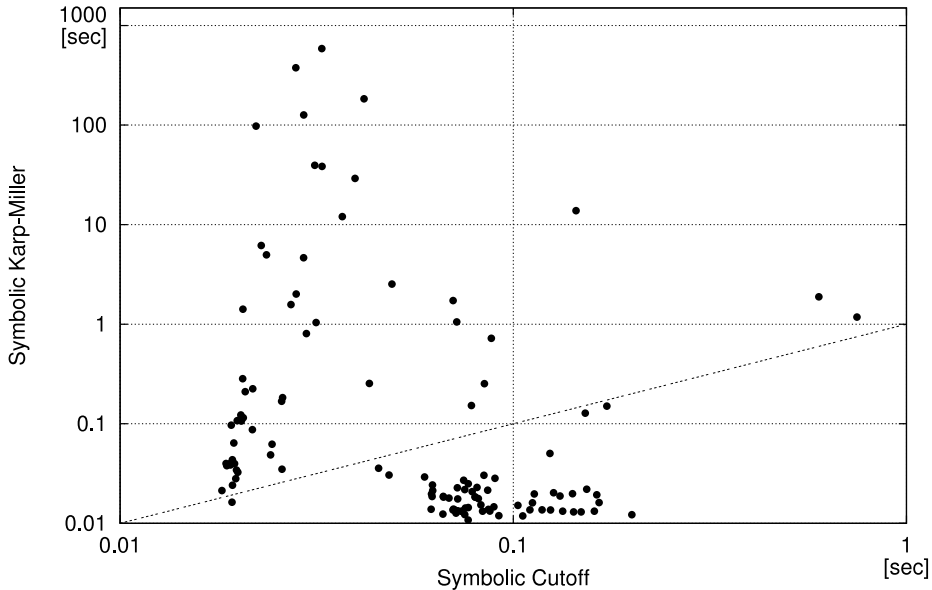


Figure 9.2: Comparing Karp-Miller and Cutoffs (symbolic)

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 [41]: “*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 reasoning² 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.

9.4 Summary

In order to explore a potentially yet more efficient solution to the coverability problem, we investigated the *reachability cutoff* of concurrent Boolean programs. We have demonstrated empirically that this

²i.e., inferring correctness for all from correctness for a few

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 suggest a frequently exhaustive reachability method.

9.5 Bibliographic Notes

Much of the work on verifying concurrent programs using cutoffs restricts communication [41, 65, 32, &c.]. For example, fixed cutoffs are known for ring networks communicating only by token passing [66]. Recently, multi-threaded programs communicating solely using locks were investigated by Kahlon, Gupta and Ivančić [92]. 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 [27]. 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.

Related Approaches. Similar 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 [118, &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.

Part III

Implementation

10

The Model Checker BOOM

THE previous chapters described the model checking algorithms from a high-level perspective. However, these high-level descriptions need to be divided into many small steps that are suitable for implementation. There are many design choices that influence the running time of an actual implementation and could potentially hamper an efficient algorithm in such a way that it performs worse than a less sophisticated algorithm.¹

This chapter presents BOOM [18], our model checking platform for Boolean programs. It contains an implementation of every algorithm presented in the previous chapters. We describe relevant ingredients of the implementation and compare alternative choices and their effects on efficiency.

BOOM and the benchmark set used in this thesis are available under the following URL:

<http://www.cprover.org/boom>.

10.1 Brief History of BOOM

The library supports concurrency mechanisms, which is important since the current elementary mode BOOM has been developed during four years and has grown to over 120'000 lines of code written in C++. The goal was a competitive reachability checker for the SATABS-framework [44]. It started as the first tool that implemented the summarization algorithm for Boolean programs using solvers for SAT and QBF formulas. Currently, MINISAT [59], QUANTOR [24] and SKIZZO [22] are supported.

In the course of about the past two years, support for concurrent Boolean programs has been added; we focused here on BDD-based data structures. Beginning with a simple symbolic exploration engine, we added various forms of partial-order reduction to optimize the performance. This was

¹Practice seems to favor simple solutions. The author was quite surprised by the improved runtime, when he replaced the classic logarithmic algorithm for computing dominators in a control-flow graph (needed for static single assignment form) with a very simple algorithm with quadratic complexity [49]! The constant of the logarithmic algorithm was in a region, where even a Boolean programs of substantial size was not large enough to outperform the quadratic algorithm.

followed by our implementation of counter abstraction (Chapter 7), which is conceptually easy, but technically non-trivial in a symbolic setting. From there, it was a relatively small step to extend the method to unbounded threads, realizing that converting a Boolean program to a vector-addition system is tantamount to applying unbounded counter abstraction to it. The challenge is rather to make the exploration of the resulting counter machine efficient.

The remainder of this chapter describes in detail our efforts to obtain an implementation that is efficient on one hand and on the other hand modular enough to allow reuse of core components as, e.g., partial-order reduction.

10.2 Explicit vs. Symbolic Program Counters

The idea of partitioning the transition relation has been known in the literature for a while (see e.g. [17]). The purpose of this optimization is to minimize formula size and existentially quantify variables as soon as possible.

We adopt a disjunctive partitioning over the program counter as proposed in [13]. Its authors propose to use a hybrid state representation that keeps only the variables in BDDs and stores the program counter in explicit form. This comes natural for Boolean programs. We find several advantages of this approach compared to an encoding that uses a single formula to represent the whole Boolean program:

- In combination with a *live variable analysis* (see Section 10.9), *dead* variables can be identified and subsequently quantified existentially in a BDD or removed from the QBF-formula that is used for fix-point detection.
- *Partial order reduction* methods need to know about the dependency of transitions (see, e.g., Section 6.2.1). This information can be computed over the set of variables read and written in a statement.
- Since the *program counter* is not encoded symbolically, the formula and therefore the BDD becomes smaller.
- Our implementation of a symbolic counter-abstraction algorithm needs to identify states that have to be split in order to avoid a loss of precision in the representation. Having the program counter available in explicit form allows an up-front identification of possible locations where a splice-test has to be applied. Moreover, we found empirical evidence that counter-abstraction without explicit program counters performs worse compared to our chosen approach (see Section 7.5.5).

In combination with a SAT-based approach, the *hybrid* state representation using an explicit program counter offers some additional benefits:

- Only the parts of the transition relation that are relevant for the reachable locations are unrolled. This slows down formula growth.
- Since the program counter is not encoded in the formula, states that only differ in the program locations result in the same QBF formula for fix-point computation and caching thereof improves runtime substantially.
- A SAT-solver can detect infeasible paths and prevents them from being explored further.

Experimental Evaluation The first BDD-based symbolic algorithm for reachability checking of non-recursive Boolean programs we implemented within BOOM used a symbolic representation of the program counter. Its advantage was its simplicity. The transition relation of the whole Boolean program can be represented within a single BDD, leading to a simple saturating loop as the model checkers core. The more sophisticated hybrid variant was subsequently developed could then be cross-checked against the simple and less error-prone implementation.

We compared both implementations on the benchmark set explained in appendix B. These concurrent benchmarks were instantiated with only one thread and stripped of any skip statements. The timeout was set to 3600s and any benchmark, which finished in less than 0.05s, has been filtered out. As Figure 10.1 depicts, the hybrid algorithm that uses an explicit program counter outperformed the symbolic encoding by three orders of magnitude on average. This confirms that the common practice of keeping the program counter explicitly is faster than storing it in the BDD .

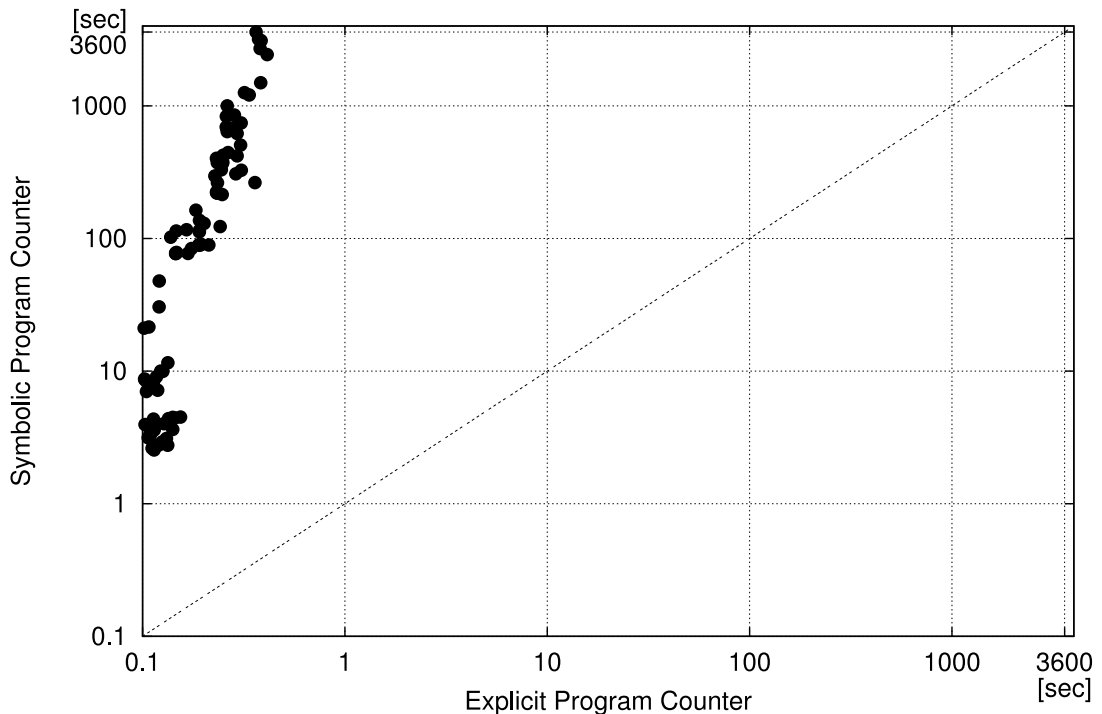


Figure 10.1: Comparing a symbolic with an explicit program counter

Variable Ordering The BDD-based parts of BOOM use a heuristic in hope to find an order over the variables such that the BDDs are small. A static analysis of the Boolean program determines any variables that are involved in assignments and tries to place the dependent variables as close as possible. We borrowed this idea from [13]. A detailed descriptions on how the variable ordering of a BDD-based model checker can be tuned to Boolean programs can be found on page 148 ff. in [122]. The optimal variable ordering for a symbolic encoding of the program counter is to place the program counter first and then the global / local variables according to the dependency.

Example 10.1 We visualize a possible variable ordering for a fictive non-recursive Boolean program,

that has n replicated threads. The (bit-blasted) program counters are represented by ranges of variables pc_t for each thread t . The global and local variables are ordered in a concatenated manner, that is, replications of a local variable l are placed adjacent (i.e., as $l[0], \dots, l[n]$).

pc_0, \dots, pc_n	g_0	g_1	$l_0[0], \dots, l_0[n]$	g_2	$l_1[0], l_2[0], \dots, l_1[n], l_2[n]$	\dots
---------------------	-------	-------	-------------------------	-------	---	---------

10.3 State Representation

We use the *hybrid* state representation (as mentioned in the last section) for the SAT- and BDD-based algorithms. As a logical consequence of the fact that a symbolic state is not represented by a single formula alone, we choose build complex state objects. Each state contains a Boolean flag that indicates, if any thread resides in an atomic section. The thread identifier of the last thread scheduled is stored explicitly as an integer number.

10.3.1 States in a SAT-based Environment

Figure 10.2 sketches the layout of a concurrent SAT-state. The *guard* γ (see also Definition 3.3), the *shared*, and the *local* variables are shared pointers to formulæ. Representing the variables of a Boolean program as a vector of shared formulas has several advantages. The translation into conjunctive normal form (CNF) does not lead to an unnecessary blow-up in the size of the formula, e.g., if two variables get assigned to the same expression.

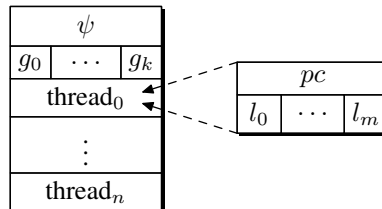


Figure 10.2: Representation of a concurrent SAT-state

Example 10.2 Let us recapitulate Example 3.3 while illustrating the manipulated data structures. Recall that we would like to construct the symbolic successor for the state

$$\left\langle \ell_{10}, (\star_1 \vee \star_2), \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto (\neg \star_1 \wedge \star_2) \end{array} \right\} \right\rangle \quad (10.1)$$

which is drawn in Figure 10.3. The arguments of the Boolean connectives are shared pointers (denoted by $\boxed{\bullet}$), that point to objects of type formula representing the expression. The statement executed has been slightly modified. We added an assignment of a non-deterministic value to b :

$$a, b := \neg a, \star \text{ constrain } (\neg b \vee a')$$

According to Table 3.1, we obtain $\omega_2 = \{a \mapsto \neg \star_1, b \mapsto \star_3\}$ and $\gamma_2 = (\star_1 \vee \star_2) \wedge (\star_1 \vee \neg \star_2 \vee \neg \star_1) \equiv (\star_1 \vee \star_2)$. Thus, the symbolic successor state results in:

$$\left\langle \ell_{11}, (\star_1 \vee \star_2), \left\{ \begin{array}{l} a \mapsto \neg \star_1, \\ b \mapsto \star_3 \end{array} \right\} \right\rangle \quad (10.2)$$

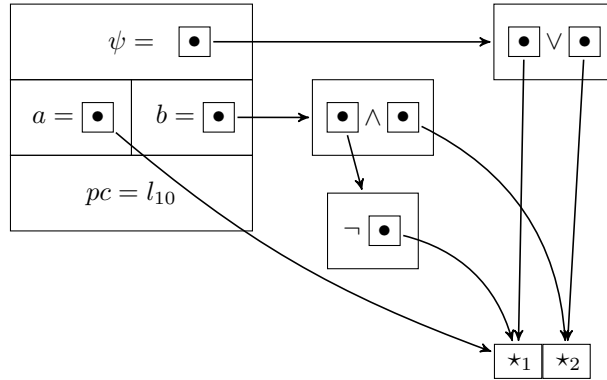


Figure 10.3: SAT-state with two shared variables

Note that, the corresponding data structure illustrated in Figure 10.4 cannot simplify the expression of the guard γ_2 nor is it able to detect the redundant negation of the non-deterministic variable \star_1 . The non-determinism in the assignment to b results in allocation of a fresh variable \star_3 .

As Example 10.2 already indicates, every non-deterministic variable, which is used in an expression, is added to the guard or assigned to a variable, needs to be replaced first with a fresh non-deterministic variable. Thus, whenever an expression of the abstract syntax tree is used or a path-summary (Definition 4.2) is applied, the corresponding formula representation is translated into a new formula with a fresh set of non-deterministic variables. Simplifications of the formula such as constant propagation can be done during this translation process.

Optimizations. Many automatically generated Boolean programs contain a large amount of **skip** and single-successor **goto** statements. They do not change the symbolically encoded part of the state, namely the guard and the variables. Thus they induce the same fix-point formulæ. Although an up-front optimization step is able to remove many of these statements, we found a significant improvement in runtime, if the QBF-formulæ used for fix-point computations are stored in a cache that is organized as a hash table.

10.3.2 States in a BDD-based Environment

The representation of a concurrent symbolic state using BDDs is illustrated in Figure 10.5. A single BDD encodes the values of the shared and local variables whereas the program counters of the threads are stored explicitly. An alternative method of storing hybrid states is proposed by the authors of BEBOP for the case of sequential Boolean programs. They associate an array of BDDs with each statement in the syntax tree. The index i in the array contains the symbolic values of the variables, reachable after i steps from the initial location. In contrast to their approach, our solution works for sequential as well as for concurrent Boolean programs.

of the summary edge to be equal to the entry variables of the call.

Unrolling the transition relation only along reachable paths has the advantage that infeasible paths are not explored.

10.5 State Merging

Exploring the set of reachable states path by path in a naïve way causes an explosion in the number of paths. Thus, we *merge* multiple symbolic states into one. However, for some settings, such as the symbolic counter abstraction (see Section 7.5.1), this is not always possible.

Merging states in a BDD-based implementation is straightforward: a new state is built by taking the disjunction of the BDD-part of all merged states. The SAT-based implementation is more complex: we need to introduce a fresh non-deterministic auxiliary variable, whenever two states are merged into one.

Example 10.3 Consider the code snippet in Listing 10.1 and the following symbolic entry state of procedure $f_{\circ\circ}$:

$$\left\langle \ell_0, true, \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto \star_2 \end{array} \right\} \right\rangle \quad (10.3)$$

Applying the rules of Table 3.1 results in:

$$\left\langle \ell_1, \neg\star_1, \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto \star_1 \end{array} \right\} \right\rangle \quad (10.4)$$

There are now two paths that produce a state at location ℓ_2 , one from the transition $\ell_1 \rightarrow \ell_2$:

$$\sigma' = \left\langle \ell_2, \star_1, \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto \star_2 \end{array} \right\} \right\rangle \quad (10.5)$$

and one from the transition $\ell_0 \rightarrow \ell_2$:

$$\sigma'' = \left\langle \ell_2, \neg\star_1, \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto \star_1 \end{array} \right\} \right\rangle \quad (10.6)$$

Let \star_m be the non-deterministic variable used for the merging process. Hence, the merged state is:

$$\begin{aligned} \sigma &= \text{if } \star_m \text{ then } \sigma' \text{ else } \sigma'' \\ &= \left\langle \ell_2, \star_m \wedge \star_1 \vee \neg\star_m \wedge \neg\star_1, \left\{ \begin{array}{l} a \mapsto \star_1, \\ b \mapsto \star_m \wedge \star_2 \vee \neg\star_m \wedge \star_1 \end{array} \right\} \right\rangle \end{aligned} \quad (10.7)$$

Example 10.3 demonstrates that merging states is the essential ingredient to obtaining a single formula that represents the unrolled transition relation of a function. Recall that we presented in Section 4.2 *universal summaries* as an alternative to path summaries. An implementation that builds these universal summaries needs to unroll each loop, until the longest path without a recurring state is found. One possibility to achieve this, is to copy the loop body for each unrolling and generate static single assignment form (SSA) [51]. Its ϕ -functions could be used to build a disjunctive formula for each variable of converging paths. However, in combination with a work-list that delays the expansion of states closer to the exit location of a function, the state merging approach completely avoids the need for SSA.

Listing 10.1: Corresponding code snippet for Example 10.3

```

decl a, b;

void foo() begin
10: if a then goto 12;
11: b := a;
12: return;

end

```

10.6 Organization of Work-lists

BOOM employs two work-lists to organize the states. One work-list, called the *history*, collects any state visited during the search. Its purpose is to circumvent re-expansion of states that have been seen before. After the expansion of a state, its successor states are compared against the history and any new state is added to the so-called *frontier* work-list. It contains any state that still needs to be expanded.

At this point, we list some possible flavors of frontier work-lists:

- A *depth-first* search is obtained by using a first-in-first-out stack: states are always added and removed from the same end.
- In order to perform a *breadth-first* search, it is advantageous to use two frontier work-lists instead of a last-in-first-out stack: the first work-list contains any states that can be reached after n steps whereas the second work-list contains states reachable after $n + 1$ steps. The states to expand are removed from the first work-list, while the new states are added to the second one. When the first work-list is empty, it is swapped with the second work-list. This approach does not imply an ordering on the states in the work-lists. Hence, we are able to choose a data structure that allows a fast lookup of states with specific program counters. Before a state is added to the work-list, we check, if it already contains a state suitable for merging and if this is the case, we merge both states. This results in a shorter work-list, because we merge states whenever possible.
- A work-list that aggressively merges states is obtained by prioritizing states according to 1) the distance from the entry node of a function and 2) the type of instruction that is executed next. If we assign the lowest priorities to the **call** and **return** statement, then we are able to favor internal transitions of a procedure. As the authors of [131] argue, internal transitions occur more often than calls or returns from a procedure and are also more expensive, since, e.g., in the setting of BDDs, two quantification operations are needed. Algorithm 10.1 assigns a priority to each node in the CFG such that nodes that are further away from a return node have higher priority. The idea is that states with nodes that are closer to a return statement are *delayed* until any state with a predecesing node has been executed.

BOOM uses breadth-first search work-lists for the counter abstraction / Karp-Miller construction and the maximal merging work-list for the sequential algorithms. In the context of the plain symbolic algorithm and the symbolic counter abstraction, the underlying data structure of the work-list is a mapping from a multiset of program locations / counter tuples to a list of states residing at these program locations (i.e., these states differ only in the values of the variables).

Clearly, state merging (see Section 10.5) is always appropriate in a BDD-based setting but the same is not true for a SAT-based implementation. State merging could, in the worst case, let the formulæ

Algorithm 10.1 Assigning priorities to statements such that merging is maximized

```

1: procedure ASSIGNPRIORITIES(pr)
2:   for all nodes  $n \in \text{CFG}(\mathbf{pr})$  do                                ▷ initialize data structures
3:     SIZE( $n$ ) := 0;
4:     VISITED :=  $\emptyset$ ;
5:   COMPUTESIZE(ENTRYNODE(CFG(pr)));                                ▷ start with procedure entry
6:   sort nodes in CFG according to size of each node
7:   assign highest priority to node with biggest subtree

8: procedure COMPUTESIZE( $n$ )                                          ▷ compute size of successor subtree
9:   insert  $n$  into VISITED;
10:  size := 1;
11:  for all  $s \in \text{SUCCESSORS}(n)$  do
12:    if  $s \notin \text{VISITED}$  then                                     ▷ ignore cycles
13:      COMPUTESIZE( $s$ );                                           ▷ recursively compute size
14:      size := size + SIZE( $s$ );
15:  SIZE( $n$ ) := size;

```

grow exponentially. A compromise, that works well in practice, is to merge only states that are in the frontier work-list and not use merging for states in the history work-list.

10.7 Optimizations for the Karp-Miller Coverability Tree

Section 8.4.2 demonstrates that our Karp-Miller implementation times out on many instances, despite our efforts to obtain an efficient implementation. This section explains optimizations, which improved the runtime of the first implementation by a factor of 4-5. However, a more sophisticated algorithm than the original one from Karp and Miller, combined with tailored data structures (as done by the authors of [77]), could change the picture dramatically.

10.7.1 Efficient State Subsumption

During the development of the first implementation, we could significantly improve the performance on our benchmarks, by switching to a history work-list that keeps only copies of the maximal states with respect to the covering relation as a partial order (the coverability tree is stored implicitly, since every state has a pointer to its predecessor(s)). We over-approximate the subsumption test, since an exact test would require solving a problem similar to the subset cover problem whenever there are thread states that have the same program locations and counter but different values of the variables. Recall that every symbolic state contains a mapping from a local variables / program locations pair of its counter (we order it according to the hash key of the BDD). This ordering permits an over-approximating subsumption check that is linear in the size of this map.

Not surprisingly, maintaining the history became the bottleneck. Since the number of threads is not constant anymore, a mapping from program locations / counter values to a list of states is not appropriate to sort out states that cannot be supersets or subsets. Thus, the first yet naïve implementation kept any states in a list and applied the subsumption check for every new state against any state in this list, see Algorithm 10.2. We observed that the check for subsumption (both directions) seems to find a subsumed state quicker if a new state is compared to the most recent states first. We attribute this to

our finding that in up to 20% of the cases, a state subsumes its predecessor.

Algorithm 10.2 Adding a state to a work-list of reachable and *maximally* covering states

```

1: procedure ISHISTORY( $\sigma$ )
2:   for all  $\sigma' \in History$  do                                ▷ Traverse History forwards
3:     if  $\sigma \subseteq \sigma'$  then return true                    ▷  $\sigma$  is subsumed by  $\sigma'$ 
4:     if  $\sigma \supseteq \sigma'$  then                                ▷  $\sigma$  subsumes  $\sigma'$ 
5:       Remove  $\sigma'$ 
6:   Add  $\sigma$  to front of History
7:   return false

```

An Improvement Using Bitsets. Algorithm 10.2 can be optimized by adapting a trick that is widely used in compiler construction. Namely, the idea to use bitsets as the data structure to represent sets. Their main advantage is that set union and intersection operations are performed in parallel by the hardware. With help of these operations we can build a fast subset check, which is essential for our improved history check. It picks up the same concept of mapping program counters to a list of states as for the case of bounded threads. We project only the program counters of the threads in a state to a bitset. Thus we get an efficient over-approximation for the subsumption check: a state can only be subsumed by another state, if the corresponding bitset is a subset. This avoids many negative subsumption checks and results in the mentioned speedup.

10.8 Handling of Non-determinism in Expressions

Expressions occurring in assignments and conditions of Boolean programs can contain an arbitrary number of non-deterministic values (see Section 3.1). The extra non-determinism in expressions is necessary to preserve the soundness of the abstraction. This section describes how non-deterministic values in expressions are handled by a symbolic model checker. A naïve implementation can result in a considerable loss of performance.

The usage of symbolic algorithms in model checkers is often motivated by data non-determinism. Symbolic state representations allow a compact representation of non-deterministic values. Similarly, non-determinism must be allowed in transition functions in order to keep them compact. However, explicit non-deterministic values in expressions (denoted by \star) raise a problem: the value of \star can not be represented by a Boolean formula, since it must resolve both to *true* and to *false*. Thus an explicit-state model checker would evaluate such a formula once for every possible value of the non-deterministic value \star (we will coin the term “ \star -variable” for them).

Example 10.4 Consider the expression “ $a := b \vee \star$ ”: If b is false, then the expression non-deterministically evaluates to false or true, but if b is true the expression becomes true as well.

Computing every possible value of an expression containing non-determinism is exponential in the number of \star -variables and runs contrary to the purpose of a symbolic transition relation. Unfortunately there is a catch in a BDD-based model checker: adding auxiliary variables can hamper an optimal variable ordering and increase the size of the BDD significantly. Thus we are interested in a method that does not need any auxiliary variables to handle the \star -variables.

In the remainder of this section we will show how \star -variables can be handled efficiently using BDDs.

Conditionals: Within **assume** expressions, every non-deterministic value \star can be considered as existentially quantified (i.e. if the expression evaluates to *false*, there is no succeeding state) and thus the formula can be simplified accordingly.

Example 10.5 The instruction “**assume** ($b \wedge \star$);” can be simplified to “**assume** (b);”.

Assignments: Assignments of the form “ $v := \star$ **constrain** (c);” (where c does not contain any \star -variables) do not need any auxiliary variables during the next state computation. Consider Ω representing the formula of the current state and Ω' of the next state accordingly. An image function for such a statement would build the conjunction of the formula representing the actual state and subsequently quantify existentially all variables that are set to \star : $\Omega' \equiv \exists v. \Omega \wedge b$. Then a renaming step permutes every primed variable with its unprimed counterpart.

Assignments in Boolean programs generated by SLAM: They feature no \star values within conditions. Only nondeterministic assignments using the **schoose** statement are allowed (e.g., “ $a := \star$;” is written as “ $a :=$ **schoose** [F, F];”). The image function of the general form “ $v :=$ **schoose** [a, b];” can be built in a similar way as before: $\Omega' \equiv \Omega \wedge (a \wedge v') \vee (\bar{a} \wedge b \wedge \bar{v}')$

The question that arises now is:

How can we treat an assignment like “ $v := \star \vee b$;” in a BDD-based Model Checker without having to resort to auxiliary variables for the \star -variables?

Any assignment of the form “ $v := e$;” can be rewritten as “ $v := \star$ **constrain** ($v' = e$);”. Since the **constrain**-expression is added using a conjunction to the formula of the next state, every \star -variable can be quantified universally without affecting the satisfiability of the formula that represents the transition relation.

Example 10.6 Consider the statement: “ $a := (b \wedge \star \vee (c \vee \star)) \vee \star$;”:

The same rewriting boils down to “ $a := \star$ **constrain** $a' = (b \wedge \star \vee (c \vee \star)) \vee \star$;”.

Now the \star -variables are quantified universally:

$$\begin{aligned}
 a' = (b \wedge \star \vee (c \vee \star)) \vee \star &\equiv & a' &= [(b \wedge 0 \vee (c \vee 0)) \vee 0] \\
 && \vee & a' &= [(b \wedge 0 \vee (c \vee 0)) \vee 1] \\
 && \vee & a' &= [(b \wedge 0 \vee (c \vee 1)) \vee 0] \\
 && \vee & a' &= [(b \wedge 0 \vee (c \vee 1)) \vee 1] \\
 && \vee & a' &= [(b \wedge 1 \vee (c \vee 0)) \vee 0] \\
 && \vee & a' &= [(b \wedge 1 \vee (c \vee 0)) \vee 1] \\
 && \vee & a' &= [(b \wedge 1 \vee (c \vee 1)) \vee 0] \\
 && \vee & a' &= [(b \wedge 1 \vee (c \vee 1)) \vee 1] \\
 &\equiv & a' &= c \\
 && \vee & a' &= \top \\
 && \vee & a' &= b \wedge c
 \end{aligned}$$

This analysis reveals that a can always become true or false if $c = 0$, but if c is true (since $c \vee (b \wedge c) \equiv c$), a must be true as well. Therefore the assignment can be rewritten as: “ $a := \star$ **constrain** $c \implies a'$;”.

However, in the event of many \star -variables this approach could become very expensive. Thus this observation gives rise to an algorithm based on BDDs (Algorithm 10.3) that normalizes every

assignment of the form “ $v := e$ ”, where e may contain \star -variables, into an equivalent assignment of the form “ $v := \star \text{constrain } c$ ” such that c does not contain any \star -variables.

Algorithm 10.3 Normalize an assignment “ $v := e$ ” into the form “ $v := \star \text{constrain } c$ ”

```

1: procedure NORMALIZEASSIGNMENT(“ $v := e$ ;”)
2:    $\Omega := \text{BUILDBDDFOREXPRESSION}(\text{“}v := e\text{”}) \otimes v'$ 
3:    $\Omega_\star := \forall \star_{1,\dots,k} \in \Omega$  ▷ Quantify universally every  $\star$ -variable
4:    $[\text{maxterms}^-, \text{maxterms}^+] := \text{EXTRACTIMPLICATION}(\Omega_\star, v)$ 
5:   if  $\Omega_\star \wedge v' = \neg(\Omega_\star \wedge v')$  then
6:     return “ $v := \text{maxterms}^-$ ;” ▷ Replace with assignment
7:   if  $\bigvee \text{maxterms}^- \neq \text{false}$  then
8:      $e^- := \text{“}\bigvee \text{maxterms}^- \implies v\text{”}$ 
9:   else
10:     $e^- := \text{true}$ 
11:   if  $\bigvee \text{maxterms}^+ \neq \text{false}$  then
12:     $e^+ := \text{“}\bigvee \text{maxterms}^+ \implies \bar{v}\text{”}$ 
13:   else
14:     $e^+ := \text{true}$ 
15:   return “ $v := \star \text{constrain } e^+ \wedge e^-$ ”

16: procedure EXTRACTIMPLICATION( $\Omega, v$ )
17:    $\text{maxterms}^- := \text{maxterms}^+ := \emptyset$ 
18:   for all  $\text{cube} \in \text{CUBES}(\neg\Omega)$  do
19:      $\mathcal{L}^- := \text{NEGATIVELITERALS}(\text{cube}) \setminus \text{LITERALS}(v)$ 
20:      $\mathcal{L}^+ := \text{POSITIVELITERALS}(\text{cube}) \setminus \text{LITERALS}(v)$ 
21:      $\text{maxterms}^- := \text{maxterm}^- \cup \mathcal{L}^-$ 
22:      $\text{maxterms}^+ := \text{maxterm}^+ \cup \mathcal{L}^+$ 
23:   return  $[\text{maxterms}^-, \text{maxterms}^+]$ 

```

10.9 Dead Variable Optimization

The idea suggests itself to use well-known compiler optimization techniques to reduce the size of the BDDs respectively QBF-formulae. For this purpose [13, 46] already propose a **live variable** analysis and a **mod/ref** analysis. The idea of a *dead* variable optimization is to exploit the fact some variables are defined but never read again during analysis.

Definition 10.1 A variable is called *live* at a particular point n in a Boolean program \mathcal{B} if there exists a path starting at n in the control-flow graph of \mathcal{B} along which its value is used before being defined. The variable is *dead* if there is no such path.

If a variable is dead at node n , then the model checker can safely discard its value when reaching n .

10.9.1 Interprocedural Live Variable Analysis

Unfortunately, an off the shelf live variable analysis algorithm (e. g. as presented in [111]) is not sufficient for Boolean programs with function calls which are for example generated by SATABS or

SLAM². A *local* live variable analysis could consider a global variable as dead although its value is read at a later point. A trivial solution to this problem would be to treat any global variable as live at any location, regardless if it is used or not in that function. However, if we aim for a more optimized solution, then the problem is a bit more complicated.

Example 10.7 In Listing 10.2 the global variable g is dead in function `foo`. However, if its value is discarded in this function, then the assertion in the `main` function would fail. A tentative solution to this problem would be to “save” the value of the global variable over the call, and infer it again after the summary of the function `foo` as been inserted. However that would at first require that g is marked as live in `foo`, which needs an interprocedural live variable analysis.

Listing 10.2: The assertion fails if g is not considered live in `foo`

```

decl g;

void main ()
begin
  g := T;
  foo ();
  assert (g);
end

void foo ()
begin
  skip;
end

```

The *interprocedural* live variable analysis implemented in BOOM is basically a combination of a mod/ref analysis followed by a standard live variable analysis. More details can be found in appendix C.

Both SAT- and BDD-based implementations of a model checker can exploit the fact that a variable became dead:

- SAT-based implementations may simplify the QBF formula used for fixed-point detection because only the variables that are alive must be included. (There is no positive effect on state merging, since the resulting formulæ are never read.)
- After every image computation in a model checker that uses BDDs, the “dead” variables can be existentially quantified out, such that the BDDs become more compact. Variations that allow only a restricted form of state merging (such as symbolic counter abstraction, see Chapter 7) can benefit from more opportunities to merge states.

²The **dead** statement specifies variables that are not alive anymore but this statement is only supported by SLAM so far.

10.10 A GUI for Analyzing Boolean programs

Using Boolean programs as the choice of abstracting higher-level programs has the advantage that abstraction generation and analysis can be decoupled, and both components can be tested separately during development. This proved to be quite useful for debugging and benchmarking the Boolean model checker. Once both components work satisfactory, it is an easy task to directly integrate the Boolean model checker into the abstraction and refinement framework. For example BOOM could be directly integrated into SATABS via a dynamically linked library and the Boolean program could be passed as an abstract syntax tree, thus bypassing the generation and parsing of an intermediate Boolean program file. A major drawback of this strict separation is that after every iteration, the Boolean model checker has to start from scratch, although the refined model might differ from the previous one only by a small incremental change. However, keeping the cache of BDDs and QBF formulæ between iterations should alleviate these effects.

Boolean programs are — although automatically generated — still humanly readable. Therefore, within the scope of this thesis, a plug-in for the Eclipse IDE[89] has been developed. A screenshot of the editor component is illustrated in Figure 10.6

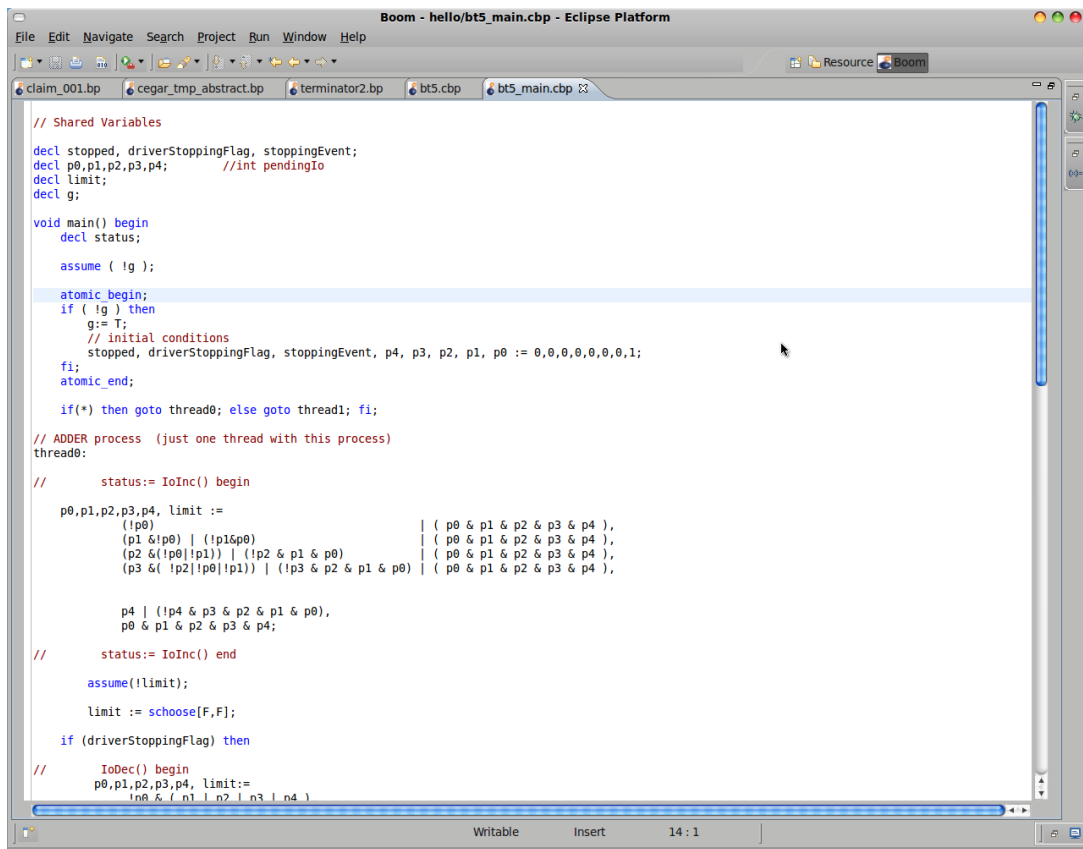


Figure 10.6: The editor with syntax highlighting

In order to present a counterexample through the Boolean program to the user, the "Debug View",

which is usually applied in the context of languages like Java, is imitated (see Figure 10.7). The windows that display the values of the program variables and the Boolean program itself keep their usual meaning. However, the window that usually shows the call stack is altered such that it displays the path of the counterexample (left side). The user can navigate forwards and backwards along the path in this window and the variable and source code windows will update their contents accordingly.

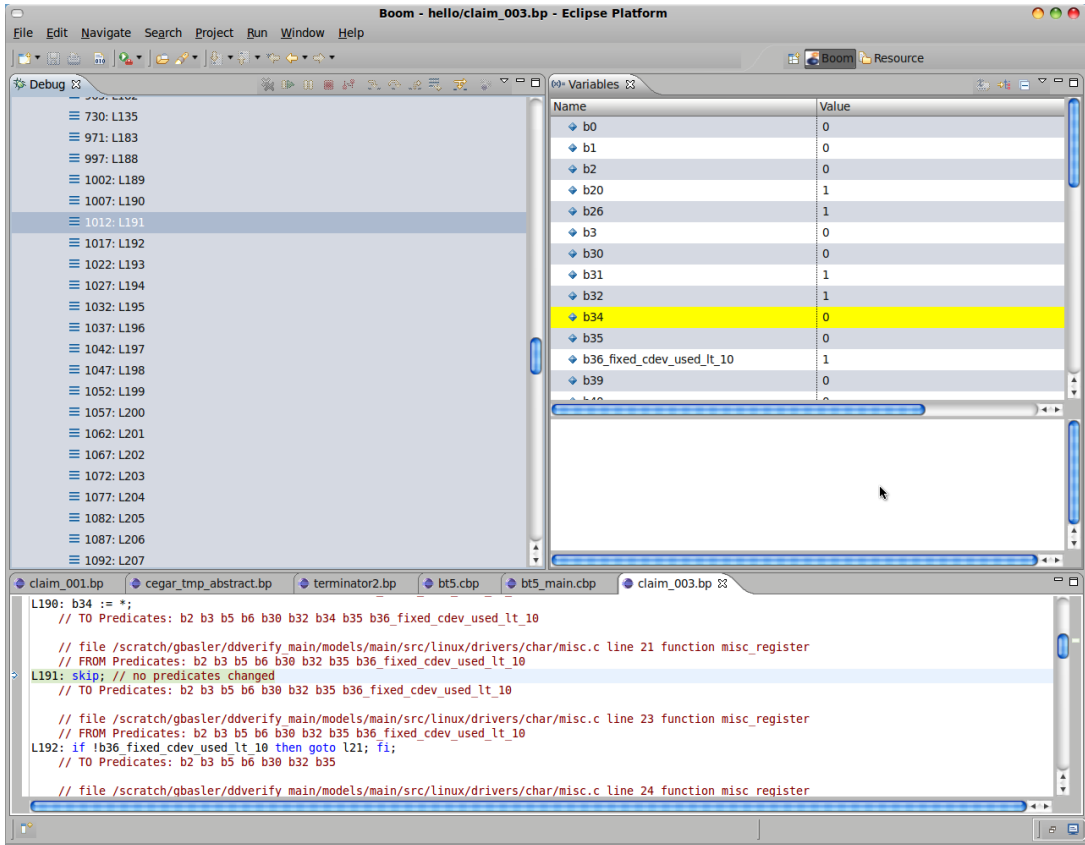


Figure 10.7: Displaying a counterexample in the "Debug View" of Eclipse

11

Conclusions

WE conclude the thesis with a brief summary of the main contributions and discuss areas of possible future research. This thesis focuses on algorithms and their implementation for assertion checking of Boolean programs. We successively developed the tool BOOM, which incorporates every algorithm presented in the thesis. We compared BOOM with similar tools using Boolean program benchmarks that were generated by the CEGAR-based frameworks SATABS and SLAM.

11.1 Summary of Contributions

This thesis makes the following key contributions:

Algorithms for Reachability Checking of Sequential Boolean Programs We presented a SAT based model checking algorithm for sequential recursive Boolean programs that uses a QBF solver to compute the least fix-point of the set of summary edges of a procedure. Furthermore, we introduced the concept of *universal summaries*, that relate arbitrary input states to their respective return states according to the transition relation of the function. Universal summaries can be applied in any calling context. We implemented an algorithm that uses bounded model checking to compute universal summaries for functions without recursive calls, and QBF to compute summaries in the presence of recursion. Furthermore, we described an over-approximating algorithm for procedures that computes and applies universal summaries in a *lazy* manner.

Our benchmarks show that this approach performs better than BDD based algorithms when it comes to detecting bugs, but is less efficient for proving the unreachability of error states. This is a very useful result, since CEGAR-based model checkers generate Boolean programs with reachable error locations in all but the last iteration.

Algorithms for Reachability Checking of Concurrent Boolean Programs We have presented an algorithm for BDD-based symbolic state space exploration of concurrent Boolean programs. The algorithm draws its efficiency from counter abstraction as a reduction technique, without resorting to approximation at any time. It is specifically designed to cope with large numbers of local states and thus addresses a classical bottleneck in implementations of counter abstraction. We have shown how to avoid the *local state space explosion* problem using a combination of two techniques: 1) achieving context-awareness by interleaving the translation with the state space exploration, and 2) ensuring that only non-zero counters and their corresponding local states are kept in memory.

We have also investigated in detail the relationship between our implementation of counter abstraction and partial-order methods. Our experiments seem to confirm the folk wisdom that symmetry and partial-order reduction are, although not independent, certainly complementary and can be combined for yet more effective compression.

Furthermore, we have presented a symbolic implementation that permits unbounded dynamic thread creation, or the parameterized version of the concurrent reachability problem. The enabling feature was our initial implementation of counter abstraction, since this facilitated a reduction to VASS coverability. 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.

The following table visualizes the types of Boolean programs that have been investigated in this thesis and refers to the corresponding chapter:

	Sequential Boolean Programs	Concurrent Boolean Programs	
		Bounded #Threads	Unbounded #Threads
No recursion Recursion	Chapter 4	Undecidable	
	Chapter 4	Chapter 7	Chapter 8

11.2 Future Work

Unsolved Problems Chapter 8 demonstrated empirically that the reachability cutoff of concurrent Boolean programs tends to be very small in practice. In fact, it is often so small that running a model checker with the exact cutoff-number of threads is more, sometimes much more, efficient than building the Karp-Miller tree. Thus, an interesting area of follow-up work is the search for a method that over-approximates this cutoff.

During the time this thesis has been written, [Kaiser et al.](#) continued the work presented in Chapter 9. Given a fixed number of threads n and a reachability problem, the authors found a dynamic criterion to decide, if n is a cutoff. Their iterative model checking algorithm works as follows. In the first step, the number of threads is bounded to a fixed number n . During reachability analysis, a sufficient condition

is checked that allows to conclude if n is a cutoff. If n is not a cutoff, then the maximal number of threads to be simulated is increased and the reachability analysis is re-run. This algorithm has been tested and implemented in BOOM (see Chapter 10) by the authors of that paper.

Outlook As the experiments in Section 4.5 show, a SAT/QBF-based approach to solve the sequential reachability analysis problem works as well as a BDD-based algorithm but not much better or worse. A promising method that gained a lot of attention in the last few years is interpolation. An interesting problem to solve is to figure out, how SAT-based interpolation [108] could be integrated into the summarization algorithm for sequential Boolean programs.

This thesis has dealt with decidable problems only. The next logical step is to look at concurrent recursive Boolean programs because they are already undecidable. Since summarization has been applied successfully to sequential Boolean programs, one idea worth trying, is to apply it to concurrent recursive Boolean programs, as done in the model checker ZING [117].

An other idea is to bound the stack height and over-approximate function calls, whenever this height has been reached. An iterative algorithm could refine the approximation by increasing this limit, if a spurious counterexample has been found.

An aspect that has been ignored so far, as it goes beyond the scope of this thesis, is the automatic generation of Boolean programs. For example there is currently no support for thread-local variables, neither from BOOM nor from SATABS. Interestingly, Boolean programs with references are still decidable [116]. Maybe such an improved abstraction could be used to efficiently check programs that perform a lot of pointer manipulations. Furthermore, enriching Boolean programs with support for integers and using SMT-solvers or multi-terminal BDDs in the back-end would be an interesting approach to try.

Appendices



Syntax of Boolean Programs and Boolean Program Examples

A.1 Syntax of Concurrent Boolean Programs

Syntax	Description
$prog ::= decl^* proc^*$	A program is a list of global variable declarations followed by a list of procedure definitions
$decl ::= \mathbf{decl} id^+ ;$	Declaration of variables
$id ::= [a-zA-Z_][a-zA-Z0-9_]^*$	An identifier is a regular C-style identifier
$proc ::= type id(id^*) \mathbf{begin} enforce sseq \mathbf{end}$	Procedure definition
$type ::= \mathbf{void}$ \mathbf{bool} $\mathbf{bool} <id^+ >$	Procedures can return an arbitrary number of values
$enforce ::= \mathbf{enforce} (expr) ;$ 	Restriction on valuations of variables
$sseq ::= lstmt^+$	Sequence of statements
$lstmt ::= stmt$ $id:^+ stmt$	Labeled statement
$stmt ::= \mathbf{skip} ;$ $\mathbf{goto} id^+ ;$ $\mathbf{return} id^* ;$ $id^+ := expr^+ \mathbf{constrain} expr ;$ $\mathbf{if} (expr) \mathbf{then} sseq \mathbf{else} sseq \mathbf{fi}$ $\mathbf{assert} (expr) ;$ $\mathbf{assume} (expr) ;$ $id^* := id (expr) ;$ $\mathbf{start_thread} id ;$ $\mathbf{end_thread} ;$ $\mathbf{atomic.begin} ;$ $\mathbf{atomic.end} ;$	Nondeterministic goto Parallel assignment Conditional statement Assert statement Assume statement Procedure call Thread creation Thread termination Beginning of atomic section Ending of atomic section
$expr ::= expr \mathit{binop} expr$ $! expr$ $(expr)$ $const$ id $*$ $\mathbf{schoose} [expr, expr] ;$	Negation Variable Non-deterministic choice Non-deterministic choice
$binop ::= ' ' \& ' ' \wedge ' ' = ' ' ! = ' ' \implies ' ;$	Logical connectives
$const ::= \mathbf{0} / \mathbf{1}$	False / True

A.2 Original Bluetooth Driver Example

Listing A.1: I/O Procedures

```

// Shared Variables
decl stopped, driverStoppingFlag, stoppingEvent;
decl p0,p1,p2,p3,p4;           //int pendingIo

void IoDec()
begin
    decl q0,q1,q2,q3,q4;      // int Pio;

    // decrement pendingIo
    p0,p1,p2,p3,p4,q0,q1,q2,q3,q4 :=
        !p0 & ( p1 | p2 | p3 | p4),
        (p1&p0)|(!p1 & !p0 & (p2 | p3 | p4)),
        ((p0|p1)&p2)|(!p2 & !p1 & !p0 & (p3 | p4)),
        (!p0&!p1 &!p2 &!p3 & p4)|((p0|p1|p2)&p3),
        ((p0|p1|p2|p3)&p4),
        !p0 & ( p1 | p2 | p3 | p4),
        (p1&p0)|(!p1 & !p0 & (p2 | p3 | p4)),
        ((p0|p1)&p2)|(!p2 & !p1 & !p0 & (p3 | p4)),
        (!p0&!p1 &!p2 &!p3 & p4)|((p0|p1|p2)&p3),
        ((p0|p1|p2|p3)&p4);

    // test pendingIo for zero
    if (!q4 & !q3 & !q2 & !q1 & !q0) then
        stoppingEvent := 1;
    fi
    return;
end

bool IoInc()
begin
    decl status;

    // increment pendingIo
    p0,p1,p2,p3,p4 :=
        !p0 | ( p0 & p1 & p2 & p3 & p4),
        (p1&p0)|(!p1&p0)|( p0 & p1 & p2 & p3 & p4),
        (p2&!p0|!p1)|(!p2&p1&p0)|( p0 & p1 & p2 & p3 & p4),
        (p3&!p0|!p1|!p2)|(!p3&p2&p1&p0)|( p0 & p1 & p2 & p3 & p4),
        p4 |(!p4&p3&p2&p1&p0);

    if (driverStoppingFlag) then
        IoDec();
        status :=0;
    else
        status :=1;
    fi
    return status;
end
// example continues in next listing ...

```

Listing A.2: Instantiation of 2 adders and 2 stoppers

```

void thread0()
begin
    decl status;
    status := IoInc();
    if (status) then
        skip;
        if (stopped) then
            SLIC.ERROR: skip; //error
        fi
    fi
    IoDec();
    assume(F);
end

void thread1()
begin
    decl status;
    status := IoInc();
    if (status) then
        skip;
        if (stopped) then
            SLIC.ERROR: skip; //error
        fi
    fi
    IoDec();
    assume(F);
end

void thread2()
begin
    driverStoppingFlag := 1;
    IoDec();
    while (!stoppingEvent)
    do
        skip;
    od
    OK: stopped := 1;
    assume(F);
end

void thread3()
begin
    driverStoppingFlag := 1;
    IoDec();
    while (!stoppingEvent)
    do
        skip;
    od
    OK: stopped := 1;
    assume(F);
end
// example continues in next listing ...

```

Listing A.3: Initialization code

```

// initial conditions
void init()
begin
    stopped, driverStoppingFlag, stoppingEvent, p4, p3, p2, p1, p0 := 0,0,0,0,0,0,1;
    return;
end

```

A.3 Converted Bluetooth Driver Example

Listing A.4: Initialization of Bluetooth driver example

```
// Shared Variables

decl stopped, driverStoppingFlag, stoppingEvent;
decl p0,p1,p2,p3,p4;           // int pendingIo
decl limit;
decl g;

void main() begin
  decl status;

  assume ( !g );

  atomic_begin;
  if ( !g ) then
    g:= T;
    // initial conditions
    stopped, driverStoppingFlag, stoppingEvent, p4, p3, p2, p1, p0 := 0,0,0,0,0,0,0,1;
  fi;
  atomic_end;

  if(*) then goto adder_process; else goto stopper_process; fi;
  // example continues in next listing ...
```

Listing A.5: Stopper process of Bluetooth driver example

```
// STOPPER process — any number of threads with this process
stopper_process:
  driverStoppingFlag := 1;

// IoDec() begin
p0,p1,p2,p3,p4, limit:=
  !p0 & ( p1 | p2 | p3 | p4 ),
  (p1&p0)|(!p1 & !p0 & (p2 | p3 | p4 )),
  ((p0|p1)&p2)|(!p2 & !p1 & !p0 & (p3 | p4)),
  ((p0|p1|p2|p3)&p4) | (!p3 & !p2 & !p1 & !p0 & (p4)),
  ((p0|p1|p2|p3)&p4),
  p0 & !p1 & !p2 & !p3 & !p4;

// test pendingIo for zero
if ( limit ) then
  stoppingEvent := 1;
fi
// IoDec() end

while (!stoppingEvent)
  do skip; od
OK: stopped := 1;
thread_end;
// example continues in next listing ...
```

Listing A.6: Adder process of Bluetooth driver example

```

// ADDER process (just one thread with this process)
adder_process:

// status:= IoInc() begin
p0,p1,p2,p3,p4, limit := (!p0) | ( p0 & p1 & p2 & p3 & p4 ),
(p1 &!p0) | (!p1&p0) | ( p0 & p1 & p2 & p3 & p4 ),
(p2 &(!p0|!p1)) | (!p2 & p1 & p0) | ( p0 & p1 & p2 & p3 & p4 ),
(p3 &( !p2|!p0|!p1)) | (!p3 & p2 & p1 & p0) | ( p0 & p1 & p2 & p3 & p4 ),
p4 | (!p4 & p3 & p2 & p1 & p0),
p0 & p1 & p2 & p3 & p4;

// status:= IoInc() end
assume(!limit);

limit := choose[F,F];

if (driverStoppingFlag) then

// IoDec() begin
p0,p1,p2,p3,p4, limit:=
!p0 & ( p1 | p2 | p3 | p4 ),
(p1&p0)|(!p1 & !p0 & (p2 | p3 | p4 )),
((p0|p1)&p2)|(!p2 & !p1 & !p0 & (p3 | p4)),
((p0|p1|p2|p3)&p4) | (!p3 & !p2 & !p1 & !p0 & (p4)),
((p0|p1|p2|p3)&p4),
!p0 & !p1 & !p2 & !p3 & !p4;

// test pendingIo for zero
if ( limit ) then
stoppingEvent := 1;
fi

status:=0;
else
status:=1;
fi

// IoDec() end

if (status) then
if (stopped) then
SLIC_ERROR: skip; //error
fi

// IoDec() begin
p0,p1,p2,p3,p4, limit:=
!p0 & ( p1 | p2 | p3 | p4 ),
(p1&p0)|(!p1 & !p0 & (p2 | p3 | p4 )),
((p0|p1)&p2)|(!p2 & !p1 & !p0 & (p3 | p4)),
((p0|p1|p2|p3)&p4) | (!p3 & !p2 & !p1 & !p0 & (p4)),
((p0|p1|p2|p3)&p4),
!p0 & !p1 & !p2 & !p3 & !p4;

// test pendingIo for zero
if ( limit ) then
stoppingEvent := 1;
fi

// IoDec() end
fi

end;

```

B

Experimental Setup for Concurrent Benchmarks

This section describes the setup that has been used for benchmarking the algorithms described in Chapter 6, 7, and 8.

We applied BOOM to 444 examples from two sources: a set of 208 Boolean programs generated by SATABS that abstract part of the Linux kernel components, and a set of 236 Boolean programs generated at Microsoft Research using SLAM.

The remainder of this section describes 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 [137], a harness for Linux device drivers. The resulting concurrent model supports synchronization primitives (such as semaphores and spinlocks) and memory-mapped IO-registers for communication with the underlying hardware. Figure B.1 illustrates how parallel execution is handled. An environment thread models the operating system threads and parallelism caused by hardware events, e.g., interrupts. These functions are non-preemptive since interrupt service routines cannot be switched out during execution by the operating system kernel. 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 access to a driver can be shared among clients.

The Boolean program that results from the abstraction process exhibits the same structure as the original code. In order to simplify our experiments, we create only one client thread in the original program and instantiate (dynamically) up to N copies of this thread in the model checker. There is no message-passing communication between the threads, but they access the same shared data structures within the driver's code. The benchmarks feature, on average, 123 program locations, 21 thread-local

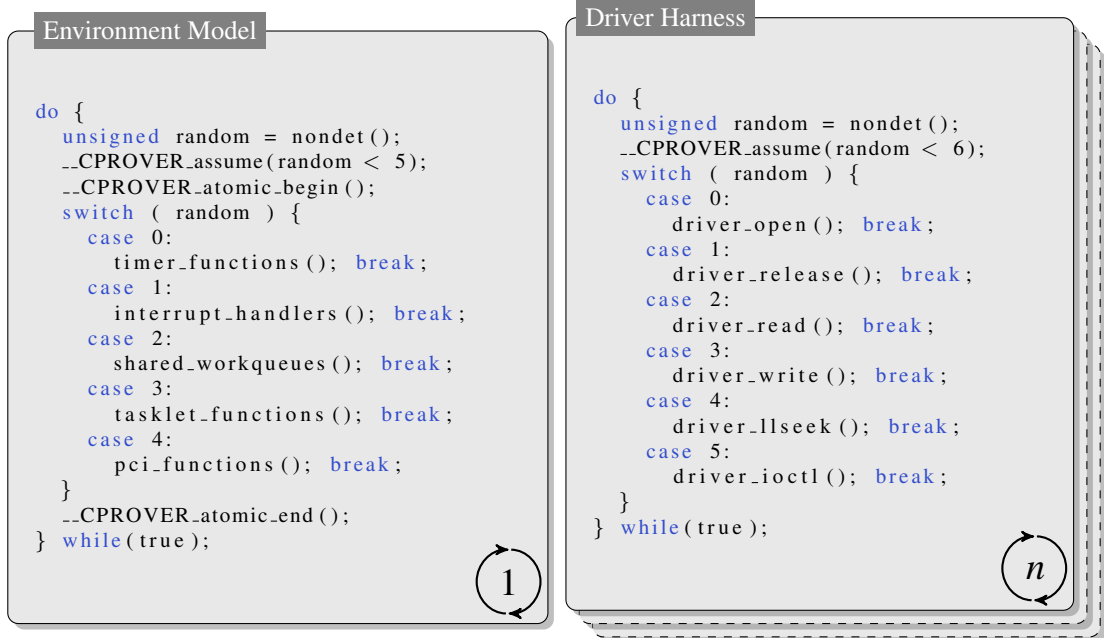


Figure B.1: Concurrent DDVERIFY execution model ('ddverify --model con2')

variables, and 12 shared variables.

The experimental setup is as follows. For each tool and benchmark, we run full reachability analysis with $n_0 = 1$ initial threads and a bound of $N = 2$ on thread creation. We then increase the bound N until the tool times out. The timeout is set to 720 s and the memory limit to 12 GB. The experiments are performed on a 3 GHz Intel Xeon machine running the 64-bit variant of Linux 2.6. We use the value 1000s to indicate timeouts.



Interprocedural Data-Flow Analysis

We implemented in BOOM (see Section 10.9) an *interprocedural* live variable analysis as described by [Srivastava and Wall](#). The data-flow equations and the explanations are reprinted below.

Their algorithm proceeds in two phases:

Phase 1 Figures out which variables are modified and referenced in each function (**mod/ref** analysis). Note that “live” information is an over-approximation and can only be used to compute the **ref** sets for each function. In order to compute the **mod** sets, an under-approximation (“dead”) is needed because, e. g., a variable could be modified only along one path to the function’s exit.

normal blocks:

$$\begin{aligned} \text{DEF}(B) &= \text{variables defined by } B \text{ before any use in } B \\ \text{USE}(B) &= \text{variables used in } B \text{ before any definition in } B \\ \text{LIVE}_{in}(B) &= \text{USE}(B) \cup \text{LIVE}_{out}(B) - \text{DEF}(B) \\ \text{LIVE}_{out}(B) &= \bigcup_{S \in \text{Succ}(B)} \text{LIVE}_{in}(S) \\ \text{DEAD}_{in}(B) &= \text{DEF}(B) \cup \text{DEAD}_{out}(B) - \text{USE}(B) \\ \text{DEAD}_{out}(B) &= \bigcap_{S \in \text{Succ}(B)} \text{DEAD}_{in}(S) \end{aligned}$$

call and return blocks:

$$\begin{aligned} \text{LIVE}_{out}(\text{call}) &= \text{LIVE}_{in}(\text{entry}) \cup \text{LIVE}_{out}(\text{return}) - \text{DEAD}_{in}(\text{entry}) \\ \text{LIVE}_{out}(\text{return}) &= \bigcup_{S \in \text{Succ}(\text{return})} \text{LIVE}_{in}(S) \\ \text{DEAD}_{out}(\text{call}) &= \text{DEAD}_{in}(\text{entry}) \cup \text{DEAD}_{out}(\text{return}) - \text{LIVE}_{in}(\text{entry}) \\ \text{DEAD}_{out}(\text{return}) &= \bigcap_{S \in \text{Succ}(\text{return})} \text{DEAD}_{in}(S) \end{aligned}$$

IN sets for a call or return block are identical to its OUT sets

When this converges, define, for each procedure P:

$$\begin{aligned}\text{REF}(P) &= \text{LIVE}_{in}(B) \text{ where } B \text{ is the entry to } P \\ \text{MOD}(P) &= \text{DEAD}_{in}(B) \text{ where } B \text{ is the entry to } P\end{aligned}$$

and then throw away the LIVE and DEAD sets computed in this phase

Phase 2 Corresponds to a **local live variable** analysis but uses the mod/ref sets that were computed in phase 1.

normal blocks:

$$\begin{aligned}\text{DEF}(B) &= \text{variables defined by } B \text{ before any use in } B \\ \text{USE}(B) &= \text{variables used in } B \text{ before any definition in } B \\ \text{LIVE}_{in}(B) &= \text{USE}(B) \cup \text{LIVE}_{out}(B) - \text{DEF}(B) \\ \text{LIVE}_{out}(B) &= \bigcup_{S \in \text{Succ}(B)} \text{LIVE}_{in}(S)\end{aligned}$$

call and return blocks, for call to procedure P:

$$\begin{aligned}\text{LIVE}_{out}(\text{call}) &= \text{REF}(P) \cup \text{LIVE}_{out}(\text{return}) - \text{MOD}(P) \\ \text{LIVE}_{out}(\text{return}) &= \bigcup_{S \in \text{Succ}(\text{return})} \text{LIVE}_{in}(S)\end{aligned}$$

IN sets for a call or return block are identical to its OUT sets

Bibliography

- [1] Alur, R., M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **27**:786–818.
- [2] Alur, R., R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. 2001. Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design (FMSD)* **18**:97–116.
- [3] Andrews, T., S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. 2004. Zing: A Model Checker for Concurrent Software. In: Alur, R., and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*. Springer, 484–487.
- [4] Apt, K., and D. Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters (IPL)*.
- [5] Ayewah, N., W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. 2007. Using FindBugs on production software. In: Gabriel, R. P., D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA Companion*. ACM, 805–806.
- [6] Baier, C., and J.-P. Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.
- [7] Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. 2006. Thorough static analysis of device drivers. In: Berbers, Y., and W. Zwaenepoel, editors, *EuroSys*. ACM, 73–85.
- [8] Ball, T., S. Chaki, and S. Rajamani. 2001. Parameterized Verification of Multithreaded Software Libraries. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. London, UK: Springer-Verlag, 158–173.
- [9] Ball, T., B. Cook, S. Das, and S. K. Rajamani. 2004. Refining approximations in software predicate abstraction. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 388–403.
- [10] Ball, T., B. Cook, V. Levin, and S. K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: *Integrated Formal Verification (IFM)*, volume 2999. Springer, 1–20.
- [11] Ball, T., R. Majumdar, T. Millstein, and S. K. Rajamani. 2001. Automatic predicate abstraction of C programs. In: *Programming Language Design and Implementation (PLDI)*. ACM, 203–213.
- [12] Ball, T., A. Podelski, and S. K. Rajamani. 2001. Boolean and Cartesian abstractions for model checking C programs. In: [105], 268–283.

- [13] Ball, T., and S. K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In: Model Checking and Software Verification (SPIN), volume 1885 of *LNCS*. Springer, 113–130.
- [14] ———. 2000. Boolean Programs: A Model and Process for Software Analysis. Technical Report 2000-14, Microsoft Research.
- [15] ———. 2002. The SLAM project: debugging system software via static analysis. In: Principles of Programming Languages (POPL). 1–3.
- [16] Barner, S., and O. Grumberg. 2005. Combining Symmetry Reduction and Under-Approximation for Symbolic Model Checking. *Formal Methods in System Design (FMSD)*.
- [17] Barner, S., and I. Rabinovitz. 2003. Efficient Symbolic Model Checking of Software Using Partial Disjunctive Partitioning. In: CHARME. 35–50.
- [18] Basler, G., M. Hague, D. Kroening, C.-H. L. Ong, T. Wahl, and H. Zhao. 2010. Boom: Taking Boolean Program Model Checking One Step Further. In: Esparza, J., and R. Majumdar, editors, TACAS, volume 6015 of *Lecture Notes in Computer Science*. Springer, 145–149.
- [19] Basler, G., D. Kroening, and G. Weissenbacher. 2007. A Complete Bounded Model Checking Algorithm for Pushdown Systems. In: Haifa Verification Conference. 202–217.
- [20] ———. 2007. SAT-based Summarisation for Boolean Programs. In: Model Checking and Software Verification (SPIN), volume 4595 of *LNCS*. Springer, 131–148.
- [21] Basler, G., M. Mazzucchi, T. Wahl, and D. Kroening. 2009. Symbolic Counter Abstraction for Concurrent Software. In: [31], 64–78.
- [22] Benedetti, M. 2005. sKizzo: a Suite to Evaluate and Certify QBFs. In: Nieuwenhuis, R., editor, CADE, volume 3632 of *Lecture Notes in Computer Science*. Springer, 369–376.
- [23] Beyer, D., T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The software model checker Blast. *STTT* 9:505–525.
- [24] Biere, A. 2004. Resolve and Expand. In: Hoos, H. H., and D. G. Mitchell, editors, SAT (Selected Papers, volume 3542 of *Lecture Notes in Computer Science*. Springer, 59–70.
- [25] Biere, A., A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. 2003. Bounded model checking. *Advances in Computers* 58:118–149.
- [26] Biere, A., A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 1579 of *LNCS*. 193–207.
- [27] Bingham, J. 2005. A New Approach to Upward-Closed Set Backward Reachability Analysis. *Electronic Notes in Theoretical Computer Science*.
- [28] Bollig, B., and I. Wegener. 1996. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Computers* 45:993–1002.
- [29] Bouajjani, A., and J. Esparza. 2006. Rewriting Models of Boolean Programs. In: Pfenning, F., editor, RTA, volume 4098 of *Lecture Notes in Computer Science*. Springer, 136–150.

- [30] Bouajjani, A., J. Esparza, and T. Touili. 2003. A generic approach to the static analysis of concurrent programs with procedures. In: *Principles of Programming Languages (POPL)*. ACM, 62–73.
- [31] Bouajjani, A., and O. Maler, editors. 2009. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009*. Proceedings, volume 5643 of *Lecture Notes in Computer Science*. Springer.
- [32] Bouajjani, A., M. Müller-Olm, and T. Touili. 2005. Regular symbolic analysis of dynamic networks of pushdown systems. *CONCUR* .
- [33] Bouajjani, A., and T. Touili. 2005. On Computing Reachability Sets of Process Rewrite Systems. In: *Term Rewriting and Applications (RTA)*, volume 3467 of *LNCS*. Springer, 484–499.
- [34] Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* **35**:677–691.
- [35] Büchi, J. R. 1964. Regular canonical systems. *Archive for Mathematical Logic* **6**:91.
- [36] Christensen, S., L. M. Kristensen, and T. Mailund. 2001. A Sweep-Line Method for State Space Exploration. In: [105], 450–464.
- [37] Ciardo, G., G. Lüttgen, and R. Siminiceanu. 2000. Efficient Symbolic State-Space Construction for Asynchronous Systems. In: *ICATPN*. 103–122.
- [38] Claessen, K., N. Eén, M. Sheeran, N. Sörensson, A. Voronov, and K. Åkesson. 2009. SAT-Solving in Practice, with a Tutorial Example from Supervisory Control. *Discrete Event Dynamic Systems* **19**:495–524.
- [39] Clarke, E., R. Enders, T. Filkorn, and S. Jha. 1996. Exploiting Symmetry In Temporal Logic Model Checking. *Formal Methods in System Design (FMSD)* .
- [40] Clarke, E., O. Grumberg, and D. Peled. 1999. *Model Checking*. MIT Press.
- [41] Clarke, E. M., O. Grumberg, and M. C. Browne. 1986. Reasoning About Networks With Many Identical Finite-State Processes. In: *Principles of Distributed Computing (PODC)*. 240–248.
- [42] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-Guided Abstraction Refinement. In: Emerson, E. A., and A. P. Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 154–169.
- [43] Clarke, E. M., O. Grumberg, and D. E. Long. 1992. Model checking and abstraction. In: *POPL*. ACM Press, 343–354.
- [44] Clarke, E. M., D. Kroening, N. Sharygina, and K. Yorav. 2005. SATABS: SAT-based Predicate Abstraction for ANSI-C. In: [83], 570–574.
- [45] Clarke, E. M., and H. Veith. 2003. Counterexamples Revisited: Principles, Algorithms, Applications. In: *Verification: Theory and Practice*. 208–224.
- [46] Cook, B., D. Kroening, and N. Sharygina. 2005. Symbolic model checking for asynchronous Boolean programs. In: *Model Checking and Software Verification (SPIN)*, volume 3639 of *LNCS*. Springer, 75–90.

- [47] ———. 2006. Over-Approximating Boolean Programs with Unbounded Thread Creation. In: *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 53–59.
- [48] ———. 2007. Verification of Boolean Programs with Unbounded Thread Creation. *Theoretical Computer Science (TCS)*.
- [49] Cooper, K., T. Harvey, and K. Kennedy. 2001. A Simple, Fast Dominance Algorithm.
- [50] Couvreur, J.-M., E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. 2002. Data Decision Diagrams for Petri Net Analysis. In: Esparza, J., and C. Lakos, editors, *ICATPN*, volume 2360 of *Lecture Notes in Computer Science*. Springer, 101–120.
- [51] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* **13**:451–490.
- [52] Das, S., and D. L. Dill. 2001. Successive Approximation of Abstract Transition Relations. In: *Logic in Computer Science (LICS)*. 51–60.
- [53] Davis, M., G. Logemann, and D. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* **5**:394–397.
- [54] Davis, M., and H. Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* **7**:201–215.
- [55] Delzanno, G., J.-F. Raskin, and L. V. Begin. 2002. Towards the Automated Verification of Multithreaded Java Programs. In: Katoen, J.-P., and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 173–187.
- [56] ———. 2004. Covering sharing trees: a compact data structure for parameterized verification. *STTT*.
- [57] Delzanno, G., J.-F. Raskin, and L. Van Begin. 2002. The BABYLON Project. <http://www.ulb.ac.be/di/ssd/lvbegin/CST/index.html>.
- [58] Donaldson, A. F., and A. Miller. 2006. Symmetry Reduction for Probabilistic Model Checking Using Generic Representatives. In: Graf, S., and W. Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*. Springer, 9–23.
- [59] Eén, N., and N. Sörensson. 2004. An Extensible SAT-solver. In: *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*. Springer, 502–518.
- [60] Emerson, A., and P. Sistla. 1996. Symmetry and Model Checking. *Formal Methods in System Design (FMSD)*.
- [61] Emerson, A., and T. Wahl. 2004. Efficient Reduction Techniques for Systems with Many Components. In: *Brazilian Symposium on Formal Methods (SBMF)*.
- [62] Emerson, E. A. 1990. *Temporal and modal logic*. Cambridge, MA, USA: MIT Press.
- [63] Emerson, E. A., J. Havlicek, and R. J. Trefler. 2000. Virtual Symmetry Reduction. In: *LICS*. 121–131.

- [64] Emerson, E. A., S. Jha, and D. Peled. 1997. Combining Partial Order and Symmetry Reductions. In: Brinksma, E., editor, TACAS, volume 1217 of *Lecture Notes in Computer Science*. Springer, 19–34.
- [65] Emerson, E. A., and V. Kahlon. 2000. Reducing Model Checking of the Many to the Few. In: McAllester, D. A., editor, CADE, volume 1831 of *Lecture Notes in Computer Science*. Springer, 236–254.
- [66] Emerson, E. A., and K. S. Namjoshi. 1995. Reasoning about Rings. In: POPL. 85–94.
- [67] Emerson, E. A., and R. J. Trefler. 1999. From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking. In: Pierre, L., and T. Kropf, editors, CHARME, volume 1703 of *Lecture Notes in Computer Science*. Springer, 142–156.
- [68] Emerson, E. A., and T. Wahl. 2005. Dynamic Symmetry Reduction. In: [83], 382–396.
- [69] Esparza, J., D. Hansel, P. Rossmanith, and S. Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In: Computer Aided Verification (CAV), volume 1855 of *LNCS*. Springer, 232–247.
- [70] Esparza, J., and K. Heljanko. 2008. Unfoldings – A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science. Springer.
- [71] Etessami, K., and S. K. Rajamani, editors. 2005. Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, volume 3576 of *Lecture Notes in Computer Science*. Springer.
- [72] Finkel, A. 1993. The Minimal Coverability Graph for Petri Nets. In: Papers from the 12th International Conference on Applications and Theory of Petri Nets. London, UK: Springer-Verlag, 210–243.
- [73] Finkel, A., B. Willems, and P. Wolper. 1997. A direct symbolic approach to model checking pushdown systems. In: Workshop on Verification of Infinite State Systems (INFINITY), volume 9 of *ENTCS*. 27–39.
- [74] Flanagan, C., and P. Godefroid. 2005. Dynamic partial-order reduction for model checking software. In: Palsberg, J., and M. Abadi, editors, POPL. ACM, 110–121.
- [75] Ganty, P., J.-F. Raskin, and L. V. Begin. 2008. From Many Places to Few: Automatic Abstraction Refinement for Petri Nets. *Fundam. Inf.* .
- [76] Geeraerts, G., J.-F. Raskin, and L. V. Begin. 2005. Expand, Enlarge and Check... Made Efficient. In: [71], 394–407.
- [77] ———. 2007. On the Efficient Computation of the Minimal Coverability Set for Petri Nets. In: Namjoshi, K. S., T. Yoneda, T. Higashino, and Y. Okamura, editors, ATVA, volume 4762 of *Lecture Notes in Computer Science*. Springer, 98–113.
- [78] German, S., and P. Sistla. 1992. Reasoning about Systems with Many Processes. *Journal of the ACM (JACM)* .
- [79] Godefroid, P. 1996. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, volume 1032 of *Lecture Notes in Computer Science*. Springer.

- [80] Graf, S., and H. Säidi. 1997. Construction of abstract state graphs with PVS. In: *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*. Springer, 72–83.
- [81] Grahlmann, B., and E. Best. 1996. PEP - More than a Petri Net Tool. In: *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 397–401.
- [82] Gueta, G., C. Flanagan, E. Yahav, and M. Sagiv. 2007. Cartesian Partial-Order Reduction. In: *Model Checking and Software Verification (SPIN)*. 95–112.
- [83] Halbwachs, N., and L. D. Zuck, editors. 2005. Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, volume 3440 of *Lecture Notes in Computer Science*. Springer.
- [84] Heitmann, F., and D. Moldt. 2005. Petri Net Tool Database. Available from <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>.
- [85] Henzinger, T. A., R. Jhala, and R. Majumdar. 2004. Race checking by context inference. In: [115], 1–13.
- [86] Henzinger, T. A., R. Jhala, R. Majumdar, and G. Sutre. 2002. Lazy abstraction. In: *POPL*. 58–70.
- [87] Henzinger, T. A., R. Majumdar, and J.-F. Raskin. 2005. A classification of symbolic transition systems. *ACM Trans. Comput. Log.* .
- [88] Holzmann, G. J., and D. Peled. 1994. An improvement in formal verification. In: Hogrefe, D., and S. Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*. Chapman & Hall, 197–211.
- [89] IBM Corp., Armonk, New York, USA. 2004. Eclipse IDE. Available from the Eclipse Foundation <http://www.eclipse.org>.
- [90] Jr., W. A. H., and F. Somenzi, editors. 2003. Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, volume 2725 of *Lecture Notes in Computer Science*. Springer.
- [91] Jussila, T., and A. Biere. 2007. Compressing BMC Encodings with QBF. *Electr. Notes Theor. Comput. Sci.* **174**:45–56.
- [92] Kahlon, V., F. Ivancic, and A. Gupta. 2005. Reasoning About Threads Communicating via Locks. In: [71], 505–518.
- [93] Kahlon, V., C. Wang, and A. Gupta. 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In: *CAV*. 398–413.
- [94] Kaiser, A., D. Kroening, and T. Wahl. 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. In: Touili, T., B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*. Springer, 645–659.
- [95] Karp, R., and R. Miller. 1969. Parallel Program Schemata. *Computer and System Sciences* .
- [96] Kroening, D. 2006. Computing Over-Approximations with Bounded Model Checking. In: *BMC workshop*, volume 144. 79–92.

- [97] Kroening, D., and O. Strichman. 2003. Efficient Computation of Recurrence Diameters. In: Verification, Model Checking and Abstract Interpretation (VMCAI), volume 2575 of *LNCS*. 298–309.
- [98] Kurshan, R. 1995. Computer-Aided Verification of Coordinating Processes. Princeton University Press.
- [99] Kurshan, R. P., V. Levin, M. Minea, D. Peled, and H. Yenigün. 2002. Combining Software and Hardware Verification Techniques. *Formal Methods in System Design* **21**:251–280.
- [100] Lahiri, S. K., R. E. Bryant, and B. Cook. 2003. A Symbolic Approach to Predicate Abstraction. In: [90], 141–153.
- [101] Lal, A., and T. W. Reps. 2006. Improving Pushdown System Model Checking. In: Ball, T., and R. B. Jones, editors, CAV, volume 4144 of *Lecture Notes in Computer Science*. Springer, 343–357.
- [102] ———. 2008. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: Gupta, A., and S. Malik, editors, CAV, volume 5123 of *Lecture Notes in Computer Science*. Springer, 37–51.
- [103] Leino, K. R. M. 2003. A SAT Characterization of Boolean-Program Correctness. In: Model Checking and Software Verification (SPIN), volume 2648 of *LNCS*. 104–120.
- [104] Lubachevsky, B. 1984. An Approach to Automating the Verification of Compact Parallel Coordination Programs. *Acta Informatica* .
- [105] Margaria, T., and W. Yi, editors. 2001. Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings, volume 2031 of *Lecture Notes in Computer Science*. Springer.
- [106] McMillan, K. L. 1993. The SMV Model Checker. University of California at Berkeley, <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv>.
- [107] ———. 1993. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic.
- [108] ———. 2003. Interpolation and SAT-Based Model Checking. In: [90], 1–13.
- [109] Melton, R., and D. Dill. 1996. Mur ϕ Annotated Reference Manual, rel. 3.1. <http://verify.stanford.edu/dill/murphi.html>.
- [110] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In: Design Automation Conference (DAC). 530–535.
- [111] Muchnick, S. 1997. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers.
- [112] Pastor, E., and J. Cortadella. 1998. Efficient Encoding Schemes for Symbolic Analysis of Petri Nets. In: DATE. IEEE Computer Society, 790–795.
- [113] Peled, D. 1993. All from One, One for All: on Model Checking Using Representatives. In: Computer Aided Verification (CAV). 409–423.

- [114] Pnueli, A., J. Xu, and L. D. Zuck. 2002. Liveness with $(0, 1, \infty)$ -Counter Abstraction. In: Brinksma, E., and K. G. Larsen, editors, CAV, volume 2404 of *Lecture Notes in Computer Science*. Springer, 107–122.
- [115] Pugh, W., and C. Chambers, editors. 2004. Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004. ACM.
- [116] Qadeer, S., and S. K. Rajamani. 2005. Deciding assertions in programs with references. Technical report.
- [117] Qadeer, S., S. K. Rajamani, and J. Rehof. 2004. Summarizing procedures in concurrent programs. In: Jones, N. D., and X. Leroy, editors, POPL. ACM, 245–255.
- [118] Qadeer, S., and J. Rehof. 2005. Context-bounded Model Checking of Concurrent Software. In: TACAS, volume 3440 of *LNCS*. 93–107.
- [119] Qadeer, S., and D. Wu. 2004. KISS: keep it simple and sequential. In: [115], 14–24.
- [120] Rackoff, C. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science (TCS)* .
- [121] Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS* **22**:416–430.
- [122] Schwoon, S. 2002. Model-Checking Pushdown Systems. Ph.D. thesis, Technische Universität München.
- [123] Sharir, M., and A. Pnueli. 1981. Program Flow Analysis: Theory and Applications, chapter Two approaches to interprocedural dataflow analysis. Prentice-Hall, 189–233.
- [124] Sheeran, M., S. Singh, and G. Stalmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design. London, UK: Springer-Verlag, 108–125.
- [125] Sighireanu, M., and T. Touili. 2009. Bounded Communication Reachability Analysis of Process Rewrite Systems with Ordered Parallelism. *Electr. Notes Theor. Comput. Sci.* **239**:43–56.
- [126] Somenzi, F. 2009. The CU Decision Diagram Package, release 2.4.2. University of Colorado at Boulder, <http://vlsi.colorado.edu/~fabio/CUDD/>.
- [127] Srivastava, A., and D. Wall. 1992. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages* **1**:1–18.
- [128] Starke, P. H. 1991. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.* .
- [129] Suwimonteerabuth, D., J. Esparza, and S. Schwoon. 2008. Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In: Havelund, K., R. Majumdar, and J. Palsberg, editors, SPIN, volume 5156 of *Lecture Notes in Computer Science*. Springer, 270–287.
- [130] Torre, S. L., P. Madhusudan, and G. Parlato. 2007. A Robust Class of Context-Sensitive Languages. In: LICS. 161–170.

- [131] ———. 2009. Analyzing recursive programs using a fixed-point calculus. In: Hind, M., and A. Diwan, editors, PLDI. ACM, 211–222.
- [132] ———. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: [31], 477–492.
- [133] Tseitin, G. S. 1968. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic* **Part II**:115–125.
- [134] Wahl, T., N. Blanc, and E. A. Emerson. 2008. Sviss: Symbolic Verification of Symmetric Systems. In: Ramakrishnan, C. R., and J. Rehof, editors, TACAS, volume 4963 of *Lecture Notes in Computer Science*. Springer, 459–462.
- [135] Wegener, I. 2000. *Branching Programs and Binary Decision Diagrams*. SIAM.
- [136] Wei, O., A. Gurfinkel, and M. Chechik. 2005. Identification and Counter Abstraction for Full Virtual Symmetry. In: Borriane, D., and W. J. Paul, editors, CHARME, volume 3725 of *Lecture Notes in Computer Science*. Springer, 285–300.
- [137] Witkowski, T., N. Blanc, D. Kroening, and G. Weissenbacher. 2007. Model checking concurrent linux device drivers. In: Stirewalt, R. E. K., A. Egyed, and B. Fischer, editors, ASE. ACM, 501–504.
- [138] Xie, Y., and A. Aiken. 2007. Saturn: A scalable framework for error detection using Boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **29**.

Index

- *, 20
- ample set, 52
- assert**, 19, 20, 54
- assume**, 19, 20, 48, 54, 68
- atomic section, 46, 54
- atomic.begin**, 46
- atomic.end**, 46

- BDD, *see* binary decision diagram
- BEBOP, 39
- binary decision diagram, 6, 19, 23, 29, 39, 62, 64, 88, 97–99, 101, 108–110, 115
- bit-blast, 100
- bitset, 106
- BMC, *see* bounded model checking
- Boolean program, 11, 12, 17, 18, 20, 22, 24, 25, 27, 28, 53, 65–67, 80, 86, 97, 106, 110, 115
 - concurrent \sim , 45, 63, 65, 79
- BOOM, 72, 97, 109, 127

- call**, 104
- Cartesian abstraction, 68
- Cartesian product, 66
- CNF, *see* conjunctive normal form
- completeness threshold, 9
- configuration
 - covered \sim , 81
 - reachable \sim , 81
- conjunctive normal form, 7, 100
- constrain**, 68
- control location, 22–24
- control-flow graph, 12, 17–20, 22, 24, 53, 104, 108
- counter abstraction, 61, 62, 64, 67, 80
 - symbolic \sim , 65
- counterexample, 5, 110

- dead**, 19, 109

- Eclipse, 110
- edge
 - path \sim , 30
 - summary \sim , 29, 102
- end.thread**, 45, 48
- enforce**, 19

- fix-point, 20, 28, 98, 101
- formula
 - symbolic \sim , 20

- goto**, 19–21, 48, 101
- guard, 20, 21, 100

- head, 22, 23, 28, 37

- if-then-else**, 20
- image function, 6, 67
- image operation, *see* image function

- k-Induction, 10
- Karp-Miller, 79, 81, 83

- live variable analysis, 98, 109
 - interprocedural \sim , 109, 127
- lock, 46

- model checker, 5, 11, 72, 108, 109, 125
- model checking, 5, 6, 11, 18, 62
 - bounded \sim , 9, 10, 27
 - symbolic \sim , 7
- MUR ϕ , 72

- non-determinism, 17, 20, 46, 65, 72, 100
- non-deterministic, 19, 106
- non-recursive, 79

- over-approximation, 67

- partial-order reduction, 51
 - Cartesian \sim , 57
- path merging, 33
- path-wise, 102

- persistent set, 55
- POR, *see* partial-order reduction
- Predicate abstraction, 11, 17, 61
- program counter, 24, 28, 98
 - explicit \sim , 98, 99
- pushdown system, 22, 25
 - symbolic \sim , 23, 28
- QBF, *see* quantified Boolean formula
- quantified Boolean formula, 9, 27–29, 39, 97, 101, 108, 109
- reachability analysis, 6, 17
- reachability diameter, 10
 - initialized \sim , 10
- reachability recurrence diameter, 10
 - initialized \sim , 10
- replicated finite-state program, 79, 80
- replicated finite-state system, 80, 81
- return**, 19–21, 104
- return value, 25
- rule
 - contraction \sim , 22, 28
 - expansion \sim , 22
 - neutration \sim , 22
- safety property, 9, 27
- SAT, 7, 28, 39, 109
 - \sim solving, 7
- SATABS, 13, 19, 62, 97, 108
- schoose**, 18
- search
 - breadth-first \sim , 104
 - depth-first \sim , 104
- Shannon decomposition, 68
- skip**, 19–21, 48, 101
- SLAM, 17, 19, 62, 109
- stack, 5, 17, 18, 22, 28
 - alphabet \sim , 24
- start.thread**, 45, 48
- state
 - Cartesian \sim representation, 67
 - explicit \sim , 18, 20, 85, 86
 - global \sim , 68, 80
 - hybrid \sim , 52, 98–102
 - local \sim , 49, 61, 62, 64, 66, 80
 - local \sim counter, 62
 - \sim merging, 103
 - reachable states, 6
 - reachable local states, 62
 - reachable thread \sim , 80, 83
 - shared \sim , 80
 - splice \sim , 68
 - \sim subsumption, 105
 - symbolic \sim , 20, 85
 - symbolic thread \sim , 68
 - thread \sim , 68, 80
 - thread \sim reachability, 49
- state space explosion
 - local \sim , 62, 65, 80
- statement
 - splice \sim , 68
- static single assignment form, 103
- summarization, 17, 18, 22, 27, 28
- summary, 20
 - non-deterministic \sim , 35
 - universal \sim , 31
- symmetry reduction, 62, 63
- thread
 - bounded number of threads, 62
 - bounded threads, 63
 - replicated threads, 63
 - replicated threads, 61
 - unbounded dynamic \sim creation, 80
- transition
 - invisible \sim , 52
 - symbolic \sim , 24
- transition system, 9
 - finite-state \sim , 6, 9, 17
- unsatisfiable, 68
- variable
 - dead \sim , 19, 98, 108
 - global \sim , 19, 24, 25, 100
 - live \sim , 108
 - local \sim , 19, 24, 80, 100
 - ordering \sim , 99
 - primed \sim , 48
 - shared \sim , 46, 66, 68, 100
 - splice \sim , 68
 - thread-local \sim , 66
- VASS, *see* vector addition system with states
- vector addition system, 79, 83
 - \sim with states, 80, 81
- while**, 20
- work-list
 - frontier \sim , 31, 104
 - history \sim , 105

Curriculum Vitæ

Personal

Gérard Charly Basler
October 28, 1978 Born in Menziken, Switzerland

Education

1994 – 1998 Alte Kantonsschule Aarau, Switzerland
1998 – 2003 Diploma in Electrical Engineering, ETH Zürich, Switzerland
2005 – 2010 Research assistant, Formal Verification Group, ETH Zürich

Professional Experience

2001 Intern at Ergonomics AG, Zurich, Switzerland
2002 Intern at ABB Turgi, Switzerland
2003 – 2005 Software Engineer at Cinerent OpenAir AG, Zollikon, Switzerland
2005 – Teaching assistant, Formal Verification Group, ETH Zürich
2007 Internship at Microsoft Research, Redmond, USA

International Experience

2002/2003 Diploma Thesis at the University of New Mexico, USA
2007 Internship at Microsoft Research, Redmond, USA
2008/2009 Visiting Scholar at Oxford University, Great Britain

Publications

- Gérard Basler, Daniel Kroening, Michele Mazzucchi and Thomas Wahl:
Context-Aware Counter Abstraction.
In *Journal of Formal Methods in System Design*, to appear
- Gérard Basler, Matthew Hague, Daniel Kroening, Luke Ong, Thomas Wahl and Haoxian Zhao
Boom: Taking Boolean Program Model Checking One Step Further.
In *TACAS*, 2010: 145-149
- Gérard Basler, Michele Mazzucchi, Thomas Wahl, Daniel Kroening:
Symbolic Counter Abstraction for Concurrent Software.
In *CAV*, 2009: 64-78
- Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Grard Basler, Piramanayagam Arumuga Nainar,
Iulian Neamtiu:
Finding and Reproducing Heisenbugs in Concurrent Programs.
In *OSDI*, 2008: 267-280
- Gérard Basler, Daniel Kroening, Georg Weissenbacher:
A Complete Bounded Model Checking Algorithm for Pushdown Systems.
In *Haifa Verification Conference*, 2007: 202-217
- Gérard Basler, Daniel Kroening, Georg Weissenbacher:
SAT-Based Summarization for Boolean Programs.
In *SPIN*, 2007: 131-148