Modular Verification of Shared-Memory Concurrent System Software

by

Zvonimir Rakamarić

Dipl.Ing., University of Zagreb, 2002
M.Sc., University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March, 2011

Abstract

Software is large, complex, and error-prone. According to the US National Institute of Standards and Technology, software bugs cost the US economy an estimated $60 billion each year. The trend in hardware design of switching to multi-core architectures makes software development even more complex. Cutting software development costs and ensuring higher reliability of software is of global interest and a grand challenge. This is especially true of the system software that is the foundation beneath all general-purpose application programs.

The verification of system software poses particular challenges: system software is typically written in a low-level programming language with dynamic memory allocation and pointer manipulation, and system software is also highly concurrent, with shared-memory communication being the main concurrent programming paradigm. Available verification tools usually perform poorly when dealing with the aforementioned challenges. This thesis addresses these problems by enabling precise and scalable verification of low-level, shared-memory, concurrent programs. The main contributions are about the interrelated concepts of memory, modularity, and concurrency.

First, because programs use huge amounts of memory, the memory is usually modeled very imprecisely in order to scale to big programs. This imprecise modeling renders most tools almost useless in the memory-intensive parts of code. This thesis describes a scalable, yet precise, memory model that offers on-demand precision only when necessary.

Second, modularity is the key to scalability, but it often comes with a price — a user must manually provide module specifications, making the verification process more tedious. This thesis proposes a light-weight technique for automatically inferring an important family of specifications to make the verification process more automatic.

Third, the number of program behaviors explodes in the presence of concurrency, thereby greatly increasing the complexity of the verification task. This explosion is especially true of shared-memory concurrency. The thesis presents a static context-bounded analysis that combines a number of techniques to successfully solve this problem.

We have implemented the above contributions in the verification tools developed as a part of this thesis. We have applied the tools on real-life system software, and we are already finding critical, previously undiscovered bugs.

Preface

I conducted the research presented in Chapter 3 and Chapter 4 of this thesis in collaboration with my Ph.D. advisor Alan Hu. This work was also originally published with him as a co-author:

- Zvonimir Rakamarić and Alan J. Hu. Automatic inference of frame axioms using static analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 89–98, 2008.

- Zvonimir Rakamarić and Alan J. Hu. A scalable memory model for low-level code. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.

Much of the original text of those publications transitioned into the thesis. As a result, some of the sentences and paragraphs of those two chapters were authored by Alan.

In Chapter 5, I describe my research on the verification of shared-memory concurrent programs. I did most of this work during my internships at Microsoft Research, in collaboration with researchers Shaz Qadeer and Shuvendu Lahiri. We published together the results of this research:

- Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *International Conference on Computer Aided Verification (CAV)*, pages 509–524, 2009.

Most of the text of Chapter 5 comes from that publication. Therefore, although majority of the work was done by me, there are sentences and paragraphs in that chapter that were authored by Shaz and Shuvendu.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I thank my parents for teaching me early that nothing is beyond reach of those who try hard, for encouraging me to aim big although that took me far away from my home town, as well as for their endless support that helped me overcome rough periods of my graduate studies.

I am incredibly grateful to my Ph.D. advisor Alan Hu for shaping me into an independent researcher. Alan always encouraged me to explore my own ideas, while making sure my research stays on track. Although perpetually busy, he would always find time for discussions about research and beyond. His ability to put a positive spin, which I sometimes missed, on everything often turned my hard Eastern European realism into optimism, and I thank him for that.

My internships at Microsoft Research were a truly rewarding experience, and I still foster long-lasting relationships with many researchers I met there. I was fortunate to be teamed up with researchers Shaz Qadeer and Shuvendu Lahiri, amazing guys because of whom I kept coming back for more.

I thank my supervisory committee members Ken McMillan and Mark Greenstreet for their genuine interest in my work and wise insights that improved the content of this thesis. I also thank my external examiner Ranjit Jhala for his valuable feedback.

I thank Domagoj Babić, a great friend, colleague, and academic "older brother" whose work ethics, tenacity, and countless advice had great influence on my research and career as a scientist.

I thank my brother and good friends for making two continents feel like home, and for turning the pursuit of this degree into one hell of a ride.

Finally, I am thankful that my research has been supported by graduate fellowships from the University of British Columbia and Microsoft Research.

To my parents Ana and Jurica, and my grandma Paula.

Chapter 1

Introduction

1.1 Motivation

Today's software systems are large, complex, and error-prone. According to a 2002 study by the U.S. National Institute of Standards and Technology [Tas02], software bugs cost the U.S. economy an estimated $60 billion each year. Even worse, the recent trends in hardware design of switching to multi-core architectures require highly concurrent software systems to take advantage of available cores. Inherent concurrency makes software development even more complex and harder to get right. Cutting software development costs and ensuring higher reliability of software is of global interest and a grand challenge [Hoa03]. Program analysis and verification is one of the most promising solutions for this problem, and therefore is a rapidly growing area of research, with steadily emerging new techniques, applications, and tools.

System software manages and controls computer hardware, providing the foundation on top of which general application programs can operate. The correct execution of programs built on top of system software relies on it to provide the correct functionality. Therefore, system software is the most critical part of a typical software stack and ensuring its correctness is of utmost importance. For example, the correct execution of even the most mundane software relies on a vast array of supporting system software: the compiler and linker during development, and also all the OS services at runtime: application-level memory management and the underlying virtual memory system, context swaps and the underlying OS scheduler, device drivers for all I/O, etc. With the emergence of virtualization, the hypervisor becomes an even lower-level, even more critical layer that needs verification (e.g., [SXSP07]), as even the operating system relies on its correctness.

The verification of system software poses additional challenges when compared to the usual application software:

- system software is typically written in a low-level programming language such as C and uses dynamic memory allocation and low-level pointer manipulation
- system software is highly concurrent, with shared-memory communication being the main concurrent programming paradigm.

In the last couple of years, many software verification tools emerged that have been successfully applied to real-life code bases. Unfortunately, the available tools still usually perform poorly when dealing with the aforementioned challenges of system software. This is a serious problem since these challenges are also the major sources of errors and complexity of system software [CYC$^+$01, LTW$^+$06, SC91, RCKH09, GN08]. The goal of this thesis is to address the problem by enabling precise and scalable verification of low-level shared-memory concurrent programs. The main contributions are about the interrelated concepts of memory, modularity, and concurrency in the context of low-level shared-memory software verification.

1.2 Software Verification

This section gives a brief taxonomy of software verification, and situates this thesis within it. Note that the term “verification” is often overloaded: the broader software engineering community uses the term when referring to any method whose goal is to increase confidence that software satisfies the specified requirements; on the other hand, parts of the formal verification community define verification as an absolutely sound[1] approach to proving correctness of a system using formal methods of mathematics. Therefore, the intended meaning of the term has to be clarified in order to avoid confusion.

[1]In the context of formal software verification, soundness means that we are not missing bugs, while completeness means we are not reporting spurious bugs.

In this thesis, the usage of the term verification lies in-between the two mentioned extremes: verification refers to any formal method for improving software correctness, where the method is allowed to systematically make calculated trade-offs in order to achieve scalability and precision at the cost of possibly missing bugs.

The foundations of software verification were laid down by Floyd [Flo67] and Hoare [Hoa69] in the 1960s with their work on logical reasoning about the correctness of programs. They defined Floyd-Hoare logic as a set of logical rules for reasoning about program correctness. Then, Dijkstra [Dij75] built on that and introduced predicate transformer semantics, in particular the well-known weakest precondition transformer. Predicate transformers enabled automatic, algorithmic transformation of reasoning about program correctness using Floyd-Hoare rules into proving a first-order logical formula valid. This seminal work branched into a few different directions in the 1970s and 1980s.

Interactive theorem proving approaches, started in the 1970s, advocate applying rigorous methods of mathematics to prove full functional correctness of a program. The approaches usually support expressive higher-order logics, and as such are used to prove complex program properties while concentrating on absolute soundness. However, this expressive power comes at a cost, and typically interactive theorem proving requires a lot of manual effort and user ingenuity: a user does proofs by guiding the proof search in some sort of a mechanical proof assistant (e.g., PVS [ORS92], HOL [GM93], ACL2 [KM97], Coq [BC04]).

In contrast, the verification-condition-checking paradigm aims to bring more automation and scalability into software verification. The main idea behind it is to automatically generate a logical formula from a program and a desired property in such a way that the formula's validity implies program correctness (with respect to the property). The generated logical formula is called a verification condition (VC). Verification conditions are then discharged using automated decision procedures. Scalability is often achieved by taking advantage of program modularity in order to check modules (e.g., procedures, functions, methods) one at a time. However, that requires from a user to provide module specifications and invariants. The approach was pioneered in the Stanford Pascal Verifier [LGvH$^+$79]. VC-checking owes its success to the seminal work on combining decision procedures [NO79, Sho84]. This work enables seamless combination of decision procedures for different theories used for software verification (e.g., the theory of uninterpreted functions, the theory of arrays, Presburger arithmetic).

Abstract interpretation [CC77] and model checking [QS82, CE82, CES86] set complete automation as their main goal. To scale to large programs these approaches analyze abstractions of the programs: Abstract interpretation abstracts a program based on the chosen abstract domain (e.g., intervals, octagons, polyhedra), while for model checking a program has to be abstracted into a finite-state system. Both approaches are completely automatic and sound. However, because they rely heavily on program abstraction, which introduces extraneous behaviors not possible in the original program, they tend to report a high rate of spurious bugs. Predicate abstraction [GS97, CGJ$^+$00, BMMR01] partially alleviates that problem by combining model checking with automated decision procedure: Automated decision procedures are used to precisely abstract programs into finite-state systems, which are then in turn model checked.

Recently, we've seen a renaissance of verification-condition-checking in the form of extended static checking (e.g., [FLL$^+$02, FM04, BLS05, CLQR07, BH08]). The renaissance was spurred by the availability of much faster machines and the birth of Satisfiability Modulo Theories (SMT) solvers (e.g., [GHN$^+$04, BT07, dMB08, BCF$^+$08]), which dramatically improved performance of automated decision procedures. Today's extended static checking tools use extremely fast SMT solvers in the back-end. This enables them to be very precise (i.e., have low rate of spurious bugs) as well as scalable. The research contributions presented in this thesis fall into the extended static checking area of software verification.

1.3 Contributions

The main contributions of this thesis are each described in their respective chapter:

Chapter 3: Memory Models for Low-Level Code. Low-level system software must sometimes make type-unsafe memory accesses (e.g., using type casts or pointer arithmetic), but because of the vast size of available heap memory in today's computer systems, faithfully representing each memory allocation and access does not scale when analyzing large programs. Instead, verification tools rely on abstract memory models to represent the program heap. This chapter reports on several related investigations to develop an accurate (i.e., providing a useful level of soundness and precision) and scalable memory model. First, we compare a recently introduced memory model, specifically designed to more accurately model low-level memory accesses in system code, to an older, widely adopted memory model. Unfortunately, the newer memory model scales poorly compared to the earlier, less accurate model. Next, we investigate how to improve the soundness of the less accurate model. A direct approach is to add assertions to the code that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Instead, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to automatically create memory models that flexibly use the more scalable memory model for most of memory, but resort to a more accurate model for memory accesses that might need it.

Chapter 4: Automatic Frame Axiom Generation. Modularity is the key to scalable software verification. However, many approaches to modular software verification are currently semi-automatic: a human must provide key logical insights — e.g., loop invariants, class invariants, and frame axioms[2] that limit the scope of changes that must be analyzed.

[2]In practice, a user typically writes *modifies clauses*, which specify a potentially un-

This chapter describes a technique for automatically inferring frame axioms of procedures and loops using static analysis. The technique builds on a pointer analysis that generates limited information about all data structures in the heap. Then, it uses that information to over-approximate a potentially unbounded set of memory locations modified by each procedure/loop; this over-approximation is a candidate frame axiom. We have tested and demonstrated the effectiveness of this approach on a set of buffer-overflow benchmarks.

Chapter 5: Verification of Shared-Memory Concurrent Programs. Context-bounded analysis is an attractive approach to verification of concurrent programs. Bounding the number of contexts executed per thread not only reduces the asymptotic complexity, but also causes the complexity to increase gradually from checking a purely sequential program. This chapter presents an approach to context-bounded verification of concurrent system programs written in C, with the heap and accompanying low-level operations such as pointer arithmetic and casts, by translating them into sequential programs. In turn, that allows traditional sequential reasoning to be employed for verification of concurrent programs. The approach is completely automatic and evaluated on a set of real-world Windows device drivers.

1.4 Organization of the Thesis

Background information required for understanding the material presented in this thesis is given in Chapter 2. Chapter 3 treats memory modeling in verification of low-level software and introduces a class of novel, scalable, and precise memory models. Chapter 4 concentrates on bringing more automation into modular software verification by presenting a technique for auto-

bounded set of memory locations that might be modified by a piece of code (e.g., procedure, loop). The underlying verification engine then automatically translates modifies clauses into frame axioms. We use the term "frame axiom", which comes from the well-known frame problem in artificial intelligence [MH69], for historical reasons. In our context, frame axioms are logical formulas that delimit what memory locations can be modified by a program module. As such, they are used in the modular assume-guarantee reasoning and actually have to be proved for each module.

matic inference of frame axioms. Chapter 5 gives an approach to context-bounded analysis of concurrent system programs by translating them into sequential programs. Finally, Chapter 6 concludes the thesis and discusses future research directions.

Chapter 2

Background

The goal of this chapter is to provide technical background material shared by multiple following chapters that describe thesis contributions. Therefore, Section 2.1 introduces the pointer analysis employed by the techniques described in Chapter 3 and Chapter 4, and Section 2.2 gives a brief overview of the thesis tool flows.

2.1 Data Structure Analysis (DSA)

This section gives some background information on the pointer analysis that is the starting point of the approach in Chapter 3 to make memory models more scalable, as well as the technique in Chapter 4 for automatically inferring frame conditions.

There is a vast body of research on pointer and related analysis, with many published algorithms that have different scalability/precision trade-offs. (Hind [Hin01] provides a good survey of many of these algorithms.) The algorithms range from the highly scalable, like Steensgaard's [Ste96], which scale to millions lines of code, to the extremely precise, such as TVLA shape analysis algorithm [LAS00], which can infer complex shapes of unbounded heap data structures but scale to only a few hundred lines of code. In that spectrum, for my research I was looking for a pointer analysis that is scalable, but precise enough for my needs, as well as readily available in a production compiler. As it turned out, Data Structure Analysis (DSA) [LLA07] worked extremely well for my purpose.

Data Structure Analysis (DSA) [LLA07] is a highly scalable and fast, context-sensitive (with full *heap cloning*), field-sensitive[3] (even in a type-

[3]The analysis precisely tracks values of distinct pointer fields of the same structure.

unsafe[4] setting), conservative pointer analysis. It offers scalability to hundreds of thousands of lines of code, with better precision than previous, highly scalable pointer analyses. The term "heap cloning" refers to a property important for achieving true context-sensitivity — heap objects are distinguished not just by allocation site, but also by (acyclic) call paths leading to their allocation, i.e. the calling context in which they were created. Support for data structure operations is often encapsulated in a library used throughout the code, and therefore context-sensitivity is important to be able to handle such cases precisely.

DSA constructs a representation of the heap in the form of Data Structure Graphs (DS graphs); it creates one DS graph per procedure plus an additional one for global storage. The separate globals graph is a key optimization allowing procedure graphs to contain only the parts of global storage reachable from that procedure. A DS graph consists of a set of nodes (DS nodes) and a set of edges. As an example, a simplified part of the globals DS graph for the applicom Linux device driver[5] is shown in Figure 2.1. DS graphs have two types of DS nodes: heap nodes with a number of fields at different offsets (e.g., rectangular nodes in the example graph), and pointer variable nodes that point into heap nodes (e.g., oval nodes in the example graph). A pointer variable node is named after the pointer variable it represents and has one outgoing edge. A heap node has one outgoing edge per pointer field. Each heap node has a type and represents a potentially unbounded number of objects in memory of that type. A DS graph edge is defined by its source node and offset (i.e. offset of the respective pointer field in the source node), and its end node and offset. For instance, if the word size in Figure 2.1 is 4 bytes, the second edge coming out of the *genhd_registered* node is defined by $\langle genhd\_registered, 8\rangle \rightarrow \langle block\_device, 0\rangle$.

Instead of just providing the usual pairs of references that may alias (points-to/alias information), the explicit heap representation DSA con-

[4]We say a program is type-safe if pointers to objects (or fields) of different types do not alias. Otherwise, we say a program is type-unsafe.

[5]The applicom driver can be found in the `/drivers/char` directory of the Linux kernel source code.

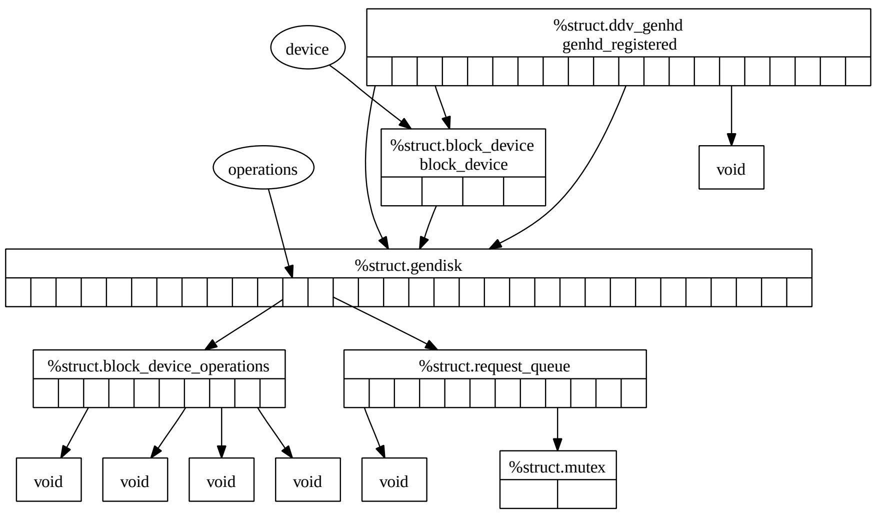

Figure 2.1: Example of a Data Structure Graph. The graph shows a simplified part of the globals DS graph for the applicom Linux device driver. Oval nodes in the graph are pointer variable nodes (e.g., *device* and *operations*); rectangular nodes are heap nodes (e.g., *genhd_registered* and *block_device*). Each heap node has a type. For instance, the type of the *genhd_registered* node is *struct.ddv_genhd*, the type of the *block_device* node is *struct.block_device*, etc. Pointer fields of heap nodes have outgoing edges, while fields of other types are just empty boxes.

structs includes objects that might not be directly necessary for identifying aliases. That additional information about linked data structures in the heap and explicit tracking of reachability relations between heap objects makes DSA a simple form of shape analysis. It can identify different instances of data structures and provide structural and type information for each identified instance.

The conservative type information for each heap object is one of the key features of DSA. The approach on scaling memory models described in Chapter 3 takes advantage of this feature. DSA defines memory accesses as operations on pointers that point into the node and actually interpret the type: load and store operations, and structure and array indexing operations on pointers. Operations such as memory allocation and pointer casts (e.g., from `void*`) are not counted as accesses and don't influence a node type. In particular, if all accesses to objects that a node represents obey a consistent type, such node is called "type-homogeneous". If accesses with incompatible types are found, the type of the node is marked as *Unknown*. Therefore, DSA tracks types precisely in the type-safe parts of the heap/program, while in the presence of type-unsafe operations, it conservatively treats nodes as having an unknown type.

The automatic inference technique presented in Chapter 4 involves describing sets of objects that DS nodes represent by traversing paths through which they are reachable from global variables, procedure parameters, etc. Hence, it is crucial to have explicit representation of heap objects and their connectives (and therefore also reachability information). Another important feature of the algorithm is conservative field-sensitivity in a type-unsafe language such as C. DSA tracks fields precisely in the type-safe parts of the heap/program, while in the presence of type-unsafe operations it conservatively collapses all fields of an object. The field-sensitivity of DSA enabled us to take advantage of the precise memory models described in Chapter 3, and therefore to generate more precise modifies sets.

DSA is a flow-insensitive and context-sensitive analysis. By definition, a flow-insensitive analysis doesn't take the order of program statements into account when analyzing a program. Therefore, it trivially extends from sequential to concurrent programs since it conservatively over-approximates all of the interleavings of parallel executions [RR99]. Furthermore, context-sensitivity comes from relying on a call graph in the analysis, which is again independent of the order of program statements. As such, it is straightforward to use DSA in the context of analysis of concurrent programs from Chapter 5 as well.

2.2 Thesis Tool Flows

An important part of this thesis consists of several verification tools that I implemented. The implemented tools interact with many others, and therefore having a high-level picture of how they are connected will allow easier

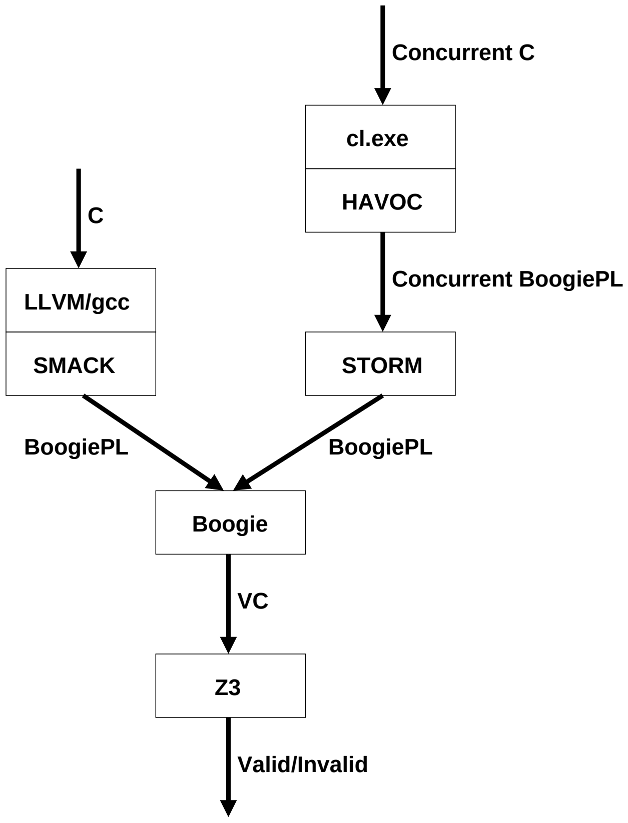

Figure 2.2: Thesis Tool Flows.

understanding of the material that follows in the later chapters. For this purpose, understanding the details is not important as the tools are explained in greater detail later on in the thesis. The tool flows are given in Figure 2.2.

During my first internship at Microsoft Research back in 2006, I was involved in creating the foundations of Havoc [CLQR07]. Havoc is a verifier for C programs built on top of Microsoft's Visual C compiler (cl.exe), and therefore is targeting Windows programs. It transforms a C program into a BoogiePL [DL05] program, which is the input of the well-known Boogie verification-condition (VC) generator [BCD+05]. Generated VCs are handed over to the Z3 theorem prover [dMB08], currently one of the fastest in the world. Based on the VC's validity, the verification either succeeds or the tool returns an error trace.

After this internship, I wanted to try out my own novel ideas in this research area. However, HAVOC was developed at Microsoft and is not open-source. Therefore, I implemented my own verifier in the spirit of HAVOC and called it SMACK [RH08, RH09]. SMACK uses the open-source LLVM compiler infrastructure [LA04], which in turn relies on *gcc* as the front-end; LLVM essentially uses the gcc parser to convert source code into LLVM's intermediate representation. Therefore, SMACK can check a different spectrum of programs — a large number of gcc-based, mainly open-source applications that are readily available. The contributions in Chapter 3 and Chapter 4 are based on this work.

During my second internship at Microsoft Research in 2008, I built a checker for concurrent C programs called STORM [LQR09]. STORM takes a multithreaded BoogiePL program generated by HAVOC as input, and generates a sequential BoogiePL program as output. The sequential BoogiePL program is then handed over to BOOGIE as usual. STORM currently uses HAVOC as the front-end and therefore aims at checking Windows programs. The contribution in Chapter 5 is based on this work.

Chapter 3

Memory Models for Low-Level Code

This chapter presents novel techniques for modeling memory in low-level code verification. It is largely based on my published paper [RH09]. In addition, Section 3.7 introduces a very recent, unpublished memory model, influenced by discussions with Ken McMillan. Overall, the described techniques rely on other front- and back-end tools orthogonal to this work (see Section 2.2), which have been significantly improved since the paper was published. Therefore, I have re-run all of the experiments of this chapter with the current tools, in order to be able to present consistent results. Understandably, the runtimes of the experiments in the thesis have changed compared to the ones from the paper. Despite that, the main message and conclusions of the chapter remain the same.

The chapter starts by giving a short introduction in Section 3.1. Section 3.2 introduces an encoding of the operational semantics of C into an intermediate language suitable for program verification. Section 3.3 explains the modeling of the semantics of memory in software verification and the related concept of memory models. It also introduces two commonly used memory models. Then, Section 3.4 compares the two introduced memory models. Section 3.5 describes a novel approach to modeling memory in low-level code verification that eliminates the weaknesses found in the previous models. Section 3.6 gives the experimental results and compares the three different techniques used in the novel approach. Section 3.7 introduces the new alias-analysis-based memory model, along with the supporting experimental results. Section 3.8 presents related work. Finally, Section 3.9 briefly summarizes the chapter by listing main contributions.

3.1 Introduction

All formal software analysis must model memory in some way. At one extreme, the entire memory space could be modeled as a single, giant array of bytes/words (e.g., [CHRF00, CKSY04, CKL04], early versions of VCC [SXSP07] also supported byte-level reasoning). Doing so makes the verification completely accurate (sound and precise with respect to the effect of any memory access), but does not scale beyond very small segments of code. At the other extreme, we can restrict our analysis to handle only code that has no dynamic memory allocation and is completely type-safe (e.g., [BCC$^+$03])[6]. Such an approach has scaled to millions of lines of code [BCC$^+$03], but obviously precludes verification of typical mainstream software. Most software verification tools (e.g., [BMMR01, HJMS02, ISG$^+$05, CCG$^+$03, FM04]) try to strike a balance, assuming some degree of type-safety, e.g., assuming that pointers to different types of objects do not alias. Note that most tools do not check these assumptions — if the code violates the assumption, the tool might report wrong answers without any warning.

The choice of memory model is particularly challenging for low-level system software, because such software must sometimes make type-unsafe memory accesses. For example, common idioms include casting a data structure from/into an array of bytes or integers for efficiency or to interface to hardware, and accessing a structure via differently-typed pointers as a way to implement sub-typing in C. Address arithmetic is also common, usually to offset before or after a given pointer in order to access a nearby data field. Verification tools for low-level software must find an intermediate memory model that relies on some type information to provide scalability, yet accurately captures the effects of lower-level, type-unsafe memory accesses.

In this chapter, we develop such a model. We start by defining an encod-

[6]Astrée now supports type casts, but still does not support dynamically allocated memory [Min06].

ing of the operational semantics of C into BoogiePL [DL05], an intermediate language suitable for program verification. Then, we give some background information on modeling the semantics of memory in software verification and by introducing two memory models: the monolithic memory model, which is a recently introduced memory model specifically designed to more accurately model low-level memory accesses in systems code, and Burstall's memory model, which is an older, widely adopted memory model that is less accurate. We compare the two memory models and find that the monolithic memory model scales poorly compared to Burstall's memory model. Then, we investigate how to improve the soundness of Burstall's model. We first consider adding to the code type-checking assertions that each memory access does not break the assumptions of the memory model, but this causes verification complexity to blow-up. Then, we develop a novel, extremely lightweight static analysis that quickly and conservatively guarantees that most memory accesses safely respect the assumptions of the memory model, thereby eliminating almost all of these extra type-checking assertions. Furthermore, this analysis allows us to automatically create memory models that flexibly use the more scalable memory model for most of memory, but resort to a more accurate model for memory accesses that might need it. Our experimental results show that the static analysis is very fast, maintaining the scalability of the less accurate memory model. In the published work, we used the conservative type information generated by the static analysis to split the memory based on computed types of memory locations. Recently, we went a step further and introduced a conservative memory splitting built on the alias information generated by the static analysis.

3.2 Operational Semantics of C

This section presents an encoding of the operational semantics of C into the BoogiePL language, which has been designed to be an intermediate language for program verification tools that use automated theorem provers. The language is simple, easy to understand, and has a well-defined semantics, so we'll use it throughout this thesis.

$$
\begin{array}{llcl}
\textit{Locs} & l & ::= & *e \mid e \rightarrow f \\
\textit{Expr} & e & ::= & x \mid n \mid l \mid \&l \mid e_1 \; \texttt{op} \; e_2 \mid e_1 \oplus_n e_2 \\
\\
\textit{Command} & c & ::= & \texttt{skip} \mid c_1; c_2 \mid x := e \mid l := e \\
 & & \mid & \texttt{if}\; e \;\texttt{then}\; c \mid \texttt{while}\; e \;\texttt{do}\; c
\end{array}
$$

Figure 3.1: Simplified Subset of C.

$$
\begin{array}{lcl}
E(x) & = & x \\
E(n) & = & n \\
E(\&e \rightarrow f) & = & E(e) + \textit{Offset}(f) \\
E(\& * e) & = & E(e) \\
E(e_1 \; \texttt{op} \; e_2) & = & E(e_1) \; \mathsf{op} \; E(e_2) \\
E(e_1 \oplus_n e_2) & = & E(e_1) + n * E(e_2)
\end{array}
$$

$$
\begin{array}{lcl}
C(\texttt{skip}) & = & \mathsf{skip} \\
C(c_1; c_2) & = & C(c_1); C(c_2) \\
C(x := e) & = & x := E(e); \\
C(l := e) & = & E(l) := E(e); \\
C(\texttt{if}\; e \;\texttt{then}\; c) & = & \mathsf{if}\; E(e) \;\mathsf{then}\; C(c) \\
C(\texttt{while}\; e \;\texttt{do}\; c) & = & \mathsf{while}\; E(e) \;\mathsf{do}\; C(c)
\end{array}
$$

Figure 3.2: Translation from C into BoogiePL.

Figure 3.1 shows a simplified subset[7] of C for illustrating the translation from C into BoogiePL. We assume all structures, global variables, and local variables whose address can be taken are allocated on the heap. The field names are assumed to be unique and $\textit{Offset}(f)$ provides the offset of a field f in its enclosing structure. In the figure, *Locs* denotes the set of heap expressions that can be used or assigned to, and *Expr* denotes the set of C expressions. The expressions include variables (x), constants (n), *Locs* and their addresses, binary operations (such as $\leq$), and pointer arithmetic $\oplus_n$ over n-byte pointers. The language contains `skip`, sequential composition, assignments, conditional statements, and loops.

Figure 3.2 shows our translation from C into BoogiePL. Note that here

[7]We are using a simplified subset of C to illustrate key concepts. Obviously, the thesis tool flows handle the full C language (see Section 2.2).

we omitted giving translations for the heap expressions $e \rightarrow f$ and $*(e : \tau)$ because those access memory. We are going to introduce the translation of the heap expressions later on, when we'll show how to model the semantics of memory using different memory models. In the figure, the operator $E(e)$ describes the translation of a C expression e. Addresses of fields and pointer arithmetic are compiled away in terms of arithmetic operations. The operator $C(c)$ translates a C statement into BoogiePL and is self-explanatory.

3.3 Modeling the Semantics of Memory

Because of the vast size of available memory in today's computer systems, faithfully representing each memory allocation and access in a static verifier does not scale. Therefore, verification tools rely on memory models that trade precision for scalability, and in turn, they define their programming language operational semantics with respect to the chosen memory model. This section introduces two memory models that are typically used in modular deductive verification tools and describes their advantages and drawbacks, as well as underlying assumptions in the context of low-level code verification.

3.3.1 Monolithic Memory Model

The monolithic memory model is heavily influenced by the one used in early versions of Havoc [CLQR07], and also similar to the one used in the first incarnation of VCC [SXSP07]. The main idea behind this memory model is to divide the memory into disjoint objects (or regions). Each object is identified by its reference, and has a fixed size determined when the object is allocated. A pointer in the memory model is therefore a pair, consisting of a reference and an offset; the reference uniquely defines the object into which the pointer points; the byte offset defines the byte in the object being pointed to.

To be able to translate a program into a representation that uses a memory model, we have to define the semantics of its source language with respect to the chosen memory model. In the monolithic memory model, the semantics of programs depends on three fundamental types: the uninterpreted type `ref` of object references, the type `int` of integers[8], and the type $\texttt{ptr} = \texttt{ref} \times \texttt{int}$ of pointers. For convenience, each variable in a program, regardless of its declared type, contains a pointer value: a pointer is a pair containing an object reference and an integer offset, and an integer value is encoded as a pointer value whose first component is the special constant `null` of type `ref`. Note that because of the integer offset component, the memory model can precisely capture byte offsets and low-level pointer arithmetic inside an object. On the other hand, since object references are uninterpreted, the objects are essentially "infinitely apart", and the memory model cannot precisely model pointer arithmetic between objects. This is not really a drawback since according to the current C standard [C99] such pointer arithmetic operations result in undefined behavior.

The heap of a program is modeled using two map variables, Mem and Alloc, and a map constant Size:

$$\begin{array}{rcl} \mathsf{Mem} & : & \texttt{ptr} \rightarrow \texttt{ptr} \\ \mathsf{Alloc} & : & \texttt{ref} \rightarrow \{\texttt{UNALLOCATED, ALLOCATED}\} \\ \mathsf{Size} & : & \texttt{ref} \rightarrow \texttt{int} \end{array}$$

The variable Mem maps pointers to pointers and represents the contents of memory at a location. The variable Alloc maps object references to the set $\{\texttt{UNALLOCATED}, \texttt{ALLOCATED}\}$ and is used to model memory allocation. The constant Size maps object references to positive integers and represents the size of the object. For instance, the procedure call `malloc(n)` for allocating a memory buffer of size `n` returns a pointer $\texttt{Ptr}(o, 0)$ where o is an object reference such that $\mathsf{Alloc}[o] = \texttt{UNALLOCATED}$ and $\mathsf{Size}[o] \geq n$ before the call, and $\mathsf{Alloc}[o] = \texttt{ALLOCATED}$ after the call. Memory allocation failure can be easily modeled by nondeterministically returning null-pointer

[8]Although currently the memory model supports only the integer data type, using the new polymorphic type system of the latest version of BoogiePL (called BOOGIE 2) [Lei08] other data types could be supported as well. However, since systems code, which is the main focus of this thesis, uses mainly integers, supporting additional data types hasn't been needed in our experiments.

```
 C1 typedef struct {
 C2   int a;
 C3   int b;
 C4 } S;
 C5
 C6 void main() {
 C7   S* s, t; int i;
 C8   s = (S*)malloc(
 C9     10*sizeof(S));
C10   i = 0;
C11   while(i < 10) {
C12     t = s[i];
C13     t->a = i;
C14     t->b = 5;
C15
C16     i++;
C17   }
C18 }
```

```
 B1
 B2
 B3
 B4
 B5
 B6 procedure main() {
 B7   var s:ptr, t:ptr, i:ptr;
 B8   call s := malloc(Ptr(null,80));
 B9
B10   i := Ptr(null,0);
B11   while(LT(i, Ptr(null,10))) {
B12     t := PLUS(s,Ptr(null,Off(i)*8));
B13     Mem[PLUS(t,Ptr(null,0))] := i;
B14     Mem[PLUS(t,Ptr(null,4))] :=
B15       Ptr(null,5);
B16     i := PLUS(i,Ptr(null,1));
B17   }
B18 }
```

Figure 3.3: Monolithic Memory Model Example. The example illustrates translation of the simple C program on the left into the BoogiePL program on the right using the monolithic memory model. We assume that the word size is 4 bytes, and that pointers and integers are one word each.

$\texttt{Ptr}(\texttt{null}, 0)$ instead.

Using the monolithic memory model, we extend Figure 3.2 with the following translations of the heap expressions (i.e., expressions that access memory):

$$\begin{aligned} E(e \rightarrow f) &= \mathsf{Mem}[E(e) + \mathit{Offset}(f)] \\ E(*(e)) &= \mathsf{Mem}[E(e)] \end{aligned}$$

Pointer arithmetic is compiled away in terms of arithmetic operations, and dereferences are simply translated as lookups into the Mem map.

The operational semantics of C with respect to the monolithic memory model are illustrated with a simple example in Figure 3.3. The figure shows the procedure `main` written in C on the left and its translation based on the monolithic memory model into the BoogiePL program on the right. The example starts by defining the C structure type `S` on line C1. Then, the procedure `main` begins by defining three variables: the variables `s` and `t` of type `S`, and the variable `i` of type `int`. Note that all variables are translated uniformly as variables of type `ptr` on line B7. Assigning 0 to `i` on line C10 shows how the integer value 0 is translated as the pointer value `Ptr(null,0)` whose first component is the special constant **null**. Then, the example illustrates how both array and field accesses are translated uniformly as pointer arithmetic. The array access on line C12 is translated as essentially $s + i * 8$ on line B12 since the size of each array element is 8. Similarly, the translation of lines C13 and C14 shows how field accesses are translated as pointer arithmetic. For instance, since the field `b` is at offset 4 in the structure type `S`, `t->b` is treated as $t + 4$ on line B14.

3.3.2 Burstall's Memory Model

The second memory model is a type-indexed memory model (also known as Burstall's memory model [Bur72]) that has been commonly used in the deductive verification of type-safe languages (e.g., [BLS05, FLL^{+}02]). The main idea behind this model is that, apart from dividing memory into disjoint objects as in the previous model, the memory is also split according to a set of possible *types*. To achieve this splitting, a set of unique type constants of type **type** is introduced, which represent types in the original program. The common types found in a language, such as `int`, `int*`, `char`, etc., are going to be translated as type constants `$int`, `$intP`, `$char`, etc. Furthermore, apart from all of the commonly found types, the set of type constants also contains a unique type constant for each structure field. For instance, the structure

```
struct {
  int a;
  int b;
} S;
```

introduces unique type constants `$S#a` and `$S#b`. It turns out that this "type-awareness" in the model, caused by adding type constants and splitting the memory according to those, is exactly what gives this model an edge when it comes to scalability over the monolithic model.

Therefore, instead of mapping pointers to pointers as in the previous memory model, the map Mem is going to map type-pointer pairs to pointers.

We also introduce in the model an additional map constant Type that maps pointers (memory locations) to types and represents the allocation type of memory locations. Each type in the memory model is a unique constant distinct from all other types. The type-indexed memory model therefore has four maps:

$$\begin{array}{rcl} \mathsf{Mem} & : & (\texttt{type} \times \texttt{ptr}) \rightarrow \texttt{ptr} \\ \mathsf{Alloc} & : & \texttt{ref} \rightarrow \{\texttt{UNALLOCATED, ALLOCATED}\} \\ \mathsf{Size} & : & \texttt{ref} \rightarrow \texttt{int} \\ \mathsf{Type} & : & \texttt{ptr} \rightarrow \texttt{type} \end{array}$$

Using Burstall's memory model, we extend Figure 3.2 with the following translations of the heap expressions:

$$\begin{array}{rcl} E(e \rightarrow f) & = & \mathsf{Mem}[f][E(e) + \mathit{Offset}(f)] \\ E(*(e:\tau)) & = & \mathsf{Mem}[\tau][E(e)] \end{array}$$

Here, we use $e : \tau$ to denote that τ is the static type of e, where static type refers to the declared type of e. The translation is mostly the same as the translation using the monolithic memory model given in the previous section. As before, pointer arithmetic is compiled away in terms of arithmetic operations. The only difference is the introduction of types into the memory model. Then, dereferences are translated as lookups into the Mem map using the appropriate type.

Figure 3.4 shows the translation of the simple example from the previous section using Burstall's memory model. First, based on the definition of the C structure type `S` on line C1, a new type for each structure field is introduced in the BoogiePL code; the introduced types `$S#a` and `$S#b` are defined on lines C1 and C2. Then, memory map Mem is going to be indexed with type-pointer pairs, as illustrated with the translation of assignments to `t->a` and `t->b` on lines C13 and C14, respectively.

Adding types to the memory model makes proving programs easier and faster:

- One can conclude that updates to different fields of a structure don't

```
typedef struct {            const unique $S#a:type;
  int a;                    const unique $S#b:type;
  int b;
} S;

void main() {               procedure main() {
  S* s, t; int i;             var s:ptr, t:ptr, i:ptr;
  s = (S*)malloc(             call s := malloc(Ptr(null,80));
    10*sizeof(S));
  i = 0;                      i := Ptr(null,0);
  while(i < 10) {             while(LT(i, Ptr(null,10))) {
    t = s[i];                   t := PLUS(s,Ptr(null,Off(i)*8));
    t->a = i;                   Mem[$S#a][PLUS(t,Ptr(null,0))] := i;
    t->b = 5;                   Mem[$S#b][PLUS(t,Ptr(null,4))] :=
                                  Ptr(null,5);
    i++;                        i := PLUS(i,Ptr(null,1));
  }                           }
}                           }
```

Figure 3.4: Burstall's Memory Model Example. The example illustrates translation of the simple C program on the left into the BoogiePL program on the right using Burstall's memory model. We assume that the word size is 4 bytes, and that pointers and integers are one word each.

influence each other without reasoning about integer offsets and pointer arithmetic, as would be needed in the monolithic memory model. Such reasoning is often hard in the presence of quantifiers.

- Memory locations of different fields of two distinct objects usually don't alias, which is nicely captured by this memory model. This also greatly simplifies the task of proving many interesting assertions.
- When a field is being updated, based on its type, only the corresponding submap of Mem changes, which simplifies expressing and checking frame axioms[9].

[9]As noted earlier, we use the term "frame axiom" for historical reasons. In our context, frame axioms are not really axioms, but rather formulas that define what is not changed by a piece of code and therefore limit the scope of changes that must be analyzed. As such, apart from being assumed, they have to be proved as well (see Chapter 4 for details).

3.3.3 Underlying Assumptions

This section gives simplifying underlying assumptions made in order to achieve scalability. The soundness of our memory models relies on these assumptions. If any are violated, soundness of a memory model is not guaranteed.

Aligned Memory Accesses. The memory models understand memory accesses to primitive types only through their first byte. It also doesn't correctly handle memory accesses that reinterpret primitive types. For example, writing a 4-byte integer value to the memory location $\texttt{Ptr}(o, 20)$ is not going to affect the value stored at the memory location $\texttt{Ptr}(o, 21)$. Similarly, reading a byte from the memory location $\texttt{Ptr}(o, 21)$ is not going to return the second byte of the integer that was written to $\texttt{Ptr}(o, 20)$, but rather an unconstrained value. Furthermore, reading a 2-byte integer value from the memory location $\texttt{Ptr}(o, 20)$, which reinterprets the 4-byte integer primitive type that was stored there earlier, is going to return the incorrect 4-byte value.

Here is a short example illustrating how such memory accesses break the soundness of our memory models:

```
int x = 0;
char y[] = &x;
y[1] = 1;
assert x == 0;
```

If we translate this example using, for instance, the monolithic memory model:

```
var x:ptr, y:ptr;
Mem[x] := Ptr(null, 0);
y := x;
Mem[PLUS(y,Ptr(null,1))] := 1;
assert Mem[x] == Ptr(null, 0);
```

it is easy to see that although the assertion in the original example should clearly fail because of the use of the unaligned pointer `y`, it actually doesn't in the given translation.

Note that not supporting byte-level memory accesses is not a serious issue in practice. Typically, casts don't reinterpret memory at the byte level, but are used to simulate object-oriented language features, such as subtyping, that are not supported directly in C. In fact, according to empirical studies [SCB$^+$99, CHM$^+$03], more than 90% of the structure casts in C fall into that category.

Whole Program Analysis. In theory, our approach is modular and doesn't require a whole program to be present: the behavior of external functions whose source code is not available can be specified using user-provided annotations in the form of pre- and postconditions. DSA also correctly handles incomplete programs: at the end of the analysis, a node in a DS graph that is reachable from unavailable external functions is marked as *Incomplete* and all information associated with it is partial and has to be treated conservatively. However, such incomplete nodes can substantially decrease the precision of our analysis, and therefore currently we require a whole program to be available. How to enable precise modular analysis by making DSA understand annotations provided for missing external functions is an open research question not addressed in this thesis.

Mathematical Integers. The operational semantics of C and memory are modeled using mathematical integers. Therefore, the effects of arithmetic operations that require precise reasoning about bit-vectors, such as overflows, are not going to be captured correctly. While I was working on the thesis, Boogie and Z3 added support for bit-vectors, but at the time the efficiency was poor. As the research on bit-vector decision procedures progresses, the modular architecture of the thesis tool flows (see Section 2.2) will enable an easy upgrade to using bit-vectors.

3.4 Comparing the Two Memory Models

This section gives empirical results on using the models to verify a number of Linux device drivers. We have implemented the preceding memory models as part of our tool Smack (Static Modular Assertion ChecKer [RH08]), which is a modular, annotation-based, extended static property checker of C programs. In the spirit of modular verification, Smack verifies programs annotated with procedure specifications and loop invariants. It uses the LLVM compiler framework [LA04] to parse input programs and annotations. The LLVM output is translated by Smack into a BoogiePL [DL05] program based on the operational semantics of C memory accesses according to the selected memory model. BoogiePL is the input language of the Boogie verifier [BCD+05], which, in turn, generates a verification condition (VC) from the input program whose validity implies partial correctness of the input. The VC generation in Boogie is performed using a variation [BL05] of the standard *weakest precondition* transformer [Dij75]. We check the generated VC using the accompanying Z3 theorem prover [dMB08]. We report only the runtimes of Boogie required to verify the examples since the transformation Smack performs takes only a small fraction of that time.

We applied Smack to check correct locking behavior of several device drivers from the Linux kernel. The source code of the examples, the models and stubs of the relevant kernel routines, and the test harness are taken from the DDVerify suite [WBKW07, DDV07]. Ensuring correct locking behavior amounts to checking that locks are initialized before they are used and that locks are alternately acquired and released. Table 3.1 lists the drivers and gives the runtimes for the verification using the monolithic and Burstall's memory models. All experiments were executed on an Intel Core2Duo at 1.6GHz.

Seven of the drivers were arbitrarily picked character device drivers that contain spinlocks, usually as one or two global variables. In addition, we handpicked the applicom driver, since this driver has a global array of structures where each structure is protected by its own spinlock. This makes it much more interesting and challenging to verify, requiring from a tool the ability to reason precisely about such unbounded data structures. Figure 3.5 illustrates the complexity of checking locking behavior in the applicom driver. Current tools that are typically used in the verification of device drivers (e.g., [BMMR01, HJMS02, ISG+05, CCG+03, CKSY04, CKL04]) have trouble handling unbounded data structures. One of the goals of Smack is to

Driver	LOC	Memory Model		Speedup
		Monolithic (s)	Burstall (s)	
ib700wd	346	25.3	12.3	2.1
w83877f_wdt	421	27.1	13.6	2.0
sc520_wdt	443	27.7	13.4	2.1
machzwd	494	34.9	15.5	2.3
wdt977	519	30.7	15.4	2.0
ds1286	633	61.0	16.6	3.7
efirtc	815	22.1	12.7	1.7
applicom	934	470.7	55.5	8.5

Table 3.1: Checking Correct Locking Behavior in Linux Device Drivers. The column "LOC" given the number of lines of code; "Monolithic" gives the total runtime of BOOGIE using the monolithic memory model; "Burstall" gives the total runtime of BOOGIE using Burstall's memory model with assumed types; "Speedup" compares the runtimes.

address that weakness.

The runtimes in Table 3.1 show that Burstall's memory model is the clear winner. It always outperforms the monolithic memory model — the speedup factor is from 1.7 to 3.7 on easier examples, and 8.5 on the more complex applicom example. As pointed out, the applicom example requires proving complex quantified invariants over fields from an array of structures. For example, the loop invariant on line 10 in Figure 3.5 states that for all pointers `x` pointing to an `applicom_board` structure belonging to the array `apbs` either `x→RamIO == NULL` or `x→mutex == UNLOCKED`. The key to fast verification of this example is structure field disambiguation (e.g., between fields `RamIO` and `mutex`): Burstall's memory model provides this for free, whereas in the monolithic model, it requires reasoning about offsets and pointer arithmetic, which is further exacerbated by the presence of quantifiers.

However, the much better runtimes of Burstall's memory model come at a price: it relies on the additional assumption that memory is strongly typed. In the examples, when we use Burstall's model, we are assuming the type of a memory location before each memory access, which is unsound

```
 1 struct applicom_board {
 2   unsigned long PhysIO;
 3   void __iomem *RamIO;
 4   wait_queue_head_t FlagSleepSend;
 5   long irq;
 6   spinlock_t mutex;
 7 } apbs[MAX_BOARD];
 8
 9 irqreturn_t ac_interrupt(int vec, void *dev_instance) {
10   invariant Forall(x, Array(apbs, sizeof(struct applicom_board),
11                             MAX_BOARD),
12                    x→RamIO == NULL || x→mutex == UNLOCKED);
13   for (i = 0; i < MAX_BOARD; i++) {
14     if (!apbs[i].RamIO) continue;
15     spin_lock(&apbs[i].mutex);
16     if(readb(apbs[i].RamIO + RAM_IT_TO_PC)) {
17       spin_unlock(&apbs[i].mutex);
18       i--;
19     } else {
20       spin_unlock(&apbs[i].mutex);
21     }
22   }
```

Figure 3.5: Simplified Code Excerpt from the applicom Linux Device Driver. The code illustrates the complexity of checking correct locking behavior. The loop on line 13 iterates over array elements. If the field `RamIO` of the element at index `i` is not null (line 14), the lock (field `mutex`) is acquired on line 15 and then later released. The verification requires checking complex invariants (e.g., line 10) over all elements of the array (i.e. quantified) that involve values of the `RamIO` fields as well as the status of locks (initialized, locked, unlocked).

and can cause bugs to be missed in a type-unsafe setting such as C. The next section describes how to deal with this problem.

3.5 Ensuring Soundness with Burstall's Memory Model

Burstall's memory model relies on the additional assumption that memory is strongly typed, as in type-safe languages such as Java. That means that a type of the object is established when it is created, via a call to

```
1  typedef struct {
2    int x;
3  } S1;
4
5  typedef struct {
6    int a;
7    int b;
8  } S2;
9
10 void main() {
11   S2* s2 =
12     (S2*)malloc(sizeof(S2));
13   S1* s1 = (S1*)s2;
14
15   s2->a = 3;
16   s1->x = 4;
17
18   assert(s2->a == 3);
19 }
```

```
1  const unique $S1#x:type;
2  const unique $S2#a:type;
3  const unique $S2#b:type;
4
5  procedure main() {
6    var s1:ptr, s2:ptr;
7    call s2 := malloc(Ptr(null,8));
8    s1 := s2;
9
10   Mem[$S2#a][s2] := Ptr(null,3);
11   Mem[$S1#x][s1] := Ptr(null,4);
12
13   assert(Mem[$S2#a][s2] ==
14     Ptr(null,3));
15 }
```

Figure 3.6: Unsoundness in Burstall's Memory Model. Example illustrating a simple upcasting in C that causes unsoundness in Burstall's memory model. The right column shows simplified BoogiePL code of the translation of the function `main`, assuming Burstall's model. Because of the assumption of type-safety, the two assignments on BoogiePL lines 10 and 11 do not alias, resulting in the assertion incorrectly passing.

`new`, and the object is always accessed using that original type. However, low-level languages like C allow reinterpretation of the original type and therefore type-unsafe memory accesses. Such operations are not uncommon in systems code and are typically done in C using casts or unions[10]. Often, casts don't reinterpret memory at the byte level, but are used to simulate object-oriented language features, such as subtyping, that are not supported directly in C. In fact, according to empirical studies [SCB$^+$99, CHM$^+$03], more than 90% of the structure casts in C fall into that category.

Figure 3.6 gives a simple example illustrating "upcasting" in C. The structure `S2` is a subtype of the structure `S1`, and the cast on line 13 represents an upcast. The example shows how such a simple cast can cause

[10]We can consider unions a special case of casts since a union can be compiled away by splitting into separate structures and introducing appropriate cast operations where needed.

Burstall's memory model to become unsound: the field update on line 16 overwrites the value that was written to the same memory location on line 15, and the assertion on line 18 fails. However, in Burstall's model this overwrite does not happen, since different field names (i.e. different unique types) denote different memory locations in the model: the write to `s2->a` is translated as the write to `Mem[$S2#a][s2]` on line 10 of the BoogiePL translation in the right column, while the write to `s1->x` is translated at the write to `Mem[$S1#x][s1]` on line 11, and doesn't overwrite the location `Mem[$S2#a][s2]` although the pointers `s1` and `s2` are equal.

In a type-safe program, Burstall's memory model captures the same non-aliasing constraints as the monolithic memory model. Therefore, in a type-safe setting, the two memory models have equivalent behavior during program analysis. In a type-unsafe program, Burstall's memory model generates spurious non-aliasing constraints (see Figure 3.6). Hence, Burstall's memory model yields different, incorrect behaviors during program analysis. These incorrect behaviors can easily break the soundness of program analysis using Burstall's memory model.

A simple way of ensuring soundness in the presence of such casts is to syntactically analyze the source code and just give up on the verification if we find one (e.g., [FM04]). Our goal is to go a step further and verify the code even in the presence of type-unsafe structure casts, while preserving soundness. The following sections describe three different techniques of how to achieve that goal.

3.5.1 Guarding Memory Accesses with Type Assertions

Our initial attempt to prevent unsoundness described in the previous section from happening in Burstall's memory model is to add *type checks* before each memory access. The checks are added in the form of assertions on the Type map. Every access to a memory location x with type τ is going to be preceded with the assertion $assert(\mathsf{Type}(x) == \tau)$ that will have to be discharged. Note that, as explained earlier in Section 3.3.2, type τ incorporates structure field names.

```
 1 const unique $S1#x:type;
 2 const unique $S2#a:type;
 3 const unique $S2#b:type;
 4
 5 procedure main() {
 6   var s1:ptr, s2:ptr;
 7   call s2 := malloc(Ptr(null,8));
 8   assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
 9   s1 := s2;
10
11   assert(Type[s2] == $S2#a);
12   Mem[$S2#a][s2] := Ptr(null,3);
13   assert(Type[s1] == $S1#x); // Fails!
14   Mem[$S1#x][s1] := Ptr(null,4);
15
16   assert(Type[s2] == $S2#a);
17   assert(Mem[$S2#a][s2] == Ptr(null,3));
18 }
```

Figure 3.7: Adding Type-check Assertions. Translation of the example from Fig. 3.6 with type-check assertions added before each memory access (lines 11, 13, and 16). The type-check assertion on line 13 will fail, indicating a violation of the assumption of type-safety.

Figure 3.7 shows the translation of the example in Figure 3.6 with the inserted type checks before each memory access (lines 11, 13, and 16). The map Type represents the compile-time allocation type of memory locations. Therefore, the correct allocation type has to be assumed on line 8 in Figure 3.7 just after the allocation. The type returned by the procedure `malloc`[11] is just `(void*)`. However, right after the call to `malloc`, the `(void*)` type will typically get cast down into the type of the pointer the allocated object is assigned to. This forms the basis for assuming the allocation type on line 8. If there is no such cast after `malloc`, the allocation type is simply assumed to be `(void*)`, in which case manual annotations equating `(void*)` with the actual allocation type will have to be provided.

The type check assertion on line 13 will clearly fail: `s1` = `s2`, and the type of `s2` is `$S2#a`, not `$S1#x`. Whenever a memory location is accessed

[11] Apart from `malloc`, our memory model allows for a user to specify a list of additional custom allocators.

through a type that is not the allocation type of the memory location (i.e., whenever two pointers to different types alias), the added type check assertion will fail. This preserves the soundness of the verification in Burstall's model.

However, proving such type check assertions for each memory access in the program is a big overhead, as the experiments in Section 3.6 show. Furthermore, discharging those assertions often requires adding more manual annotations to the code which poses an additional burden on the user. Both of these drawbacks are an unacceptable burden that is not justified since most parts of the code usually obey the type restrictions imposed by Burstall's memory model. Therefore, the next section introduces a lightweight static analysis that eagerly removes most of the required type-check assertions by conservatively guaranteeing that those memory accesses safely respect the assumptions of the model.

3.5.2 Eagerly Eliminating Type Check Assertions

This section introduces our algorithm for eagerly eliminating type check assertions. The algorithm is relatively simple and straightforward, but as the experiments in Section 3.6 show, extremely effective. First, we run DSA (pointer analysis introduced in Section 2.1) on the code we are analyzing, outputting a DS graph for each procedure and the globals graph. Then, for each memory read or write through a pointer, we find the type of the memory location it points to using the appropriate DS graph. If the computed type is the same as the actual type of the pointer, we omit the type check (assertion) that would be otherwise generated. If the types are not the same or if the type of the node the pointer points to is *Unknown*, we will generate the type check assertion to preserve soundness.

Figure 3.8 illustrates the benefits of our technique, removing two type-check assertions compared to the code in Figure 3.7. However, the soundness is preserved, since the assertion on line 12 couldn't be safely eliminated and is going to fail again: According to DSA, pointer `s1` is going to point to the field `a` of structure `S2`, and therefore its type is going to be `$S2#a` and not

```
const unique $S1#x:type;
const unique $S2#a:type;
const unique $S2#b:type;

procedure main() {
  var s1:ptr, s2:ptr;
  call s2 := malloc(Ptr(null,8));
  assume(Type[s2] == $S2#a && Type[s2 + 4] == $S2#b);
  s1 := s2;

  Mem[$S2#a][s2] := Ptr(null,3);
  assert(Type[s1] == $S1#x); // Fails!
  Mem[$S1#x][s1] := Ptr(null,4);

  assert(Mem[$S2#a][s2] == Ptr(null,3));
}
```

Figure 3.8: Using the Eager Type Check Elimination Algorithm. Translation of Fig. 3.6 using the eager type check elimination algorithm. Compared to Fig. 3.7, the unneeded type checks have been eliminated, but the type-safety violation will still be caught.

`$S1#x` as expected by the memory access.

The algorithm essentially compares compile-time pointer types used by Burstall's memory model with the sound over-approximation of the run-time types that DSA generates: if the two agree, we can safely omit the type check; if not, which could happen either because of actual type-unsafe casts or because of the imprecision of DSA, the type check stays. To sum up, using the extremely fast, cheap, and yet relatively precise pointer analysis, the algorithm eagerly gets rid of most of the type checks that are usually hard and expensive to prove later on.

In order for the remaining assertions to be discharged, either the user has to provide additional manual annotations that will essentially unify the types, which is the approach taken in some related work [Moy07, CHLQ09], or such types can be unified automatically, which is our approach described in the next section.

3.5.3 Eager Type Unification

The type check elimination algorithm from the previous section doesn't remove the type check assertion for which the compile-time type of a pointer and the one computed by DSA don't agree. Proving those leftover assertions might still require the addition of manual annotations by a user. Instead, we describe a simple, completely automatic technique that will soundly remove the leftover assertions.

For each memory access for which the type check elimination algorithm couldn't agree on types, we unify the two types. Unification simply means that the type constants are not unique anymore, which is in BoogiePL achieved by removing the keyword `unique`. There is an obvious trade-off between the type check elimination algorithm and the type unification algorithm: the first one might require additional runtime and manual annotations from a user to discharge the leftover assertions; the second one is completely automatic, but with each unification, the memory model is closer to the monolithic one and the performance might suffer (in the worst case, all types are unified and we essentially have the monolithic model). Another side-effect of using the type unification based memory model is that the map constant Type used to track types of memory locations is not needed anymore. Therefore, the type unification based memory model has only three maps:

$$\begin{aligned} \mathsf{Mem} &: (\texttt{type} \times \texttt{ptr}) \rightarrow \texttt{ptr} \\ \mathsf{Alloc} &: \texttt{ref} \rightarrow \{\texttt{UNALLOCATED, ALLOCATED}\} \\ \mathsf{Size} &: \texttt{ref} \rightarrow \texttt{int} \end{aligned}$$

Figure 3.9 shows the translation using the eager type unification algorithm. Instead of the type-check assertion on line 12 in Figure 3.8, the types `$S1#x` and `$S2#a` are unified and are not unique constants any more (lines 1 and 2). Now, `Mem[$S2#a][s2]` and `Mem[$S1#x][s1]` possibly refer to the same location, which is sound, and therefore the assertion on line 13 will fail. Note that only the types `$S1#x` and `$S2#a` involved in the actual type-unsafe access got unified, while the type `$S2#b` not involved in

```
 1 const $S1#x:type;
 2 const $S2#a:type;
 3 const unique $S2#b:type;
 4
 5 procedure main() {
 6   var s1:ptr, s2:ptr;
 7   call s2 := malloc(Ptr(null,8));
 8   s1 := s2;
 9
10   Mem[$S2#a][s2] := Ptr(null,3);
11   Mem[$S1#x][s1] := Ptr(null,4);
12
13   assert(Mem[$S2#a][s2] == Ptr(null,3));
14 }
```

Figure 3.9: Using the Eager Type Unification Algorithm. Translation of Fig. 3.6 using the eager type unification algorithm. Instead of flagging the type-safety violation, this translation handles type-unsafety by allowing `$S1#x` and `$S2#a` to be possibly the same type. Thus, the verifier will correctly catch the assertion violation on line 13.

type-unsafe operations didn't. Therefore, the over-approximation caused by unification is localized only to the places that actually need it in order to preserve soundness. In the limit, eager type unification degenerates into the monolithic memory model, but for code that is mostly type-safe, it should have most of the efficiency of Burstall's model and the soundness of the monolithic model.

3.6 Comparing the Three Approaches

The results in Table 3.2 compare the runtimes for checking correct locking behavior while ensuring soundness using the three different approaches: guarding memory accesses with type assertions, eagerly eliminating type check assertions, and eagerly unifying types. We report only Boogie runtimes. This is because total runtimes are dominated by the verification done by Boogie. The algorithm that inserts type checks for each memory access is a simple linear scan of the code and is extremely fast. Also, DSA scales to hundreds of thousands of lines of code in less than 4s [LLA07].

Driver	Assuring Soundness			Speedup	Speedup
	Every Acc. (s)	Eager Elim. (s)	Eager Unif. (s)	EA/EE	EA/EU
ib700wd	57.9	23.6	11.6	2.5	5.0
w83877f_wdt	68.4	24.9	13.1	2.8	5.2
sc520_wdt	80.4	25.3	13.5	3.2	6.0
machzwd	92.5	28.1	15.3	3.3	6.1
wdt977	92.5	33.7	14.7	2.8	6.3
ds1286	88.4	35.7	15.9	2.5	5.6
efirtc	68.1	22.8	12.0	3.0	5.7
applicom	549.9	155.1	68.4	3.6	8.0

Table 3.2: Soundly Checking Correct Locking Behavior in Linux Device Drivers. The column "Every Acc." gives the total runtime of BOOGIE when checking type assertions on every access; "Eager Elim." gives the total runtime of BOOGIE when our eager elimination technique is used to soundly remove most of the required type checks; "Eager Unif." gives the total runtime of BOOGIE when our eager unification technique is used to ensure soundness; "Speedup EA/EE" compares the runtimes of Every Access vs Eager Elimination; "Speedup EA/EU" compares the runtimes of Every Access vs Eager Unification.

As expected, blindly generating type check assertions for each memory access does not scale well — verification times after using both eager techniques are roughly 3-8 times faster. Furthermore, the verification times using eager unification are roughly 2 times faster than the ones using eager elimination because of the type-check assertions that eager elimination couldn't eliminate. Based on these experimental results the eager unification technique is the clear winner.

To sum up, our new memory models can completely automatically ensure, rather than assume, type-safety, and yet the experiments prove they are still scalable enough to handle real, complex code.

3.7 Alias-Analysis-Based Memory Model

As we discussed previously, Burstall's memory model nicely captures the fact that memory locations of different types typically don't alias, even in a type-unsafe language such as C. Our experiments also showed that having the type-based, non-alias information explicit in a memory model, such as in Burstall's, simplifies the theorem prover's task and substantially speeds up proving many interesting assertions. In this section, we explore if we can benefit from adding even more explicit non-aliasing information into a memory model using results of an alias analysis.

Alias analysis divides the program's memory into *alias classes*. Alias classes are disjoint sets of memory locations, i.e. memory locations from two alias classes cannot alias each other. Therefore, instead of having memory map Mem that maps type-pointer pairs to pointers, as in Burstall's memory model from Section 3.3.2, here we define memory map Mem that maps (alias class)-pointer pairs to pointers:

$$\begin{array}{rcl} \mathsf{Mem} & : & (\texttt{aliasclass} \times \texttt{ptr}) \rightarrow \texttt{ptr} \\ \mathsf{Alloc} & : & \texttt{ref} \rightarrow \{\texttt{UNALLOCATED, ALLOCATED}\} \\ \mathsf{Size} & : & \texttt{ref} \rightarrow \texttt{int} \end{array}$$

Each alias class in the memory model is a unique constant of type `aliasclass` distinct from all other alias class constants. We call such a memory model that uses the results of an alias analysis to explicitly capture non-aliasing information an alias-analysis-based memory model.

Figure 3.10 gives a simple example illustrating the translation of C into BoogiePL using the alias-analysis-based memory model. Note that we modified the example slightly to illustrate the potential advantage of using the results of an alias analysis when modeling memory (the previously used example would behave exactly the same as when using eager unification). Based on the results of the alias analysis (e.g., DSA), we can easily infer that pointers `s1` and `s2` point to memory locations belonging to different alias classes. Therefore, the translation introduces two unique constants `$ac1` and `$ac2` of type `aliasclass` on lines B1 and B2. Then, the in-

```
 C1 typedef struct {
 C2   int a;
 C3   int b;
 C4 } S;
 C5
 C6 void main() {
 C7   S* s1, s2;
 C8   s1 = (S*)malloc(sizeof(S));
 C9   s2 = (S*)malloc(sizeof(S));
C10
C11   s1->a = 3;
C12   s2->a = 4;
C13
C14   assert(s1->a == 3);
C15
C16 }
```

```
 B1 const unique $ac1:aliasclass;
 B2 const unique $ac2:aliasclass;
 B3
 B4
 B5
 B6 procedure main() {
 B7   var s1:ptr, s2:ptr;
 B8   call s1 := malloc(Ptr(null,8));
 B9   call s2 := malloc(Ptr(null,8));
B10
B11   Mem[$ac1][s1] := Ptr(null,3);
B12   Mem[$ac2][s2] := Ptr(null,4);
B13
B14   assert(Mem[$ac1][s1] ==
B15     Ptr(null,3));
B16 }
```

Figure 3.10: Alias Analysis Based Memory Model. This example illustrates translation of the simple C program on the left into the BoogiePL program on the right using the alias-analysis-based memory model.

troduced constants are used as the first index into memory map Mem on lines B11, B12, and B14. Note that by explicitly introducing alias classes into the memory model, one can conclude that memory updates on lines B11 and B12 are independent without reasoning about memory allocation, although the types of the respective memory locations are the same. This should further simplify reasoning about memory and therefore speed up the verification process.

The results in Table 3.3 compare the Boogie runtimes using the eager unification and alias-analysis-based memory models. While there is not a significant difference in the verification runtimes on the easier examples, on the harder applicom example the alias-analysis-based memory model is roughly 15% faster than the eager unification one. This suggests that the alias-analysis-based memory model has an advantage over eager unification on the more complex examples that generate more objects on the heap.

Driver	Eager Unification (s)	Alias Analysis (s)
ib700wd	11.6	11.4
w83877f_wdt	13.1	12.9
sc520_wdt	13.5	14.5
machzwd	15.3	15.5
wdt977	14.7	15.5
ds1286	15.9	17.1
efirtc	12.0	13.8
applicom	68.4	58.0

Table 3.3: Comparison of Eager Unification and Alias Analysis Based Memory Models. The column "Eager Unification" gives the total runtime of Boogie using the eager unification based memory model; "Alias Analysis" gives the total runtime of Boogie using the alias-analysis-based memory model.

3.8 Related Work

All software verification tools must model memory in some way. Therefore, there are a large number of different approaches to this problem and covering all of them is beyond the scope of this thesis. This section focuses on modular deductive verification tools similar to Smack and their memory models.

Burstall's memory model [Bur72] has been successfully employed by many modular software verification tools for type-safe languages. For instance, ESC/Java [FLL^{+}02] and Boogie [BLS05], as modular verification tools for Java and Spec# respectively, use Burstall's memory model. The monolithic memory model was initially appealing to verifiers for type-unsafe languages such as C, and therefore memory models similar to the monolithic were used in early versions of Havoc [CLQR07] and also in the first incarnation of VCC [SXSP07]. Caduceus [FM04] is a verification tools for C that is based on Burstall's memory model, but doesn't have a clean solution for type-unsafe operations — it gives up in the presence of type-unsafe casts. In the Caduceus framework, Moy [Moy07] introduced an approach for handling unions and casts that requires user-provided annotations as a guidance.

Havoc recently moved away from the monolithic memory model by switching to a novel memory model for low-level code that includes type information [CHLQ09]. Types can be checked using an SMT solver, and they also provide a decision procedure for checking type-safety. Using these techniques, they type-checked a number of Windows device drivers. Their work is complementary to ours: we conservatively and eagerly remove as many type checks as possible using a cheap pointer analysis, whereas they provide an efficient technique to prove the inserted type checks using a much more heavyweight SMT solver. Also, while our type unification approach is completely automatic, they require manual annotations for merging types.

The authors of VCC also independently confirmed benefits of having a typed memory model as opposed to the monolithic one, and very recently published their work on a novel memory model for C [CMST09]. VCC relies heavily on user-provided specifications and is targeting completely sound functional verification of C code. Therefore, their memory model supports precise byte-level reasoning reasoning as well as partial object overlaps, while the focus of ours is on scalability. A direct performance comparison would therefore be somewhat unfair. As well as Havoc, VCC could also benefit from the approach described in this section of automatically and conservatively splitting memory based on results of a static pointer analysis.

This thesis concentrates on using completely automatic SMT solvers for discharging verification conditions, and therefore proposes memory models suitable for that task. Alternatively, there has been some work on more expressive and powerful memory modeling and, in turn, using interactive theorem proving when discharging generated formulas [TKN07, Tuc09]. However, such approaches put a substantial burden on a user to guide theorem proving, which is something we cannot accept.

3.9 Summary

Modeling memory in a verification tool for low-level system software is a challenging problem. On one hand, we showed that using type information from the source code in a memory model greatly improves scalability. On the other hand, we illustrated how a memory model that blindly relies on such type information is potentially unsound since low-level code must sometimes make type-unsafe memory accesses. In this chapter, we developed an accurate, sound, and scalable memory models suitable for verification of low-level code. The memory models are based on a lightweight static analysis that enables them to safely use type and alias information even in the presence of type-unsafe operations. On a number of benchmarks, we clearly showed advantages of using our novel memory models in the verification-condition-checking paradigm. Furthermore, the benefits of the memory models are increasing as our verification problem becomes more involved (e.g., bigger code, more complex properties and invariants). In addition, we believe that other software verification methods for low-level code that aspire to model memory at this level would benefit from our approach.

Chapter 4

Automatic Frame Axiom Generation

This chapter presents a method for automatic inference of frame axioms in the context of modular software verification. The material presented in this chapter is mostly based on my published work [RH08]. The chapter starts with a short introduction in Section 4.1 and then gives an illustrative running example in Section 4.2. Section 4.3 informally introduces parts of the specification language required for understanding the material of this chapter. Section 4.4 describes the algorithm for fast automatic inference of frame axioms that is based on a light-weight pointer analysis. Section 4.5 gives experimental results, while Section 4.6 presents related work. Finally, Section 4.7 briefly summarizes the chapter by listing main contributions.

4.1 Introduction

The goal of this thesis is sound and scalable verification of system software. To this end, the previous chapter introduced a sound and scalable memory model for low-level code, which is an important first step since modeling memory is at the foundation of every software verifier. This chapter concentrates on modularity, which is the key to sound and scalable software verification.

Good programming practice is to construct large software systems from smaller, manageable functional units or modules. Many software verification tools exploit this natural software modularity for scalability, either by computing procedure summaries (e.g., [BH08, BMMR01, HJMS02, ISG$^+$05, BCC$^+$03]), or by relying on user-provided procedure contracts (e.g., [FLL$^+$02, Bar03, FM04, BLS05, CLQR07, SXSP07]). There is a trade-off between the two approaches: the first approach offers more automation, but the computed summaries are usually not very precise, especially in the presence of unbounded data structures; the second approach requires user input and is less automatic, but can give much more precision. One of the goals of this thesis is to make the approach based on user-provided contracts more automatic while preserving scalability and precision. Typically, user-provided contracts include procedure pre- and postconditions, loop invariants, and frame axioms.

This chapter focuses on frame axioms. Formal analysis of software always confronts some version of the frame problem [MH69]: knowing what is *not* changed by a piece of code is necessary for correct and efficient verification. For straight-line code and scalar variables, computing what changes and what does not is straightforward. In the presence of pointers, unbounded arrays, and heap-allocated data structures, however, with the corresponding looping/recursive code to manipulate them, computing precisely what changes is exceedingly difficult (undecidable in general). Frame axioms allow the user to aid this computation by suggesting candidate logical formulas that delimit what memory locations can be modified by a loop or procedure body; if the verification tool can prove these formulas to be inductive invariants, then it can use them as assumptions during verification of other assertions. Because frame axioms are so helpful to the verification process, many tools and specification styles support them, e.g.: *modifies* clauses in Spec# [BLS05] and HAVOC [CLQR07], *assignable* clauses in JML [LBR06], and *assigns* clauses in Caduceus [FM04]. Unfortunately, these frame axioms are often very complex and difficult to write, as they must carefully balance between looseness and tightness in order to be inductive, as well as being strong enough to prove desired properties of the program. We have found writing frame axioms to be the most tedious part of annotating a program's procedures and loops.

This chapter presents a novel, automatic method to infer candidate frame axioms. Our two main goals are scalability to non-trivial code bases and sufficient precision to replace most or all manual annotation of frame axioms. We are targeting lightweight verification of simpler data-oriented safety properties (e.g., buffer-overflows), which are the most common properties users are typically interested in. Because of the scalability issues with full functional verification of complex shape-sensitive properties (e.g., red-black tree rotation), those are not the focus of our method.

The method starts from a recent shallow shape analysis approach that was introduced in Section 2.1. This analysis summarizes the points-to relation as a graph; our algorithm performs a graph traversal to create a logical formula characterizing what could be modified via *any* sequence of pointer-chasing. This formula is the candidate frame axiom. The worst-case complexity of the algorithm is exponential in the size of a graph, so we must evaluate our approach empirically. We have implemented the algorithm in our modular extended static checker SMACK (see Section 3.4). To evaluate scalability, we ran our tool on several medium-sized open-source C programs and had no difficulty scaling to several tens of thousands of lines of code. To evaluate the precision of our analysis, we tested our tool on a benchmark suite of challenging buffer-overflow examples proposed at ASE 2007 [KHCL07]. With manually provided specifications (pre- and postconditions, loop invariants, and frame axioms), our tool could verify/falsify 226 of the 289 benchmarks. Using our new automatic inference approach we replaced manually provided frame axioms, which amount to more than half of the total annotation burden, with the automatically generated ones. We showed that the completely automatically inferred frame axioms are precise enough to verify/falsify 203 of the 226 benchmarks, demonstrating the effectiveness of our inference approach.

4.2 Illustrative Example

Throughout this chapter, we will use a simple running example to illustrate the basic concepts as well as our new automatic inference approach. The code of the example is presented in Figure 4.1. First, we define type `Elem` that is a structure consisting of two integer fields `f1` and `f2`. The example has two procedures called `alloc` and `init`. The procedure `alloc`

```
 1 typedef struct {int f1;
 2                 int f2;} Elem;
 3
 4 Elem* alloc(int size) {
 5   return (Elem*)malloc(size * sizeof(Elem));
 6 }
 7
 8 void init(int size) {
 9   Elem *a1 = alloc(size), *a2 = alloc(size);
10
11   // set fields f1 of a1 to 1
12   for (int i = 0; i < size; i++) {
13     a1[i].f1 = 1;
14   }
15
16   // set fields f1 of a2 and
17   // fields f2 of a1 to 0
18   for (int i = 0; i < size; i++) {
19     a2[i].f1 = 0;
20     a1[i].f2 = 0;
21   }
22
23   // check if fields f1 of a1 are set to 1
24   for (int i = 0; i < size; i++) {
25     assert(a1[i].f1 == 1);
26   }
27 }
```

Figure 4.1: Example Illustrating Frame Problem.

allocates an array of `size` elements of type `Elem`. The procedure `init` starts by allocating arrays `a1` and `a2` by calling procedure `alloc` on line 9. Both arrays have an unspecified size `size`. Then, two loops are executed:

1. The loop on line 12 sets field `f1` of all elements in `a1` to 1.
2. The loop on line 18 sets field `f1` of all elements in `a2` to 0, and also field `f2` of all elements in `a1` to 0.

In the end, we check whether field `f1` of all elements in `a1` is set to 1 using the assertion on line 25. Obviously, the assertion is not going to fail: First of all, it is clear that fields `f1` of `a1` are set to 1 in the first loop on line 13. Second, the loop on line 18 does not change those fields: it updates fields `f1` of different array `a2` and also different fields `f2` of array `a1`. Precisely such important facts about preservation of values of memory locations are necessary for verification of this example. We capture them using modifies clauses (i.e. frame axioms). As we'll see in the next section, it is often tedious to specify the modifies clauses manually. Therefore, the goal of this work is to infer as much as possible, completely automatically.

4.3 Specification Language

The modular style of verification we are employing requires a specification language for program annotations, in the form of invariants and procedure pre- and postconditions. The specification language of SMACK is the same as the one used by HAVOC [CLQR07]. It allows succinct expression of many interesting properties of low-level programs that manipulate unbounded data structures.

In this section, we will informally introduce the specification language on our illustrative example from Figure 4.1. The example with manually provided annotations required for the verification to go through is given in Figure 4.2.[12] As usual, we denote preconditions with **`requires`**, postconditions with **`ensures`**, loop invariants with **`invariant`**, and modifies clauses with **`modifies`**.

The procedure `alloc` has one precondition, `size>0`, requiring that its integer parameter `size` be greater than 0 at every call. Furthermore, it ensures that the heap object pointed to by the return pointer (denoted with `$return`) is allocated, its size is equal to `size*sizeof(Elem)`, and also that the offset component of `$return` is 0.

In procedure `init`, all three loops had to be annotated with loop invariants and modifies clauses to be able to prove the assertion on line 38. Each loop has a necessary invariant `0<=i<=size` that bounds the counter `i`. Apart from the usual basic expressions, such as `0<=i<=size`, the spec-

[12]For better readability, we omit the syntactic clutter that pushes the annotations through the C front-end of SMACK.

```
typedef struct {int f1;
                int f2;} Elem;

requires size > 0;
ensures Allocates($return);
ensures Size($return) == size*sizeof(Elem);
ensures OffsetOf($return) == 0;
Elem* alloc(int size) {
  return (Elem*)malloc(size * sizeof(Elem));
}

requires size > 0;
void init(int size) {
  Elem *a1 = alloc(size), *a2 = alloc(size);

  invariant 0 <= i <= size;
  invariant Forall(x, Array(a1, sizeof(Elem), i), x→f1 == 1);
  modifies Incr(Array(a1, sizeof(Elem), New(i)),
                OFFSET(Elem, f1));
  // set fields f1 of a1 to 1
  for (int i = 0; i < size; i++) {
    a1[i].f1 = 1;
  }

  invariant 0 <= i <= size;
  modifies Union(
    Incr(Array(a2, sizeof(Elem), New(i)), OFFSET(Elem, f1)),
    Incr(Array(a1, sizeof(Elem), New(i)), OFFSET(Elem, f2)));
  // set fields f1 of a2 and fields f2 of a1 to 0
  for (int i = 0; i < size; i++) {
    a2[i].f1 = 0;
    a1[i].f2 = 0;
  }

  invariant 0 <= i <= size;
  // check if fields f1 of a1 are set to 1
  for (int i = 0; i < size; i++) {
    assert(a1[i].f1 == 1);
  }
}
```

Figure 4.2: Example Annotated Using our Specification Language. The example is annotated with necessary preconditions, postconditions, loop invariants, and modifies clauses.

ification language also supports annotations, again borrowed from HAVOC, convenient for constructing potentially unbounded sets of pointers (such as `Array`) and for manipulating those sets (such as `Incr` and `Union`).

The expression `Array`$(p, \mathit{size}, \mathit{idx})$, where p is a pointer and *size* and *idx* are integers, refers to the unbounded set of pointers

$$\{p, p + \mathit{size}, p + 2 * \mathit{size}, \ldots, p + (\mathit{idx} - 1) * \mathit{size}\}.$$

We use it to specify a set of memory locations up to index *idx* belonging to an array whose element size is *size*. For instance, in the invariant on line 17, the expression `Array(a1,sizeof(Elem),i)` captures elements of the array `a1` up to index `i`.

The set expression `Incr`(C, n) increments each element of the set of pointers C by the offset n. On line 18, it is used to increment all pointers in the set defined with `Array` by the offset of field `f1` in the structure type `Elem`. Similarly, the set expression `Decr`(C, n) decrements each element of the set of pointers C by the offset n.

To be able to reason about sets of pointers, we use the expression `Forall`$(\mathtt{x}, S, \phi)$, which says that for all elements `x` of some set of pointers S, formula ϕ has to hold. For example, on line 17, we use `Forall` to say that fields `f1` of all elements in `a1` up to index `i` are set to 1.

Each modifies clause `modifies` C refers to a set of pointers C in the pre-state of the respective procedure or loop. It specifies which memory locations get modified by the procedure/loop. The set C has to be carefully specified. If the set is a subset of the memory locations that actually get modified, the frame axiom generated from the modifies clause will fail when the verifier checks it. If the set is too coarse of an over-approximation, the verifier will not be able to prove many interesting properties later on. Modifies clauses are therefore often complex, as can be seen from the one on line 26, which says that the loop modifies only fields `f1` of the array `a2` (first `Incr` expression of the `Union`) and fields `f2` or the array `a1` (second `Incr` expression). Note that in the loop modifies clauses, `New(i)` indicates that we are not referring to the value of `i` in the pre-state, but to the value of `i` being changed by the loop (i.e. in the post-state).

Smack needs two important facts to be able to discharge the assertion on line 38. This facts are captured by the annotations on line 17 and 26 we just explained. First, the invariant on line 17 ensures that after the loop, the field `f1` of all elements in `a1` is set to 1. In addition, the modifies annotation on line 26 ensures that the second loop does not modify the `f1` fields of `a1` that the first loop just set. The modifies clause says that the loop modifies `f1` fields of elements of array `a2` and `f2` fields of elements of `a1`, leaving therefore `f1` fields of `a1` unchanged. In order to generate these modifies sets automatically, we have to be able to distinguish `a1` from `a2` although they are allocated using the same `malloc` instruction on line 9 called from different contexts, as well as to conclude which fields (offsets) of array elements are being modified.

4.4 Automatic Frame Axiom Generation Algorithm

Given the DS graphs generated by DSA (see Section 2.1), our tool chain creates candidate frame axioms for the verifier via a three-step process. First, we process the DS graph to compute an over-approximation of the set of memory locations that can be modified by each function or loop body, which we call the *modifies set*. Next, we encode this set into a typical program specification logic, for use as a modifies clause annotation. The final step is the standard conversion of the modifies clauses into frame axioms used internally by the verification tool.

4.4.1 From DS Graphs to Modifies Sets

The goal of the first step is to compute an over-approximation of the set of memory locations that can be modified by parts of the program code. Because we intend to generate annotations for loops and procedures (for use inductively as frame axioms), our analysis centers on characterizing what memory locations can be modified by a given procedure or loop body.

The algorithm is a straightforward traversal and marking of the DS graphs. The analysis is ordered by a bottom-up traversal of the program's call graph (cycles in the call graph are broken arbitrarily). For each procedure or loop body, we can identify all store operations and mark the target address's corresponding location, which is defined by its node and offset, in the procedure's or globals DS graph as (potentially) modified. In addition, for any procedures called from this procedure or loop body, we copy the markings from the callee's DS graphs to the corresponding node in the current DS graph, if any. In other words, we mark any changes made by the current procedure/loop body, as well as copying over any changes made by any callee that is visible to the current procedure/loop body. Note that all modified locations marked in the globals graph do not have to be copied over since the globals graph is shared by all procedures. Therefore, the globals graph is always searched first when marking modified locations.

The result is that we compute the following sets of $\langle \mathit{DSnode}, \mathit{offset} \rangle$ pairs:

- For each procedure, we find a set of $\langle \mathit{DSnode}, \mathit{offset} \rangle$ pairs (i.e. memory locations) in the respective DS graph that are being modified by that procedure or its callees, and that are visible to its callers, i.e. nodes reachable from globals or procedure parameters.

- For each loop, we find a set of $\langle \mathit{DSnode}, \mathit{offset} \rangle$ pairs modified by that loop or by procedures called from the loop, and that are visible outside the loop, i.e. nodes reachable from globals or loop variables that are live at the loop header.

Note that each $\langle \mathit{DSnode}, \mathit{offset} \rangle$ pair can represent an unbounded set of memory locations.

A cycle in the call graph indicates recursive procedure calls. The modifies set we compute might not be guaranteed to be an over-approximation, because when we break cycles of recursion in the call graph, we may lose behaviors of the original program. Fortunately, this localized unsoundness in our analysis does not compromise the overall soundness of the verification, because the candidate frame axioms (like any other annotation) are checked when they are used during the verification process.

4.4.2 Modifies Sets to Modifies Clauses

The modifies sets are then passed to the second stage of our algorithm, which tries to characterize these sets of memory locations as formulas in a logic for program specification and verification, such as was presented in Section 4.3. One issue is that our specification language, like most others, does not currently have constructs for describing unbounded recursive data structures. Accordingly, the second stage starts by breaking any cycles in the DS graphs, which can represent such data structures, yielding directed acyclic graphs (DAGs). As before, this localized potential unsoundness does not compromise the overall soundness of the verification process.

For each node in the DS graph, we generate a logic formula that tries to over-approximate the set of memory locations that the pair $\langle DSnode, 0\rangle$ represents. The formulas are generated by walking over the topologically sorted (each node before all nodes to which it has outbound edges) nodes of a DS graph (DAG) starting from variables that can appear in the respective modifies clause. We call such variables *root variables.* The root variables for modifies clauses for procedures are the globals and the procedure parameters; the root variables for loops are globals and variables live at the loop header. A path in a DS graph to a node represented as a formula will be a sequence of pointer arithmetic operations, memory dereferences, and `Array` set constructors. The pseudocode of the algorithm is given in Figure 4.3.

The input of the algorithm is a set of root nodes R and a DS graph. For each node in the graph reachable from the root nodes, the algorithm generates a list of expressions, each expression representing one path to the $\langle DSnode, 0\rangle$ pair from a root node. We call such expressions *path expressions.* If a node n is a pointer variable node and its outgoing edge is e, the path expression to the beginning of the object the edge e points to is simply

$$\texttt{Decr}(n.varName, e.endOffset)$$

and is generated on line 6. Note that while $n.varName$ and $e.endOffset$ are actually evaluated by our algorithm, `Decr` becomes a part of the path expression we are recursively constructing and is not evaluated. Before we

```
1:  $Q \leftarrow$ nodes with no predecessor
2:  **while** $Q$ is non-empty **do**
3:     pop node $n$ from $Q$
4:     **if** $n$ is a pointer variable node and $n$ in $R$ **then**
5:        $e \leftarrow n.edge$
6:        $path \leftarrow \texttt{Decr}(n.varName, e.endOffset)$
7:        **if** $e.endNode$ is array node **then**
8:           $path \leftarrow$
                $\texttt{Array}(path, \texttt{sizeof}(e.endNode),$
                      $\texttt{Size}(path))$
9:        **end if**
10:       $e.endNode.addPath(path)$
11:    **else if** $n$ is a heap node **then**
12:       **for all** edges $e$ of $n$ **do**
13:          **for all** paths $p$ of $n$ **do**
14:             $path \leftarrow \texttt{Decr}(\texttt{Deref}($
                   $\texttt{Incr}(p, e.startOffset)), e.endOffset)$
15:             **if** $e.endNode$ is array node **then**
16:                $path \leftarrow$
                      $\texttt{Array}(path, \texttt{sizeof}(e.endNode),$
                            $\texttt{Size}(path))$
17:             **end if**
18:             $e.endNode.addPath(path)$
19:          **end for**
20:       **end for**
21:    **end if**
22:    **for all** edges $e$ of $n$ **do**
23:       remove edge $e$ from graph
24:       **if** $e.endNode$ has no other incoming edges **then**
25:          push $e.endNode$ into $Q$
26:       **end if**
27:    **end for**
28: **end while**
```

Figure 4.3: Automatic Frame Axiom Generation Algorithm. The algorithm generates formulas that describe DS graph nodes reachable from a set of root nodes R.

add the path expression to the *e.endNode*, we always have to case-split on whether the *e.endNode* represents an array or not. If a node n is a heap node, the algorithm iterates through all of its outgoing edges on line 12. For each edge e, it loops on line 13 through all path expression to the current node n. Then, for each path p, the path expression

$$\texttt{Decr}(\texttt{Deref}(\texttt{Incr}(p, e.\mathit{startOffset})), e.\mathit{endOffset})$$

to the beginning of the object the edge points to is generated on line 14. The path expression captures the fact that each outgoing edge from a heap node represents a memory dereference, which is represented by the `Deref` expression. Again, before adding the newly generated path expression to the end node, we have to case-split on whether the *e.endNode* represents an array or not on line 15.

An array heap node represents an array of unbounded number of elements that the algorithm captures using the `Array` expression introduced in Section 4.3. The algorithm generates array expressions

$$\texttt{Array}(\mathit{path}, \texttt{sizeof}(e.\mathit{endNode}), \texttt{Size}(\mathit{path}))$$

on lines 8 and 16. Note that while the size of each array element denoted `sizeof`(*e.endNode*) can be known at compile time, the total size of the array is usually not known since arrays tend to be dynamically allocated. However, because our memory model described in Chapter 3 has the map Size where size of each object is stored during allocation, with the expression `Size`(*path*) we are referring to this map when looking for a dynamic size of the array object *path* points to. The ability of the DSA to recognize array heap nodes, and the ability of our algorithm to precisely express the potentially unbounded set of memory locations the array nodes represent, is crucial for the precision of the generated modifies sets.

In the end, because the generated path expressions point to the beginning of an object, we have to offset them to point exactly to the modified memory location inside the object. The modifies set for the respective procedure or loop is then the union of such path expressions.

4.4.3 Modifies Clauses to Frame Axioms

Finally, frame axioms are constructed from the modifies clauses in the standard manner of the many tools that support modifies clauses. Formally, the modifies clause `modifies` C, where C is a set of pointers, is translated into the following frame axiom: [13]

$$\forall \mathtt{x} : \mathtt{ptr} \left(\begin{array}{l} \mathtt{Old}(\mathsf{Alloc})[\mathtt{Obj(x)}] == \mathtt{UNALLOCATED} \\ \mathtt{||}\ \ (\mathtt{x} \in \mathtt{Old}(C)\ \mathtt{\&\&}\ \mathtt{Obj(x)!=null}) \\ \mathtt{||}\ \ \mathtt{Old}(\mathsf{Mem})[\mathtt{x}] == \mathsf{Mem}[\mathtt{x}] \end{array} \right).$$

Informally, the axiom states that the contents of Mem remains unchanged at each pointer that is allocated and both not a member of C and not `null` in the pre-state of the procedure/loop. Because of the flow-insensitivity of DSA and also of our algorithm (i.e. flow-insensitive marking of modified locations even if they had not been allocated), a loop frame axiom might contain memory locations that are allocated only later on. Such locations are uninitialized and can point to essentially anything. Therefore, leaving them in the frame axioms would mean that anything could be modified, which is highly imprecise and would prevent proving many interesting properties. We prevent this by adding path-sensitivity by restricting the set of modified locations just to the ones that have been allocated (i.e. not equal to `null`) at the point where frame axiom had been asserted (procedure or loop entry).

The automatically generated frame axioms are our best effort to be as precise as possible, and in general do not have to be sound. However, as mentioned already, this does not affect the soundness of the verification, since all of the generated frame axioms are checked during verification.

4.4.4 Example Run

We now illustrate how the presented algorithm generates path formulas on the DS graph of our illustrative example given in Figure 4.4. The root nodes

[13] The expression `Old`(ϕ) denotes the value of ϕ in the pre-state of the procedure/loop.

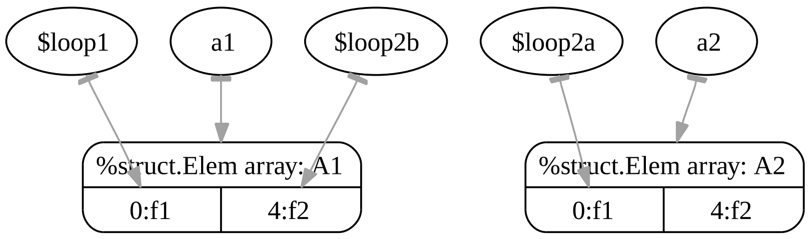

Figure 4.4: Simplified Data Structure Graph of our Illustrative Example. The graph is for procedure `init`. The nodes $\$loop1$, $\$loop2a$, and $\$loop2b$ are temporary helper pointer variable nodes not visible in the source code; $\%struct.Elem$ denotes the type of a node; flag $array$ marks array nodes; $f1$ and $f2$ are fields at offsets 0 and 4, respectively. The fields $f1$ and $f2$ are integer and not pointer fields, and therefore have no outgoing edges.

for the second loop in the example are $a1$ and $a2$. The algorithm starts by putting all pointer variable nodes (i.e. nodes with no predecessor) into Q. Nodes $\$loop1$, $\$loop2a$, and $\$loop2b$ are just going to be popped on line 3 and their edges removed in the loop on line 22 since these pointer variable nodes are not in root nodes. Node $a1$ is a root node. It has one edge whose end node is the array heap node $A1$. Therefore, the path

$$\texttt{Array}(\texttt{Decr}(a1,0),\texttt{sizeof}(A1),\texttt{Size}(\texttt{Decr}(a1,0)))$$

will be added to the paths of node $A1$. Also, in the loop on line 22, the node $A1$ will be pushed onto Q since it has no more incoming edges. The same thing will happen with $a2$ in the next iteration of the while loop. Then, $A1$ and $A2$ will be removed from Q since they have no outgoing edges, Q is empty, and we are done.

The paths generated by the algorithm are

$$\texttt{Array}(\texttt{Decr}(a1,0),\texttt{sizeof}(A1),\texttt{Size}(\texttt{Decr}(a1,0)))$$

to the memory location $\langle A1, 0\rangle$, and

$$\texttt{Array}(\texttt{Decr}(a2, 0), \texttt{sizeof}(A2), \texttt{Size}(\texttt{Decr}(a2, 0)))$$

to the memory location $\langle A2, 0\rangle$. The loop modifies memory locations pointed by $\$loop2a$ and $\$loop2b$, which correspond to pairs $\langle A2, 0\rangle$ and $\langle A1, 4\rangle$. Therefore, the expression

$$\begin{aligned}&\texttt{Incr}(\texttt{Array}(\texttt{Decr}(a2, 0),\\ &\quad \texttt{sizeof}(A2), \texttt{Size}(\texttt{Decr}(a2, 0))), 0)\end{aligned}$$

represents the first set of modified memory locations, while the expression that offsets all pointers by 4

$$\begin{aligned}&\texttt{Incr}(\texttt{Array}(\texttt{Decr}(a1, 0),\\ &\quad \texttt{sizeof}(A1), \texttt{Size}(\texttt{Decr}(a1, 0))), 4)\end{aligned}$$

represents the second set. The final modifies set for the second loop is the union of these two sets, which corresponds to the modifies set we provided manually on line 26 in Figure 4.2.

4.5 Experimental Results

We have implemented the inference algorithm in SMACK. We assessed the usability of our technique and the precision of the generated modifies clauses on the buffer-overflow benchmark suite proposed at ASE 2007, containing testcases derived from a number of buffer-overflow vulnerabilities in open-source programs [KHCL07].[14] The suite has 22 vulnerabilities from 12 programs, totaling 289 testcases (faulty and patched versions) with different difficulty levels and around 18000 LOC. First, we manually annotated most of the benchmarks with pre- and postconditions, loop invariants, and modifies

[14]For easier exposition, we presented the work on memory models in the previous chapter, although chronologically it comes after this work. Using improved memory modeling SMACK could handle real-life device drivers from Section 3.6 as opposed to smaller buffer-overflow benchmarks from this section.

Program	#TCs	#Annot	#Mod	#Infer
apache	24	24/24	90	9/24
bind	20	4/20	8	4/4
edbrowse	6	6/6	14	6/6
gxine	2	2/2	0	2/2
libgd	8	4/8	4	4/4
MADWiFi	6	6/6	8	2/6
NetBSD-libc	24	24/24	72	20/24
OpenSER	102	102/102	204	102/102
samba	4	4/4	2	4/4
sendmail	67	46/67	58	46/46
SpamAssassin	2	2/2	4	2/2
wu-ftpd	24	2/24	2	2/2
Total	289	226/289	466	203/226

Table 4.1: Quality of the Generated Modifies Clauses. Results showing the quality of the automatically generated modifies clauses. "#TCs" is the number of testcases; "#Annot" is the number of testcases our tool discharged with the manually provided annotations; "#Mod" is the number of required modifies clauses; "#Infer" is the number of testcases with the automatically generated modifies clauses our tool successfully discharged.

clauses necessary for the verification/falsification to go through. We checked NULL pointer dereference, buffer-overflow, and buffer-underflow properties for each pointer dereference. Then, we removed all of the manually provided modifies clauses and, instead, used the ones generated by our automatic approach. We again ran all of the experiments to measure the quality of the automatically generated modifies clauses. The results for this set of experiments are presented in Table 4.1 and Table 4.2.

Table 4.1 assesses the quality (i.e., precision) of the automatically generated modifies clauses. We managed to manually annotate and check with Smack 226 out of the 289 testcases. We had to skip 63 testcases because they either require bit-precise reasoning, which our tool currently does not support, or they were too complex to be completely annotated manually with the limited time we had. The annotation of these 226 testcases required 2087 annotations total (i.e., pre- and postconditions, loop invariants, and modifies clauses). Modifies clauses alone amounted to 466 annotations (or 22%), ranging in complexity from simple lists of variables that get modified to complex expressions involving unions of unbounded sets of pointers (`Array` expressions) and pointer arithmetic. Furthemore, modifies clauses were usually the most complex annotations and were hard to come up with even when other annotations were already written down.

After removing all of the manually provided modifies clauses and replacing them with the automatically generated ones, Smack discharged successfully 203/226 testcases (or 90%). This clearly shows the effectiveness of our technique: in most cases, the automatically generated modifies clauses are precise enough for the verification to succeed, or for finding a bug without introducing false errors.

We analyzed the 23 testcases for which the automatically generated modifies clauses are not good enough. In all cases, the problems are loop modifies clauses, in particular, certain idioms of loops iterating over arrays. The root cause is the loss of precision because DSA conservatively over-approximates an unbounded array by a single element. The resulting overly conservative modifies clauses can cause either verification complexity to blow-up or some annotation to fail erroneously. Typically, in these 23 testcases, a loop modifies clause that is too conservative causes the proof of some other needed loop invariant (on the same loop) to fail. Therefore, Smack never erroneously reported a violation of the overall correctness properties, but instead marked the loop invariants it couldn't prove; the failure was manifestly a failure of the analysis, not a false bug report.

Table 4.2 gives cumulative execution times of Smack for the 203 testcases that Smack could discharge with automatically generated modifies clauses. The results again support our automatic inference technique — the performance penalty we paid for using automatically generated modifies clauses is negligible compared to the effort needed for manually specifying them when the technique was not available.

Because the size of the testcases in the buffer-overflow benchmark suite is relatively small, the runtimes of DSA and our automatic inference algorithm are just a few milliseconds. Therefore, although annotating and checking

Program	MTime(s)	ITime(s)
apache	42.5	44.1
bind	11.5	11.5
edbrowse	14.5	13.6
gxine	3.8	3.8
libgd	13.3	13.3
MADWiFi	3.9	3.9
NetBSD-libc	61.6	94.5
OpenSER	275.3	276.4
samba	7.7	7.8
sendmail	120.4	120.9
SpamAssassin	4.3	4.3
wu-ftpd	3.9	3.9
Total	559.1	585.1

Table 4.2: Comparison of Manually Provided and Automatically Inferred Modifies Clauses. "MTime" is the verification time for testcases with manually provided modifies clauses; "ITime" is the verification time for testcases with automatically inferred modifies clauses. Verification was run on an Intel Pentium D at 2.8GHz.

these programs using SMACK is beyond the scope of this thesis, we assessed the scalability of the inference algorithm on a number of open-source applications: the bftpd FTP server, the muh irc-bouncer, the gzip compression utility, the Pure-FTPd FTP server, the CUDD decision diagram package (we actually run the analysis on nanotrav — a simple reachability analysis program included with the CUDD package), and the Spin explicit-state model-checker.

The runtimes are in Table 4.3 and clearly show the scalability of the prototype implementation of our approach. Our biggest example, CUDD, took only 61s. We believe the runtime for Spin is the longest because it has an unusually big DS graph of around 2500 nodes for global storage. Since the complexity of our expression generator algorithm is worst-case exponential in the size of a DS graph, it is understandable that slowdown of the analysis is possible on big graphs, which is confirmed by the Spin example. However, as can be seen from the published DSA results [LLA07], the usual maximal

Benchmark	LOC	Time(s)
bftpd 2.0	3843	2.5
gzip 1.2.4	5809	2.9
muh 2.2a	6294	2.7
Pure-FTPd 1.0.21	26320	3.8
Spin 5.1.4	29672	122.5
CUDD/nanotrav 2.4.1	67578	61.0
Total	139516	195.4

Table 4.3: Results of the Automatic Inference Algorithm. Total runtimes of DSA and our automatic inference algorithm on a number of open-source benchmarks. These experiments were executed on an AMD Opteron 254 at 2.8GHz.

graph size is only a couple hundred of nodes and such big graphs do not occur often in practice.

4.6 Related Work

There has been lots of previous work on automatic inference of data-oriented procedure preconditions, postconditions, and loop invariants (e.g., [CC77, FL01, BMMR01, FQ02, HJMM04, LL05, EPG$^+$07]). For example, the Houdini algorithm [FL01] is a simple and yet effective approach to automatic inference of module annotations in the extended static checking framework. The algorithm starts with a set of (relatively simple) candidate annotations. The set is generated from program source using simple heuristics about which annotations might be useful. Then, the algorithm iteratively removes candidate annotations that don't hold until it reaches a consistent set of annotations. However, no heuristics are provided for generating candidate modifies clauses, which is exactly what our automatic inference algorithm provides. Our work shares the same motivation with these works: making semi-automatic program verification more automatic. However, we address the different problem of inferring frame axioms, so this body of work is complementary to ours.

Separation logic [Rey02] is an extension of Floyd-Hoare logic [Flo67, Hoa69] that facilitates reasoning about programs that manipulate pointers. It allows for succinct specifications of procedures (pre- and postconditions) and loops (loop invariants) since it avoids the need to explicitly state frame axioms [ORY01]. Separation logic makes such succinct specification possible using a proof rule called the *frame rule*. However, automatic application of the frame rule often requires inference of frame axioms. Berdine et al. [BCO05b] present an automatic method for extracting frame axioms from incomplete proofs. They implemented the method in their tool for symbolic execution with separation logic called Smallfoot [BCO05a] and used it on a few small examples. This work was followed by approaches based on an automatic fixed-point computation over an abstract domain built from assertions expressed in separation logic (e.g., [GBC06, DOY06, YLB$^+$08]). Such a fix-point computation also automatically infers frames. Recently, Calcagno et al. [CDOY09] showed how to do compositional shape analysis that scales to very large programs (e.g., Linux kernel) and also infers frames.

Note that the separation-logic-based approaches focus almost exclusively on the verification of program's memory safety (i.e., no double frees, memory leaks, dereferences of dangling pointers). On the other hand, the goal of Smack is to be able to check any user-provided assertions, which typically talk about program data and not heap. Discharging such assertions requires reasoning about program data using theories such as the theory of linear arithmetic, which can't be done using separation logic exclusively. Furthermore, combinations of separation logic with other theories required for reasoning about program data are not readily available. The algorithm described in this chapter makes this connection possible: It essentially moves frame axioms generated in frameworks suitable for reasoning about the heap into the extended static checking environment supported by SMT solvers suitable for reasoning about program data. Therefore, the approaches based on separation logic are orthogonal to the results presented in this chapter, and as such could potentially be used instead of the pointer analysis we are currently employing in the initial step of our algorithm. For instance, if we would need frame axioms that talk about complex linked data structures, it might be promising to use separation-logic-based pointer analysis to generate frame axioms in expressive logics for linked data structures (e.g., [KMS02, BPZ05, BR06, RZ06, CLQR07]).

Separation-logic-based approaches to shape analysis are just one portion of the vast literature on shape analysis (and the related analysis of side-effects, pointers, etc.). We believe our automatic inference method could be adapted to other pointer and shape analyses that produce similar summary graphs of the data structures in the program to what DSA does (e.g., [LAS00, FRD00, LH01, HR05]).

Like our work, Taghdiri et al. [TSJ06] check data-oriented properties of programs. However, they attack the more difficult problem of inferring procedure summaries that are sufficiently precise to prove verification conditions. Frame axioms form part of these procedure summaries. Because they are attempting a more ambitious objective, they created their own static analysis, which is more precise (flow-sensitive as well as context-sensitive), and is therefore by design more expensive and less scalable. We cannot compare results directly, because their tool is for Java, whereas ours is for C, but we report results on more and larger examples. Our tool also can analyze the much more complicated pointer manipulation that occurs in C programs, which theirs cannot. On the other hand, for small Java procedures, their tool infers usable, complete procedure summaries, whereas our goal is only to infer frame axioms.

4.7 Summary

This chapter describes a technique for automatically inferring frame axioms of procedures and loops using static analysis. Our inference technique automatically generates frame axioms of sufficient quality to discharge approximately 90% of the benchmark examples that we could solve with manually provided frame axioms. In no cases did the automatic frame axioms produce false error reports or fail to falsify the buggy examples in the benchmark suite. The inference algorithm is also very fast. We have demonstrated scalability to several tens of thousands of lines of code.

The soundness of the verification process using the automatic frame axiom inference technique described in this chaper doesn't rely on the generated frame axioms being conservative — the axioms are still going to be discharged as part of proving the program correct. However, the underlying assumptions that memory models rely on (see Section 3.3.3) apply to this chapter as well.

Chapter 5

Verification of Shared-Memory Concurrent Programs

This chapter introduces a completely automatic approach to context-bounded analysis of concurrent programs. The material presented in this chapter is largely based on my published work [LQR09]. Our approach starts with a bug-preserving encoding of concurrent programs written in C into sequential programs. The approach handles the heap and accompanying low-level operations such as pointer arithmetic and casts. Then, it applies traditional verification techniques on the resulting sequential programs: correctness of the sequential program implies correctness of its concurrent counterpart under the given context-bound.

Section 5.1 introduces the related background work and contributions of this chapter. Section 5.2 gives the translation of concurrent C programs under the given context-bound into sequential ones. Section 5.3 presents the *field abstraction* algorithm that is crucial for the scalability of the translation. Section 5.4 describes the implementation of the approach, the benchmarks, and the experimental results. Section 5.5 presents related work. Finally, Section 5.6 briefly summarizes the chapter by listing main contributions and underlying assumptions.

5.1 Introduction

Context-bounded analysis is an attractive approach to verification of concurrent programs. This approach advocates analyzing all executions of a concurrent program in which the number of contexts executed per thread is bounded by a given constant K. Bounding the number of contexts executed per thread reduces the asymptotic complexity of checking concurrent programs: while reachability analysis of concurrent boolean programs is undecidable, the same analysis under a context-bound is NP-complete [QR05, LTKR08]. Moreover, there is ample empirical evidence that synchronization errors, such as data races and atomicity violations, are manifested in concurrent executions with small number of context switches [QW04, MQ07]. These two properties together make context-bounded analysis an effective approach for finding concurrency errors. At the same time, context-bounding provides for a useful trade-off between the cost and coverage of verification.

In this chapter, we apply context-bounded verification to concurrent C programs such as those found in low-level systems code. In order to deal with the complexity of low-level concurrent C programs, we take a three-step approach. First, we eliminate all the complexities of C (e.g., dynamic memory allocation, pointer arithmetic, casts) by compiling into the Boogie programming language (BoogiePL) [DL05] using the techniques described in Chapter 3. Thus, we obtain a concurrent BoogiePL program from a concurrent C program. Second, we eliminate the complexity of concurrency by appealing to the recent method of Lal and Reps [LR08] for reducing context-bounded verification of a concurrent boolean program to the verification of a sequential boolean program. By adapting this method to the setting of concurrent BoogiePL programs, we are able to construct a sequential BoogiePL program that captures all behaviors of the concurrent BoogiePL program (and therefore of the original C program as well) up to the context-bound. Third, we generate a verification condition from the sequential BoogiePL program and check it using a Satisfiability Modulo Theories (SMT) solver [dMB08].

In order to scale our verification to realistic C programs, we introduce the idea of *field abstraction*. The main insight is that the verification of a given property typically depends only on a small number of fields in the data structures of the program. Our algorithm partitions the set of fields into *tracked* and *untracked* fields; we only track accesses to the tracked fields and abstract away accesses to the untracked fields. This approach not only reduces the complexity of sequential code being checked, but also allows us to soundly drop context-switches from the program points where only untracked fields are accessed. Our approach is similar to localization-reduction [Kur95], but adapted to work with arrays. We present an algorithm for refining the set of tracked fields based on the counterexample-guided-abstraction-refinement (CEGAR) loop, starting with the fields in the property of interest. Our refinement algorithm is effective; on a number of examples it discovered the field abstraction that was carefully picked by a manual inspection of the program.

We implemented our ideas in a prototype tool called Storm. We applied Storm on several real-life Windows device drivers that operate in a highly concurrent setting. Storm has the ability to check any safety property specified by a user in the form of assertions in the code. In our experiments, we checked the *use-after-free* property for one of the main data structures used by device drivers. Typically, multiple driver routines, which are executing concurrently, access and may complete this data structure. To satisfy our property, the code must follow the proper and often complex synchronization protocol. Therefore, the property violations will often occur only in highly concurrent scenarios, which makes this property a natural target for Storm.

The experiments clearly demonstrate Storm's usability and scalability. Furthermore, we assess its performance with respect to code size, number of contexts, and number of places where a context-switch could happen. In the process, we found a bug in one of the drivers that could not be detected by extensive application of previous tools. The bug was confirmed and fixed by the driver developers.

5.2 Translation

In earlier work, Lal and Reps [LR08] presented a mechanism for transforming a multithreaded program operating on scalar variables into a sequential program, with a fixed context-bound. In this section, we show the main steps to transform a multithreaded program written in C into a sequential program, using Lal and Reps' method. The input C programs support pointers, dynamic memory allocation, unbounded arrays, and low-level operations such as casts and pointer arithmetic that are prevalent in system software. Our translation is performed in two steps:

1. Translate a multithreaded C program into a multithreaded BoogiePL program using the Havoc tool [CLQR07][15]. The resultant BoogiePL program contains scalars and maps, and operations on them. The translation compiles away the complexities of C programs related to pointers, dynamic memory allocation, casts, and pointer arithmetic. It is very similar to the one described in Chapter 3.

2. Translate the multithreaded BoogiePL program into a sequential BoogiePL program, for a fixed context-bound. We show how to extend Lal and Reps method to deal with programs with maps or arrays.

In the next two subsections, we describe these two steps in details.

5.2.1 Translating from C into BoogiePL

This section presents a translation of C into BoogiePL programs. The translation is used by Havoc and very similar to the one described in Chapter 3 that is used by Smack. In particular, Havoc's translation uses a variation of Burstall's memory model described in Section 3.3.2:

$$\begin{aligned} E(e \to f) &= \mathsf{Mem}^f[E(e) + \mathit{Offset}(f)] \\ E(*(e:\tau)) &= \mathsf{Mem}^\tau[E(e)] \end{aligned}$$

[15] Havoc is the tool I was working on during my first internship at Microsoft Research in 2006, and was later on the inspiration for writing Smack. The details explaining different tool flows and how they are related can be found in Section 2.2.

Here, instead of having one Mem map, the Mem map is split into a set of maps where there is a map Mem^f for each (word-valued) field f and Mem^τ for each pointer type τ. We use $e : \tau$ to denote that τ is the static type of e. Then, a dereference is translated as a lookup into the appropriate Mem map. Soundness of such a memory model in the presence of type-unsafe C operations can be assured using the techniques described in Chapter 3.

5.2.2 Eliminating Concurrency Under a Context-Bound

The previous section showed how to convert a concurrent C program into a concurrent BoogiePL program. In this section, we show how to reduce a concurrent BoogiePL program into a sequential BoogiePL program while capturing all behaviors within a context-bound, i.e. within a certain number of contexts per thread [LR08].

For the rest of this section, we fix the number of threads in the input program to a positive number n and the context-bound to a positive number K. Without loss of generality, we assume that the input concurrent program is provided as a collection of procedures containing $n + 1$ distinguished procedures Init, T_1, …, T_n, each of which takes no parameters and returns no value. The concurrent program is then given by $P \triangleq \mathit{Init}(); (T_1()|| \cdots || T_n())$. Our goal is to create a sequential program Q that captures all behaviors of P up to the context-bound K. More precisely, Q will capture all round-robin schedules of P starting from thread T_1 in which each thread can execute at most K times. Each thread is allowed to stutter in each turn, thereby enabling Q to model even those schedules that are not round-robin.

The global store of the concurrent C program is captured in the BoogiePL program as a collection of global maps from integers to integers, as described in the previous section. We assume that the program has been transformed so that every statement either reads (into a local variable) or writes (from a local variable) a global map at a single index, and that the condition for every branch depends entirely on local variables. We will also assume that each such read or write to the global memory executes atomically. To model synchronization constructs, the grain of atomicity can be explicitly increased by encapsulating statements inside an atomic block. For example, the acquire operation on a lock stored at the address a is modeled using a global map variable *Lock* and a local scalar variable *tmp* as follows:

$$\mathsf{atomic}\ \{\ \mathit{tmp} := \mathit{Lock}[a];\ \mathsf{assume}\ \mathit{tmp} = 0;\ \mathit{Lock}[a] := 1;\ \}$$

Finally, we assume that assertions in the program are modeled using a special global boolean variable *error* that is set to true whenever the condition in the assert statement evaluates to false.

To convert the concurrent program P into the semantically-equivalent sequential program Q, we introduce several extra global variables. First, we introduce a global variable k to keep track of the number of contexts executed by each thread. Second, for each global map G, we introduce $K - 1$ new symbolic map constants named V_2^G to V_K^G. Finally, we replace each global map G with K new global maps named G_1 to G_K. Intuitively, the sequential program Q mimics a concurrent execution of P as follows. First, each map G_i is initialized to the arbitrary symbolic constant V_i^G for all $2 \leq i \leq K$. The initialization procedure *Init* runs using the global map G_1 (with an arbitrary initial value) and initializes it. Then, the procedure T_1 starts executing using the global map G_1. Context switches in T_1 are simulated by a sequence of $K - 1$ nondeterministic choices using calls to procedure *Schedule* defined below. The i-th such choice enforces that the program stops using the map G_i and starts using the map G_{i+1}. Then, each of T_2 to T_n is executed sequentially one after another under the same policy. Note that when T_{j+1} starts executing on the map G_i, the value of this map is not arbitrary; rather, its value is left there by T_j when it made its i-th context switch. Finally, when T_n has finished executing, we ensure that the final value of map G_i is equated to V_{i+1}^G, which was the arbitrary initial value of the map G_{i+1} at the beginning of the $i + 1$-th context of T_1.

We capture the intuition described above by performing the following transformations in sequence:

1. Replace each statement of the form $tmp := G[a]$ with

   ```
   atomic {
     if $(k = 1)$ $tmp := G_1[a]$
     elsif $(k = 2)$ $tmp := G_2[a]$
     ...
     else $tmp := G_K[a]$
   }
   ```

   and each statement of the form $G[a] := tmp$ with

   ```
   atomic {
     if $(k = 1)$ $G_1[a] := tmp$
     elsif $(k = 2)$ $G_2[a] := tmp$
     ...
     else $G_K[a] := tmp$
   }
   ```

2. After each atomic statement that is not within the lexical scope of another atomic statement, insert a call to procedure *Schedule* that simulates an effect of a context switch happening nondeterministically. *Schedule* has the following specification:

   ```
   modifies $k$
   ensures old$(k) \leq k \wedge k \leq K$
   exsures true
   void $Schedule$(void);
   ```

   Here, exsures true means that *Schedule* may terminate either normally or exceptionally; under normal termination, k is incremented by an arbitrary amount but remains within the context-bound K. The possibility of incrementing k by more than one allows the introduction of stuttering into the round-robin schedules. The possibility of exceptional termination allows a thread to stop executing at any point. The raised exception is caught by handlers (as shown below) that wrap the invocation of each T_i. We assume that *Init* does not share any code with the threads and we do not add a call to *Schedule* to any of the procedures called from *Init*.

For each procedure f, let the procedure obtained by the transformation above be denoted by f'. Let us assume that there is a single map variable G in the original program. The sequential program Q is then defined to be as follows:

```
$G_2 := V_2^G;\ \ \ldots;\ \ G_K := V_K^G;$
$Init();$
$error := false;\ \ k := 1;$
try { $Schedule();\ \ T_1'()$ } finally $k := 1;$
...
try { $Schedule();\ \ T_n'()$ } finally $k := 1;$
assume $G_1 = V_2^G;\ \ \ldots;$ assume $G_{K-1} = V_K^G;$
assert $\neg error$
```

Note that all constraints involving the symbolic map constants are *assumed* equalities. These equalities can be handled by the select-update theory of arrays without requiring the axiom of extensionality. Consequently, these constraints do not present any impediment to the use of an off-the-shelf SMT solver. The transformed program contains control flow due to exceptions which can be easily compiled away if the underlying verification-condition generator does not understand it. Furthermore, since the transformed program is sequential, the verification-condition generator can ignore the atomic annotations in the code.

5.3 Field Abstraction

Once we have the sequential BoogiePL program generated from the multithreaded C program, the next step is to try to verify the program using

Boogie. Boogie performs precise reasoning across loop-free and call-free code, but needs loop invariants and procedure contracts to deal with loops and procedure calls modularly. In order to have an automatic tool, we inline procedures and unroll loops (with some exception discussed later). Since recursion is rare in system programs, inlining procedures is acceptable; however, the size of inlined procedures can be very large. Our initial attempt at verifying these inlined programs did not succeed. On the other hand, we may lose coverage when we unroll loops a fixed number of times. In this section, we illustrate the use of a *field abstraction* technique to achieve scalability when checking large inlined call-free programs without sacrificing precision; in some cases, our method enables us to avoid unrolling loops and therefore obtain greater coverage.

5.3.1 Abstraction with Tracked Fields

The high-level idea of this section is fairly simple: our translation of C programs described in Section 5.2.1 uses a map Mem^f for dereferencing a field f, and a map Mem^τ for dereferencing pointers of type τ. We assume that the input C program has been proven *field-safe* for this split, i.e. the type checker has verified the assertions about the Type map as described earlier. We guess a subset of these fields and types as *relevant* and abstract the program with respect to these fields. If the abstracted program can be proved correct, then we have proved the correctness of the sequential BoogiePL program. Otherwise, we have to *refine* the set of relevant fields and try again. While proving the abstracted program, we can skip loops (without the need to unroll them) that do not modify any of the relevant fields.

In this section, we formalize how we perform the abstraction with respect to a set of fields, while in the next section we show how to refine the set of fields we track. Let us define the operation `Abstract`(P, F) that takes a BoogiePL program P generated in the last section and a set of fields F, and performs the following operations:

1. For any field $g \notin F$, translate the writes $\mathsf{Mem}^g_i[e] := \mathit{tmp}$ for all $1 \leq$

$i \leq K$ as skip.

2. For any field $g \notin F$, translate the reads $tmp := \mathsf{Mem}_i^g[e]$ for all $1 \leq i \leq K$ as $\mathsf{havoc}\ tmp$, which scrambles the value of tmp.

3. Finally, remove the call to *Schedule* that was inserted after the atomic section for a read or write from a field $g \notin F$.

It is easy to see that the first two steps are property-preserving, i.e. they do not result in missed bugs. Since statements such as $\mathsf{havoc}\ tmp$ and skip do not access any global state, context switches after them will not introduce any extra behavior. Consequently, the trailing calls to *Schedule* can be removed, thereby eliminating a significant number of redundant context switches.

In addition to reducing code size and eliminating context switches, checking the abstraction `Abstract`$(P,\ F)$ has another benefit: It enables us to create a simple summary for a loop whose body does not contain any reads or writes from F. The summary leaves the memory maps unchanged and puts nondeterministic values into the local variables modified by the loop. This simple heuristic for creating loop summaries is applicable in practice: it could be applied on 5 out of a total of 15 loops in our benchmarks from Section 5.4.1. Note that such summaries are further over-approximations, and therefore are sound but not complete — if program correctness depends on local state that is modified by a summarized loop, the abstraction will generate a spurious counterexample. However, although one can always construct an artificial example leading to such behavior, we haven't seen spurious counterexamples of this form in practice.

Both of these factors improve the scalability of our approach and improve coverage by not requiring every loop to be unrolled. In particular, we can avoid the problem with unrolling loops whose exit condition does not depend on any input values (e.g., a loop that goes from 1 to 100) — for such loops any unrolling less than 100 times would block the execution after the loop.

5.3.2 Refining Tracked Fields

In this section, we provide an algorithm for inferring the set of relevant fields that affect the property being checked. Our inference algorithm is a variant of the counterexample-guided abstraction refinement (CEGAR) framework [CGJ$^+$00, Kur95]. Figure 5.1 gives the pseudo-code for the algorithm. The algorithm takes a program P and checks if the assertion in the program holds. We start with initializing *trackedFields* with an empty set, and then we add fields to the set based on the analysis of counterexamples. The outer loop in lines 3 to 26 refines *trackedFields* from a single abstract counterexample *absErrTrace* obtained by checking the abstract program A. If the abstract program A is not correct, we concretize the abstract counterexample trace *absErrTrace* and check if the trace is spurious. If the trace is not spurious, then we have a true error in line 23. The operation `Concretize` simply restores the reads and writes of fields that were abstracted away (we do not add the context switches back, although adding them would not break the algorithm). The inner loop in lines 13 to 21 greedily finds a minimal set of fields from *allFields* such that abstracting them would result in a spurious counterexample. Those fields are added to *trackedFields* and the outer loop is iterated again. Since each iteration of the inner loop increases the size of *trackedFields* and the total number of fields is finite, the algorithm terminates.

5.4 Experimental Results

In this section, we describe our prototype implementation Storm, and our experience with applying the tool on several real-life benchmarks. As described earlier, Storm first uses Havoc to translate a multithreaded C program along with a set of relevant fields into a multithreaded BoogiePL program (Section 5.2.1), then reduces it to a sequential BoogiePL program (Section 5.2.2), and finally uses Boogie and the SMT solver Z3 to check the sequential program.

```
Input: Program $P$
Output: Program $P$ checked or error trace
 1: $allFields \leftarrow$ all fields in P
 2: $trackedFields \leftarrow \emptyset$
 3: loop
 4:   $A \leftarrow$ Abstract($P$, $trackedFields$)
 5:   ($checked$, $absErrTrace$) $\leftarrow$ Check($A$)
 6:   if $checked = $ true then
 7:     return CHECKED
 8:   else
 9:     $concTrace \leftarrow$ Concretize($P$, $absErrTrace$)
10:     $checked \leftarrow$ Check($concTrace$)
11:     if $checked = $ true then
12:       $F \leftarrow allFields$
13:       for all $f \in allFields$ do
14:         $absTrace \leftarrow$ Abstract($concTrace$, $trackedFields \cup F \setminus \{f\}$)
15:         $checked \leftarrow$ Check($absTrace$)
16:         if $checked = $ true then
17:           $F \leftarrow F \setminus \{f\}$
18:         else
19:           $trackedFields \leftarrow trackedFields \cup \{f\}$
20:         end if
21:       end for
22:     else
23:       return BUG($concTrace$)
24:     end if
25:   end if
26: end loop
```

Figure 5.1: Algorithm for Tracked Fields Refinement. The algorithm is based on the CEGAR loop.

Driver	LOC	TLOC	Routine	#F	#T	Scenario
`daytona`	105	21720	`ioctl`	53	2	D | CA
`mqueue`	494	14075	`read` `write` `ioctl`	72	4	D | CA | CP | DPC
`usbsamp`	644	4040	`read` `write` `ioctl`	113	3	D | CA | CP
`usbsamp_fix`	643	4040	`read` `write` `ioctl`	113	3	D | CA | CP
`serial`	1089	32560	`read` `write`	214	3	D | CA | DPC

Table 5.1: Windows Device Drivers Used in the Experiments. "LOC" is the bare number of lines of code in the checked scenarios that the harness we wrote executes. It excludes whitespaces, comments, variable and function declarations, etc.; "TLOC" is the total number of lines of code in the checked driver. This is the number of lines that usually gets reported. However, it can be very misleading since a tool is usually checking only the code executed by a harness, and therefore it is very hard to achieve high coverage. "Routine" lists the dispatch routines we checked; "#F" gives the total number of fields; "#T" is the number of threads in the checked scenario; "Scenario" shows the concurrent scenario being checked, i.e. which driver routines are executed concurrently as threads by our harness (D – dispatch routine, CA – cancel routine, CP – completion routine, DPC – deferred procedure call).

5.4.1 Benchmarks

We evaluated Storm on a set of real-world Windows device driver benchmarks.[16] Table 5.1 lists the device drivers used in our experiments and the corresponding driver dispatch routines we checked. It also provides their size, total number of fields, number of threads, and the scenario in which they are checked. Storm found a bug in the `usbsamp` driver (see Section 5.4.3) and `usbsamp_fix` is the fixed version of the example.

[16] Storm is in the branch of the tool flows (see Section 2.2) whose front-end can't process gcc-based Linux device drivers we used previously in the experiments in Section 3.6.

We implemented a common harness for putting device drivers through different concurrent scenarios. Each driver is checked in a scenario possibly involving concurrently executing driver dispatch routines, driver request cancellation and completion routines, and deferred procedure calls (column "Scenario" in Table 5.1). The number of threads and the complexity of a scenario depend on the given driver's capabilities. For example, for the `usbsamp` driver, the harness executes a dispatch, cancel, and completion routine in three threads. Apart from providing a particular scenario, our harness also models synchronization provided by the device driver framework, as well as synchronization primitives, such as locks, that are used for driver-specific synchronization.

Storm has the ability to check any user-specified safety property. In our experiments, we checked the *use-after-free* property for the `IRP` (*IO Request Packet*) data structure used by the device drivers. A driver may complete and free an `IRP` it receives by calling a request completion routine (e.g., `WdfRequestComplete` in Figure 5.2), and must not access an `IRP` object once it has been completed. To check this property, we introduced assertions via automatic instrumentation before each access to an `IRP` object; our examples have up to a hundred of such assertions. Typically, drivers access and may complete the same request in multiple routines executing concurrently. To satisfy our crucial *use-after-free* property, the code must follow the proper and often complex synchronization protocol. Bugs often manifest only in highly concurrent scenarios; consequently, this property is difficult to check with static analysis tools for sequential programs.

5.4.2 Evaluation

Our empirical evaluation of Storm consists of two sets of experiments. In the first one (Table 5.2 and Table 5.3), we run Storm on the benchmarks described in the previous section using a manually provided, fixed set of tracked fields. We assess the scalability of Storm with respect to code size, number of threads, number of contexts, and number of locations where a context switch could potentially happen. In the second set of experiments

Example	Routine	# of contexts per thread (K)				
		1	2	3	4	5
`daytona`	`ioctl`	3.4	3.8	4.2	4.5	5.6
`mqueue`	`read`	62.1	161.5	236.2	173.0	212.4
	`write`	48.6	113.4	171.2	177.4	192.3
	`ioctl`	120.6	198.6	204.7	176.1	199.9
`usbsamp`	`read`	17.9	37.7	65.8	66.8	85.2
	`write`	17.8	48.8	52.3	74.3	109.7
	`ioctl`	4.4	5.0	5.1	5.3	5.4
`usbsamp_fix`	`read`	16.9	28.2	38.6	46.7	47.5
	`write`	18.1	32.2	46.9	52.5	63.6
	`ioctl`	4.8	4.7	5.1	5.1	5.2
`serial`	`read`	36.5	95.4	103.4	240.5	281.4
	`write`	37.3	164.3	100.8	233.0	649.8

Table 5.2: Results When Varying the Number of Contexts per Thread. Note that if the number of threads is n and the number of contexts per thread is K, then the number of possible context-switches is $n * K - 1$. For instance, in the `mqueue` example with 4 threads and 5 contexts per thread, Storm checked all behaviors with up to 19 context switches assuming round-robin schedule.

(Table 5.4), instead of using manually provided tracked fields, we determine the usability of our tracked fields refinement algorithm by using it to completely automatically check our benchmark drivers. All experiments were conducted on an Intel Pentium D at 2.8GHz running Windows XP, and all runtimes are in seconds.

Table 5.2 shows the result of varying the number of contexts per thread from 1 (sequential case) to 5. We managed to successfully check all of our benchmarks with up to 5 contexts per thread, which clearly demonstrates the scalability of our approach. In the process, our tool discovered a bug in the `usbsamp` driver (details can be found in Section 5.4.3).

Table 5.3 demonstrates how the runtimes vary with the number of places in the code where a context switch can be introduced. For the `usbsamp` example that has a bug, removing the context switches results in the bug not being discovered. The runtime decreases as the number of context-

Example	Routine	#CS	% of switches removed			
			0	40	80	100
`daytona`	`ioctl`	26	3.9	3.7	3.6	3.5
`mqueue`	`read`	201	161.1	121.3	112.1	57.8
	`write`	198	112.7	101.5	100.6	25.2
	`ioctl`	202	197.7	192.8	168.5	73.1
`usbsamp`	`read`	90	37.7	42.2	*22.6	*17.9
	`write`	90	48.9	37.7	*22.7	*18.9
	`ioctl`	22	5.0	4.8	4.5	4.4
`usbsamp_fix`	`read`	89	28.2	25.9	22.6	17.0
	`write`	89	32.2	28.2	22.5	16.5
	`ioctl`	21	4.7	4.7	4.5	4.3
`serial`	`read`	307	95.4	92.7	66.3	47.6
	`write`	309	164.8	120.2	94.3	29.7

Table 5.3: Results When Varying the Number of Locations Where a Context Switch Could Happen. The number of contexts per thread is fixed to 2. "CS" represents the total number of places where a context switch could happen. The examples where we missed the `usbsamp` bug because of randomly (unsoundly) removing context switch locations are marked with *.

switch locations decreases. This observation justifies that removing context switches during field abstraction is important for scalability.

Table 5.4 describes the results of applying the abstraction-refinement algorithm from Section 5.3 to discover the set of relevant fields and completely automatically check the examples. Using the refinement algorithm, we were always able to obtain a set of relevant fields that is just a small fraction of the set of all fields and that closely matches the set of manual fields that we used previously. Most of the runtime is actually spent in scripts to perform the abstraction, and can be significantly reduced. Without the use of field abstraction, Storm was unable to run on large examples. For example, even checking the `mqueue read` routine with only two contexts does not terminate in one hour if we do not use field abstraction.

Example	Routine	#F	#MF	#AF	#IT	Time(s)
`daytona`	`ioctl`	53	3	3	3	244.3
`mqueue`	`read`	72	7	9	9	3446.3
	`write`			8	8	3010.0
	`ioctl`			9	9	3635.6
`usbsamp_fix`	`read`	113	1	3	3	4382.4
	`write`			4	4	2079.2
	`ioctl`			0	0	21.7
`serial`	`read`	214	5	5	5	3013.7
	`write`			4	3	1729.4

Table 5.4: Results of the Tracked Fields Refinement Algorithm. "#F" gives the total number of fields; "#MF" is the number of manually provided tracked fields; "#AF" denotes the number of tracked fields generated by the refinement algorithm; "#IT" is the number of CEGAR loop iterations; "Time" is the total runtime.

5.4.3 Bug Found

By applying STORM on the Windows device drivers listed in Table 5.1, we found a concurrency bug in the **`usbsamp`** driver. We reported the bug, and the driver developers confirmed and fixed it. Figure 5.2 illustrates the bug with a simplified code excerpt from the driver. The code excerpt contains two routines, the `UsbSamp_EvtIoRead` dispatch routine and the `UsbSamp_EvtRequestCancel` cancellation routine. The routines get executed by threads `T1` and `T2`, respectively. The example proceeds as follows:

1. Thread `T1` starts executing on a request `Request`, while thread `T2` is blocked since cancellation for `Request` has not been enabled.

2. `T1` enables cancellation and sets the cancellation routine with the call to the driver framework routine `WdfRequestMarkCancelable` on line 8. Then the context switch on line 10 occurs.

3. `T2` can now start executing `UsbSamp_EvtRequestCancel`, and another context switch happens on line 20 of `T2`.

4. `T1` completes `Request` on line 11 and context switches again on line 12.

5. On line 21, `T2` tries to access `Request` that has been completed in the previous step, which is an error.

It is important to note that although the scenario leading to this bug might seem simple, the bug had not been found before by extensively applying other software checkers on **`usbsamp`**. For instance, SLAM [BMMR01] failed to discover this bug since SLAM can check only sequential code. KISS [QW04], on the other hand, can check concurrent code, but only up to 2 context switches, and would therefore also miss this bug since the bug occurs only after at least 3 context switches.

5.5 Related Work

We roughly divide the related work into two areas — bounded approaches to concurrency and other techniques for analysis of concurrent C programs.

Bounded approaches to concurrency. The idea of context-bounded analysis of concurrent programs was proposed by Qadeer and Wu and implemented in their tool called KISS [QW04]. KISS transforms a concurrent program with up to two context switches into a sequential one by mimicking context switches using procedure calls. However, restricting the number of context switches can be limiting, as evidenced by the bug in Section 5.4.3 that STORM discovered. From the theoretical perspective, context-bounded reachability analysis for concurrent boolean programs was shown to be decidable [QR05].

Rabinovitz and Grumberg [RG05] propose a context bounded verification technique for concurrent C programs based on bounded model checking and SAT solving. Their algorithm applies traditional BMC on each thread separately and generates sets of constraints for each. The constraints are instrumented to account for concurrency, by introducing copies of global variables and additional constraints for context switches. The resulting formula is solved by a SAT solver. Our work offers several important advantages: we support memory maps to deal with a possibly unbounded heap[17]; our

[17]Granted, our current implementation unrolls loops and recursion and therefore the

```
// Thread T1
VOID UsbSamp_EvtIoRead(
    WDFQUEUE   Queue,
    WDFREQUEST Request,
    size_t     Length
  ) {
  ...
  WdfRequestMarkCancelable(
    Request, UsbSamp_EvtRequestCancel);
  ... // SWITCH 1: T1->T2
  WdfRequestComplete(Request, status);
  ... // SWITCH 3: T1->T2
}

// Thread T2
VOID UsbSamp_EvtRequestCancel(
    WDFREQUEST Request
  ) {
  PREQUEST_CONTEXT rwContext;
  ... // SWITCH 2: T2->T1
  rwContext = GetRequestContext(Request);
  ...
}
```

Figure 5.2: Discovered Concurrency Bug. Simplified version of the code illustrating the concurrency bug STORM found in the `usbsamp` example. Places where context switches happen when the bug occurs are marked with `SWITCH`.

source-to-source program transformation allows us to leverage any sequential verification technique, including annotation-based modular reasoning; our experiments are performed on real-world benchmarks, whereas they apply their technique to handcrafted microbenchmarks. Finally, it is unclear how to exploit techniques such as field abstraction using their method.

CHESS [MQ07] is a tool for testing multithreaded programs that dynamically explores thread interleavings by iteratively bounding the number of contexts. In contrast, STORM is a static analysis tool and therefore does not have to execute the code using tests and offers more coverage since it explores all possible paths in a program up to a given context bound.

Bounded model checking of concurrent programs was also explored by Ganai and Gupta [GG08], where concurrency constraints are added lazily and incrementally during bounded unrolling of programs. The number of context switches is not bounded a priori, but the heap and stack are, and the number of program steps the bounded model checker explores is limited by the available resources.

Suwimonteerabuth et al. [SES08] present a context-bounded analysis of multithreaded Java programs. Their approach is different from ours because it translates a multithreaded Java program to a concurrent pushdown system by bounding the size of the program heap and using finite bit-vector encoding for integers.

Similarly to the method of Lal and Reps, very recently La Torre et al. [LMP09] proposed another method for reducing context-bounded reachability of a concurrent boolean program to the reachability of a sequential boolean program. Their method permits lazy analysis: Whereas Lal and Reps' eager analysis guesses the values of shared variables and therefore also explores unreachable states, La Torre et al.'s lazy analysis recomputes the values of shared variables when needed and hence only explores reachable states. They show performance benefits of having lazy analysis by model-checking manually generated boolean program microbenchmarks. On the other hand, we are automatically checking real-life Windows device drivers

heap is not unbounded. However, preventing unrolling by doing modular verification, in which case the heap becomes unbounded, is an area of future work (see Section 6.2).

using a verification-condition generator and SMT solvers. Therefore, in my recent work [GHR10], we do an extensive empirical comparison of different context-bounded translations in a verification-condition-checking paradigm. Our comparison clearly shows that La Torre et al.'s lazy approach does not benefit in the VC-checking paradigm. The main reason behind this result is the power of today's state-of-the-art SMT solvers to quickly prune away irrelevant parts of the search space and to propagate relevant information in any direction.

Analysis of concurrent C programs. Witkowski et al. [WBKW07] describe their experience with applying CEGAR-based predicate abstraction on concurrent Linux device drivers. Their results indicate that concurrency rapidly increases the number of predicates inferred by the refinement loop, which in turn causes a fast blow-up in the model checker. Before we derived our current technique based on SMT solvers, we attempted a similar approach where we used the Lal-Reps method to create a source-to-source transformation from a multithreaded to a sequential C program, which is then analyzed by the SLAM [BMMR01] verifier. Our experience was similar as we could not scale this approach beyond even simple microbenchmarks. Henzinger et al. [HJM04] present a more scalable approach for CEGAR-based predicate abstraction of concurrent programs; their method checks each thread separately in an abstract stateful context that is iteratively constructed by a refinement loop.

Gotsman et al. [GBCS07] construct a thread-modular shape analysis that avoids explicitly exploring thread interleavings. The analysis automatically infers a resource invariant associated with each lock: the invariant describes the part of the help protected by the lock and preserved across threads. This enables sequential analysis to be used on each thread. Their prototype implementation was used to prove memory safety of concurrent C programs operating on doubly-linked lists and ranging from 50 to 300 LOC. While our approach cannot soundly analyze unbounded linked data structures, it is much more scalable. Furthermore, while they require concurrent programs to be properly locked and data-race free, our approach is more general and handles data-races.

Chugh et al. [CVJL08] introduce a framework for converting a sequential dataflow analysis into a concurrent one using a race detection engine. The race detection engine is used to ensure soundness of the sequential analysis by invalidating the dataflow facts influenced by concurrent writes. The analysis is scalable, but yields many false positives; our approach is much more precise, but not as scalable.

Kahlon et al. [KSG09] focus their efforts on iteratively reducing the number of thread interleavings using invariants generated by abstract interpretation. The described techniques are complementary to our approach, since we could also use them to reduce the number of interleavings in our instrumented program. The authors then apply model checking, but only on program slices in order to resolve data-race warnings, and therefore fair comparison with our experiments would be hard.

There also exists work that targets analysis of concurrent boolean program models [CKS05, PST07]. However, these approaches do not clarify how to obtain these models from real-world programs, while our approach can automatically analyze C programs.

5.6 Summary

In this chapter, we introduced an encoding of context-bounded verification of a concurrent C program into the verification of a sequential program. The encoding works for system software written in C with the heap and accompanying low-level operations such as pointer arithmetic and casts. Our approach is completely automatic: we use a verification-condition generator and SMT solvers, instead of a boolean model checker used by previous similar techniques, in order to avoid manual extraction of boolean programs and false alarms introduced by the abstraction. We demonstrated the use of field abstraction for improving the scalability and (in some cases) coverage of our checking. We evaluated our tool STORM on a set of real-world Windows device drivers, and we discovered a bug that could not be detected by extensive application of other software verification tools.

This chapter, apart from relying on the assumptions from Section 3.3.3

for memory models, has additional potential sources of unsoundness that were purposely introduced in order to improve scalability and precision.

Context-Bounding. Using context-bounded analysis to check concurrent programs means that the analysis is going to miss bugs that require more than the given bound of context switches to be discovered. For example, bounding the number of context switches to only 2 wouldn't allow us to find the concurrency bug described in Section 5.4.3 that needs at least 3 context switched to be discovered. Our technique can increment the context-bound, and therefore also improve coverage, until it runs out of resources. We showed that STORM easily scales to 5 contexts per thread (see Table 5.2), and there is also ample empirical evidence that bugs are manifested in concurrent executions with small number of context switches [QW04, MQ07]. Supporting unbounded number of contexts is an area of future work (see Section 6.2).

Type-Safety. As illustrated in Section 3.5, the variation of Burstall's memory model used by STORM (see Section 5.2.1) is potentially unsound in the presence of type-unsafe memory accesses. Soundness of such a memory model can be assured using the techniques described in Chapter 3. Since the DSA pointer analysis is sound under concurrency (see Section 2.1), the described DSA-based approaches in theory trivially extend to concurrent programs. However, this extension hasn't been implemented due to practical issues related to different tool flows (see Section 2.2). In particular, DSA is a part of the open-source LLVM compiler infrastructure and as such it is not readily available in HAVOC, which uses Microsoft's infrastructure. On the other hand, HAVOC's approach to assuring memory model soundness [CHLQ09] wasn't turned on in our experiments of Section 5.4 because of the negative impact it would have on the performance of STORM.

Loops and Recursion. As currently implemented, STORM is unrolling loops and inlining recursive calls a bounded number of times. This is another potential source of unsoundness. For example, if the loop test cannot fail before the unrolling bound is reached, all assertions after the loop will be unreachable and therefore trivially satisfiable. Similarly, if the recursive call condition cannot fail before the inlining bound is reached, all assertions after the call will be unreachable. This coverage problem typically occurs when code contains loops that iterate a fixed, constant number of times. Obviously, one of the future work directions is addressing this issue (see Section 6.2).

Sequential Consistency. In the context of concurrent program verification, note that HAVOC's memory model from Section 5.2.1 assumes *sequential consistency* [Lam79]. Therefore, STORM will miss bugs that manifest only in non-sequentially-consistent executions. This is justified since our target programs are typically properly synchronized and have no data races, and therefore have only sequentially consistent executions [SJMvP07]. If necessary, data race freedom, and therefore also the soundness of STORM, can be assured using a lightweight static analysis (e.g., [KYSG07, EA03, PFH06]).

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis improves the current state-of-the-art in the area of verification of low-level shared-memory concurrent system software. The main contributions are motivated by practical problems related to memory, modularity, and concurrency, and boost precision, scalability, and automation of software verification tools. We have implemented the contributions in the verification tools SMACK and STORM developed as part of this thesis. We have applied the tools on real-life system software, and we are already finding critical, previously undiscovered bugs.

Chapter 3 is motivated by issues related to memory modeling in the presence of pointers and low-level memory operations. In the first part of the chapter, we investigated the usage of two previously known memory models for verification of low-level software. We showed on a number of experiments that the less accurate memory model heavily outperforms the more accurate one. Then, in the second part of the chapter, we investigated how to improve the soundness of the less accurate, but scalable, model. We developed a lightweight static analysis and used it to automatically create memory models that combine the more scalable memory model (used for most of memory) with the more accurate one (used only when additional precision is needed). Experimental results proved the scalability and precision of the novel memory modeling approach.

Chapter 4 is motivated by the need for more automation in modular software verification. In this chapter, we developed an automatic technique for inferring frame axioms. The technique uses pointer analysis to approximate a set of memory locations possibly modified by each procedure/loop, which is then turned into a candidate frame axiom. Experimental results demonstrated the precision and effectiveness of this technique.

Chapter 5 is motivated by the need for practical approaches to verification of concurrent system programs. Concurrent system programs are typically written in C, perform low-level memory operations, and use shared-memory communication, and as such are particularly challenging to verify. In this chapter, we introduced an approach to context-bounded verification of concurrent system programs by translating them into sequential programs. The approach is completely automatic and extremely appealing since it leverages the mature verification techniques for sequential programs. We have implemented the approach in Storm. Experimental results on a set of real-world Windows device drivers showed Storm can be used in practice to find important concurrency bugs. In follow-up work at Microsoft Research, Storm has been applied on many additional device drivers and has found more critical concurrency bugs. The Storm project is ongoing at Microsoft.

6.2 Future Work

Using Must-Alias Information in Memory Models. The alias-analysis-based memory model (see Section 3.7) leverages may-alias information returned by an alias analysis in order to split the program's memory into disjoint alias classes. Each alias class then gets its own memory map (i.e. array) in the memory model. Ken McMillan suggested that in addition to using may-alias information, we could further improve the alias-analysis-based memory model by relying on must-alias information as well. For example, if all of the pointers belonging to an alias class must alias, then they point to a single object. This object can then be modeled as a scalar variable in the memory model and not turned into an array. This would likely further improve the performance since discarging the generated verification condition would be easier without the need to reason about arrays when dealing with such scalar objects. The increase in the performance would depend on the number of opportunities in real code to perform this optimization and would have to be determined experimentaly.

Optimizing Frame Axioms. As already mentioned, the algorithm for generating modifies clauses described in Section 4.4.2 is exponential in the size of the input DS graph: the algorithm walks over a potentially exponential number of paths through the DS graph, and therefore generates a potentially exponential number of path expressions for each modified memory location. As we showed in the experiments (see Section 4.5), this exponential behavior rarely occurs in practice. However, pruning unfeasible or extraneous path expressions would still make the inferred frame axioms smaller; having smaller frame axioms would likely improve performance since smaller verification conditions are typically easier to discharge. Therefore, exploring different heuristics for pruning path expressions is a promising future work direction. For example, one simple heuristics would be to drop path expressions that dereference variables or structure fields that are never dereferenced in the source code. Thorough empirical comparison of such different heuristics would have to be done to assess what works well in practice.

Modular Verification and Concurrency. The first step towards modular verification of concurrent programs using the translation described in Chapter 5 is deriving an approach for handling loops with provided loop invariants. Therefore, we briefly introduce an initial idea of how to do the translation in the presence of loops and loop invariants.

In the weakest-precondition-based verification-condition generation, loops are usually first expanded into loop-free pieces of code, and then a traditional weakest precondition computation can be applied. The while loop:

$$\begin{array}{l}\mathsf{invariant}\ I\\ \mathsf{while}(C)\ \{B\}\end{array}$$

where I is the loop invariant, C is the loop condition, and B is the loop body, is expanded as follows:

```
assert $I$;
havoc $Modified(B)$;
assume $I$;
goto body, exit;
body : assume $C$; $\{B\}$ assert $I$; assume $false$;
exit : assume $\neg C$;
```

The code first checks whether the loop invariant holds before entering the loop. Then, an arbitrary loop iteration is created by "havocking" (i.e. assigning to nondeterministic values) all variables that are modified by the loop body and assuming the loop invariant. After that, either one more loop iteration is performed, in which case the loop condition is assumed, or the loop is terminated, in which case the negated loop condition is assumed. The inductive step of loop verification is checked after executing B using the asserted loop invariant.

The crucial problem with such loop expansion in the context of our translation from Chapter 5 is in the way we translate assertions. Because the translation introduces unconstrained symbolic global variables, assertions in general cannot be proven before the introduced unconstrained values are constrained with the assumes in the end of the translated program. Therefore, we introduce the notion of *delayed asserts*. An assertion

```
assert $F$;
```

is translated as an if statement

```
if $(\neg F)$ $error$ := $true$;
```

which delays checking the assertion to checking if the *error* bit has been set after the constraints on the shared globals have been assumed (see Section 5.2.2).

Therefore, applying our translation transforms the expansion as follows:

```
delayed_assert $I$;
havoc $Modified(B)$;
assume $I$;
goto body, exit;
body : assume $C$; $\{B\}$ delayed_assert $I$; assume $false$;
exit : assume $\neg C$;
```

The key unresolved problem is proving that applying our translation on the expansion and introducing delayed assertions in the process is sound.

Unbounded Number of Contexts. Another exciting research direction would be generalizing our translation for a fixed number of contexts from Chapter 5 to an unbounded number of contexts. We roughly sketch the initial idea next.

First, instead of predefining a bounded number of contexts K as in Section 5.2.2, in the generalized translation, K is a symbolic value representing the unbounded number of contexts. Then, instead of splitting each global map into a bounded number of new global maps (one for each context), we extend each global map with an additional dimension that is indexed using our context counter k. Therefore, instead of case-splitting on k as in

```
atomic {
  if $(k = 1)$ $tmp := G_1[a]$
  elsif $(k = 2)$ $tmp := G_2[a]$
  ...
  else $tmp := G_K[a]$
}
```

the generalized translation uses k to index into the global maps

```
atomic {
  $tmp := G[k][a]$
}
```

Since both Boogie and Z3 support quantifiers, the final assumes for constraining the introduced symbolic values are translated as

$$\textsf{assume}\ \forall i : 1 \leq i \leq K \Rightarrow G[i] = V[i+1]^G;$$

Note that introducing quantifiers usually leads to incompleteness issues. However, we still might be able to prove some interesting properties of concurrent programs with unbounded numbers of contexts.

Bibliography

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCC$^+$03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[BCD$^+$05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2005.

[BCF$^+$08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *International Conference on Computer Aided Verification (CAV)*, pages 299–303, 2008.

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 115–137, 2005.

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 52–68, 2005.

[BH08] Domagoj Babić and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering (ICSE)*, pages 211–220, 2008.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 82–87, 2005.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, pages 49–69, 2005.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 164–180, 2005.

[BR06] Jesse Bingham and Zvonimir Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 207–221, 2006.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[Bur72] Rod M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[C99] *Programming languages – C*. ISO/IEC Standard 9899:1999.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CCG$^+$03] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Symposium on Principles of Programming Languages (POPL)*, pages 289–300, 2009.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1982.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification (CAV)*, pages 154–169, 2000.

[CHLQ09] Jeremy Condit, Brian Hackett, Shuvendu Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Symposium on Principles of Programming Languages (POPL)*, pages 302–314, 2009.

[CHM$^+$03] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, 2003.

[CHRF00] David W. Currie, Alan J. Hu, Sreeranga Rajan, and Masahiro Fujita. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135, 2000.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *International SPIN Workshop on Model Checking Software (SPIN)*, pages 75–90, 2005.

[CKSY04] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[CLQR07] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.

[CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.

[CVJL08] Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 316–326, 2008.

[CYC$^+$01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[DDV07] The DDVerify verification tool. `http://www.cprover.org/ddverify/`, 2007. Cited: September 1, 2010.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[DL05] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302, 2006.

[EA03] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, 2001.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, 1967.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 15–29, 2004.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Symposium on Principles of Programming Languages (POPL)*, pages 191–202, 2002.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 253–263, 2000.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium (SAS)*, pages 240–260, 2006.

[GBCS07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 266–277, 2007.

[GG08] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In *International SPIN Workshop on Model Checking Software (SPIN)*, pages 114–133, 2008.

[GHN$^+$04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *International Conference on Computer Aided Verification (CAV)*, pages 175–188, 2004.

[GHR10] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamarić. Context-bounded translations for concurrent software: An empirical evaluation. In *International SPIN Workshop on Model Checking Software (SPIN)*, pages 227–244, 2010.

[GM93] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GN08] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft — an exploratory survey. In *Workshop on Exploiting Concurrency Efficiently and Correctly (EC2)*, 2008.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification (CAV)*, pages 72–83, 1997.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Symposium on Principles of Programming Languages (POPL)*, pages 232–244, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[ISG$^+$05] Franjo Ivančić, Ilya Shlyakhter, Aarti Gupta, Malay K. Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking C programs using F-Soft. In *International Conference on Computer Design (ICCD)*, pages 297–308, 2005.

[KHCL07] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *International Conference on Automated Software Engineering (ASE)*, pages 389–392, 2007.

[KM97] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[KSG09] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic reduction of thread interleavings in concurrent programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 124–138, 2009.

[Kur95] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1995.

[KYSG07] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer-Aided Verification (CAV)*, pages 226–239, 2007.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, 2004.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301, 2000.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes*, 31(3):1–38, 2006.

[Lei08] K. Rustan M. Leino. *This is Boogie 2*, 2008. Draft. Available from `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[LGvH$^+$79] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford Pascal Verifier user manual. Technical report, Stanford University, 1979.

[LH01] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium (SAS)*, pages 279–298, 2001.

[LL05] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 119–134, 2005.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[LMP09] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *International Conference on Computer Aided Verification (CAV)*, pages 477–492, 2009.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *International Conference on Computer Aided Verification (CAV)*, pages 509–524, 2009.

[LR08] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *International Conference on Computer Aided Verification (CAV)*, pages 37–51, 2008.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 282–298, 2008.

[LTW$^+$06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33, 2006.

[MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[Min06] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 54–63, 2006.

[Moy07] Yannick Moy. Union and cast in deductive verification. In *C/C++ Verification Workshop (CCV)*, pages 1–16, 2007.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 446–455, 2007.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1:245–257, 1979.

[ORS92] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction (CADE)*, pages 748–752, 1992.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic (CSL)*, pages 1–19, 2001.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 320–331, 2006.

[PST07] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *International Conference on Computer Aided Verification (CAV)*, pages 254–257, 2007.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 93–107, 2005.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Colloquium on International Symposium on Programming*, pages 337–351, 1982.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 14–24, 2004.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *European Conference on Computer Systems (EuroSys)*, pages 275–288, 2009.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *International Conference on Computer-Aided Verification (CAV)*, pages 82–97, 2005.

[RH08] Zvonimir Rakamarić and Alan J. Hu. Automatic inference of frame axioms using static analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 89–98, 2008.

[RH09] Zvonimir Rakamarić and Alan J. Hu. A scalable memory model for low-level code. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.

[RR99] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 1999.

[RZ06] Silvio Ranise and Calogero Zarba. A theory of singly-linked lists and its extensible decision procedure. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 206–215, 2006.

[SC91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability — a study of field failures in operating systems. In *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 2–9, 1991.

[SCB$^+$99] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas W. Reps. Coping with type casts in

C. In *European Software Engineering Conference/Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 180–198, 1999.

[SES08] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *International SPIN Workshop on Model Checking Software (SPIN)*, pages 270–287, 2008.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM (JACM)*, 31:1–12, 1984.

[SJMvP07] Vijay A. Saraswat, Radha Jagadeesan, Maged M. Michael, and Christoph von Praun. A theory of memory models. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 161–172, 2007.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[SXSP07] Wolfram Schulte, Songtao Xia, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop (CCV)*, 2007.

[Tas02] Gregory Tassey. *The economic impact of inadequate infrastructure for software testing.* National Institute of Standards and Technology, 2002. Planning Report 02-3.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 97–108, 2007.

[TSJ06] Mana Taghdiri, Robert Seater, and Daniel Jackson. Lightweight extraction of syntactic specifications. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 276–286, 2006.

[Tuc09] Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42(2-4):125–187, 2009.

[WBKW07] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *International Conference on Automated Software Engineering (ASE)*, pages 501–504, 2007.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer-Aided Verification (CAV)*, pages 385–398, 2008.