Exploiting Structure for Scalable Software Verification

by

Domagoj Babić

Dipl.Ing., University of Zagreb, 2001
M.Sc., University of Zagreb, 2003

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

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

August, 2008

Abstract

Software bugs are expensive. Recent estimates by the US National Institute of Standards and Technology[1] claim that the cost of software bugs to the US economy alone is approximately 60 billion USD annually. As society becomes increasingly software-dependent, bugs also reduce our productivity and threaten our safety and security. Decreasing these direct and indirect costs represents a significant research challenge as well as an opportunity for businesses.

Automatic software bug-finding and verification tools have a potential to completely revolutionize the software engineering industry by improving reliability and decreasing development costs. Since software analysis is in general undecidable, automatic tools have to use various abstractions to make the analysis computationally tractable. Abstraction is a double-edged sword: coarse abstractions, in general, yield easier verification, but also less precise results.

This thesis focuses on exploiting the structure of software for abstracting away irrelevant behavior. Programmers tend to organize code into objects and functions, which effectively represent natural abstraction boundaries. Humans use such structural abstractions to simplify their mental models of software and for constructing informal explanations of why a piece of code should work. A natural question to ask is: How can automatic bug-finding tools exploit the same natural abstractions? This thesis offers possible answers.

More specifically, I present three novel ways to exploit structure at three different steps of the software analysis process. First, I show how symbolic execution can preserve the data-flow dependencies of the original code while constructing compact symbolic representations of programs. Second, I propose

[1] For details, see [1].

structural abstraction, which exploits the structure preserved by the symbolic execution. Structural abstraction solves a long-standing open problem — scalable interprocedural path- and context-sensitive program analysis. Finally, I present an *automatic tuning* approach that exploits the fine-grained structural properties of software (namely, data- and control-dependency) for faster property checking. This novel approach resulted in a 500-fold speedup over the best previous techniques. Automatic tuning not only redefined the limits of automatic software analysis tools, but also has already found its way into other domains (like model checking), demonstrating the generality and applicability of this idea.

Table of Contents

List of Tables

List of Figures

List of Symbols

Abbreviations

List of Algorithms

Acknowledgments

I have been really fortunate to have Alan J. Hu as my Ph.D. advisor. Most of what I learned about how research is done and how the research community works can be attributed to him. His insightful comments and advice immensely improved this thesis.

I am extremely grateful to my committee members, Tom Ball and Gail Murphy, for their insightful suggestions for improving the content of this thesis. Encouraging me to do research on software verification, Tom has played a crucial role in the formulation of my thesis topic. His enthusiasm, wisdom, and knowledge have been a great source of inspiration for me. I also very much appreciate the directness and shrewdness of the career advice he has given me.

While I was a research intern at Microsoft Research (MSR), Rustan Leino gave me an unforgettable crash-course in extended static checking, which has impacted much of my later work. Sriram Rajamani provided invaluable suggestions and wise feedback on many of my early ideas for my thesis topic, both while I was at MSR and later at the Lipari Summer School. I can freely say that Sriram's comments saved me months (if not years!). To Madan Musuvathi, I owe my interest in bit-vector arithmetic — his genuine enthusiasm for research is simply contagious. Byron Cook's interest in my work and his encouragement have been a constant source of inspiration.

My internship at MSR created a number of long-lasting personal connections, which seem to multiply over time. Leonardo de Moura and Nikolaj Bjørner became close friends and their constructive criticism was often a guideline in deciding what constitutes a publishable result and what doesn't. Their incredible work on decision procedures really sets the research standard that I strive to achieve.

Zvonimir Rakamarić, a close friend and an honest critic, has spent countless hours listening my practice talks. Our discussions and his invaluable feedback have had significant influence on my research. All in all, I have been very fortunate to benefit from his knowledge and intuition. I would also like to thank Jesse Bingham, Shaun Xiushan Feng, Naghmeh Ghafari, and Flavio de Paula for their support, friendship, and valuable feedback on my research.

I benefited a lot from a close collaboration with Holger Hoos and Frank Hutter. It has been a joy to work and discuss research with them. Some of the material presented in this thesis is a joint work with the two of them. Under the influence of Holger's legendary time organization and efficiency, I became a better and more efficient researcher.

Many other people influenced my research and/or the presentation of the material in this thesis. Just to mention a few... The discussions I had with Aarti Gupta at various conferences kept reassuring me that I was on the right track. Ed Clarke's excitement and curiosity about my research, as well as the discussions I had with him and his group during my visit to Carnegie Mellon University, were extremely rewarding and motivating. I am also grateful to Bob Brayton, David Dill, Arie Gurfinkel, Sava Krstić, Alan Mishchenko, Ofer Strichman, and Karen Yorav for the career advice, encouragement, and research feedback that they provided me on multiple occasions.

My research has been supported by graduate fellowships from Microsoft Research and the University of British Columbia. The Natural Sciences and Engineering Research Council of Canada also supported my research indirectly (through my advisor). I am deeply grateful to all three organizations.

Finally, I would like to thank my wife for putting up with me on a long journey through all my academic degrees. All the way, she has been a reliable source of encouragement, support, and motivation.

Chapter 1

Introduction

1.1 Motivation

Error removal via verification and testing is one of the most time-consuming parts of the software development life cycle — for instance, the Windows XP testing team is larger than the development team [95]. Accordingly, an enormous range of techniques have been developed to support this task. Although existing techniques offer different scalability-precision-automation tradeoffs, programmers most often rely only upon debuggers, compiler type-checking, and ad-hoc testing. Techniques based on formal underpinnings are rarely used, especially those requiring a non-trivial amount of manual effort.

The core of the problem seems to be in the inadequate balance between scalability and precision of automatic software analysis techniques. The basic hypothesis that motivated this thesis was that a highly scalable (to several hundred thousand lines of code) interprocedural analysis can be conceived, without sacrificing precision, by exploiting the structure of the program code and customization of the entire tool chain. Experimental results (Chapter 6) not only confirm the hypothesis, but also suggest that with more research, the approach proposed in this thesis could be pushed further, perhaps even up to applications that have several million lines of code.

The two main ideas running through the thesis are exploitation of structure and customization. The motivation to exploit the structure of the problem came from my own observations of how programmers think about program code and how they manage to get large applications, which defy automatic analysis, almost correct. In different chapters, *structure* will represent a different set of properties of the problem being solved. In particular: symbolic execution (Chapter 3) preserves the original dataflow dependencies among variables and memory locations in the analyzed program and thus reflects the structure of the original code in the computed symbolic expressions; structural abstraction (Chapter 4) exploits those dependencies for precise, yet scalable, interprocedural analysis; and automatic tuning of decision procedures (Chapter 5) exploits the fine-grained properties of the code being checked. The motivation to customize the entire tool chain came from my previous experience with decision procedures — it is much harder to design a decision procedure that will perform well on a wide range of problems than a decision procedure that will perform exceptionally well on a specific problem. Design of decision procedures is a complex task that requires intimate knowledge of the problem being solved, understanding of a very wide range of algorithms, and engineering skills. So, why make that task even harder by requiring superb performance on a very broad set of problems? The main barrier that prevented this customization was the simple fact that there are many more classes of problems than there are skilled people that could design highly customized decision procedures. In Chapter 5, I show that a combination of light-weight manual customization and smart automatic customization can be very effective, giving up to two orders of magnitude speedup. Taken altogether, exploitation of structure and customization applied at all levels of software analysis give an unprecedented combination of precision and scalability.

1.2 Research Context

This section places my research in context of previous work (a much more detailed survey of the related work is given in Chapter 7). More precisely, the section discusses the role of abstraction and structure in software analysis (Section 1.2.1), situates my research in between two of the most important techniques for formal software analysis: model checking and static analysis (Section 1.2.2), and concludes by discussing the scalability-precision tradeoff, which is the key tradeoff of any verification technique based on formal underpinnings. By verification in this thesis, I mean checking that the program conforms to the specification provided in terms of asserted predicates (assertions) in code. Such assertions can be either written by programmers or inserted automatically.

1.2.1 Abstraction and Structure

Despite the increasing popularity of formal methods and significant progress made in the last two decades, the scalability of software formal verification and bug-finding tools is inadequate. The rule of the thumb for production compilers is that all algorithms used for code transformations and optimizations must be of almost linear complexity. Higher complexity algorithms are allowed only rarely and only for problems that are known to be localized and of small size in practice. In contrast, formal verification methods are notoriously slow. For instance, model checking [33] is linear with the size of the state space (e.g., [113]), which is exponential (or worse). Lighter-weight methods, like static checking, offer a broader range of precision-performance tradeoffs, although, in general, static analysis is known to be undecidable [85]. This negative result has a far more serious impact on verification than optimization. Sound and very imprecise analyses are often good enough for optimization, but produce an unacceptably high percentage of false positives if used for verification. More precise approaches typically do not even have polynomial computational complexity. Steadily increasing computing power certainly helps, but let us not forget that a 1000-fold increase in computing power is not that much if we are dealing with problems of exponential complexity.

It is hard to ignore such a huge complexity gap. However, programmers seem to be able to deal somehow with that complexity and write more-or-less usable code. How do they do that? Although no one has yet given a clear answer, it is obvious that abstraction and the structure of the code play an important role.

Abstraction has received well-deserved attention, and a large number of papers have been published on the topic. A seminal work on abstraction in the context of program analysis was done by Cousot and Cousot [39]. The abstract interpretation framework they invented forms a theoretical basis of essentially all software model checkers [112] and static checkers.

Code structure, on the other hand, has received much less attention. In particular, most research on exploiting structure has been focused on function summaries. Reps et al. [109] proposed summarization of functions, as natural abstraction boundaries, and reuse of summaries in different calling contexts. The SLAM software model checker [13] uses Binary Decision Diagrams (BDDs) [24] for the representation of function summaries. The Saturn static checker [135] also uses BDDs for summaries, but in a slightly different way. Summaries in Saturn represent the effect of the function on a single property that is being verified, rather than the predicate abstraction of the effect of the function on a set of variables related to the checked property. Recently, Conway et al. [38] have shown that summaries can be stored and reused for incremental verification of large bodies of code.

The focus of this thesis is on improving the scalability of software formal verification by exploiting the structure present in code. The presented work addresses two main bottlenecks: interprocedural path-sensitivity and decision procedures for proving the validity of logical formulas constructed from checked software and given properties (also known as Verification Conditions (VCs)). I show how the code structure can be used to mitigate the computational costs of those two bottlenecks.

1.2.2 Model Checking and Static Checking

A model checker is an automatic tool that does more or less brute force exploration of the state space (e.g., [33]). Arguably the most famous software model checkers are SLAM [13] and SPIN [76], which have both been applied to commercial code bases. SLAM is a symbolic model checker designed for the verification of Windows operating system device drivers, which typically have up to 10 thousand lines of code (KLOC). SPIN is a model checker for verification of distributed software systems.

SLAM verifies a finite abstraction, while SPIN verifies a finite description of the system in the Promela language. The finite approximation of infinite state software systems doesn't seem to be a serious hurdle for applicability of SLAM and SPIN — SLAM has found its way to driver developers by becoming a part of Microsoft Development Studio, and SPIN seems to be widely accepted too. The static checker CALYSTO[2], which I developed for the experimental evaluation of the innovations presented in this thesis, continues this tradition, and focuses on checking finite approximations of software.

Static checkers trade imprecision for scalability. A number of systems have been presented and used commercially. Astrée [22, 41] has been used for verification of some parts of Boeing's airplane control software. According to the reported results, Astrée scales up to 400 KLOC, but requires some code annotations from the programmer and a fairly high level of expertise. The systems verified by Astrée contain no `goto` statements, no dynamic memory allocation, no recursive calls, no recursive data structures, and no pointer arithmetic. These properties drastically simplify the requirements on the static checker. In contrast, CALYSTO is designed to be applicable to general-purpose software.

MC [57], on the other side of the spectrum, is a relatively imprecise, but very scalable (up to 4.7 million lines of code (MLOC)) static checker. The high rate of false positives is somewhat remedied by a bug probability rating that uses a number of heuristics to try to detect the most likely and the most critical bugs. CALYSTO, on the other hand, strives to eliminate as many false positives as possible through a very precise analysis.

Static checkers have some advantages over model checkers, the most important one being scalability. Static checkers are able to go over larger code bases and therefore usually find more bugs. Soundness and a proof of correctness

[2]The name is an intentionally misspelled version of Callisto, who is a nymph in Greek mythology. Callisto was a hunting companion of the goddess of the hunt — Artemis. The focus of the work in this thesis is bug finding, rather than the computation of proofs of correctness. Accordingly, CALYSTO is a bug hunting companion.

that the model checkers can give (as opposed to static checkers which are often unsound and cannot give a proof of correctness) seem to be of little value in mainstream software development where the optimization function is to cram as much functionality into an application as cheaply, as quickly, and as correctly as possible.

As the emphasis of the presented work is scalable software verification, static checking was a natural path to choose. More precisely, the work focuses on extended static checking [50], which is a combination of static checking and decision procedures targeted at achieving a practical balance between the precision of model checking and scalability of static checking.

1.2.3 Scalability and Precision

The focus of this thesis is on a scalable, yet precise, combination of static checking with automatic theorem proving for checking properties of general-purpose software. The presented approach is essentially language-independent, but targeted towards imperative procedural languages.

The proposed analysis is general enough to handle user-provided assertions. The assertions are essentially predicates expressible in the programming language in which the checked application is written and can be either provided by the programmer or automatically inserted in the code.

In order to achieve a high level of precision (and therefore a low false-positive rate), the analysis is field-, path-, and context-sensitive. The computational complexities of the last two properties are fairly high (even intraprocedural path-sensitivity for Boolean programs is $\mathcal{P}$-space complete, while context-sensitivity is exponential[3]). In addition, path-sensitive analyses require some sort of a decision procedure for checking feasibility of paths. The decision procedures suitable for precise software verification are computationally expensive.

The path-sensitivity, context-sensitivity, and decision procedures are the

[3] k-CFA (Control Flow Analysis), where k is the maximum call depth in the original call graph that the analysis tracks has exponential complexity for $k > 0$ [81], [101, pages 187–197].

main performance bottlenecks for precise static analyses. The work presented in this thesis takes a holistic approach, considers their interactions, and proposes innovative solutions based on exploitation of the structure present in the software:

Path-Sensitivity. Chapter 3 shows how local structure in the Control Flow Graphs (CFGs) can be used to generate path-sensitive summaries and VCs in linear time and space.

Context-Sensitivity. Chapter 4 proposes *structural abstraction* — an abstraction for achieving interprocedural path-sensitivity and context-sensitivity. Structural abstraction is based on symbolic relational summaries [40], and exploits the structure of software at the function level. The chapter also discusses the tradeoffs between the precision of summaries and the cost of interprocedural analysis.

Decision Procedure. Chapter 5 focuses on a bit-vector arithmetic decision procedure designed especially for checking the validity of VCs. Software VCs have specific low-level structural properties that can be exploited by customizing the decision procedure. Such customization speeds up the main analysis by two orders of magnitude.

To summarize, the thesis demonstrates how to exploit the structure present in:

1. the CFG — for efficient generation of compact verification conditions,
2. the Call Graph (CG) — for mitigating the cost of context-sensitivity and achieving interprocedural path-sensitivity, and
3. within the VCs themselves — for efficient path-sensitivity, and scalable validity checking.

1.3 Contributions

This section outlines the main contributions of the thesis. The contributions are mostly based on exploitation of the structure of checked programs at various levels of granularity.

Chapter 3: Structure-Preserving Symbolic Execution. One of the main contributions of this thesis is a novel structure-preserving symbolic execution, which maintains the program dataflow dependencies (structure) in the computed intraprocedural summaries. That structure is later exploited during the interprocedural analysis. The proposed symbolic execution algorithm computes fully path-sensitive summaries for each side-effect[4] of each function. Besides summaries, the symbolic execution algorithm also computes partial VCs, which are assembled into fully interprocedurally path-sensitive VCs later during the interprocedural analysis. The highlights of this novel intraprocedural symbolic execution algorithm are the following:

- The data-flow dependencies in the original program are reflected in the computed function summaries and VCs. Structural abstraction (Chapter 4) exploits this structure for selecting the parts of the formula that need to be refined to refute spurious counterexamples.

- The computed symbolic expressions are compact and immutable. Immutability, in practice, means faster symbolic execution that is less memory-demanding.

- The algorithm exploits the structure of the CFG — more precisely the (post)dominance relation — to avoid expensive recursive memory lookups.

Chapter 4: Structural Abstraction and Refinement. Scalable interprocedural path- and context-sensitive analysis has been a long-standing open

[4] In this thesis, both the returned value and modifications of the caller's state will be considered as side-effects.

problem. Structural abstraction, one of the main contributions of this thesis, is an effective and novel solution. The proposed interprocedural analysis uses a decision procedure to lazily assemble function summaries into formulas that represent the checked assertion predicate and the conditions under which that assertion is reachable (i.e., global control-flow context). This assembly is done in a lazy manner through structural abstraction and refinement, which both exploit the dataflow dependencies (preserved by the above mentioned symbolic execution) in the original code. The most important aspects of the proposed approach are:

- Structural abstraction is the first completely incremental abstraction framework for interprocedural analysis — when analyzing a single VC, the decision procedure does not need to repeat the same work.

- Since the proposed symbolic execution computes summaries for each side-effect, structural refinement can be more precise and, instead of inlining the whole function call, can inline only the needed side-effect.

- While abstraction at function interfaces has been done in various forms before (e.g., [126]), structural refinement is, to the best of my knowledge, the first proposed refinement that exploits the dataflow dependencies (structure) in code for refining abstracted VCs.

  Structural refinement is computationally cheaper than other previously proposed alternatives. More precisely, structural refinement identifies missing subexpressions in time that is linear with the size of the abstract counterexample, while predicate abstraction, for instance, has to solve an $\mathcal{NP}$-complete problem to compute new predicates for refinement.

- The core insight of structural abstraction is that decision procedures can be used to effectively perform precise interprocedural analysis. While this insight was exploited before for interprocedural pointer alias analysis [132] (BDDs can also be seen as a decision procedure), it hasn't been exploited for analysis of program functionality.

Chapter 5: Customized Bit-Vector Arithmetic Decision Procedure. Proving the logical validity of interprocedurally path-sensitive and bit-precise VCs is computationally very challenging, and requires a high-performance customized decision procedure. Chapter 5 presents my custom-made decision procedure for VC-solving, named SPEAR.[5] I achieved customization through the design of the decision procedure, simplifications that I built in, and through automatic tuning. To the best of my knowledge, no prior work has been published on automatic tuning of decision procedures. Another important contribution of my work on SPEAR is a practical proof that a decision procedure based on a Boolean satisfiability (SAT) solver can be highly effective for solving bit-vector arithmetic problems.[6]

Chapter 6: Prototype and Experimental Evaluation. From a practical perspective, one contribution of this thesis, especially important for readers who wish to implement structure-preserving symbolic execution and structural abstraction, is the description of CALYSTO's architecture in Chapter 6. The chapter also provides valuable experimental results that show that the combination of the techniques in CALYSTO and SPEAR is highly effective in finding bugs on large real-world applications.

1.4 Organization of the Thesis

The background material required for understanding the thesis is presented in Chapter 2. Chapter 3 presents the intraprocedural structure-preserving symbolic execution for computation of function summaries and VCs. The presented symbolic execution constructs summaries and VCs in the form of maximally-shared graphs, which preserve the data-flow dependencies in the original code. Chapter 4 presents structural abstraction, which uses a decision procedure to perform interprocedural VC-checking, abstracting away irrelevant side-effects

[5] Callisto, a hunting companion of the goddess of the hunt, uses spears for hunting.

[6] SPEAR won the bit-vector division of the Satisfiability Modulo Theories (SMT) 2007 competition.

of function calls. The work presented in Chapter 4 appeared in [8]. Chapter 5 dives into the details of SPEAR— my bit-vector decision procedure for software verification. The material presented in that chapter is rooted in several of my publications on decision procedures [6, 10, 11, 78], but focuses on my insights, conclusions, and results that are relevant to software analysis. Chapter 6 is based on [9] and presents a prototype implementation — the CALYSTO extended static checker — of the analyses presented in the previous chapters. The same chapter also gives experimental results obtained by checking a number of real-world applications. Related work is surveyed in Chapter 7. Finally, the last chapter gives conclusions and discusses future work.

Chapter 2

Background

This chapter introduces a number of concepts that will be routinely used later. Section 2.1 introduces less standard notations that will be used throughout the thesis. Section 2.2 introduces VCs and the underlying logic. Section 2.3 provides some background on program analysis and compilers. The final section overviews the Low Level Virtual Machine (LLVM) compilation framework used for implementation of the CALYSTO static checker. The section on LLVM focuses on the details of LLVM that are of special importance for verification.

2.1 Notational Conventions

Throughout this thesis, the conditional selection if-then-else operator will be denoted by ITE. More formally:

$$\mathtt{Var} = ITE(\Psi, Term_T, Term_F)$$
$$\equiv$$
$$(\Psi \Rightarrow \mathtt{Var} = Term_T) \wedge (\neg\Psi \Rightarrow \mathtt{Var} = Term_F)$$

Some other notation used throughout the thesis includes the following. Frequently, two objects appear together for logical or practical reasons. Pairs will be denoted by $\langle a, b\rangle$. The finite set of variable names in a logical expression ψ is denoted by $\mathsf{supp}\,(\psi)$. For example, $\mathsf{supp}\,(x \cdot y + z > 0) = \{x, y, z\}$. The operator $\phi\,[a \leftarrow b]$ is the substitution operator that substitutes all free instances of a in ϕ with b. Parallel substitution will be denoted by $[range : x \leftarrow y]$ — every element x for which the $range$ predicate evaluates to $\mathtt{true}$ is substituted with y. The parallel substitution operator performs all substitutions at once. Hence, if

y contains x, the substitution will be performed only once. Equality for terms is denoted by $=$, equality for logical expressions as $\equiv$, the assignment operator as $:=$, and the grammar production as $::=$. Existential quantification will be denoted by $\exists$; universal, by $\forall$; and $\exists!$ will stand for "exists a unique."

2.2 Verification Conditions

This section begins by introducing the notion of Verification Conditions (VCs), and proceeds by describing the underlying logic that will be used in this thesis for reasoning about VCs. An example at the end of the section shows a VC constructed from a simple piece of code.

VCs are logical formulas, constructed from a system and desired correctness properties, such that the validity[7] of VCs corresponds to the correctness of the system. Commonly, correctness properties in programs are specified with assertions, which can be either written by programmers, or automatically generated (e.g., [15]). If all the assertions in the program are valid, the program is considered to be correct with respect to the given set of assertions.

The VCs in this thesis are quantifier-free First-Order Logic (FOL) formulas over finite structures (Fig. 2.1), namely bit-vectors. The used logic enables us to precisely model both hardware and software, which commonly use bit-vectors to represent data, including boundary behaviors (over- and under-flows), bit-wise operators (AND, XOR,...), and non-linear operators (multiplication, division,...). The finite nature of the logic also makes it decidable (whereas FOL is in general undecidable [30]). More precisely, checking the validity of a formula in the given logic is $\mathcal{NP}$-complete [103].

Any formula in Finite Fragment of First-Order Logic (F3OL) can be represented as a Boolean logic circuit, because `Var` (resp. `Constant`) represents finite

[7] A logical formula ϕ is said to be satisfiable if there exists a partial or complete assignment to its variables, such that the formula is `true`. Otherwise, it is said to be unsatisfiable. A logical formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable, written as $\vDash \phi$.

$$
\begin{array}{rcl}
\mathit{Formula} & ::= & \neg \mathit{Formula} \mid (\mathit{Formula}) \mid \mathit{Atom} \mid \mathit{Formula} \Rightarrow \mathit{Formula} \\
 & \mid & \mathit{Formula} \wedge \mathit{Formula} \mid \mathit{Formula} \vee \mathit{Formula} \\
 & \mid & \mathit{Formula} \equiv \mathit{Formula} \mid \mathit{ite}(\mathit{Formula}, \mathit{Formula}, \mathit{Formula}) \\
\mathit{Atom} & ::= & \texttt{BoolConstant} \mid \texttt{BoolVar} \mid \mathit{Term}\ \mathit{Rel}\ \mathit{Term} \\
\mathit{Term} & ::= & \texttt{Var} \mid \texttt{Constant} \mid \mathit{UnaryOp}\ \mathit{Term} \mid \mathit{Term}\ \mathit{BinaryOp}\ \mathit{Term} \\
 & \mid & \mathit{ITE}(\mathit{Formula}, \mathit{Term}, \mathit{Term}) \\
\mathit{Rel} & ::= & = \mid \neq \mid <_u \mid \leq_u \mid >_u \mid \geq_u \mid <_s \mid \leq_s \mid >_s \mid \geq_s \\
\mathit{UnaryOp} & ::= & \neg \mid \mathit{ZeroExtend} \mid \mathit{SignExtend} \mid \mathit{Truncate} \\
\mathit{BinaryOp} & ::= & + \mid - \mid * \mid \div_u \mid \div_s \mid \mathit{Rem}_u \mid \mathit{Rem}_s \mid \wedge \mid \vee \mid \mathit{Shl} \mid \mathit{Ashr} \mid \mathit{Lshr}
\end{array}
$$

Figure 2.1: Grammar of the F3OL.

integers (bit-vectors) from the set $\mathbb{Z}_n$[8] and `BoolVar` (resp. `BoolConstant`) represents Boolean variables (of type $\mathbb{B}$). Such a circuit can be translated into a Conjunctive Normal Form (CNF) formula over Boolean variables by Tseitin's transform [128] in linear time and by introducing at most a linear number (linear with the size of the circuit) of fresh variables. Such a translation is usually called *bit-blasting*.

The formulas from F3OL can be constructed from the grammar in Fig. 2.1. All the operators are pretty much standard. The operator $\neg$ (resp. $\wedge$, $\vee$) is overloaded to represent both logical and bit-wise NOT (resp. AND, OR). The left side of an implication $\alpha \Rightarrow \beta$ is known as the antecedent (α), while the right side is called the consequent (β). Equality of formulas is denoted by $\equiv$. Unary relations include zero (resp. sign) extension, which extends a bit-vector by zero (resp. its most significant bit). Truncation discards a number of most significant bits so that the operand fits into the type of the result. Binary relations over terms are standard equality ($=$), disequality ($\neq$), as well as unsigned and signed comparisons. Binary operators include standard addition ($+$), subtraction ($-$), multiplication ($*$), unsigned and signed division ($\div_u$ and $\div_s$), unsigned and signed remainder (Rem_u and Rem_s), bit-wise AND (resp. OR), and shifts (left,

[8] The set of integers modulo n.

arithmetic right, logical right). Two versions of the `if-then-else` operators will be distinguished: The $ite(\Psi_c, \Psi_a, \Psi_b)$ operator for formulas is syntactic sugar for $(\Psi_c \wedge \Psi_a) \vee (\neg\Psi_c \wedge \Psi_b)$, whereas the second is the operator for terms: $ITE(\Psi, Term_a, Term_b)$, is equal to $Term_a$ if $\Psi \equiv \mathsf{true}$, and $Term_b$ otherwise.

Example 1:

The following example illustrates a VC computed from some code and a checked property. Let us assume that we want to check the validity of the asserted condition on the last line of:

```
x0 = a + b;
if (x0 < 0) {
    x1 = -y0;
    y1 = y0;
} else {
    x1 = x0;
    y1 = y0 + x0;
}
assert(0 <= y1);
```

Using either symbolic execution [82] or the weakest precondition predicate transformer [52], one can compute the following VC:

$$0 \leq ITE(a + b < 0, y_0, y_0 + a + b)$$

The computed verification condition is not valid. One possible falsifying assignment is $a = 0, b = -1, y_0 = -1$. Thus, the asserted condition does not hold and either the program or the assertion has to be fixed.

2.3 Program Analysis

Many concepts that originated in early work on optimizing compilers are useful in software verification. After introducing the basic concepts, this section proceeds with some elementary data structures used for software analysis and transformations. Section 2.3.2 describes the commonly used static single assignment forms that have played a major role in the development of compilers and also serve as basic data structures used in this thesis. The subsequent section introduces the problem of aliasing in software analysis. The last section reviews several approaches to interprocedural analysis by summarization.

2.3.1 Basic Definitions

Control Flow Graph (CFG) and Related Concepts. A basic block consists of a sequence of program statements and a single branch or return statement at the end. The flow of control enters the block only at its beginning and leaves it by execution of the branch or return at its end. A basic block B can be split into several basic blocks $B_0, \ldots, B_n$ connected with unconditional branch statements.

Above the basic-block level, functions are the next higher organizational unit. Each function can be represented with a single connected, directed graph $G = (N, E, \texttt{Entry}, \texttt{Exit})$, called the Control Flow Graph. A function consists of a set of basic blocks $\mathcal{B}$, which are represented by the nodes N in the graph. If there is a branch from some basic block B_1 to B_2, those two are connected by an edge in the CFG. The set of edges is denoted by E. The start node $\texttt{Entry} \in N$ has no incoming edges, while the end node $\texttt{Exit} \in N$ has no outgoing edges. An example of a CFG is given in Fig.2.2. It will be assumed that G has a single `Exit` node. This is not a serious constraint — all return nodes can be easily unified into a single node. We will assume that G contains no blocks unreachable from `Entry`. Unreachable blocks can never be executed and therefore can be removed without having any effect on the outcome of the execution of the function. A point of definition of a variable will be known simply as *definition*, and the points at which it is read will be known as *uses*.

If there is an edge connecting two nodes, $x \longrightarrow y \in E$, then x and y are called the source and the destination of the edge. A path of length n is a sequence of edges $x_0 \longrightarrow x_1 \longrightarrow \cdots \longrightarrow x_n$, where $x_i \longrightarrow x_{i+1} \in E$. If $x_0 = x_n$, the path

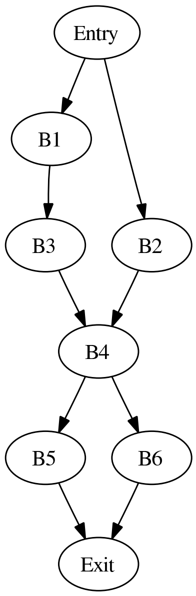


Figure 2.2: CFG Example and Parental Relations. Some examples of the parental relations are: $Pred(B_4) = \{B_2, B_3\}$ (predecessors), $Succ(B_4) = \{B_5, B_6\}$ (successors), $Ancs(B_3) = \{\texttt{Entry}, B_1, B_3\}$ (ancestors), $PropAncs(B_3) = \{\texttt{Entry}, B_1\}$ (proper ancestors).

is a cycle. In the context of the analysis presented in later chapters, CFGs are going to be acyclic. Section 3.1.3 on page 38 explains how the analysis can be extended to cyclic CFGs. Paths of length zero or more are denoted by $x \xrightarrow{*} y$ and of length one or more as $x \xrightarrow{+} y$. Given two basic blocks B_1, B_2 such that $B_1 \longrightarrow B_2$, we say that B_1 is a predecessor of B_2 ($B_1 \in Pred(B_2)$) and B_2 is a successor of B_1 (denoted $B_2 \in Succ(B_1)$). If $B_1 \xrightarrow{*} B_2$, then B_1 is an ancestor of B_2 ($B_1 \in Ancs(B_2)$ or $B_1 \preceq B_2$) and B_2 is a descendant of B_1 (denoted $B_2 \in Desc(B_1)$). If the path is of length one or more, we qualify the parental relation with the word "proper", written as $B_1 \in PropDesc(B_2)$ or $B_1 \prec B_2$ (resp. $B_2 \in PropAncs(B_1)$).

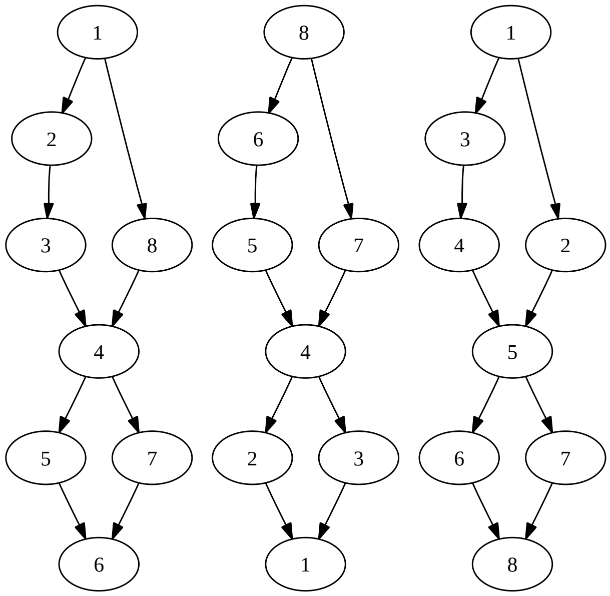
Two basic blocks B_1 and B_2, such that $B_1 \npreceq B_2 \wedge B_2 \npreceq B_1$, are called siblings. A basic block with more than one successor is called a branch node, and a node with more than one predecessor is called a join node. Basic blocks will be frequently identified by labels (for example, `ReturnLabel`), and the two notations will be used interchangeably.

Two basic types of traversal, preorder and postorder, are frequently used in the analysis of CFGs. Both types of traversal can be defined in terms of a depth-first search that marks each newly discovered node as visited (avoiding revisiting the already marked nodes) and assigns a unique consecutive number to each node. The preorder (resp. postorder) traversal assigns numbers to nodes before (resp. after) recursing on their unmarked descendants. The preorder (resp. postorder) sequence number of a node B will be denoted by $Pre[B]$ (resp. $Post[B]$). By using preorder and postorder sequence numbers, ancestry and descendancy queries in a CFG can be answered in constant time [72]. Let $B_0, B_1, \ldots, B_n$ be a sequence of nodes in the postorder traversal. The reverse-postorder traversal is defined as the inverted sequence $B_n, \ldots, B_1, B_0$. An important property of reverse-postorder is that all the predecessors of a join node are visited before the node itself. This property will be used later for computation of VCs.

A program is considered to be structured if it is composed only of so called structured constructs, like `if-then-else`, `do-while`, and statement sequences. The CFG of a structured program has the following property: every subgraph that represents an `if-then-else` or a loop construct has a single point of entry. Furthermore, all loops in a structured program are strictly nested.

Dominance Relations. A node B_i dominates node B_j if and only if all the paths from the `Entry` node to B_j go through B_i, written as $B_i \underline{\gg} B_j$. If $B_i \neq B_j$, B_i strictly dominates B_j, denoted by $B_i \gg B_j$. A node B_j postdominates node B_i if and only if all the paths from B_i to the `Exit` node go through B_j, written as $B_j \underline{>} B_i$.[9] If $B_i \neq B_j$, B_j strictly postdominates B_i, denoted by $B_j > B_i$.

[9] The pointed side of the postdominance relation symbol points to the ancestor in $\mathcal{B}$. The reverse holds for the dominance relation symbol. It seemed natural to turn the pointed side of $\underline{>}$ symbol to the basic block that is being postdominated. Essentially, the postdominance



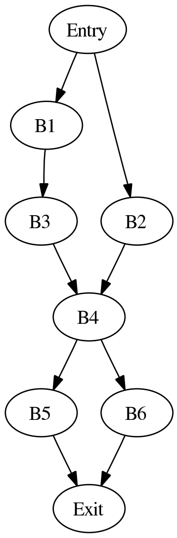
(a) Preorder. (b) Postorder. (c) Rev. Postorder.

Figure 2.3: Traversal Orders.

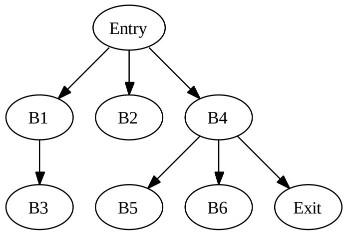
A node B_i is an immediate dominator of B_j $(idom(B_j))$ if $B_i \gg B_j$ and $\neg\exists B_k \in \mathcal{B} : B_i \gg B_k \gg B_j$. A nearest common dominator of two nodes, $ncdom(B_i, B_j)$, is a node B_k such that $B_k \gg B_i \wedge B_k \gg B_j \wedge \neg\exists B_h \in \mathcal{B} : (B_h \gg B_i \wedge B_h \gg B_j \wedge B_k \gg B_h)$. An immediate postdominator and a nearest common postdominator are denoted by $ipostdom(B_i)$ and $ncpostdom(B_i, B_j)$ respectively. The Dominance Frontier (DF) of a node B, denoted by $DF(B)$, is a set of nodes $\{B_x \mid \exists B_i \in Pred(B_x) : B \gg B_i \wedge B \not\gg B_x\}$. The DF naturally extends to a set $\mathcal{S} \subseteq \mathcal{B}$ of basic blocks: $DF(\mathcal{S}) = \cup_{B_i \in \mathcal{S}} DF(B_i)$. The Iterated Dominance Frontier (IDF), denoted by $DF(\mathcal{S})^+$, is defined as the limit of the increasing sequence of sets of nodes:

$$DF(\mathcal{S})^{i+1} = DF\left(\mathcal{S} \cup DF(\mathcal{S})^i\right)$$

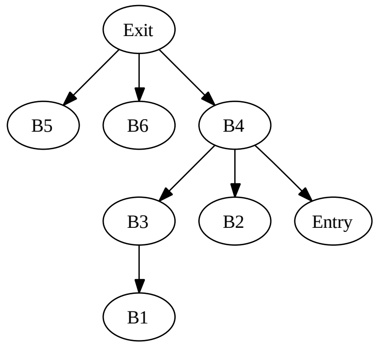
relation is a dual of the dominance relation. If $B_1 \underline{\gg} B_2$ in the direct graph, $B_1 \underline{\gg} B_2$ in the reverse graph.



(a) CFG.



(b) DT.



(c) PDT.

Figure 2.4: Dominance Relations. Some examples of the dominance relations are: `Entry` $\gg B_4$ (dominance), $idom\,(\texttt{Exit}) = B_4$ (immediate dominator), $B_4 \gtrdot B_2$ (postdominance), $ncdom\,(B_5, B_2) = \texttt{Entry}$ (nearest common dominator), $DF\,(B_1) = \{B_4\}$ (dominance frontier).

Basic dominance relations are illustrated in Fig. 2.4.

Dominator Tree (DT). The dominance relation [104] is a partial order (reflexive, antisymmetric, and transitive). Furthermore, the set of dominators of a node are linearly ordered by the dominance relation [102]. Thus, the dominance relation can be represented by a tree, called the Dominator Tree (resp. Postdominator Tree (PDT) for the postdominance relation). Dominator trees can be computed in $\mathcal{O}(E \cdot \alpha(E, N))$ time [91], where α is the extremely slowly growing inverse of Ackermann's function. An example of a DT is given in Fig. 2.4(b), and an example of a PDT is given in Fig. 2.4(c).

The dominator tree level of a basic block B, denoted by $dtl[B]$, is the breadth-first-search level of the corresponding node in the DT. The root node of the dominator tree has level 1, its children 2, and so on.

Call Graph (CG). Programs are composed of one or more functions. A call graph is a directed graph $G_f = (N_f, E_f)$, such that each node $f \in N_f$ represents a single function and each edge represents a call. Let $f_i \longrightarrow f_j \in E_f$, then f_i is the caller, and f_j the callee. Call graphs might be cyclic and can have multiple roots. For languages that support function pointers, a simple pointer analysis can be used for call graph construction. It has been shown that even a very simple field-sensitive version of Steensgaard's [123] points-to analysis works very well for call graph construction [96]. For data memory locations, on the contrary, cheap analyses tend to be very imprecise, implying that manipulations of data pointers are much more complex than manipulations of function pointers in practice.

Abstract Syntax Tree (AST). Although not directly used in the work presented in this thesis, ASTs have been commonly used as a basic code representation in several related works, as discussed in Chapter 7. An AST is a data structure that represents a parsed program. Each node represents an operator or statement and its children represent the operands [4]. Code transformations and analyses are commonly performed on simpler lower-level representations, like CFGs and Static Single Assignment (SSA), which are also more language-independent than ASTs.

2.3.2 Single Assignment Forms

Single assignment forms are intermediate representations used in compilers for more efficient optimizations and transformations of the program code. A number of single assignment forms have been proposed. Their common property is that local variables have a single point of definition. This is achieved by introducing a fresh variable name for each definition and propagating the name to the uses dominated by the definition. Merges of values are handled through special functions, which in most cases appear at the beginning of each join block.

Static Single Assignment (SSA). The SSA form [110] is a directed flow graph in which each definition defines a distinct name and each use refers to a single definition. In addition, each definition dominates all uses. At join points, the names are merged by the use of ϕ-functions, which take as arguments a number of definitions. One of the definitions is chosen depending on the predecessor node from which the flow of control enters the join node. However, the ϕ-functions have no special arguments that signal where the flow of control came from. So, strictly speaking, ϕ-functions are not really functions, but a non-deterministic choice operator. If the exact definition matters, the analysis that relies on the ϕ-functions has to track where the flow of control came from, and pick the appropriate definition. All ϕ functions are considered to be executed instantaneously in parallel at the beginning of the block.

The first efficient (quadratic worst case, linear in practice) algorithm for computing SSA was proposed by Cytron et al. [43]. Sreedhar and Gao have more recently proposed a simple and elegant linear algorithm [122].

SSA handles pointers and globals in a slightly unexpected way. All globals are accessed through constant pointers, while all other pointers are handled as local variables. For instance:

```
int *a = malloc(...);
int *b = malloc(...);
int *ptr;
if (x) {
    ptr = a;
} else {
    ptr = b;
}
```

is in SSA represented as $ptr = \phi(a, b)$. This is nowadays a common approach, which has evolved from the Hashed SSA form proposed by Chow et al. [29]. The approach is commonly found in modern compilers, like the GNU C compiler (GCC) and the LLVM compilation framework [86]. Because of this feature, all pointers (not the abstract memory locations to which they may point!) have unique points of definition. At this point, I introduce the concept of a terminal pointer:

Definition 1 (Terminal Pointer)**.** *A terminal pointer is a constant pointer to a global, a formal pointer parameter, or a pointer initialized with some memory allocation function (like* `alloca()`, `malloc()`, `realloc()`,`...`*).*

For instance, the symbolic definition of the *ptr* pointer in the above given example would be $ITE(x, a, b)$, where a and b are terminal pointers. Note that the definition of *ptr* would be the same if `int *ptr;` were replaced with the `int *ptr = malloc(...);` statement.

The intraprocedural symbolic execution algorithm for function summarization presented in Chapter 3 defines all pointers in terms of terminal pointers, by performing definition table lookups, and always replacing *uses* with definitions. The replacement is simplified by the unique point of definition property of SSA.

Gated Single Assignment (GSA). SSA is not precise enough for verification purposes. First, the ϕ-functions are not really functions. Second, the ϕ-functions contain no information about the conditions under which a certain definition is chosen. As a result, the GSA form will be used in this thesis.

GSA form was first introduced by Ballance et al. [17]. GSA is an appropriate intermediate representation for a number of code transformations [129]. An efficient algorithm (running in $\mathcal{O}(E \cdot \alpha(E, N))$ time) for GSA computation was proposed by Tu and Padua [129]. GSA extends SSA with gating functions that replace the ϕ-functions. Besides the definitions, the gating function also contains conditions under which the flow of control reaches the block in which the function is placed. Tu and Padua [129] define three types of gating functions:

- For each definition, the γ-function represents the condition under which that definition reaches a join node, and it directly replaces the ϕ-function in SSA. For example, the following piece of code

```
if (c1) {
    x = 1;
} else {
    if (c2) {
        x = y;
    } else {
        x = z;
    }
}
```

is translated to $x_1 = \gamma\,(c_1, 1, \gamma\,(c_2, y, z))$.

- The μ-functions are inserted in the loop headers. These functions choose between the initial and the loop-carried definitions.
- The η-function is inserted at loop exits, and it captures the flow of definitions from within the loop to subsequent uses [17].

The static analysis proposed in this thesis unrolls loops once and terminates them with an assumption that the loop test has failed, as in ESC/Java [63]. Consequently, μ and η functions become unnecessary. The γ-function is more formally defined as:

Definition 2 (γ-Function [129])**.**
A γ-function for a block B is recursively defined by the following grammar:

$$\Gamma \quad ::= \quad \gamma\,(Cond, \Gamma, \Gamma) \mid Object \mid \emptyset$$

In the grammar, Cond is a Boolean variable. The object can be of an arbitrary type. The second and the third operand of the γ-function must be of the same type. An infeasible path is represented as $\emptyset$.

As the γ-function is semantically equivalent to the `if-then-else` construct, it will be interpreted either as a term *ITE* or a logical formula *ite*, depending on the context. The terminal symbols in the given grammar of the γ-functions will be simply known as terminals. Terminals in the standard γ-functions are variable definitions. In the previously given example, the terminals are $\{1, y, z\}$. All terminals are of the same type.

Gating paths are another important concept related to GSA, and they will be heavily used for the construction of VCs:

Definition 3 (Gating Path [129]). *A gating path of a node B is a path in the CFG from $idom\,(B)$ to B, such that every node on the path is dominated by $idom\,(B)$.*

In the context of the analysis presented in later chapters, CFGs are going to be acyclic, and therefore a gating path is just a path $idom\,(B) \xrightarrow{+} B$. Definition 3 is written in a more general way to allow cycles.

To represent a set of gating paths, we are going to introduce an expression similar to γ-functions that has only basic-block labels as terminals. These are known as gating path expressions, and will be denoted with an overline ($\overline{\gamma}$), as a mnemonic for the *path* expression.

Definition 4 (Gating Path Expression).
Let G be an acyclic CFG. A gating path expression $GP\,(B)$ of a basic block $B \in G$ is an expression that tracks the conditions that select which $B_p \in Pred(B)$ would precede B on a path $idom\,(B) \xrightarrow{*} B_p \longrightarrow B$. It is defined recursively as:

$$\Gamma \quad ::= \quad \overline{\gamma}\,(Cond, \Gamma, \Gamma) \mid B_p \mid \emptyset$$

where $B_p \in Pred()$. As a special case, $GP\,(\texttt{Entry}) = \texttt{Entry}$.

Intuitively, one can think of the conditions *Cond* as switches that determine which path is taken from $idom\,(B)$ to $B_p \in Pred(B)$.

Gating path expressions for a given CFG can be easily canonicalized by simplifying expressions of the form $\overline{\gamma}\,(c, B_i, B_i)$ to B_i. As each unconditional

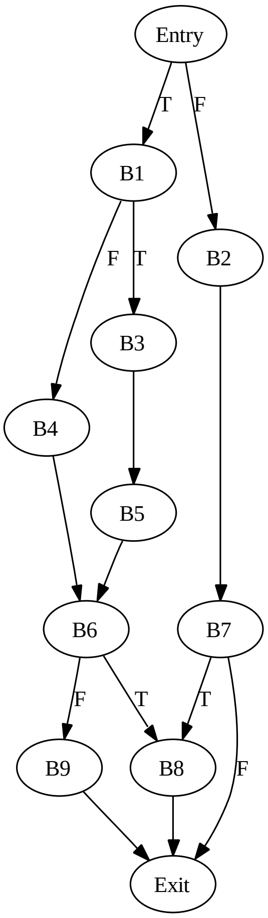


Figure 2.5: Gating Path Expression Examples. Only blocks `Entry`, B_1, B_6, and B_7 are terminated with conditional branch statements. Let us denote the corresponding branch conditions as $c_{\texttt{Entry}}$, c_{B_1}, c_{B_6}, and c_{B_7}. Assume that the left branch is taken if the condition is `true`, and the right branch otherwise. Some examples of the gating path expressions are: $GP\,(B_1) = \texttt{Entry}$, $GP\,(B_6) = \overline{\gamma}\,(c_{B_1}, B_5, B_4)$, $GP\,(B_8) = \overline{\gamma}\left(c_{\texttt{Entry}}, B_6, B_7\right)$, and $GP\,(\texttt{Exit}) = \overline{\gamma}\left(c_{\texttt{Entry}}, \overline{\gamma}\,(c_{B_6}, B_9, B_8)\,, \overline{\gamma}\,(c_{B_7}, B_8, B_7)\right)$.

branch statement *branch* B_s can be seen as a special case of a conditional branch statement $branch(*, B_s, B_s)$, the same simplification rule applies to basic blocks terminated by unconditional branches. Consequentially, given a canonicalized $GP\,(B)$, the basic blocks $B_i \in Ancs(B)$ terminated by unconditional branches will never be in $\mathsf{supp}\,(GP\,(B))$, unless $B_i \in Pred(B)$.

In Chapter 3, we shall perform operations on these gating path expressions (illustrated in Fig. 2.5) to find definitions of variables and abstract memory locations quickly.

Memory Access. The work presented in this thesis is based on the single assignment forms. SSA and GSA handle pointers, local variables, and globals in a specific way. To avoid ambiguity, I provide the definitions of some quite standard notions that might have a slightly unexpected meaning in this thesis:

- A local variable denotes a location on the stack that holds a value of an arbitrary type, it is visible only within a single function, and it is immutable (because it has a single point of definition). If a function passes its local variable as a parameter to another function, the value of the local variable is copied to the context of the callee.

- A pointer is a local variable that holds an address of a memory location. Pointers inherit all the properties of local variables. In addition, pointers can be dereferenced. The memory location to which a pointer points is being written (resp. read) if the dereferenced pointer is on the left (resp. right) side of the assignment operator.

- An abstract memory location is a piece of memory to which one or more pointers can point. Abstract memory locations can be globally visible, and can be modified at multiple points in the program, even in different functions.

- A global variable is an abstract memory location. All functions that access a global variable have to access it by dereferencing a pointer. Compilers usually keep pointers to global variables in the symbol table. To simplify the formalization. I assume that all pointers are passed as parameters to functions. CALYSTO handles global variables by performing symbol table look-ups.

2.3.3 Alias Analysis

Aliasing in software occurs when two or more names in the program refer to a single memory location, either on the heap or on the stack. For example, given this code:

```
x = 17;
*y = b;
z = x;
```

the constant propagation optimization can assign 17 to z if and only if y does not alias x.

Pointer analysis answers whether two pointers can refer to the same memory location, while points-to analysis returns a set of abstract memory locations to which a single pointer can point. In general, both analyses are undecidable [85, 106].

This negative result led to a wide range of algorithms with different precision-performance tradeoffs. The plethora of alias analysis algorithms that has been proposed can be classified according to the following properties:

Context-Sensitivity. Alias analyses are mostly interprocedural. If the analysis differentiates the contexts from which a function is called, it is said to be context-sensitive. Otherwise, it's context-insensitive. The cost of context-sensitive analysis depends on the used abstract domain. Context-sensitive algorithms that handle bit-vector operations precisely have exponential worst-case run times.

Flow-Sensitivity. If an analysis takes into account the flow of control between statements in a function, it is said to be flow-sensitive. Flow-insensitive analyses treat the entire function as a set of statements ignoring the flow.

Path-Sensitivity. An analysis that considers the exact path along which a certain statement is executed is known as path-sensitive. For example, given the program:

```
x = 7;
if (c) {
    *y = a;
}
...
if (!c) {
    z = x; /* Block B */
}
```

a path-sensitive analysis can infer that $z = 7$ in block B even if y aliases x. In contrast, a path-insensitive variant must conservatively assume that writing to y can modify the value of x if the two may alias. Path-sensitivity is an expensive property, and path-sensitive analyses typically require some sort of a decision procedure to distinguish feasible from infeasible paths. An analysis that is path-sensitive is also necessarily flow-sensitive.

Field-Sensitivity. Field-sensitive analyses distinguish the individual fields within a structure, while field-insensitive ones fold the entire structure into a single "super-field."

Handling of Recursive Data Structures. Most alias analyses used in practice start with a single abstract location per each allocation site, and refine the information by an iterative algorithm. Such an approach is oblivious to recursive data structures. Early work on the alias analysis of recursive data structures has been done by Chase, Wegman and Zadeck [28]. Later, the analysis of recursive data structures became known as shape analysis [133]. Shape analysis is computationally very expensive [80].

The interprocedural analysis presented in this thesis is context-sensitive, interprocedurally path-sensitive (and therefore also flow-sensitive), and directional, but does not handle recursive structures precisely — rather, it uses a logical memory model [16] and represents the heap by a single memory location per allocation site to avoid the high computational complexity associated with shape analysis.

2.3.4 Summarization for Interprocedural Analysis

Summaries are used in software analysis to avoid or mitigate the potentially exponential cost of context-sensitivity. Cousot and Cousot [40] made a classification of approaches to modular static analysis:

Worst-case analysis assumes that absolutely no information is known on the interfaces of modules. This form of analysis is very imprecise, but also scalable and easy to parallelize.

Simplification-based approaches simplify the summaries before using them. Simplification can range from existential quantification of local variables to abstractions over various abstract domains [39].

Analysis based on user annotations requires the users to provide interface specifications in the forms of preconditions and postconditions. Users can often also provide assertions and loop invariants. An example of such a system is the Boogie [90] Spec# static checker.

Symbolic relational analysis represents summaries as relations, similar to circuit modules. The environment assigns symbolic names to each object that a module can use or modify. The analysis instantiates the names in the environment as relational parameters when the module is used. Such an approach is used in the SLAM software model checker [14], in which finite-state summaries are represented as BDDs [24] and used for image computation.

Structural abstraction in Chapter 4 builds upon symbolic relation summaries, but also allows various simplifications and abstractions to be performed before the summaries are used.

2.4 Low Level Virtual Machine

My experimental static checker Calysto uses the Low Level Virtual Machine (LLVM) compiler framework [86] as a front-end. LLVM consists of a number of front-ends for different languages, an extensive framework of analyses and transformations that operate on the LLVM intermediate representation, an interpreter, and a number of back-ends for several mainstream processors. The LLVM intermediate representation is in SSA form, and the instructions are similar to standard three-address Reduced Instruction Set Computer (RISC) instructions (e.g., [73]). All abstract memory locations are accessed through `load` and `store` instructions, so the standard address-of operator is unnecessary.

Several LLVM features are relevant to the work presented in this thesis:

1. The intermediate form is language-independent.
2. LLVM eliminates all the complexity of dealing with numerous official and unofficial standards of C, which is the main verification target for Calysto.
3. LLVM performs a standard batch of semantic and syntactic checks on the program before producing the intermediate code. Calysto assumes that the code is syntactically and semantically checked.
4. Local variables (including pointers) in LLVM SSA have a unique point of definition and are immutable.
5. LLVM handles all address-of, field-access, and array-access constructs with a single instruction — `getelementptr`. Handling of `getelementptr` will be abstracted away from the algorithms presented in this thesis.

6. Global variables can be only accessed through constant pointers.
7. Multi-level pointers are flattened by using temporary variables.
8. LLVM handles pointer arithmetic.

The features related to pointers are important for the work presented in this thesis, and the rest of this section provides more details.

Global Variables. All identifiers used within SSA must have a unique point of definition. If the globals were accessed directly, it would not be possible to enforce the single point of definition requirement. For that reason, globals can be accessed only through constant pointers.

Multi-Level Pointers. LLVM expands multi-level (non-recursive) pointers into a number of temporary variables making the analysis simpler and more uniform. All abstract memory locations are read and written by a single pointer dereference. Since Calysto always substitutes definitions for uses, it can handle multi-level pointers. The intraprocedural analysis need not be aware of the multi-level pointers, because they are all expanded into single-level ones.

Pointer Arithmetic. LLVM resolves most pointer arithmetic that can be resolved statically without using a decision procedure. Furthermore, it does a number of global optimizations, like constant propagation and strength reduction, that make this resolution more precise.

Calysto makes additional efforts to resolve pointer arithmetic through several optimizations of the constructed summaries. Variable offsets resulting from arbitrary pointer arithmetic are treated in the same way as the variable array indices — they are considered to be zero, in the spirit of the logical memory model [16]. This approximation is unsound (bugs can be missed) and incomplete (spurious bugs can be reported), but renders the constructed VCs easier for the decision procedures. Section 3.1.3 explains how this approximation can be eliminated at the cost of additional complexity.

Chapter 3

Structure-Preserving Symbolic Execution

This chapter will focus on the intraprocedural analysis of call-free functions, and the following chapter will extend the intraprocedural into an interprocedural analysis. Section 3.1 introduces a programming language grammar that reflects the most important features of the LLVM intermediate form, while using γ-functions instead of ϕ-functions.[10] The introduced language makes the presentation more readable. The implementation in CALYSTO works on the full LLVM intermediate form. The same section also surveys the design decisions, and discusses the main contributions. Section 3.2 introduces a novel structure-preserving symbolic execution algorithm. The structure preservation property is going to be exploited later by structural abstraction in Chapter 4. Section 3.3 discusses the space complexity and correctness of the algorithm.

3.1 Introduction

The real implementation of the presented symbolic execution algorithm works on a standard SSA form (as implemented in the LLVM framework). To make the exposition easier to follow, I will use a simpler language. The language picks up the most important features of the LLVM intermediate form and replaces ϕ-functions with their γ-function counterparts. The language captures all the important features of commonly used programming languages, except for func-

[10] As previously mentioned, SSA is not precise enough for symbolic execution, so the implementation of the algorithm replaces ϕ-functions with γ-functions on the fly.

$$
\begin{array}{rcl}
\mathit{BasicBlock} & ::= & \mathit{Terminator} \mid \mathit{Stmt}; \mathit{Terminator} \\
\mathit{Terminator} & ::= & \textbf{branch}\,(\mathit{BoolVar}, \texttt{Label}_T, \texttt{Label}_F) \mid \textbf{branch}\ \texttt{Label} \\
 & \mid & \textbf{return}\,\mathit{IntVar} \\
\mathit{Stmt} & ::= & \mathit{IntVar} := \mathit{IntExpr} \\
 & \mid & \mathit{IntVar} := *\mathit{IntVar} \\
 & \mid & *\mathit{IntVar} := \mathit{IntVar} \\
 & \mid & \mathit{BoolVar} := \mathit{BoolExpr} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt} \\
 & \mid & \textbf{assert}\,(\mathit{BoolVar}) \\
 & \mid & \textbf{assume}\,(\mathit{BoolVar}) \\
 & \mid & \textbf{abort} \\
\mathit{IntExpr} & ::= & \text{IntConst} \mid \mathit{IntVar}\ \mathit{BinaryOp}\ \mathit{IntVar} \mid \mathit{UnaryOp}\ \mathit{IntVar} \\
 & \mid & \gamma\,(\mathit{BoolVar}, \Gamma_I, \Gamma_I) \mid \mathit{Function}\,(\mathit{Params}) \\
\mathit{BoolExpr} & ::= & \text{BoolConst} \mid \mathit{IntVar}\ \mathit{Rel}\ \mathit{IntVar} \mid \gamma\,(\mathit{BoolVar}, \Gamma_B, \Gamma_B) \\
 & \mid & \mathit{BoolVar} \wedge \mathit{BoolVar} \mid \mathit{BoolVar} \vee \mathit{BoolVar} \mid \neg \mathit{BoolVar} \\
\mathit{Params} & ::= & \texttt{EMPTY} \mid \mathit{IntVar}, \mathit{Params} \\
\Gamma_B & ::= & \gamma\,(\mathit{BoolVar}, \Gamma_B, \Gamma_B) \mid \mathit{BoolVar} \mid \emptyset \\
\Gamma_I & ::= & \gamma\,(\mathit{BoolVar}, \Gamma_I, \Gamma_I) \mid \mathit{IntVar} \mid \emptyset
\end{array}
$$

Figure 3.1: The Grammar of a Simple Language.

tion pointers. As explained earlier in Section 2.3.1 on page 21, a simple points-to analysis suffices in practice for resolving the function pointers. The implementation of the algorithm performs such a simple points-to-analysis and explicitly quantifies over all side-effects of functions that can be called at an indirect call site. The maximal number of allowed calls is determined by a user-controllable parameter.

The grammar of the language is given in Fig. 3.1. Two types of variables are defined: Boolean type $\mathbb{B}$ and integers $\mathbb{Z}_N$. Abstract memory locations, function parameters, and function return values can only be of the integer type. As noted in Section 2.3.2, pointers are handled in the same manner as local variables. The only point when this distinction matters is when a pointer is dereferenced.

A program consists of one or more functions. Each function consists of one or more basic blocks and can have zero or more input parameters. Each basic block contains one or more statements and is terminated with a statement that changes the flow of control: either a **branch** or a **return** statement. The **branch** statement can be conditional or unconditional. The conditional branch jumps to the basic block denoted by label $\texttt{Label}_T$ if *BoolVar* is true, and to $\texttt{Label}_F$ otherwise. Note that such a branch statement allows the representation of unstructured code. The **return** statement returns the flow of control and a returned value back to the caller. The dereference operator $*$ is used to represent access to an abstract memory location — pointer read if the dereferenced pointer is on the right side of an assignment, and pointer write if it is on the left. Sequential composition is denoted by the ";" operator. The statements **assert**(*BoolVar*) and **assume**(*BoolVar*) do not influence the state of the program if *BoolVar* is true. If *BoolVar* is false, **assert** terminates erroneously and **assume** terminates with no error [89]: the difference between the two is that erroneous termination says it's the programmer's fault that the condition does not hold, while the other type of termination relieves the programmer of any responsibility. The **abort** statement is syntactic sugar for *assume*(false). Expressions allow a standard set of logic and arithmetic operators. All relations (*Rel*), as well as all unary (*UnaryOp*) and binary operators (*BinaryOp*) specified in Fig. 2.1 on page 14, are supported.

3.1.1 Sources of Complexity

Symbolic execution (also known as symbolic simulation) [82] is a technique that computes symbolic expressions that give the values of all modified locations in a program for all possible execution paths. The technique has numerous applications, e.g., verification of software partial correctness [36, 54, 42], software testing [35, 12], program slicing [93], and equivalence checking between high- and low-level hardware descriptions [83, 34, 61].

Symbolic execution is inherently difficult. Even if loops are unrolled (as is often done in software verification), and recursive data structures are imprecisely represented by a single node per allocation site, symbolic execution remains very expensive because:

1. There are a potentially exponential number of paths that need to be considered.
2. Pointer reads and writes can access a potentially large number of abstract memory locations, due to pointer aliasing.
3. Computed symbolic expressions can be of exponential size if represented naïvely.

Fig. 3.2 on the next page illustrates some of these difficulties. Accordingly, precise symbolic execution has been considered too expensive to use on large bodies of code. For example, many previously proposed tools considered paths one-by-one (e.g., [42, 138, 26]) and can not handle the join operation. Software model checking tools (e.g., [13, 74]) also typically perform symbolic execution on a single path at a time.

This chapter introduces a novel symbolic execution algorithm that addresses the above three problems. I attack the first two problems by exploiting structure in the CFG to keep the analysis local. The third problem could be solved by introducing a linear number of additional variables (as originally proposed by Tseitin [128]) to flatten the computed expressions without blowup. However, flattening computed expressions loses the original structure, which is valuable for later analysis. The approach I use is to represent the computed expressions in the form of maximally-shared graphs, which not only avoids the blowup, but also preserves the original structure of problem.

3.1.2 Assumptions

The presented analysis makes several assumptions:

1. gating path expressions are computed for each basic block;

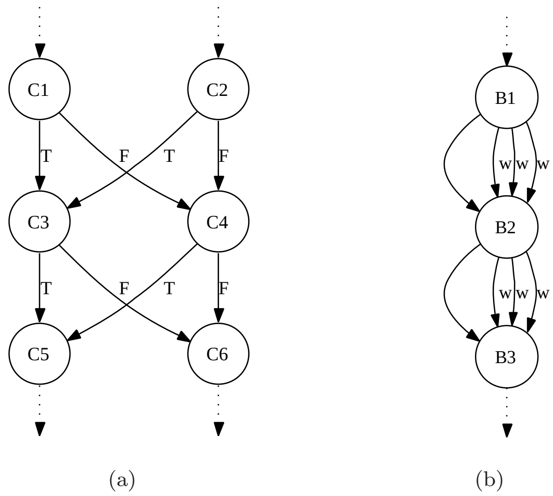


Figure 3.2: These two fragments of CFGs illustrate some sources of complexity in symbolic execution. Nodes denote basic blocks, and edges represent flow of control. Graph (a) illustrates the exponential number of paths possible. Symbolic execution computes exact symbolic expressions for modified locations for all possible paths, so the algorithm must consider all (exponential number of) paths through the code and encode the conditions under which each path is executed. For example, basic block C_6 can be reached from C_1 or C_2 if $(c_1 \wedge \neg c_3) \vee (\neg c_1 \wedge \neg c_4) \vee (c_2 \wedge \neg c_3) \vee (\neg c_2 \wedge \neg c_4)$, where the branching condition of block B_x is denoted as c_x. Graph (b) shows that the number of live definitions of an abstract memory location* can be much larger than the cut-width of the CFG. Assume that some abstract memory location, say $*p$, is written on all the paths from B_1 to B_2, except on one path. The same holds for paths from B_2 to B_3. Assuming that $*p$ is defined in B_1, seven possible definitions reach B_3. The interactions of pointer reads and writes tend to get even more complex in practice.

*Local variables always have a unique point of definition, unlike abstract memory locations.

Data structure	Computational complexity	Due to
gating path expressions	$\mathcal{O}(N \cdot \alpha(E, N))$	[129]
dominator trees	$\mathcal{O}(N \cdot \alpha(E, N))$	[91]
iter. dom. frontiers	"almost linear"	[107]

Table 3.1: Prerequisite Algorithms and Their Computational Complexities.

2. dominator and postdominator trees are available;
3. the iterated dominance frontiers are computed;
4. all loops are unrolled at least once and terminated with **assume**(¬loop_test)

The computational complexities of the algorithms for computing the required data structures are given in Table 3.1. All loops can be unrolled in linear time by breaking the back edges in the call graph.

3.1.3 Design Decisions

To achieve scalability, I made several simplifying assumptions about loops, memory model, and terminal pointers. This section discusses the tradeoffs and possible improvements.

Loops. Like ESC/Java [63], Calysto approximates loops by unrolling them once and terminating them with an assumption that the loop test has failed. Although this approach cannot introduce spurious errors, it can result in decreased code coverage — if the loop test cannot fail after the first iteration, the conjunction of the loop test and its negation is going to be false. Since anything can be implied from false, all assertions reachable from the loop exit block become trivially satisfiable.

The approach can be easily extended with a more precise treatment of loops in several ways:

- By unrolling loops multiple times and omitting the assumption that the loop test has failed. This approach is unsound, but gives higher code coverage.

- By using the framework of abstract interpretation [39]. Loops can be replaced with their invariants computed by abstract interpretation before running the presented symbolic execution algorithm, like in [88]. This approach can be expensive and its effectiveness depends on the templates used for invariant guessing.

- By considering loop-carried values to be unconstrained. This approach is very imprecise, but sound and simple.

  The loop-carried values can be simply detected in both GSA and SSA. In GSA, the μ-functions select between the initial and the loop-carried values. In SSA, definitions are joined through the ϕ-functions. In many implementations of SSA, operands of a ϕ-function are definition-source pairs, where source is the basic block from which the corresponding definition reaches the ϕ-function. If a CFG is traversed in the reverse postorder, the definitions that come from sources that haven't yet been visited are the loop-carried values.

- By requiring the users to provide loop invariants. This approach requires a significant amount of manual effort, but is sound, and potentially very precise.

Memory Model. The algorithm, as implemented in Calysto, uses a simple memory model, similar to the logical memory model [16], in which $*(ptr + i)$ and $*ptr$ are assumed to refer to the same object, except that Calysto does distinguish those two locations if the expression simplifier can simplify i to a constant. Such constant offsets are often used for access to structure fields (making the analysis field-sensitive as well), so this added precision is important in practice. Recursive data structures are imprecisely abstracted by a single node per allocation site. All other locations in the program (including dynamically allocated structures) are handled precisely. Multi-level pointers and statically known pointer offsets are handled precisely as well.

It would be relatively simple to extend the proposed analysis with support for the non-extensional theory of arrays [124] if a more precise memory model is required. The tradeoff between the precision and the additional complexity is unclear at this point and requires further research.

Non-Aliased Terminal Pointers. The last design decision worth mentioning is my assumption that the terminal pointers do not alias each other. This is a reasonable assumption in practice: Pointers returned by the memory allocation functions (like `malloc()`) can be safely considered unique. The pointers to globals are always constants and therefore unique. The pointers passed to functions as parameters frequently do not alias each other, but if they do, CALYSTO computes a join of the corresponding definitions by explicit existential quantification over the symbolic definitions of modified abstract memory locations.

For instance, let f be a function that calls function g, which takes two pointer parameters. Let p_1 and p_2 be the pointers, which may alias, that f passes to g as parameters. If g executes $*p_1 = d_1$ and $*p_2 = d_2$, where d_1 and d_2 are arbitrary symbolic expressions, CALYSTO will represent the side-effects of the call as $*p_1 = ITE(c_1, d_1, d_2)$ and $*p_2 = ITE(c_2, d_1, d_2)$, where c_1 and c_2 are fresh unconstrained Boolean variables.

If the may-alias analysis is sound, so is handling of aliased pointer parameters. CALYSTO, however, only checks whether the pointer parameters are aliased in the caller. This could be easily changed by substituting a sound and arbitrarily precise may-alias analysis. The tradeoff between the precision and soundness in handling aliased pointer parameters requires further research.

3.1.4 Contributions

The main technical contribution of this chapter is an efficient symbolic execution algorithm that preserves the data-flow dependencies (structure) of the original program in computed symbolic expressions. Structural abstraction (Ch. 4) uses that structure for fast validity checking of the computed VCs.

The proposed symbolic execution algorithm uses gating path expressions (Definition 4 on page 25) to reconstruct only as much high-level structure as needed and does not require a structural analysis (e.g., [98, page 236]).

The algorithm exploits the CFG structure for efficiency. The CFG structure exploitation is based on the insight that all the paths by which a basic block B is reachable from the `Entry` block have to pass through the immediate dominator of B. This property localizes the problem to subgraphs of the dominator tree. My dynamic programming algorithm iterates over the CFG in the reverse post-order, assuring that the immediate dominator of each node is visited before the node itself. After a definition of an abstract memory location is computed for the immediate dominator of some basic block B, it can be redefined only on a small number of paths that reach B. Direct application of this insight would require computation of a definition of every modified location for almost every single block. My algorithm updates definitions in a lazy manner and only for the basic blocks where an update is required.

3.2 Symbolic Execution

This section starts by giving a high-level introduction into the construction of VCs. VCs are composed of two components: the control-flow context and the asserted condition. Symbolic representations of both components can be computed with the proposed symbolic execution algorithm. After introducing the notion of a VC and describing its components, this section continues with two novel operators (Section 3.2.1) that will be instrumental for performing efficient symbolic execution. The structure-preserving symbolic execution algorithm is introduced in Section 3.2. Section 3.2.3 defines maximally-shared graphs for representation of computed symbolic expressions, while Section 3.2.4 provides a number of examples that illustrate the newly introduced concepts and the dynamics of the proposed algorithm.

The logical formula representing a VC is composed of two parts: the control-flow context, and the condition being checked. The control-flow context, say ψ, represents the conditions under which a particular statement is reachable. Let us assume that we want to check that an assertion *assert*(α) can never fail. Intuitively, the VC should be $\psi \Rightarrow \alpha$. For example:

```
assume(!c1);
if (c2) {
    assert(c3);
}
```

the control-flow context of the assertion is $\neg c_1 \wedge c_2$, and the constructed VC would be: $\neg c_1 \wedge c_2 \Rightarrow c_3$. More formally, VCs are defined as follows:

Definition 5 (Verification Condition)**.** *Let α stand for an asserted condition and ψ for the predicate that specifies the control-flow context, i.e. the conditions under which* **assert**(α) *statement is reachable. The VC for the asserted condition is a logical formula ϕ defined to be equal to $\psi \Rightarrow \alpha$. The assertion α holds if and only if ϕ is valid.*

Both ψ and α can be computed with the proposed algorithm. The symbolic value of α is computed through symbolic execution. To compute ψ, Calysto uses the following trick [87]: For every conditional branch statement **branch**(c, B_1, B_2), an assumption **assume**(c) (resp. **assume**($\neg c$)) is prepended to the statements in B_1 (resp. B_2). The passive statements (like **assert**(ϕ) and **assume**(ϕ)) that influence the flow of control, but not the state of the program itself, are assumed to write ϕ to a special Boolean abstract memory location (and **abort** writes false). Reading that memory location produces the symbolic definition of the control-flow context.

3.2.1 Restriction Operators

Even if loops are eliminated, the number of paths in the program can be exponential with the size of the program, but is finite. Analyzing paths one by one during the symbolic execution is hopelessly inefficient, so I take a different approach. Instead of executing symbolic paths one by one, the proposed algorithm computes symbolic definitions of each variable or abstract memory location in the program on all paths.

Once symbolic execution is seen as computation of symbolic definitions, it is obvious that at each *use* (resp. *def*) point in the program we need to construct (resp. update) symbolic definitions. Two problems are associated with the construction (resp. update) of symbolic definitions: (1) How to find which definitions matter? (2) How to avoid recursive lookups?

This section presents two restriction operators that solve these two problems. One operator is for computing symbolic definitions at the *use* points, and the other is for updating symbolic definitions after a write to an abstract memory location at the *def* points. The first operator exploits the dominance relation to reduce the number of definitions that need to be considered. The second operator takes an existing definition, and performs a suitable substitution. Neither of the operators is recursive.

The first restriction operator takes a gating path expression $\overline{\gamma}$ (representing paths) and a set of location-definition pairs $\mathcal{R}$ (representing live definitions) as parameters, and produces a γ-function in which the terminals are from $\mathcal{R}$ (new definition at the *use* point).

Definition 6 (Read)**.** *Let lower case Greek letters $o, \ldots, \rho$ stand for arbitrary objects of some type T. Let $\mathcal{R}$ be a set of pairs $\langle B_i, o\rangle$, where B_i is a basic block. Each basic block can appear only once in the set $\mathcal{R}$. Let Λ stand for a placeholder for a pair to be substituted later. The restriction of a gating path expression $GP(B)$ (see Definition 4 on page 25) of a basic block B to the set $\mathcal{R}$, denoted as $GP(B)|_{\mathcal{R}}$, is defined as follows:*

$$\overline{\gamma}\,(Cond, \Gamma, \Gamma)|_{\mathcal{R}} = \gamma\,(Cond, \Gamma|_{\mathcal{R}}\,, \Gamma|_{\mathcal{R}})$$

$$B_p|_{\mathcal{R}} = \begin{cases} \textit{if} \quad \exists\langle B, o\rangle \in \mathcal{R} \quad \textit{then} \quad o \\[2em] \textit{elsif} \;\; \exists\langle B_p, \pi\rangle \in \mathcal{R} \quad \textit{then} \quad \pi \\[2em] \textit{elsif} \;\; \exists\langle B_x, \varpi\rangle \in \mathcal{R} : \left[\begin{array}{c} B_x \gg B_p \wedge \\[1em] \forall\langle B_i, \rho\rangle \in \mathcal{R} : \begin{pmatrix} B_i \gg B_p \wedge \\ dtl[B_x] > dtl[B_i] \end{pmatrix} \end{array} \right] \textit{then}\, \varpi \\[2em] \textit{else} \quad \Lambda \end{cases}$$

$$\emptyset|_{\mathcal{R}} = \emptyset$$

$GP\,(B)|_{\mathcal{R}}$ attempts to substitute each $B_p \in Pred(B)$ with an appropriate definition from B or B_p (if there is no definition in B). If that fails (no definitions from B or B_p in $\mathcal{R}$), it finds the closest dominator B_d of B_p in $\mathcal{R}$, and replaces B_p with the definition from B_d.[11] In order to use the restriction operator defined above, the dominators of each node B have to be processed, paired with the appropriate objects, and added to the $\mathcal{R}$ before processing B. Fortunately, this is guaranteed by the reverse postorder traversal:

Lemma 1. *Let $B_x, B_y \in \mathcal{B}$ be two basic blocks. If $B_x \gg B_y$, then $Post[B_x] > Post[B_y]$.*

Proof: Assume the opposite, $Post[B_x] \leq Post[B_y]$. The equality part is trivially false as the two basic blocks are distinct. For B_y to be visited first in the reverse post-order traversal, there must exist a path $p = \texttt{Entry} \xrightarrow{*} B_y$ that avoids visiting B_x. From the definition of the dominance relation, it follows

[11] The initialization step of the symbolic execution algorithm (Algorithm 1 on page 51) assures that there always exists at least one dominating definition in `Entry`.

that B_x cannot dominate B_y, which is a contradiction. $\square$

The second restriction operator is used for constructing new conditional definitions after a write to an abstract memory location. It takes a γ-function (representing existing symbolic definition), an overwritten location L, and a substitution value y (newly written value) as parameters, and returns a γ-function (new definition) in which all the objects in terminals are modified in the following way: If a terminal object is equal to L, it is replaced with y. Otherwise, the object is replaced with a placeholder Λ. This operator is used for handling writes to abstract memory locations. Formally,

Definition 7 (Write)**.** *Let γ be an arbitrary gating function. Let x and y stand for arbitrary objects of the same type T; and Λ, for a placeholder for an object to be substituted later. The restriction of γ to location L with the substitution element y, denoted as $\gamma|_L^y$ is defined as:*

$$\begin{aligned} \gamma\,(Cond, \Gamma, \Gamma)|_L^y &= \gamma\,(Cond, \Gamma|_L^y, \Gamma|_L^y) \\ x|_L^y &= \begin{cases} y & \textit{if} \quad x \equiv L \\ \Lambda & \textit{otherwise} \end{cases} \\ \emptyset|_L^y &= \emptyset \end{aligned}$$

The example that follows illustrates how the first restriction operator works. More detailed examples in Section 3.2.4 explain how the newly introduced operators are used within the symbolic execution algorithm.

Example 2:
Let Fig. 3.3(a) on page 47 represent a CFG subgraph of some CFG.[12] Now, let us compute the definition of $*p$ for basic blocks B_2 and B_7 using the first restriction operator (Definition 6 on page 43). For block B_2, it is assumed that

[12]The focus on a subgraph emphasizes the localization of analysis achieved by using the first restriction operator.

$*p$ is read after the $*p = 2$ write statement, while the exact location of the read in B_7 is irrelevant.

The gating path expressions are:

$$\begin{aligned} GP\,(B_2) &= B_1 \\ GP\,(B_7) &= \overline{\gamma}\,(c_1, B_2, B_6) \end{aligned}$$

The branching conditions of each conditional branch node B_x are denoted as c_x. Note that B_7 sees the same definition of $*p$ from B_3 regardless of whether the flow of control passes through B_4 or B_5.

After symbolic execution of B_1 and statement $*p = 2$ in B_2, the set of location-definition pairs of the abstract memory location $*p$ is equal to $\mathcal{R} = \{\langle B_1, 1\rangle, \langle B_2, 2\rangle\}$. From Definition 6 on page 43, it follows that:

$$GP\,(B_2)|_{\mathcal{R}} = 2$$

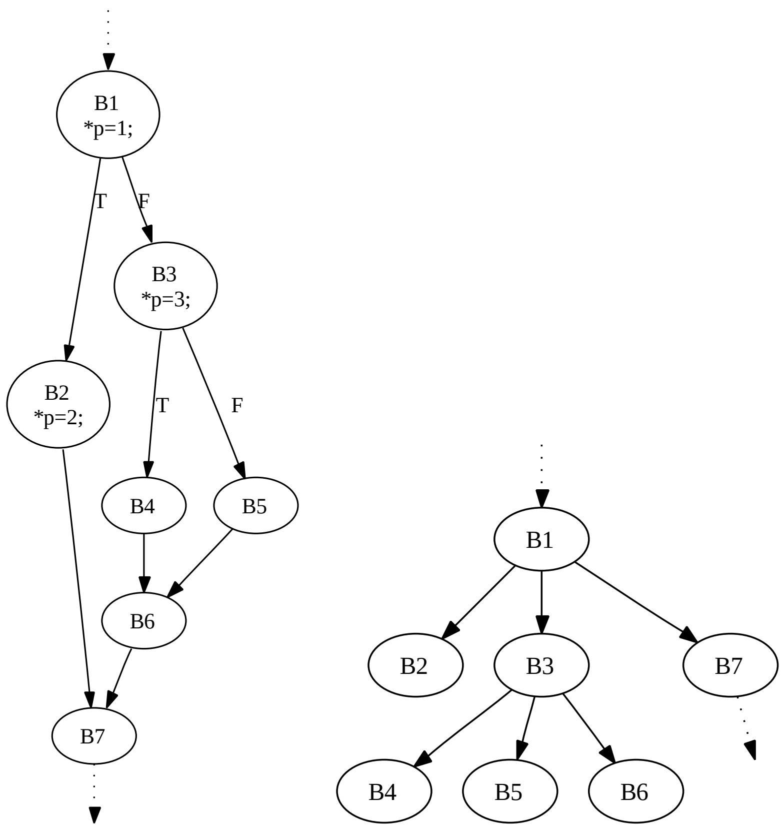
because $\langle B_2, 2\rangle \in \mathcal{R}$ (the first case of $B_2|_{\mathcal{R}}$). According to our assumptions on the number of write statements, $*p$ cannot change again in B_2. The symbolic execution of B_3 adds $\langle B_3, 3\rangle$ to $\mathcal{R}$. The definition of $*p$ in B_7 can be computed as follows:

$$GP\,(B_7)|_{\mathcal{R}} = \overline{\gamma}\,(c_1, B_2, B_6)|_{\mathcal{R}}$$

Since $\langle B_2, 2\rangle \in \mathcal{R}$, the restriction quickly determines that the definition that reaches B_7 through B_2 is equal to 2 (the second case of $B_2|_{\mathcal{R}}$). Finding the definition that reaches B_7 through B_6 triggers the third case of $B_6|_{\mathcal{R}}$: As there is no definition in B_6, the restriction operator starts traversing the DT bottom-up, until it finds a definition. In this case, the definition is found in B_3.[13] Thus, the computed symbolic definition of $*p$ in B_7 is equal to: $\gamma\,(c_1, 2, 3)$.

The handling of a write statement in, for example, block B_4 would be slightly more complex. The symbolic execution algorithm that will be presented shortly maintains an important invariant — given a read statement in some block B, the algorithm guarantees that the definition of the corresponding abstract memory

[13] The initialization step of the main symbolic execution algorithm assures that there always exists a definition in the `Entry` block of each CFG.



(a) CFG (b) DT

Figure 3.3: CFG and DT Subgraphs Used in Example 2. Nodes represent basic blocks and edges the flow of control. The edge labels correspond to the branching conditions being `true` or `false`. The basic blocks are numbered in the reverse post-order and labeled $B_1, \ldots, B_7$. Each block $B_1, \ldots, B_3$ contains one statement that writes some constant to the abstract memory location $*p$. It is assumed that no other statement in the shown CFG subgraph can change $*p$. All other statements are irrelevant for this example and therefore not drawn. The shown CFG and DT graphs are subgraphs of a potentially larger CFG and DT graphs. The dotted edges connect the subgraphs with the corresponding larger graphs.

location is either available in B or in some dominator (not necessarily proper) of each $B_p \in Pred(B)$ — by inserting additional definitions in $\mathcal{R}$ at the IDF. Thus, this little example is representative of all possible cases in which the first restriction operator will be used within this thesis.

3.2.2 Data-Flow Summaries by Symbolic Execution

This section presents the structure-preserving symbolic execution algorithm. The algorithm requires a dominator tree, iterated dominance frontiers, precomputed postdominance relation, and the gating path expressions. All these can be done in almost linear time (Table 3.1).

The restriction operators presented in Section 3.2.1 are used for computing a unique conditional definition from the definitions of abstract memory locations that reach a *use*. The first operator can be intuitively understood as a vehicle for resurrecting the single-point-of-definition property of GSA for abstract memory locations.

The state of symbolic execution is kept in three tables:

1. The expression table *ExpTable* keeps the symbolic definitions of local variables. Local variables (including pointers) can be used as unique table indices, because they have a single point of definition. For this reason, each bin of the *ExpTable* table contains a single entry.

2. Abstract memory locations do not have a single point of definition: they can be (re)defined each time a pointer is written. Thus, multiple definitions can reach a *use*. To compute precisely which definitions actually reach a *use* and under which conditions, the algorithm tracks both the locations where a memory location was (re)defined and the corresponding definitions. Such location-definition pairs are kept in the expression list *ExpList*. The expression list contains a list of all active definitions in the form of γ-functions. The fully path-sensitive definition of an abstract memory location $*p$ at some statement in a basic block B can be computed by restricting $GP\,(B)$ with the set of definitions in the $ExpList\,[*p]$.

3. The $Merge\,[B]$ table contains the set of abstract memory locations whose definitions need to be merged once the basic block B is reached. The definitions of abstract memory locations are merged at the dominance frontiers with other definitions reaching the merge point. This merging occurs exactly where the SSA construction would insert ϕ-functions if abstract memory locations were local variables. Algorithm 2 on page 52 performs the merging.

The symbolic execution algorithm frequently uses the following two patterns in the pseudocode:

1. $ITE(ExpTable\,[x] \neq \emptyset, ExpTable\,[x]\,, x)$
2. $ITE(const\,(x)\,, x, ExpTable\,[x])$

The first pattern is used for pointer variables, and its purpose is to replace each *use* with the corresponding definition. This assures that all pointers are defined in terms of terminal pointers.[14] The second pattern uses the $const\,(x)$ predicate, which is `true` only if x is a constant, to avoid performing table lookups for constants.

Arithmetic and logic operators will be denoted simply as Op. The algorithm computes symbolic expressions $x\ Op\ y$ and does not execute the operator Op. Unary operators can be handled in the same way.

Algorithm 1 starts with an initialization of abstract memory locations, corresponding to terminal pointers, to unconstrained values (unknown initial state). The implementation initializes only the locations actually accessed in the function in a lazy manner. The *Merge* table is initially empty. Basic blocks are processed in reverse postorder, which guarantees that all dominators of a basic block B are visited before the B itself, and symbolic definitions are always computed before the *uses*. If the processed basic block B is in the IDF of some other block that defines some abstract memory location $*p$, the definitions of $*p$ reaching B are merged. Statements in the block are processed from the first to the

[14] Recall that we eliminated recursive data structures from the discussion.

last instruction sequentially. Lines 7–14 handle local variables and pointers substituting each variable x with its definition, namely $\mathit{ExpTable}\,[x]$. The parallel substitution operator is used at Line 16 to substitute multiple variables by their definitions. Since I assume a `load-store` architecture of the intermediate form, abstract memory locations can never directly appear as operands of ϕ-function or any other operator. Line 18 constructs the definition of an abstract memory location. The rest of the algorithm deals with pointer writes. First, the definition of the pointer x is found in the $\mathit{ExpTable}$. For each pointer p that x can alias, the algorithm constructs the currently valid definition Def_p of the abstract memory location $*p$. The rationale behind this step is that the new write might not completely overwrite the Def_p definition (because $x = \gamma\,(\mathit{Cond}, \ldots, \ldots)$ is conditional), so Def_p could become a subexpression of the newly computed definition. In the next step, the stale definitions superceded by the newly constructed one are deleted from $\mathit{ExpList}\,[*p]$. The new definition of each abstract memory location to which some alias p of x can point to is constructed as follows: First, the definition of x is restricted to p and the definition of the variable on the right hand side of the assignment (v) is used as the substitution value. Second, all the other terminal pointers $p_i \neq p$ which x can alias (represented by the Λ placeholder after the restriction) are replaced with Def_p — note that this step assumes that the terminal pointers p_i and p cannot alias each other. Finally, all basic block in the dominance frontier of B are added to the list of merging points for $*p$.

Algorithm 2 is very similar to the innermost loop of Algorithm 1. The definitions of abstract memory locations are simply merged at the dominance frontiers. Precomputation of definitions is not needed as the algorithm has to perform only merge, not write.

3.2.3 Representation

The representation of symbolic expressions affects efficiency a lot, and therefore I define the following representation:

Algorithm 1 Symbolic Execution Algorithm.

For all dereferenced terminal pointers p, initialize $*p$: $ExpList\,[*p] = \langle \texttt{Entry}, *p \rangle$
For all parameters x of the function $ExpTable\,[x] = x$
For all $B_i \in \mathcal{B}$ $Merge\,[B_i] = \emptyset$
for all $B \in \mathcal{B}$ in reverse postorder on CFG **do**
  PerformMerge(B) ▷ Algorithm 2
  for all assignment statements S in sequential order **do**
    if $S = (v := x)$ **then**
      $ExpTable\,[v] = ITE(const\,(x)\,, x, ExpTable\,[x])$
    else if $S = (v :=\ Op\ x)$ **then**
      $ExpTable\,[v] =\ Op\ ITE(const\,(x)\,, x, ExpTable\,[x])$
    else if $S = (v := x\ Op\ y)$ **then**
      $ExpTable\,[v] =$
        $ITE(const\,(x)\,, x, ExpTable\,[x])\ Op\ ITE(const\,(y)\,, y, ExpTable\,[y])$
    else if $S = (v := \phi(\ldots))$ **then** ▷ Join for local variables
      map ϕ to Γ by an operator similar to the one in Def. 6
      $ExpTable\,[v] = \Gamma\,[x \in \mathsf{supp}\,(\Gamma) : x \leftarrow ExpTable\,[x]]$ ▷ Parallel substitution
    else if $S = (v := *x)$ **then** ▷ Memory read
      $ExpTable\,[v] =\ GP\,(B)|_{ExpList[*ITE(ExpTable[x] \neq \emptyset, ExpTable[x], x)]}$
    else if $S = (*x := v)$ **then** ▷ Memory write
      $d = ITE(const\,(v)\,, v, ExpTable\,[v])$ ▷ Get definition of the written value
      $e = ITE(ExpTable\,[x] \neq \emptyset, ExpTable\,[x]\,, x)$ ▷ Write through terminal pointers
      for all pointers $p \in \mathsf{supp}\,(e)$ **do** ▷ Locations x can alias
        $Def_p =\ GP\,(B)|_{ExpList[*p]}$ ▷ Current definition of $*p$
      end for
      for all pointers $p \in \mathsf{supp}\,(e)$ **do** ▷ Locations x can alias
        for all $\langle B_i, \beta \rangle \in ExpList\,[*p]$ **do** ▷ Delete stale definitions
          if $B \underline{\geq} B_i$, delete $\langle B_i, \beta \rangle$ from $ExpList\,[*p]$
        end for
        $ExpList\,[*p] = ExpList\,[*p] \cup \langle B, \left(e|_p^d \right) [\Lambda \leftarrow Def_p] \rangle$ ▷ New definition
        For each $B_i \in DF\,(B)^+$, $Merge\,[B_i] = Merge\,[B_i] \cup *p$ ▷ Merge at IDF
      end for
    end if
  end for
end for

Algorithm 2 Merging Abstract Memory Locations at Dominance Frontiers.

```
procedure PerformMerge(B)
    if Merge[B] ≠ ∅ then
        for all *p ∈ Merge[B] do
            Def = GP(B)|_{ExpList[*p]}
            for all ⟨B_i, β⟩ ∈ ExpList[*p] do          ▷ Delete stale definitions
                if B ≥ B_i, delete ⟨B_i, β⟩ from ExpList[*p]
            end for
            ExpList[*p] = ExpList[*p] ∪ ⟨B, Def⟩
        end for
    end if
end procedure
```

Definition 8 (Maximally-Shared Graph).
Given a graph $G = (N, E)$, let $\mathcal{L}$ stand for a labeling function $\mathcal{L} : N \longrightarrow string$. Define the arity of a node n, denoted as $|n|$, as the number of outgoing edges. The outgoing edges are ordered, and the i-th edge of a node n will be denoted as $child_i(n)$. Two operator nodes n_1 and n_2 are defined to be equivalent ($n_1 \triangleq n_2$) if and only if $|n_1| = |n_2|$, $\mathcal{L}(n_1) = \mathcal{L}(n_2)$, and:

$$\forall i \in [1, |n_1|] : child_i(n_1) \triangleq child_i(n_2)$$

Graph G is maximally-shared if $\neg\exists n_1, n_2 \in O : n_1 \neq n_2 \wedge n_1 \triangleq n_2$.

The choice of representation has several consequences: (1) the representation of computed expressions is compact, (2) it is simple to run expression simplification on top of symbolic execution, (3) the computed expressions reflect the structure of the original code. I found these three properties to be important for efficient solving of the VCs.

In order to construct such a representation, Algorithm 1 uses a hash table to check for existing definitions before constructing the new ones. This guarantees that operator nodes are unique. Assuming that all the graph nodes are hashed, variables and constants are represented by a single node. Binary and unary operators can construct at most one additional node, while the restriction operators can create more than one additional node.

The constructed symbolic expression graph is acyclic. That property trivially follows from the single point of definition property that SSA guarantees for both pointers and local variables, and the following facts: the loops carried values are considered to be unconstrained, each basic block is visited only once, and all *uses* are replaced with definitions. Algorithm 1 also assures that all the pointers are either terminal pointers or entirely defined in terms of terminal pointers.

Maximally-shared graphs make the data-flow dependency explicit. Given any two nodes, n and m, from the same maximally-shared graph, we say that n is data-flow dependent on m if there exists a path $n \xrightarrow{*} m$. If neither $n \xrightarrow{*} m$, nor $m \xrightarrow{*} n$, the nodes are data-flow independent, and cannot influence each other. Thus, if we are interested in solving a VC represented by an expression node v in a maximally-shared graph, all nodes that are unreachable from v can be sliced away without impacting v's validity. Such slicing dramatically improves performance of decision procedures used for validity checking.[15]

Besides improving performance, slicing also separates out individual side-effects of a function: The proposed symbolic execution algorithm, in general, computes a multi-rooted maximally-shared graph. Every root that corresponds to the returned value, or to a value of an updated abstract memory location visible in the calling context can be seen as a symbolic representation of one side-effect. In effect, symbolic execution slices the function with respect to its side-effects, breaking apart a single function with multiple side-effects into multiple pure functions that have only one side-effect each. Structural abstraction (Chapter 4) operates on such function slices.

[15] Example 5 on page 60 illustrates this correlation between reachability in maximally-shared graphs and data-flow dependency in programs.

3.2.4 Examples

Statement 29 of Algorithm 1 might be a bit hard to parse, especially the parallel substitution part. The following example should give a better intuition into how the statement works:

Example 3:

Let x be a pointer defined with γ-function $\gamma\left(c_1, u, \gamma\left(c_2, v, \gamma\left(c_3, z, u\right)\right)\right)$. We shall refer to that γ-function as e. Now, let us consider what happens when the statement $*x := w$ is executed.

The pointers which x can alias are $\mathsf{supp}\left(e\right) = \{u, v, z\}$. Let us consider the effect of the assignment on the abstract memory location to which u points. Construct a new expression by restricting e with $\{u\}$ and substitute with $\mathit{ExpTable}\left[w\right]$. The result is

$$e|_u^{\mathit{ExpTable}[w]} = \gamma\left(c_1, \mathit{ExpTable}\left[w\right], \gamma\left(c_2, \Lambda, \gamma\left(c_3, \Lambda, \mathit{ExpTable}\left[w\right]\right)\right)\right)$$

Effectively, $*u$ is overwritten only if $c_1 \vee (\neg c_1 \wedge \neg c_2 \wedge \neg c_3)$. In all other cases, the content of the $*u$ abstract memory location is left unchanged. The unchanged locations correspond exactly to the Λ placeholder, which is replaced with the currently valid definition of $*u$. Finally, the new definition is added to $\mathit{ExpList}\left[*u\right]$. The same is repeated for the other two pointers (v and z).

The restriction operators introduced in Section 3.2.1 avoid recursive definition lookups, and therefore localize the analysis. The following example illustrates the localization and the overall dynamics of the algorithm:

Example 4:
This example illustrates how the symbolic execution algorithm computes definitions of abstract memory locations and local variables of the function represented with the SSA representation in Fig. 3.4 on the next page, while its CFG and DT graphs are shown in Fig. 3.5(a) and 3.5(b) on page 56. The function defines local variables $a_0, \ldots, a_5$, takes pointer parameters p_0 and p_1, and ac-

cesses two global abstract memory locations through constant pointers g_0 and g_1.

The branching conditions of each conditional branch node B_x are denoted as c_x. Each c_x is a local Boolean variable with a single point of definition. Since those variables have a single point of definition and are immutable, they do not really have much impact on the dynamics of the symbolic execution algorithm and therefore this example abstracts their computation away.

Gating path expressions for the join blocks are:

$$\begin{aligned}
GP\left(B_3\right) &= \overline{\gamma}\left(c_1, B_1, \overline{\gamma}\left(c_2, B_2, \emptyset\right)\right) \\
GP\left(B_5\right) &= \overline{\gamma}\left(c_3, B_3, B_4\right) \\
GP\left(B_8\right) &= \overline{\gamma}\left(c_6, B_6, \overline{\gamma}\left(c_7, B_7, \emptyset\right)\right) \\
GP\left(B_9\right) &= \overline{\gamma}\left(c_1, B_5, \overline{\gamma}\left(c_2, B_5, \overline{\gamma}\left(c_6, B_8, \overline{\gamma}\left(c_7, B_8, B_7\right)\right)\right)\right)
\end{aligned}$$

The gating path expressions of all other non-join blocks B_i, such that $i > 1$, are equal to the single element of $Pred(B_i)$. According to Definition 4 on page 25, $GP\left(B_1\right) = B_1$.

The implementation maps ϕ- to γ-functions as follows:

$$\begin{aligned}
a_2 &= \gamma\left(c_1, a_0, a_1\right) \\
p_2 &= \gamma\left(c_1, p_0, p_1\right) \\
p_4 &= \gamma\left(c_6, p_1, p_3\right) \\
a_4 &= \gamma\left(c_3, a_2, a_3\right) \\
a_5 &= \gamma\left(c_1, a_4, \gamma\left(c_2, a_4, a_0\right)\right)
\end{aligned}$$

Note that no definition can reach a *use* along an infeasible path, so $\emptyset$ can be simplified away.

At Line 1 of Algorithm 1, all modified abstract memory locations are initialized with $ExpList\left[*m\right] = \{\langle B_1, *m\rangle\}$. The unconstrained values represent unknown initial values of the corresponding abstract memory locations.

The rest of the example describes the steps of the algorithm executed for each basic block. The algorithm visits basic blocks in the reverse post-order (which

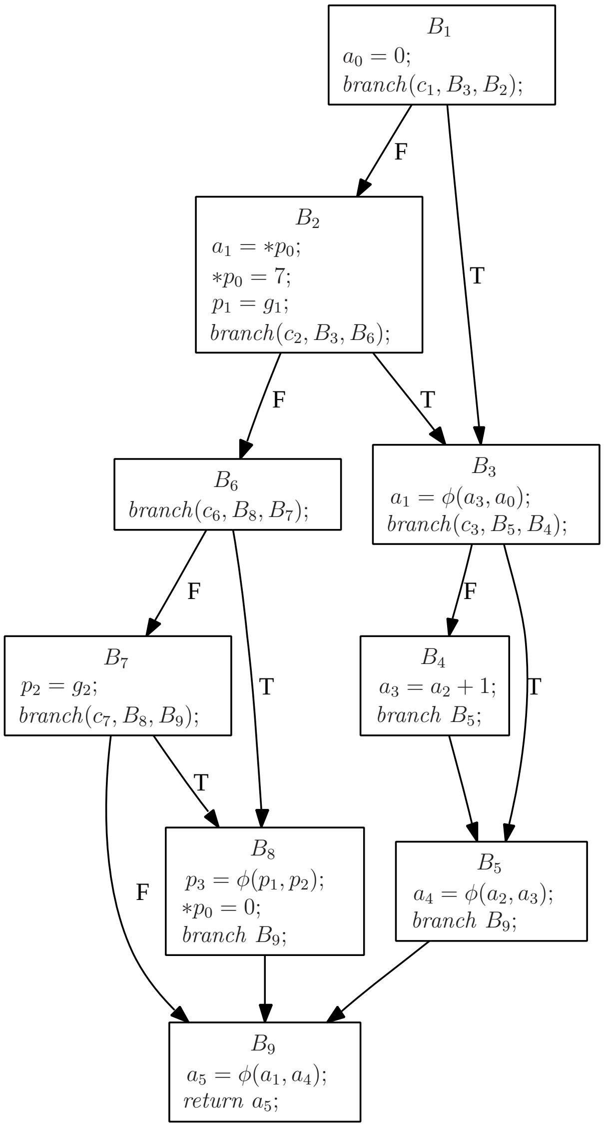


Figure 3.4: SSA of the Function in Example 4. Each Block is labelled with its name: $B_1, \ldots, B_9$. The branching conditions of each conditional branch node B_x are denoted as c_x. The code that computes the conditions is abstracted away from the example.

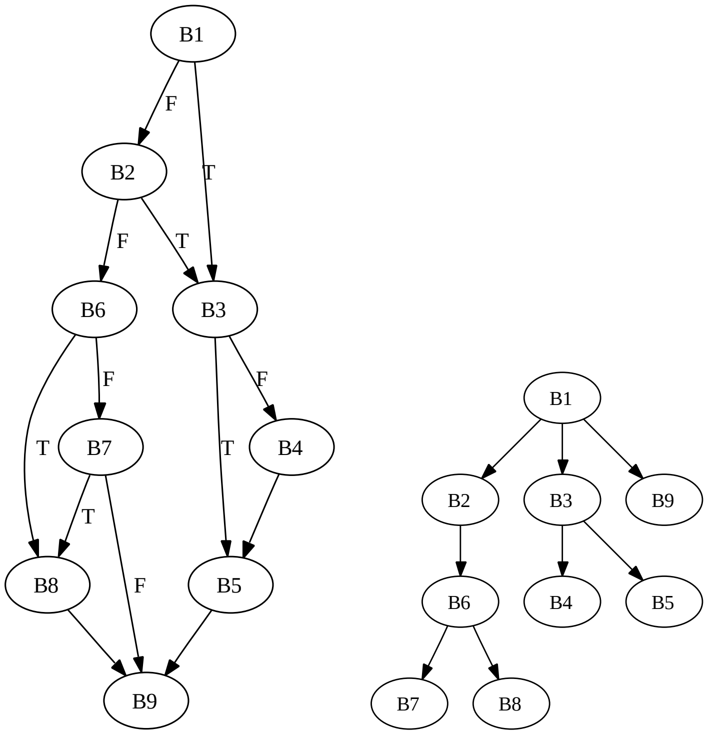


Figure 3.5: CFG and DT of the Function Defined in Example 4. Nodes represent basic blocks and edges the flow of control. The edge labels correspond to the branching conditions being `true` or `false`. The basic blocks are numbered in the reverse post-order.

corresponds to the numerical order of subscripts of blocks in Fig. 3.5(a)).

B_1. At the end of the first block, the state of the algorithm is $\mathit{ExpTable}\,[a_0] = 0$.

B_2. The second block defines a new local variable a_1, writes value 7 to the abstract memory location pointed to by p_0, and defines a new pointer p_1. After this block is symbolically executed, the state of the algorithm is: $\mathit{ExpTable}\,[a_1] = *p_0$, $\mathit{ExpList}\,[*p_0] = \{\langle B_2, 7\rangle, \langle B_1, *p_0\rangle\}$, and $\mathit{ExpTable}\,[p_1] = g_1$. Since $*p_0$ is written, the algorithm schedules merge operations at $DF(B_2)^+ = \{B_3, B_9\}$, so $Merge[B_3] = \{*p_0\}$, and $Merge[B_9] = \{*p_0\}$.

$\mathbf{B_3}$. At the beginning of B_3, the definitions of $*p_0$ are merged:

$$\begin{aligned} GP(B_3)|_{ExpList[*p]} &= \overline{\gamma}(c_1, B_1, \overline{\gamma}(c_2, B_2, \emptyset))|_{\{\langle B_2,7\rangle,\langle B_1,*p_0\rangle\}} \\ &= \gamma(c_1, *p_0, 7) \end{aligned}$$

The new definition is appended to the expression list: $ExpList[*p_0] = \{\langle B_1, *p_0\rangle, \langle B_2, 7\rangle, \langle B_3, \gamma(c_1, *p_0, 7)\rangle\}$.[16] The definition of a_2 is added to the expression table: $ExpTable[a_2] = \gamma(c_1, a_0, *p_0)$. Notice that a_1 was replaced with its definition ($*p_0$).

$\mathbf{B_4}$. The symbolic execution of the basic block B_4 only updates the table with $ExpTable[a_3] = \gamma(c_1, a_0, *p_0) + 1$.

$\mathbf{B_5}$. The symbolic execution of B_5 is simple — only one definition is added to the expression table: $ExpTable[a_4] = \gamma(c_3, \gamma(c_1, a_0, *p_0), \gamma(c_1, a_0, *p_0) + 1)$. The simplifier in Calysto would simplify that definition to: $ExpTable[a_4] = \gamma(c_1, a_0, *p_0) + \gamma(c_3, 0, 1)$.

$\mathbf{B_6}$. This basic block does not change the state of the algorithm because it contains only the branch statement.

$\mathbf{B_7}$. Only one new definition is created in this block: $ExpTable[p_2] = g_2$.

$\mathbf{B_8}$. After execution of the first statement in block B_8, we have: $ExpTable[p4] = \gamma(c_6, g_1, g_2)$. The write statement $*p_3 = 0$; is handled by lines 19–31 of Algorithm 1. First, the algorithm precomputes the definitions for $*g_1$ and $*g_2$, which are unconstrained values (equal to the values at the point of the function call).

[16] Note that B_2 and B_3 cover all paths $B_1 \longrightarrow \{B_2, B_3\} \xrightarrow{+} B_x$, but since the proposed algorithm currently does not detect k-postdominance, the definition in B_1 will not be erased.

Second, the restriction is computed as in Example 3:

$$\begin{aligned} \gamma\left(c_6, g_1, g_2\right)\big|_{g_1}^{0} &= \gamma\left(c_6, 0, \Lambda\right) \\ \gamma\left(c_6, g_1, g_2\right)\big|_{g_2}^{0} &= \gamma\left(c_6, \Lambda, 0\right) \end{aligned}$$

After the substitution of the place holder is performed we get: $\mathit{ExpList}\left[*g_1\right] = \{\langle B_1, *g_1\rangle, \langle B_8, \gamma\left(c_6, 0, *g_1\right)\rangle\}$, and $\mathit{ExpList}\left[*g_2\right] = \{\langle B_1, *g_2\rangle, \langle B_8, \gamma\left(c_6, *g_2, 0\right)\rangle\}$. Finally, merging of definitions of $*g_1$ and $*g_2$ is scheduled for B_9, so we have $\mathit{Merge}\left[B_9\right] = \{*p_0, *g_1, *g_2\}$.

B₉. The definitions are merged at B_9 as follows:

$$GP\left(B_9\right)\big|_{ExpList[*p_0]} = \tag{3.1}$$

$$\gamma\left(c_1, \gamma\left(c_1, *p_0, 7\right), \gamma\left(c_2, \gamma\left(c_1, *p_0, 7\right), \gamma\left(c_6, 7, \gamma\left(c_7, 7, 7\right)\right)\right)\right)$$

which can be simplified to: $\gamma\left(c_1, *p_0, 7\right)$. The definition from B_2 was used for predecessors B_7 and B_8, and the definition from B_3 for the predecessor B_5. Observe how the restriction operator avoids recomputing the redundant definitions and takes the definition from the nearest dominator that defines the value. As B_9 postdominates the definitions from B_1, B_2, and B_3, those three are deleted. The same is repeated for $*g_1$ and $*g_2$ (the expression for $*g_2$ is immediately simplified):

$$GP\left(B_9\right)\big|_{ExpList[*g_1]} = \tag{3.2}$$

$$\begin{gathered} \gamma(c_1, *g_1, \gamma\big(c_2, *g_1, \gamma\Big(c_6, \gamma(c_6, 0, *g_1), \gamma\Big(c_7, \gamma(c_6, 0, *g_1), *g_1\Big)\Big)\big)) = \\ \gamma\left(c_1, *g_1, \gamma\left(c_2, *g_1, \gamma\left(c_6, 0, *g_1\right)\right)\right) \end{gathered}$$

$$GP\left(B_9\right)\big|_{ExpList[*g_2]} = \tag{3.3}$$

$$\gamma(c_1, *g_2, \gamma(c_2, *g_2, \gamma\Big(c_6, *g_2, \gamma\left(c_7, 0 * g_2\right)\Big)))$$

Finally, the definition of a_5 is computed and simplified as:

$$\begin{gathered}
ExpTable\,[a_5] = \\
\gamma\left(c_1, \gamma\left(c_1, a_0, *p_0\right) + \gamma(c_3, 0, 1), \gamma\Big(c_2, \gamma\left(c_1, a_0, *p_0\right) + \gamma\left(c_3, 0, 1\right), a_0\Big)\right) = \\
\gamma(c_1, a_0 + \gamma\left(c_3, 0, 1\right), \gamma\Big(c_2, *p_0 + \gamma\left(c_3, 0, 1\right), a_0\Big))
\end{gathered}$$

Although the symbolic expression of the returned value (a_5), is relatively complex, it actually depends only on one parameter visible in the caller's context: the abstract memory location to which p_0 terminal pointer points to (assuming that the conditions do not depend on any parameters or globals).

This completes the example. Equation 3.1 represents the effect of the function on the abstract memory location(s) to which the actual parameter p_0 of the function can point. The other two equations (Eq. 3.2 and 3.3) represent the effect of the function on the globals. So, the simulated function has four side-effects: the return value, and modifications of abstract memory locations to which p_0, g_1, and g_2 point. The computed γ definitions can be interpreted as *ITE* expressions.

The following example illustrates both how symbolic execution indirectly performs slicing and how dataflow dependencies (structure) are reflected in the computed symbolic expressions.

Example 5:

For the following code fragment:

```
x0 = a + b;
if (x0 < 0) {
    x1 = -x0;
    y1 = y0;
} else {
    x1 = x0;
    y1 = y0 + x0;
}
```

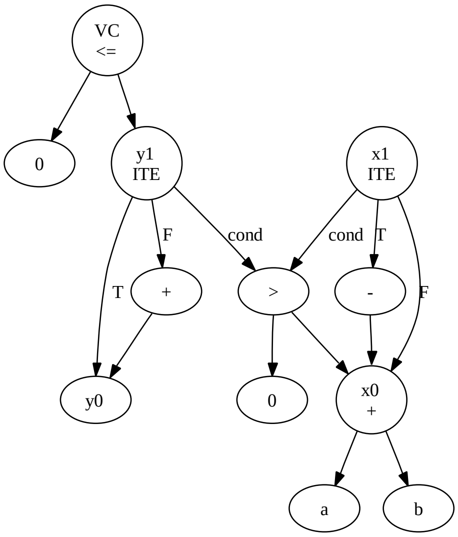


Figure 3.6: Maximally-Shared Graph. Nodes are labeled with the corresponding operators, variable names, and constants. The Zero node is duplicated to improve the graph layout. The ITE outgoing edges are labeled with `cond` for the conditional branch, `T` for the if-branch, and `F` for the then-branch.

```
assert(0 <= y1);
```

Calysto would compute the maximally-shared graph shown in Fig. 3.6. Although the maximally-shared graph representation is identical to the standard logical representation, graphs have two interesting additional properties:

Explicit Structure. Our VC is dataflow dependent on the variable `a`, and that is evident from the graph, because there is a path from the node representing the VC to the node representing `a`.

Simple Slicing. In any maximally-shared graph, a node, say x, is dataflow dependent on another node, say y, if and only if there exists a path in the graph from x to y. Otherwise, y is irrelevant for the computation of x. Going back to our example, there exists no path from the node representing the VC to the node representing `x1`. In effect, `x1` can be sliced away. More generally, all nodes that are unreachable from the node we are interested in can be sliced away. The graph representation makes such slicing very simple.

3.3 Complexity and Correctness

The first part of this section focuses on the space complexity of the proposed algorithm. The second part introduces and proves the main correctness invariants.

3.3.1 Space Complexity

This section discusses the space complexity of the proposed symbolic execution algorithm. The section assumes acyclic CFGs — cyclic CFGs can be reduced to the acyclic ones by replacing loops with appropriate loop invariants. We begin by setting a bound on the size of gating path expressions.

Lemma 2. *Let $\mathcal{B}$ represent a set of basic blocks in an acyclic CFG C. For any $B \in \mathcal{B}$, $GP\,(B)$ can be represented as a maximally-shared graph with at most $|\mathcal{B}|$ nodes.*

Proof: Let C' be equal to C with all non-branch statements eliminated. Reverse the edges of C' and discard the following nodes and their corresponding outgoing edges:

- all nodes unreachable from B,
- all nodes reachable from $idom\,(B)$, and
- block B itself,

getting C''. Reverse the edges of C''. The obtained subgraph contains only blocks that are on at least one path $idom\,(B) \xrightarrow{*} B_p$, where $B_p \in Pred(B)$. Graph C'' might contain redundant basic blocks, which can be eliminated by repeating the following graph transformation for all blocks $B_i \notin Pred(B)$ until no change:

- If B_i has two outgoing edges, and they both have the same destination, replace the conditional branch that terminates B_i with an unconditional one and discard one of the outgoing edges.

- If B_i is terminated by an unconditional branch, disconnect all incoming edges and reconnect them to the successor of B_i, short-circuiting B_i. Discard B_i and its outgoing edges.

This transformation corresponds to canonicalization of gating path expressions (explained after Definition 4 on page 25).

Let $\mathcal{B}_c$ denote a set of blocks B_i whose branching conditions c_i are used in the canonical $GP\,(B)$. Graph C''' contains exactly the set of nodes $Pred(B) \cup \mathcal{B}_c$. The first transformation guarantees that C''' cannot contain any nodes that are not on $idom\,(B) \xrightarrow{*} B_p$. The second transformation discards only blocks that would be eliminated in the canonical $GP\,(B)$, and nothing else. Thus, the canonical $GP\,(B)$ represents exactly the branching conditions on all the paths $idom\,(B) \xrightarrow{*} B_p$ in C'''. Now, we only need to translate C''' into the corresponding $GP\,(B)$.

Traverse C''' in postorder, constructing a partial gating path expression $E(B_i)$ for each block $B_i \in C'''$ as follows:

- If $B_i \in Pred(B)$, then $E(B_i) = B_i$.
- If $B_i \in C''' \backslash Pred(B)$, then B_i must be a branch node (thanks to the previously done transformations). Let B_i be terminated with **branch**$(c_i, B_T,$ $B_F)$. Create an expression $E(B_i) = \overline{\gamma}\,(c_i, B_T, B_F)$. If either of the two basic blocks is not in $Pred(B)$, replace it with $\emptyset$.

The last step creates at most $|C'''|$ expressions. Now replace all $B_i \notin Pred(B)$ in each of those expressions with $E(B_i)$. Using structural hashing, create a maximally-shared graph for $E(idom\,(B))$. The created graph is semantically equivalent to $GP\,(B)$, according to Definition 4, and contains at most $|\mathcal{B}|$ nodes. $\square$

The number of terminals of a gating path expression is limited by the maximal number of predecessors any block can have. Thus, the number of terminals is equal to the maximal cut-width of a CFG, which is typically much smaller in practice than the total number of blocks.

Now we turn our attention to Algorithm 1 and its space complexity. We begin by setting bounds on the size of expressions generated by the two restriction operators. The proof of Lemma 2 can also be used to set the bound on the size of any γ-function to $|\mathcal{B}|$, which means that a definition of any variable or pointer can be represented as a maximally-shared graph of size at most $|\mathcal{B}|$. It follows that the first restriction operator (Definition 6 on page 43) can create at most $|\mathcal{B}|$ new nodes. Note that the definitions substituted for the terminals of $GP\,(B)$ were already created elsewhere. Similarly, the second restriction operator (Definition 7 on page 45) copies the restricted γ-function and replaces the terminals either with x, or with a placeholder Λ, creating at most $|\mathcal{B}|$ new nodes.

The following theorem sets pessimistic worst-case bounds on the total size of symbolic expressions that an implementation of Algorithm 1 based on maximally-shared graphs computes for a call-free function.

Theorem 1. *Given a call-free function f with k operators and $|\mathcal{B}|$ basic blocks, an implementation of Algorithm 1 that uses maximally-shared graphs creates at most $\mathcal{O}(k \cdot |\mathcal{B}|^2)$ nodes. Each node represents a variable, a constant, or an operator.*

Proof: Symbolic execution of any operator can create at most one new node — the result of the operator. Symbolic execution of a GSA γ-function can create at most $|\mathcal{B}|$ nodes (proof similar to the proof of Lemma 2). A memory read can create at most $|\mathcal{B}|$ nodes, according to the previous analysis.

A memory write to some pointer x first reads all the terminal pointers (Line 23 in Algorithm 1), creating at most $t \cdot |\mathcal{B}|$ new nodes, where t is the number of the terminal pointers in the definition of x. Line 29 takes the definition of the written pointer x, which is of size at most $|\mathcal{B}|$, and substitutes the definitions computed in the previous step. Hence, a memory write can create at most $(t + 1) \cdot |\mathcal{B}|$ new nodes. The worst case value of t is $|\mathcal{B}|$, because each block can have at most $|\mathcal{B}| - 1$ predecessors. This is a very conservative estimate, because the maximal CFG cut-width is usually much smaller than $|\mathcal{B}|$.

Thus, we conclude that the symbolic execution of a call-free function f can construct at most $\mathcal{O}(k \cdot |\mathcal{B}|^2)$ nodes in the worst case. $\square$

This bound is very pessimistic. First, it does not take sharing and expression simplification into account. With common subexpression elimination and simplification, the proposed algorithm in practice usually creates a symbolic representation of a call-free function that is linear with the number of statements in the function. Second, functions usually have a maximal cut-width that is much smaller than the number of basic blocks. Structured programs without the *switch* statement tend to have especially small cut-widths [127].

3.3.2 Correctness

When computing a definition of an abstract memory location, Algorithm 1 considers only the definitions reaching B from $\mathit{Pred}(B)$ or immediate dominators of blocks in $\mathit{Pred}(B)$. We start this section with a lemma that explains why this localization is correct. Intuitively, the lemma shows that if a definition in the *ExpList* precedes a predecessor of B, it must also dominate it.

Lemma 3. *Let B be a basic block processed at some iteration of the algorithm. Let $*p$ be some abstract memory location, and x a term. Then it holds:*

$$\forall \langle B_w, x\rangle \in \mathit{ExpList}\,[*p] : (\forall B_p \in \mathit{Pred}(B) : B_w \prec B_p \Rightarrow B_w \underline{\gg} B_p)$$

Proof: Let us assume the opposite:
$\exists \langle B_w, x\rangle \in \mathit{ExpList}\,[*p] : (\exists B_p \in \mathit{Pred}(B) : B_w \prec B_p \wedge B_w \not\gg B_p)$ Right after the initialization, the assumption is obviously `false` because the `Entry` node has no predecessors. In every subsequent iteration of the algorithm, the definitions are merged at the nodes that belong to the IDF of B_w. With no loss of generality, we can assume that B_{w2} is the last merge point that still precedes B. So, the assumption can be rewritten as:
$\exists B_{w2} \in \mathit{DF}\,(B_w)^+ : (\exists B_p \in \mathit{Pred}(B) : B_w \preceq B_{w2} \preceq B_p \wedge B_{w2} \not\gg B_p)$

If $B_{w2} = B_p$, the assumption is trivially false. As B_{w2} is the last merge point, there can not exist another merge point B_{w3} such that $B_w \preceq B_{w2} \prec B_{w3} \prec B_p$, so there can be no more join nodes on the path $B_{w2} \xrightarrow{+} B_p$ (including B_p). Hence, B_{w2} has to dominate B_p. This contradicts the assumption. □

If Lemma 3 did not hold, the restriction operator would have to recursively traverse the CFG, searching for definitions.

Any time a new definition is appended to *ExpList*, all overwritten (postdominated) definitions can be safely deleted. The following lemma proves that such deletions do not violate Lemma 3:

Lemma 4. *Let $\langle B, Def \rangle$ be a new location-definition pair that is about to be added to ExpList $[*p]$. Deleting all $\langle B_i, x\rangle$ from ExpList $[*p]$, such that $B \gtrdot B_i$ preserves the main invariant of the algorithm (Lemma 3).*

Proof: Let B_w denote a block postdominated by B. Assume that both blocks write to $*p$. Due to the reverse postorder traversal, all predecessors of B have been already visited. Thus, only its proper descendants or siblings can be visited in the future iterations.

Let B_d be a descendant of B. Assume that there exists a path $B_w \xrightarrow{+} B_d$ that avoids B. Then there exists a path $B_w \xrightarrow{+} B_d \xrightarrow{*}$ `Exit` that avoids visiting B. This contradicts the definition of the postdominance relation. Hence, any path `Entry` $\xrightarrow{+} B_d$ has to pass through B if it passes through B_w. It follows that if B_w is a dominator of B_d, so is B. This preserves the property.

Let B_s be a sibling of B. Obviously, B cannot postdominate B_s or any of its dominators. Consequently, a write to $*p$ does not influence reads/writes in B_s. □

All the entries in the expression table must be well-defined and contain no undefined elements. The only undefined element that is used at several places is the Λ placeholder. Lemma 5 proves that Λ is not in the support of any expressions added to the expression table:

Lemma 5. *Given a non-empty set* $\mathcal{R}$*, the gating path restrictions in Algorithms 1 and 2 produce a* γ*-function that contains no* Λ *placeholder.*

Proof: The gating path expression for B contains only predecessors of B as terminal symbols. According to Definition 6 on page 43, each predecessor B_p in the gating path expression will be replaced either with the value of an abstract memory location $*p$ written in B or B_p, or with a value of that location written in the nearest dominator of B_p, or with Λ. However, from Lemma 3, it follows that for any predecessor B_p, either B_p or at least one of its dominators is in $ExpList\,[*p]$. Thus, the first restriction operator can never create a Λ subexpression.

The second restriction operator can generate Λ subexpressions, but those are replaced by parallel substitution at Line 16. $\square$

Lemma 3 proves that for any abstract memory location $*p$ and any *use* in some basic block B, the expression list $ExpList\,[*p]$ contains only definitions that either dominate or do not even precede the predecessors of B. Since the dominators of any basic block can be linearly ordered and the restriction operator takes the most recent definition reaching each predecessor, it follows that the definition of $*p$ is constructed correctly for each predecessor of B. The gating path expression handles all the paths $idom\,(B) \xrightarrow{*} B_p \longrightarrow B$ such that $B_p \in Pred(B)$. If there exists a definition d in B, all terminals in the gating path expression are replaced with d, and the expression can be trivially simplified to d. Otherwise, the restriction operator takes the definition in B_p, if there is one, or the definition that is in the nearest dominator of B_p that defines the location. Lemma 4 proves that this property is maintained even if postdominated definitions are removed from $ExpList$. Finally, Lemma 5 shows that each computed symbolic definition is completely defined.

Chapter 4

Structural Abstraction

The previous chapter dealt with the intraprocedural analysis, while this chapter discusses how the results of the intraprocedural analysis are used for interprocedurally path-sensitive VC validity checking. Section 4.1 provides the intuition behind structural abstraction, motivates the approach with a real-world example, and outlines the underlying assumptions and design decisions. Furthermore, Section 4.1 lists the most important contributions. Section 4.2 starts by enumerating the types of side-effects that each function can have on the caller's context, and discusses how different types of side-effects interact with structural abstraction. The rest of the section focuses on the abstraction and refinement, and proposes several improvements at the end.

4.1 Introduction

In general, proving software VCs requires interprocedural analysis, e.g., of the propagation of data-flow facts. Some properties, like proper nesting of lock-unlock calls, tend to be localized to a single function and are amenable to simpler analysis. Many others, especially pointer-related properties, tend to span through many function calls.

To handle the complexity of interprocedural analysis, the software analysis community has developed a number of increasingly expensive abstractions. For instance, path-insensitive analysis does not track the exact path along which a certain statement is executed, while context-insensitive analysis does not differentiate the contexts from which a function is called. These abstractions work well in optimizing compilers, but are not precise enough for verification purposes. Software verification analysis has to be both path- and context-sensitive (*-sensitive) to keep the number of false errors low.

The proposed analysis is both path- and context-sensitive. It is based on an abstraction-checking-refinement framework that exploits the natural function-level abstraction boundaries present in software.

4.1.1 Intuition

As discussed in Section 3.2.3, the symbolic execution that computes maximally-shared graphs also indirectly slices the simulated function with respect to its side-effects. Each such slice will be called a *summary* of a side-effect, and can be seen as a pure function that produces a single result and takes a number of inputs, which are a subset of the parameters passed to the simulated function.

Interprocedural analysis of pure functions is relatively simple. When a function call is symbolically executed, formal parameters of the callee have to be substituted with the symbolic definitions of actual parameters, and the abstract memory locations that the callee reads have to be replaced with their symbolic counterparts in the caller's context. That is the easy part. The harder problem is what to do with the summary itself. According to my experiments, the full expansion of each summary is infeasible. I ran experiments on a number of programs, and even on very small non-recursive programs (3–5 KLOC) full inlining results in an exponential blowup, while on smaller programs it bloats the code size 50–180 times the original size. A better approach, used in this thesis, is to represent summaries with *summary operators*, which serve only as placeholders for the full symbolic representation of the summary, and can be expanded into such a full representation later on demand, if needed. Summary operators are going to be named with $\mathit{function}:\mathit{effect}$ notation, representing the function being summarized and the particular side-effect (more on this in Section 4.2.1).

When checking a VC, structural abstraction initially treats all summary operators as unconstrained variables, effectively abstracting away the subexpressions of the VC. If the decision procedure finds a spurious counterexample, structural refinement finds which summary operator needs to be fully interpreted, and inlines the symbolic expression represented by the operator. This process is repeated until the VC is either proven valid, or a concrete counterexample is found.

Experimental results in Chapter 6 show that structural abstraction is very robust and scalable in practice, because it effectively exploits the same natural abstraction boundaries that programmers use when writing a program — functions. Programmers organize code into functions and use them as abstractions. They tend to ignore the details of the effects of the function on the caller's context — the easiest invariant to remember is to remember no invariant at all.

4.1.2 Motivating Example

This section gives an example that provides intuition about how structural abstraction approach solves *-sensitive VCs. The code used in the example is a simplified and slightly modified piece of code from a real application.[17]

Example 6:
Through the example, we shall follow a sequence of steps needed to prove the assertion on line 23. To prove an assertion, we need to prove either that the assertion itself is unreachable, or that it always evaluates to `true`.

```
1  int global1, global2;
2
3  // If *data<0, returns true and computes
4  // *data=abs(*data).
5  bool flip(int *data) {
6      if (*data < 0) {
7          *data = -(*data);
8          return true;
9      }
10     return false;
```

[17]The example is modified from the bit-vector arithmetic theorem prover SPEAR.

```
11   }
12
13   // Assume init is a pure function (no side-effects).
14   int init(int x) {
15       // Some expensive computation...
16   }
17
18   // If global1 is positive and global2 is negative,
19   // scales global1 by abs(global2).
20   void scale() {
21       global2 = init(global1);
22       if (flip(&global2)) {
23           assert(global2 != 0); // Div by zero check.
24           global1 /= global2;
25       }
26   }
```

The VC computed for the assertion at Line 23 is an implication (if Line 23 is reachable, the asserted condition should hold) that can be represented as a maximally-shared graph (Fig. 4.1(a)). Each side-effect is represented by a summary operator. Outgoing edges point to parameters required for computing the result of the particular side-effect. The antecedent contains two nested function calls. The consequent is a simple comparison of zero with the effect of *flip* on the global variable.

The function *flip* has two effects: it returns a Boolean value and modifies the abstract memory location pointed to by its parameter. In the caller's context, the side-effects of a function call are represented by a placeholder (summary operator) node. Each summary operator is named with the *function* : *effect* notation explained in Section 4.2.1. For example, the return value of a call to *flip* will be an operator node labeled *flip* : *ret*. Each summary operator represents a symbolic expression. Expansion of an operator corresponds to a round of

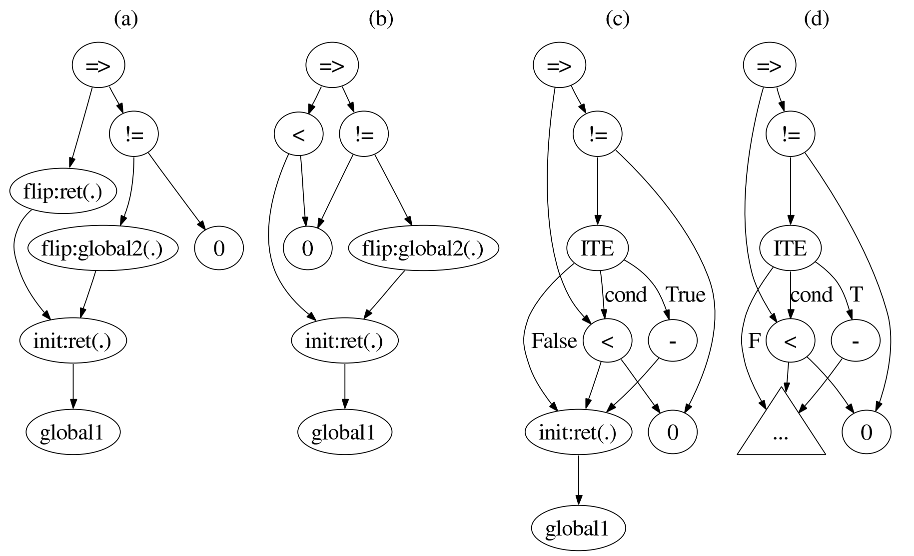


Figure 4.1: Structural Abstraction Example. Summary nodes are structurally refined in the following sequence: *flip* : *ret*, *flip* : *global2*, and finally *init* : *ret*. The subgraph obtained by the refinement of *init* : *ret* is represented by a triangle. For simplicity, these figures do not show pointer references and dereferences.

inlining.

To be fully context-sensitive, the obvious approach is to completely inline all calls. Such inlining leads to exponential blow-up even on small applications. A better approach is to track the individual effects of a function separately. This fine-grained approach makes it possible to expand only the slice of the called function that is actually in the cone of influence of the verified property. Together with the common subexpression elimination, this approach is more scalable, but does not offer satisfactory performance.

The crux of the problem is that interprocedural analysis can't decide when to stop inlining. After only three refinements, *-sensitive analysis would expand computationally expensive *init*, rendering the problem much harder for the decision procedure. However, the VC can be proven to be valid after only two refinements

$$\text{Let}\, x = init\,{:}\,ret(global1)$$

$$x < 0 \Rightarrow ITE(x < 0, -x, x) \neq 0$$

which simplifies to `true`, no matter what *init* returns. Cases like this appear frequently in practice, especially during *-sensitive verification of data-intensive properties, like checking of assertions or global pointer properties.

Structural abstraction initially considers all summary operators to be just unconstrained variables, and gradually refines the maximally-shared graph by expansion of selected summary operators, until the VC becomes valid, or the decision procedure finds a falsifying assignment that does not depend on any summary nodes.

4.1.3 Assumptions

Structural abstraction imposes only two requirements:

1. Each summary operator must represent a pure function. The symbolic execution presented in the previous chapter slices simulated functions into a number of pure sub-functions, represented by summary operators.

2. The maximally-shared graph representing a fully-refined VC has to be acyclic and of finite size. As long as there are no cycles in the CG, any VC can be represented as a finite maximally-shared graph (all function calls can be inlined).

4.1.4 Design Decisions

The undecidability of software analysis in general and the focus on scalability motivated me to make a number of simplifying design assumptions. The presented analysis assumes that CFGs are not recursive and that there are no exceptions in the code. Also, the analysis is oblivious to concurrency. The discussion below justifies those simplifying design decisions and addresses possible ways for eliminating them.

Recursion. Tail-recursive calls can be transformed into loops [4], and loops can be replaced with expressions that represent their invariants (either manually or through abstract interpretation [39]), which can then be handled by structural abstraction. General recursion (not tail-recursion) cannot be handled by structural abstraction, unless the user provides pre- and post-conditions of all functions involved in a recursive cycle.

Exceptions. If the symbolic execution presented in the previous chapter could build the effects of exceptions into the summaries and VCs, structural abstraction could reason about exceptions as well. However, the current analysis does not attempt to be smart about exceptions, and this is left for future work.

Concurrency. Recent work on iterative context-bounding by Musuvathi and Qadeer [99] and later work by Bouajjani et al. [23] could be used in combination with structural abstraction: partial order reduction [67] performed directly on the source could generate a sequence of different interleavings of the source (or the intermediate form), which could then in turn be passed to a tool based on structural abstraction, like Calysto. However, it is very unlikely that this would be efficient. A more efficient approach would require a tighter integration of structural abstraction and analysis of different interleavings.

4.1.5 Contributions

Structural abstraction and refinement have several advantages over other abstraction-based approaches to interprocedural analysis, listed below.

Incrementality. The abstraction-checking-refinement loop is strictly incremental because it always just keeps inlining symbolic expressions in place of summary operators. This is very important in practice because decision procedures use learning to speed up the solving and to avoid revisiting the same search space. So, if that knowledge is discarded between iterations, the decision procedure has to re-learn all the facts it had already learned. This is clearly a waste of resources. Abstractions that learn new facts needed for refinement from the proof of unsatisfiability, like predicate abstraction [13], cannot be completely incremental, because once a formula becomes unsatisfiable, adding new constraints does not change its status. Structural abstraction, on the other hand, uses *satisfying*[18] assignments for refinement, and therefore is fundamentally different from other proposed approaches.

Cheap Abstraction. Structural abstraction is very cheap, because abstraction is performed by *avoiding computation*, rather than by computing potentially expensive abstractions.

Cheap Refinement. Structural refinement is very efficient as well: given a falsifying assignment, the refinement walks over the maximally-shared graph representing a VC, and if it runs into a summarized node, expands it (refining the VC). Thus, structural refinement performs a number of steps that is at most linear with respect the size of the maximally-shared graph at the current iteration plus the size of the maximally-shared graph of the expanded summary operator.

4.2 Interprocedural Analysis

This section explains how the intraprocedural analysis presented in the last chapter can be extended into an interprocedural one (taking the assumptions listed in Sections 3.1.2 and 4.1.3 into account). More specifically, Section 4.2.1 explains handling of different types of side-effects, while Sections 4.2.2 and 4.2.3 introduce structural abstraction and refinement. Finally, Section 4.2.4 suggests possible improvements.

[18] *Falsifying*; if we are talking about validity.

4.2.1 Side-Effects

Functions can have multiple side-effects. This section begins by discussing several possible types of side-effects, and explains the *function* : *effect* notation for naming summary operators for each type. The second part of this section shows how the intraprocedural symbolic execution presented in the previous chapter can be extended to handle side-effects of function calls.

Types of Side-Effects. A called function can influence the caller's state in multiple ways, for instance by modifying abstract memory locations reachable through pointers passed as parameters, or by influencing the control-flow context (through assertions, assumptions, and aborts). The presented analysis distinguishes the following side-effects:

Returned value represents the result returned by the called function, and this effect is going to be represented with a keyword *ret*. So, a summary operator representing the returned value of *foo* would be denoted as *foo* : *ret*.

Abstract memory locations that are visible in the caller's context and modified in the callee's context cannot be represented with a finite number of summary operators, in general, because software can be an infinite-state system. However, assuming (1) logical memory model, (2) unrolling (or invariants) of loops, and (3) absence of recursion, the number of such modified abstract memory locations becomes finite.[19]

In the *function* : *effect* notation, those effects are going to be represented by expressions from the following grammar:

$$\begin{aligned} \mathit{effect} &::= *\mathit{eff} \\ \mathit{eff} &::= \mathit{termptr} \mid (\mathit{eff} + \mathcal{Z}) \mid *\mathit{eff} \end{aligned}$$

[19] Even with a more complex memory model based on the theory of non-extensional arrays [124], one could track a possibly infinite number of modified locations, while only a finite number of summary operators would be needed.

Where $\mathcal{Z}$ is an integer offset, terminal pointers are denoted as *termptr*, and $*$ is a dereference operator. For instance, if function $foo(p)$ modifies an abstract memory location at $p + 8$, the corresponding summary operator will be labelled with $foo : *(p + 8)$.

Control-flow context of the caller can be impacted by assertions, assumptions, and aborts executed in the context of the callee. This side-effect is going to be represented with a keyword *cfc*.

Verification conditions are very similar to the control-flow context (CFC) effects, because both can be represented as Boolean summary operators and can't be dereferenced. Unlike the CFC effects, which need to be assumed in the caller's context, VCs need to be asserted. This process of pushing the callee's VCs into the caller's context is going to be called VC lifting. The simplest (and least effective) technique is to lift VCs all the way to the root of the CG, at which point they are guaranteed to be fully interprocedurally path-sensitive. Section 4.2.4 discusses more advanced techniques for dealing with VCs.

Side-Effects and Symbolic Execution. In most cases, the intraprocedural analysis presented in the previous chapter can be easily extended to an interprocedural analysis. The rest of this section analyzes handling of the function call statement.

Handling the effect of a function call on the control-flow context is particularly simple: given a summary operator $foo : cfc$ that represents the effect of calling *foo* on the caller's context, symbolic execution only has to conjoin $foo : cfc$ with the control-flow context predicate in the caller's context. If $foo : cfc$ becomes relevant to any VCs, it can be lazily expanded later.

VCs can be handled in the same way if the interprocedural analysis constructs only one VC for the entire program. However, that is rarely useful in practice — constructing multiple VCs (one VC per assertion and per calling context) usually gives much more fine-grained feedback to the programmer, and facilitates elimination of multiple points of failure in a single run of the static checker.

The most important problem of having multiple VCs is that each VC that depends on the calling context has to be lifted and asserted in the calling context. Obviously, this can lead to exponential blowup in the number of lifted VCs. Section 4.2.4 gives the details on how this blowup can be largely avoided in practice.

Unlike the previous two side-effects, the return values and the modifications of abstract memory locations can also represent a pointer, which can be dereferenced in the calling context. To keep the symbolic execution fully precise, such summaries need to be expanded if dereferenced. Calysto expands summaries heuristically — if the number of nodes in a summary is smaller than user-defined bound, the summary is expanded, otherwise the dereferenced pointer is considered to be a terminal pointer. In the worst case, such an expansion can cause an exponential blowup, but such cases seem to be rare in practice — it is rare that a long sequence of function calls all modify the same memory location in a complex manner. It is very important to note that the expansion of dereferenced summary operators is a design tradeoff, not a feature of structural abstraction — one could imagine modeling pointer reads and writes with the theory of arrays, which would make the VCs harder for the decision procedure, but would avoid the potential exponential blowup during symbolic execution.

4.2.2 Abstraction

The proposed approach follows the general paradigm of automatic, counter-example-guided, abstraction refinement [32], but unlike typical CEGAR approaches, structural abstraction and refinement operations are entirely structural, and the refinement works incrementally on abstract counterexamples (rather than concretizing the abstract counterexample, proving it spurious, and then analyzing the proof). Locations modified by a function call (either indirectly through a pointer, or directly via returned values) are initially considered to be unconstrained variables. Those unconstrained variables are incrementally refined until the formula represented by the graph becomes valid, or the falsifying assignment does not depend on any unconstrained variables. Incremental refinement performs structural refinement on maximally-shared graphs through expansion of summary operators. The refinement step replaces an unconstrained variable with a subgraph that represents the summary expression and the edges that were pointing to the unconstrained variable are relinked to point to the newly constructed expression. Algorithm 3 is a high-level rendition of the structural abstraction-refinement cycle.

Algorithm 3 Main Abstraction-Checking-Refinement Loop. The checked VC is represented by a root F in the maximally-shared graph. The algorithm encodes F (calling $encode()$) on the fly into formula f and passes it to the decision procedure ($solve()$). Summary nodes are encoded as unconstrained variables. If the decision procedure proves f valid, we are done. Otherwise, refinement takes F and the table of current assignments to variables in $\mathsf{supp}(F)$, and returns true if the graph was refined, and false otherwise. If the graph was not refined, then all the summary nodes related to the falsifying assignment have been expanded, and the main loop terminates. Otherwise, the abstraction-checking-refinement cycle continues. Since maximally-shared graphs are acyclic, the algorithm necessarily terminates.

Let F be a node in the maximally-shared graph representing some VC.
$f = encode(F)$
while $\neg solve(f)$ **do** ▷ *solve* returns false if a falsifying assignment is found
    if $\neg\textsc{Refine}(F, current\_solution)$ **then**
        Report a bug and exit.
    end if
end while
Report VALID and exit.

The algorithm interacts gracefully with incremental decision procedures — each expansion of a summary node replaces only a single node with the expression represented by the summary node, monotonically increasing the set of constraints.

The abstraction is done in the *encode* and REFINE functions in the cheapest way possible — by avoiding computation, or more precisely, by avoiding expansion of summary operators. Other abstractions, like predicate abstraction [13], typically perform relatively expensive recomputation of the abstractions in each abstraction-checking-refinement cycle.

This lazy approach to interpretation of function summaries resembles the intuition behind lazy proof explication [62], a technique used to bridge between different theories in a theorem prover. The shared intuition is to abstract away expensive reasoning — expanding a function summary or solving a sub-theory query — as unconstrained variables, and then constrain them lazily, only as needed to refute solutions to the abstracted problem. The specifics of what to abstract and how to refine, of course, are different, since I am solving a different problem.

4.2.3 Refinement

The first few iterations of the main loop of Algorithm 3 will likely return false counterexamples, since the initial abstraction is usually very crude. So, the refinement algorithm has to identify very quickly a set of summary nodes that are relevant to the falsifying assignment.

The algorithm attempts to minimize the number of expanded summaries to avoid expensive computation. Given a falsifying assignment, my refinement scheme searches the graph and selects a single summary node to expand, thereby refining the model. In particular, the algorithm starts traversing the formula from the VC root. During the traversal, the algorithm detects don't-care values — values that are irrelevant to the current solution and can therefore be ignored. I use the concept of absorptive element to formalize don't-care values:

Definition 9 (Absorptive Element)**.** *If there exists an element a for some operator $\star$, such that $\forall x : a \star x = a$, then a is an absorptive element of $\star$, denoted* *as* $\mathsf{abelem}(\star) = a$. *For instance,* $\mathsf{abelem}(\wedge) = \mathsf{false}$ *and* $\mathsf{abelem}(*) = 0$.

If the decision procedure returns a falsifying assignment, each node F in the graph representing the checked VC has some assigned value, which will be denoted as $\mathsf{val}(F)$. If F is an operator $\star$, the algorithm checks $\mathsf{val}(x)$ for each operand x of F. If $\mathsf{val}(x)$ is an absorptive element of $\star$, it is a sufficient explanation of the value of F in the falsifying assignment (the other operand is a don't-care). Hence, it suffices to refine only x. The refinement procedure is given in Algorithm 4. As is usual for graph traversal, visited nodes are marked during traversal to avoid re-visiting nodes; marking is not shown in the pseudocode.

Algorithm 4 Structural Refinement Algorithm. F is a node in the maximally-shared graph, and x and y are its operands. The return value indicates whether a summary has been expanded.

function REFINE(graph node F, values assigned to nodes)
    if F is a summary node **then**
        expand the summary for F; return true
    else if F is a leaf node **then**
        return false
    else if $F \equiv x \star y$ **then**
        if $\mathsf{val}(x) = \mathsf{abelem}(\star)$ **then**
            return REFINE(x)
        else if $\mathsf{val}(y) = \mathsf{abelem}(\star)$ **then**
            return REFINE(y)
        else
            return REFINE(x) **or** REFINE(y)
            (The **or** is lazy: if either call succeeds, the other is skipped.)
            (The order is arbitrary. Either x or y can be refined first.)
        end if
    end if
end function

Operators like implication and if-then-else can be rewritten in terms of simple operators (AND, OR) that have absorptive elements. The implementation of the refinement algorithm in CALYSTO avoids such rewrites for efficiency, and directly applies customized rules to both implication and if-then-else, according to the following algorithm:

Algorithm 5 Additions to the Basic Refinement Algorithm.

if $F \equiv (x \Rightarrow y)$ and $\mathsf{val}\,(x) \equiv \mathsf{false}$ **then** return REFINE(x)
else if $F \equiv (x \Rightarrow y)$ and $\mathsf{val}\,(y) \equiv \mathsf{true}$ **then** return REFINE(y)
else if $F \equiv (x \Rightarrow y)$ return REFINE(x) **or** REFINE(y)

if $F \equiv ITE(c, x, y)$ and $\mathsf{val}\,(c) \equiv \mathsf{true}$ return REFINE(c) **or** REFINE(x)
else if $F \equiv ITE(c, x, y)$ return REFINE(c) **or** REFINE(y)

Returning to the example in Fig. 4.1 on page 73, in 4.1(a), the checker treats the placeholder nodes as unconstrained variables and finds a falsifying assignment where *flip* : *ret* is true and the `!=` is false. Algorithm 5 will derive the refinement in 4.1(b), where a possible falsifying solution gives the *init* : *ret* node a negative value. Next, the algorithm might choose to expand the *flip* : *global2* node, yielding the refinement in 4.1(c), which is valid. Structural abstraction is able to avoid the expensive expansion of the *init* : *ret* node.

Unlike other approaches, this approach to refinement does not require a theorem prover. The downside is that such a refinement might be less precise and result in more refinement cycles. However, each refinement cycle only adds additional constraints to the decision procedure incrementally, making the solving phase more efficient as well.

4.2.4 Improvements

Experiments revealed that the two most significant bottlenecks of naïvely implemented structural abstraction are: (1) lifting of VCs to the calling context exhausts available memory on large programs, and (2) the total run time of a static checker based on naïve structural abstraction is impractically long (a couple of days for a ~100 KLOC program). This section discusses implemented and further possible improvements. The improvements that have been already implemented significantly improve the overall performance and decrease memory consumption. The first part of this section focuses on the lifting of VCs, the second one on better refinement heuristics that localize the analysis, dramatically improving the overall performance, and the third part discusses the specifics of the implementation of common subexpression elimination in Calysto.

Lifting. A simple, but effective, approach that avoids complete lifting of VCs to the root of the CFG is to avoid lifting VCs that are completely defined in terms of constants and local variables. If such a VC is valid, it is going to be valid in any context (and vice versa). Calysto uses this approach and manages to handle several hundred KLOC without exceeding the memory resources. The only downside of this approach is that the checker can report bugs in unreachable code. For that reason, Calysto relies on the LLVM dead-code elimination pass and also implements an additional pass that eliminates functions that can never be called. This additional pass can be configured through user-controlled switches. For instance, a paranoid user who is checking a library might want to check all functions, even those with internal linkage that are not visible from the outside. In my own experience, programs either contain trivially unreachable functions, or almost all functions are reachable. Thus, these dead-code elimination passes are sufficient to eliminate almost all cases when the checker would report a warning in unreachable code.

An even better approach would be to check each VC before lifting — if a VC is already valid, there is no need to lift it any further. Likewise, if a falsifying assignment does not depend on any parameters or globals, the checker could report a bug (the same argument as above, about warnings in unreachable code, applies). This technique, dubbed localized structural abstraction, would require significantly less memory resources (because most VCs can be proven with very local reasoning, at least in my experience), and would also be applicable to checking of libraries, where unconstrained interfaces result in a large number of false positives. However, achieving incrementality with this technique would be much harder, because one would need to carry around the state of the theorem prover. With a specially designed theorem prover, this could be an interesting tradeoff, definitely requiring further research.

Heuristics. The order in which branches are refined in Algorithm 4 actually does matter. An especially effective heuristic that I found is to refine the consequents of implications before the antecedents. With that optimization, Calysto runs approximately 2–3 orders of magnitude faster. The rest of the section explains why this heuristic is so effective.

Each time a VC ϕ is lifted to the calling context, a new VC is constructed that includes the control-flow context in the callee ψ, so we get $\psi \Rightarrow \phi$. Further lifting prepends more antecedents, resulting in a formula that looks like: $\psi_0 \Rightarrow \cdots \Rightarrow \psi_n \Rightarrow \phi$. The number of antecedents is equal to the number of calls in the call chain, and usually, there is some expression sharing among different antecedents.

In practice, if programmers perform any checking (for instance, that a pointer is not `NULL`), usually those checks will be very local to the point where the checked property matters (for instance, where the pointer is dereferenced). So, very often properties can be proven with a very local analysis. If that is the case, an abstraction that avoids analyzing too many contexts actually works faster because it can prove the property to hold with very local reasoning. Refining consequents before antecedents achieves exactly that effect and localizes the analysis.

It is very possible that other heuristics would improve the performance of structural abstraction even further, but that is left for future research.

Merging Common Summary Operators. Elimination of common subexpressions can also be applied to summary operators, at least those that summarize deterministic functions. If a function is non-deterministic, Calysto flags all the side-effects that are non-deterministic with a special flag. Flagged summary operators can never be merged, because they could evaluate to a different result even if all the inputs are identical. In other words, summary operators are merged if and only if they represent the same fully deterministic side-effect and they have (structurally) equivalent operands. Common sources of non-determinism are calls to external functions (like `random()`), for which the code is not available for analysis.

Chapter 5

A Decision Procedure for Bit-Vector Arithmetic

The previous two chapters discussed the intraprocedural analysis for computing function summaries and the structural abstraction that assembles and proves interprocedural VCs. Structural abstraction often avoids the full expansion of a VC, and makes the interprocedural path-sensitive analysis feasible. However, even if the analysis is localized (as discussed in Section 4.2.4), there could be hundreds of thousands of VCs that a decision procedure has to (dis)prove. The large number of VCs and their complexity (due to interprocedural path-sensitivity) put a heavy load on the decision procedure.

This chapter discusses my bit-vector arithmetic decision procedure, Spear, which I designed especially for software analysis. While structural abstraction in Calysto achieves the precision of interprocedural path-sensitivity, Spear provides the raw speed required to make the Calysto static checker practical. More specifically, this chapter focuses on the design of an application-specific decision procedure that is capable of exploiting the properties and structure of Calysto's VCs.

The presented insights and contributions generalize to other problem domains, as demonstrated by experimental results in Section 5.2.3. Even though my main motivation for designing Spear was to solve VCs produced by Calysto as quickly as possible, thanks to a highly modular and configurable design, Spear has been (automatically) optimized for a number of different classes of problems. The end result of such automatic optimization is superior perfor-

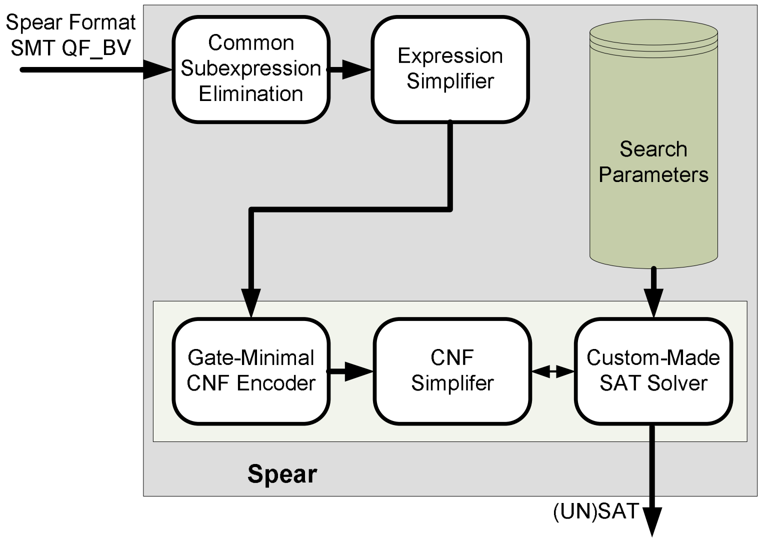


Figure 5.1: Architecture of Spear. In stand-alone mode, the solver can take as input a formula either in the logic described in Fig. 2.1 on page 14 or in the SMT QF_BV standard format [108] through a translator provided with Spear. Simplified formulas are translated into CNF and passed to the custom-made SAT solver. The solver takes the formula and a suitable parameter configuration from a simple database. During solving, the solver occasionally calls the CNF simplifier.

mance on a number of different classes of problems.

5.1 Architecture

This section describes Spear's architecture, illustrated in Fig. 5.1. The main components — expression simplifier, CNF encoder, dynamic CNF simplifier, and the custom-made SAT solver — are discussed in this section, while the role of a simple database with search parameter configurations will be clarified in Section 5.2.

5.1.1 Expression Simplification

Spear simplifies expressions in two steps: First, common subexpression elimination merges (structurally) equivalent expressions. Second, a term-rewriting engine simplifies the expressions starting at the leaves of the maximally-shared graph representing the formula and moving toward the root. The engine performs operations like constant-propagation (e.g., $a + 0 = a$), constant-collection (e.g., $a + 1 + 2 = a + 3$), simple deduction (e.g., $a < b \wedge a > b = \mathsf{false}$), redundancy elimination (e.g., $ITE(\phi, a, a) = a$), partial canonicalization (e.g., $ite(\phi, \phi \wedge a, b) = ite(\phi, a, b)$), strength-reduction (e.g., $3 * a = a << 1 + a$), and so on. This section explains how these two steps work in Spear.

Common Subexpression Elimination (CSE) is a simple, but very important optimization for decision procedures based on SAT solvers. Spear eliminates common subexpressions by simple structural hashing. The following example illustrates the importance of CSE:

Example 7:
Let us assume that we have a bit-vector arithmetic formula $a * b \neq a * b$, where $*$ represents multiplication, while a and b are bit-vectors of length N. If such a formula is bit-blasted and passed to a SAT solver, even the best SAT solvers available today[20] cannot solve the formula in a reasonable amount of time for $N > 11$. However, with trivial CSE, we get something like: $x = a * b \wedge x \neq x$, which becomes trivial even for very large N.

That is a trivial example that shows that CSE can exponentially speed-up decision procedures based on SAT solvers.

The term rewriting engine in Spear is based on approximately 160 hard-coded simplification rules[21], which can be broadly classified into several cat-

[20] As of May 2008.

[21] Different versions of Spear have different number of simplification rules. In general, I remove a rule when I discover that it is already covered by a combination of simpler rules, and add a rule when I notice an opportunity for simplification.

egories: constant-propagation, constant-collection, simple deduction, redundancy elimination, partial canonicalization, and strength reduction. In general, the engine is not recursive: it looks only at the operator and its operands, and very rarely recurses deeper into the subexpressions.[22] This was a deliberate decision to speedup the simplifier — such an approach is a good-enough solution for the common cases appearing in software analysis, leaving the hard cases to the SAT solver later. While this customization gives superior performance on Calysto's VCs, it is relatively easy to come up with small hard problems that defy Spear's simplifier.

5.1.2 Conjunctive Normal Form Encoder

Programs contain non-linear operators, and to be bit-precise, one must have a decision procedure that supports them. A number of different methods have been developed for linear bit-vector arithmetic, but few of them are applicable to non-linear operators. The usual approach is bit-blasting: Variables are encoded as bit-vectors of suitable size, and operators are replaced by digital circuits corresponding to that operator. In effect, VCs become large digital circuits, which can be converted to CNF using Tseitin's transform [128] and given to a SAT solver.

Numerous circuits have been proposed for each standard operation. Choosing the right circuit for CNF encoding is a little-researched but important problem — properly selected circuit can easily make the decision procedure an order of magnitude faster. The heuristic I found most effective is to use gate-optimal circuits, i.e., circuits that have the minimal number of gates. Such circuits tend to generate the fewest variables during the encoding, thereby avoiding flooding the SAT solver with redundant variables, which tend to confuse the solver's decision heuristics.

For example, the encoder creates an instance of Guild's array divider [70] circuit for each division operator that takes two variable inputs. Guild's array

[22] Some more complex rules require the engine to peek several subexpressions deep.

divider is very compact, as well as regular, making it an ideal choice for the CNF encoding. Operations with constants are encoded using a wide variety strength-reduction techniques [131], e.g., signed division of a 32-bit n by 13 is done as follows:

$$\begin{aligned}
n &= n + ((n \ Ashr\ 31) \wedge 12) \\
q &= (n \ Ashr\ 1) + (n \ Ashr\ 4) \\
q &= q + (q \ Ashr\ 4) + (q \ Ashr\ 5) \\
q &= q + (q \ Ashr\ 12) + (q \ Ashr\ 24) \\
q &= q \ Ashr\ 3 \\
REMAINDER &= n - 13q \\
QUOTIENT &= q + ((REMAINDER + 3) \ Ashr\ 4)
\end{aligned}$$

For the meaning of operators, see Fig. 2.1 on page 14.

Although the encoder tries to be gate-optimal, it can still create redundant variables, especially on the interfaces between multiple operators. The following section describes an effective way to eliminate such redundancies — dynamic CNF simplification.

5.1.3 Dynamic Conjunctive Normal Form Simplifier

Developing Spear and HyperSAT [6], I experimented with a large number of static and dynamic CNF simplification techniques. The only two techniques that I found to consistently improve performance on a wide set of industrial instances are: variable elimination [55] and elimination of satisfied clauses and falsified literals [56]. This section focuses on the variable elimination technique, which is the more important of the two for Calysto's VCs.

Other techniques that I have tried did not consistently improve the solver's performance. Some of those techniques are elimination of subsumed clauses [55], the pure-literal rule[23] [45], symmetry-breaking [60], and equivalence preprocessing [130]. I also developed a number of novel techniques that I haven't

[23] Called the *affirmative-negative* rule in the original paper.

found particularly effective so far: identification of clusters in formulas for localizing the search process, the B-cubing [6] learning technique, and dynamic don't-care analysis that was supposed to prevent the solver case-splitting on irrelevant variables[24]. However, I haven't researched the full potential of those three techniques.

Variable elimination is a simple and effective technique based on resolution. For example, given a variable v that appears in only two clauses[25], $(D_1 \vee \neg v)$ and $(D_2 \vee v)$, where D_1 and D_2 are disjunct of literals, variable v can be eliminated and those two clauses can be replaced with a resolvent $(D_1 \vee D_2)$.

The technique was originally proposed by Eén and Biere [55] and implemented in SatELite. SatELite heuristically selects variables for elimination, and eliminates them by resolving each clause in which a selected variable appears as a positive literal with each clause in which the variable appears as a negative literal. In the worst case, such resolution can quadratically increase the number of created clauses, so in practice, the number of clauses that can be generated is bounded heuristically. SatELite's variable elimination heuristics are not tuned to Spear's encoder and the structural properties of Calysto's VCs. Extensive automatic tuning (Section 5.2) optimized the heuristics used for variable elimination as follows:

Order of Elimination. Variable elimination in Spear sorts variables for elimination by the number of occurrences in all clauses in the clause database (both original and learned clauses), and starts with the rarest variables.

Type of Elimination. Only original clauses are resolved, while the learned clauses that contain the eliminated variable (either as a positive, or a negative literal) are simply discarded.

Frequency of Elimination. The dynamic simplifier runs variable elimination

[24] Done in collaboration with Leonardo de Moura and Nikolaj S. Bjørner.

[25] A literal is a Boolean variable or its negation. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals in which each variable appears at most once.

only after restarts, but not necessarily after each restart. In total, variable elimination can be run at most sixteen times.

Preconditions for Elimination. A variable v can be only eliminated if all the following conditions hold:

Clause Cutoff. Variable v appears in at most four clauses.

Literal Cutoff. The total sum of the number of literals of all clauses in which v appears is less than one hundred.

Max Clause Number Increment. The total number of new clauses after the resolution is allowed to be only up to four times as large as the number of original clauses.

5.1.4 Custom-Made SAT Solver

The core of Spear is a DPLL-style [44] SAT solver. Spear incorporates a number of novel optimizations, as described later in Section 5.3. However, one of the most important features of Spear and its core is configurability: every single search parameter, which is typically hard-coded in most off-the-shelf decision procedures, can be set on the command line. The full flexibility and power of this configurability becomes obvious in the following section on automatic tuning.

5.2 Automatic Tuning

The work presented in this section is a joint work with Frank Hutter, Holger H. Hoos, and Alan J. Hu [78].

This section surveys the problems related to manual tuning, which is the traditional way of optimizing decision procedures, discusses the local search technique used for automatic optimization of Spear, gives experimental results, and finally discusses the best automatically found parameter configuration for solving Calysto's VCs.

In practice, tuning a decision procedure for a specific class of problems is a challenging task. A high-performance decision procedure typically uses numerous heuristics that interact in complex ways. Some examples from the SAT-solving world include decision variable and phase selection, clause deletion, next watched literal selection, and initial variable ordering heuristics (e.g., [120, 97, 20]). The behavior and performance of these heuristics is typically controlled by parameters, and the complex effects and interactions between these parameters render their tuning extremely challenging.

In this section, I explain how ParamILS, a recent parameter optimization tool developed by Hutter et al. [79], was used for optimization and development of Spear. ParamILS optimized Spear for a number of different classes of problems, computing a single parameter configuration for each class. These search parameter configurations are stored in the simple database (see Fig. 5.1 on page 88). Users can select different parameter configurations for different problem classes through command-line switches. Calysto automatically preselects the configuration optimized for its VCs. The presented technology offers significant performance improvements (up to two orders of magnitude) and avoids tedious and time-consuming manual tuning.

5.2.1 Manual Tuning

After the first version of Spear was written and its correctness thoroughly tested[26], I spent one week on manual performance optimization, which involved:

[26] Every release of the SAT solver in Spear was tested with 30 million random CNF instances. That number seems to be sufficient to uncover even the most elusive correctness bugs. Frequently, a few thousand random instances suffices to reveal a bug in a SAT solver. If Spear's SAT solver found a test instance to be unsatisfiable, the instance was also checked with another SAT solver. The encoding of more complex operators (multiplication, division, remainder) was tested with approximately two million different tests for each bit-width. In effect, all complex operations that take operands with twenty or fewer bits were tested exhaustively. As complex operators are implemented through the simpler ones (addition, bit-wise operators,...), the simpler operators were implicitly tested as well. The expression simplifier has not been thoroughly tested, but I carefully reviewed that code multiple times. A more

(i) optimization of the implementation, resulting in a speedup by a constant factor, with no effects on the search parameters, and (ii) manual optimization of roughly twenty search parameters, most of which were hard-coded and scattered around the code at the time.

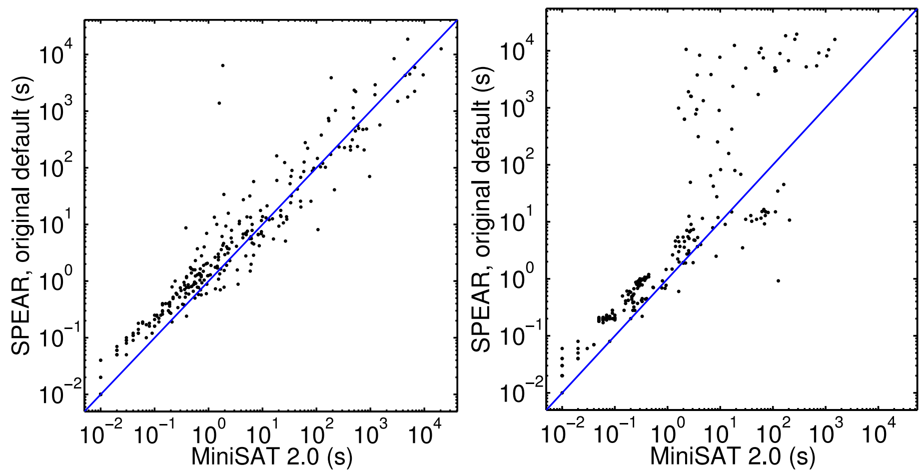
The manual parameter optimization was a slow and tedious process done in the following manner: I collected several medium-sized benchmark instances which SPEAR could solve in at most 1000 seconds and attempted to come up with a parameter configuration that would result in a minimum total runtime on this set. The benchmark set was very limited and included several medium-sized Bounded Model Checking (BMC) and some small Software Verification (SWV) instances generated by the CALYSTO static checker.[27] Such a small set of test instances facilitates fast development cycles and experimentation, but has many disadvantages.

Quickly, it became clear that implementation optimization gave more consistent speedups than parameter optimization. Even on such a small set of benchmarks, the variations due to different parameter settings were huge (2–3 orders of magnitude). Given the costly and tedious nature of the process, no further manual parameter optimization was performed after finding a configuration that seemed to work well on the chosen test set.

Fig. 5.2 on the following page compares the performance of this manually tuned version of SPEAR against MiniSAT 2.0 [56], the winner of the industrial category of the 2005 SAT Competition and of the 2006 SAT Race. The two sets of instances used for this experiment, BMC and SWV, will be introduced in detail in Section 5.2.3 As can be seen from the runtime correlation plots shown in Fig. 5.2, both solvers perform quite similarly for BMC and easy SWV. For difficult SWV instances, however, MiniSAT clearly performs better. This seems to be the effect of focusing the manual tuning on a small number of easy

systematic approach — testing through randomly generated bit-vector benchmarks — is left for future work.

[27]Small instances were selected because CALYSTO tends to occasionally generate very hard instances that would not be solved within a reasonable amount of time without automatic tuning.



(a) Bounded Model Checking (BMC) (b) Software Verification (SWV)

Figure 5.2: MiniSAT 2.0 vs. Manually Tuned Spear. (a) The two solvers perform comparably on BMC instances, with average runtimes of 298 seconds (MiniSAT) vs. 341 seconds (Spear) for the instances solved by both algorithms. (b) Performance on easy and medium software verification instances is comparable, but MiniSAT scales better for harder instances. The average runtimes for instances solved by both algorithms are 30 seconds (MiniSAT) and 787 seconds (Spear).

instances.

For most decision procedures, the process of finding default (or hard-coded) parameter settings resembles the manual tuning described above. During the typical development process of a heuristic solver, certain heuristic choices and parameter settings are tested incrementally, typically using a modest collection of benchmark instances that are of particular interest to the developer. Many choices and parameter settings thus made are "locked in" during early stages of the process, and frequently, only a few parameters are exposed to the users of the finished solver. Furthermore, most users of these tools do not change these settings, and when they do, they typically apply the same manual approach. Not surprisingly, this manual configuration and tuning approach usually fails to realize the full performance potential of a heuristic solver.

5.2.2 Parameter Optimization by Local Search

This section gives a quick overview of ParamILS [79] — the tool used for automatic tuning of Spear. More specifically, the section focuses on the underlying ParamILS algorithm, which is motivated by the following manual parameter tuning technique often used by algorithm developers:

- Start with some parameter configuration
- Iteratively, modify one algorithm parameter at a time, keeping the modification if performance on a given benchmark set improves and undoing it otherwise.
- Terminate when no single parameter modification yields an improvement, or when the best configuration found so far is considered "good enough".

Notice that this is essentially a simple hill-climbing local search process, and as such it will typically terminate in a locally, but not globally, optimum parameter configuration, in which changing any single parameter value will not achieve any performance improvement. However, since parameters of heuristic algorithms are usually not independent, changing two or more parameter values at the same time may still improve performance.

The problem of local optima is ubiquitous in local search, and many approaches have been developed to effectively deal with them; one of these approaches is *Iterated Local Search (ILS)* [105, 77], which provides the basis for ParamILS. ILS essentially alternates a subsidiary local search procedure (such as simple hill-climbing) with a perturbation phase, which lets the search escape from a local minimum. Additionally, an acceptance criterion is used to decide whether to continue the search from the most recently discovered local minimum or from some earlier local minimum. More precisely, starting from some initial parameter configuration, ParamILS first performs simple hill-climbing search until a local minimum c is reached, and then it cycles through the following phases:

1. apply perturbation (in the form of multiple random parameter changes);
2. perform simple hill-climbing search until a new local minimum c' is reached;
3. accept the better of the two configurations c and c' as the starting point of the next cycle.

ParamILS thus performs a biased random walk over locally optimum parameter configurations. To determine the better of two configurations, it can use arbitrary scalar performance metrics, including expected runtime, expected solution quality (for optimization algorithms), or any other statistic on the performance of the algorithm to be tuned when applied to instances from a given benchmark set. This benchmark set is called the *training set*, in contrast to the *test sets* that are used later for evaluating the final parameter configurations obtained from ParamILS. As is customary in the empirical evaluation of machine learning algorithms, training and test sets are strictly disjoint.

Clearly, the choice of the training set has important consequences for the performance of ParamILS. Ideally, the training set is homogeneous, i.e., the impact of parameter settings on the performance of the algorithm to be tuned is similar for all instances in the set. Frequently in practice, instances produced from a specific domain (here, software) and by a specific tool (here, Calysto) are fairly homogeneous.

A more advanced version of ParamILS that was used for Spear tuning adaptively chooses the number of training instances to use for each parameter setting: while poor settings can be discarded after a few algorithm runs, promising ones are evaluated on more instances. This mechanism avoids over-fitting to the instances in the training set. (For details, see [79].)

5.2.3 Experimental Setup

The automatic tuning technique described in the previous section was successfully used to optimize Spear for a number of different problem domains, namely: Calysto instances, BMC [31], graph coloring [66], and quasigroup completion [68]. This section presents the experimental results on the first two mentioned problem domains. After introducing the training and test instances, this section specifies the experimental environment, and finally quickly lists the heuristic search parameters of the optimized version of Spear.

Benchmark Sets. The BMC set consists of 754 IBM bounded model checking instances created by Zarpas [137], and the SWV set is comprised of 604 verification conditions generated by Calysto, as described in the previous chapters. Both instance sets, BMC and SWV, were split 50:50 into disjoint training and test sets. Only the training sets were used for tuning, and all the reported results are for the test sets.

Experimental Environment. All experiments were carried out on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB Random Access Memory (RAM), running OpenSuSE Linux 10.1. Reported times are Central Processing Unit (CPU) times per single CPU. Runs are terminated after 10 CPU hours or when they run out of memory and start swapping; both of these conditions are counted as time-outs.

Search Parameters. The availability of automatic parameter tuning encouraged me to parameterize many aspects of Spear. The first automatically tuned version exposed only a few important parameters, such as restart frequencies and variable priority increments. The results of automated tuning of those first versions of Spear prompted me to expose more and more search parameters, up to the point where not only every single hard-coded parameter was exposed, but also a number of new parameter-dependent features were incorporated. This process not only significantly improved Spear's performance, but also has driven the development of Spear itself.

The resulting version of Spear used for the experiments reported in the following has 26 parameters:

- 7 types of heuristics (with the number of different heuristics available shown in parentheses):
  - Variable decision heuristics (20)
  - Heuristics for sorting learned clauses (20)
  - Heuristics for sorting original clauses (20)
  - Resolution ordering heuristics (20)
  - Phase selection heuristics (7)
  - Clause deletion heuristics (3)
  - Resolution heuristics (3)
- 12 double-precision floating point parameters, including variable and clause decay, restart increment, variable and clause activity increment, percentage of random variable and phase decisions, heating/cooling factors for the percentage of random choices, etc.
- 4 integer parameters which mostly control restarts and variable/clause elimination
- 3 Boolean parameters which enable/disable simple optimizations such as the pure literal rule

For each of Spear's floating point and integer parameters, I chose lower and upper bounds on reasonable values and considered a number of values spread uniformly across the respective interval. This number ranges from three to eight, depending on my intuition about importance of the parameter. The total number of possible combinations after this discretization is 3.78×10^{18}. By exploiting some dependencies between parameters, I reduced the number of distinctive configurations to 8.34×10^{17}.

5.2.4 Experimental Results

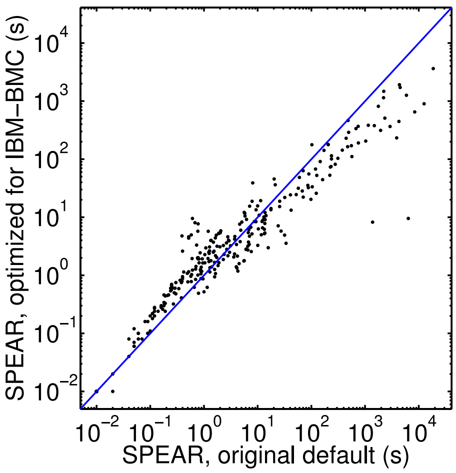
In practice, one typically cares about excellent performance on only a specific class of instances, such as BMC or SWV. This section gives experimental results of tuning Spear for these two specific sets of problems. Since users usually care most about an algorithm's total runtime, ParamILS used average (arithmetic mean) runtime as the tuning optimization objective.

The training cutoff time was set to 300 seconds, which according to Spear's internal book-keeping mechanisms turned out to be sufficient for exercising all techniques implemented in the solver. In order to speed up the optimization, 95 hard BMC instances, which could not be solved by Spear with its default parameter configuration within one hour, were removed from the training set, leaving 287 instances for training. The tuner ran on the cluster for three days in the case of SWV and for two days for BMC on the training set.

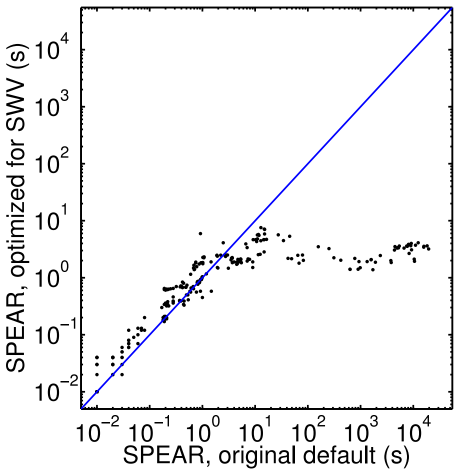
Fig. 5.3 on the next page shows the total effect of automatic tuning by comparing the performance of manually and automatically tuned Spear on the test set. For both the BMC and SWV sets, the scaling behavior of the tuned version is much better, and on average, large speedups are achieved — by a factor of 4.5 for BMC and 500 for SWV. Spear with the default settings even times out on four SWV instances after 10 000 seconds, while the tuned version solves every single instance in less than 20 seconds.

Table 5.1 on page 103 summarizes the performance of MiniSAT 2.0 and Spear with both the default and automatically tuned parameter settings. Notice that the versions of Spear specifically tuned for BMC and SWV clearly outperform MiniSAT: for BMC, Spear solves two additional instances and is faster by a factor of three on average; for SWV, the speedup factor is over 100.

The presented automatic tuning approach is both very effective and significantly simplifies the design of decision procedures: Heuristics are hard to design and evaluate, and because of that most modern decision procedures are brittle and (manually) over-tuned to specific classes of problems. Automatic tuning frees the decision procedure designers to invent novel heuristics, eliminates the



(a) Bounded Model Checking (BMC)



(b) Software Verification (SWV)

Figure 5.3: Overall Improvements Achieved by Automatic Tuning of Spear. The scatter plots (a) and (b) compare the manually engineered default parameter configuration vs. the automatically optimized version for the BMC and SWV sets. Results are on test sets disjoint from the instances used for parameter optimization.

(a) The default timed out on 90 instances after 10 000 seconds, while the tuned configuration solved four additional instances. For the instances that the default solved, mean runtimes are 341 seconds (default) and 75 seconds (tuned), a speedup factor of 4.5.

(b) The default timed out on four instances after 10 000 seconds, the tuned configuration solved all instances in less then 20 seconds. For the instances that the default solved, mean runtimes are 787 seconds (default) and 1.35 seconds (tuned), a speedup factor of over 500.

Solver	BMC		SWV	
	#(solved)	runtime for solved	#(solved)	runtime for solved
MiniSAT 2.0	289/377	360.9	302/302	161.3
Spear default	287/377	340.8	298/302	787.1
Spear tuned	291/377	113.7	302/302	1.5

Table 5.1: Summary of Automatic Tuning Results. For each solver and instance set, #(solved) denotes the number of instances solved within a CPU time of 10 hours, and the runtimes are the arithmetic mean runtimes for the instances solved by that solver. If an algorithm solves more instances, the shown average runtimes include more, and typically harder, instances. Note that the averages in this table differ from the runtimes given in the captions of Figures 5.2 and 5.3, because averages are taken with respect to different instance sets: for each solver, this table takes averages over all instances solved by that solver, whereas the figure captions state averages over the instances solved by both solvers compared in the respective figure.

need for tedious manual experimentation, and manages to find effective parameter configurations automatically, taking the properties and structure of the given class of instances into account.

5.2.5 Best Parameter Configuration Found

The analysis of the best parameter configuration found for Calysto's VCs gave some insights into their properties. According to the experimental results, the SWV instances prefer an activity-based heuristic that resolves ties by picking the variable with a larger product of occurrences. This heuristic might seem too aggressive, but helps the solver to focus on the most frequently used common subexpressions. It seems that a relatively small number of expressions play a crucial role in (dis)proving each verification condition, and this heuristic quickly narrows the search down to such expressions. The SWV instances also favored very aggressive restarts (first after only 50 conflicts), which in combination with our experimental results shows that most such instances can be solved quickly if the right order of variables is found. A simple phase selection heuristic (always assign `false` first) seems to work well for SWV, and also produces more natural bug traces (small values of variables in the satisfying assignments). The VCs used for tuning correspond to `NULL`-pointer dereferencing checks, and this phase selection heuristic attempts to propagate `NULL` values first (all `false`), which explains its effectiveness. Calysto's VCs prefer no randomness at all, which is probably the result of joint development of Calysto and Spear as a highly optimized tool chain for software verification.

The BMC instances seem to prefer a significantly different parameter configuration. Since the focus of this thesis is on the software analysis, the discussion of the best parameter configuration found for the BMC set is omitted and can be found in [78].

5.3 Improving Performance

The automatic tuning would not be very effective without having a choice of a wide range of heuristics and search parameters to tweak. The quality of choice is equally important as the quantity — if all heuristics are very similar, or if the search parameters do not have much impact on the dynamics of the search process, the size of combinatorial space is going to be pointlessly inflated. Such inflation of the combinatorial space dramatically reduces the effectiveness of automatic tuning.

This section begins by describing several novel heuristics that I implemented in Spear, and ends with a quick description of another optimization technique used in Spear— memory latency hiding.

5.3.1 Novel Heuristics

Spear features many novel heuristics and this section describes those that automatic tuning identified as a good match for certain classes of problems.

The standard activity-based (e.g., [56]) heuristic seems to be the heuristic of choice for picking variables for case-splitting. Spear uses slightly more complex versions of the activity-based heuristic: ties are resolved by picking the variable either with the maximal product, or with the maximal sum of positive and negative literal occurrences. Although Spear has been tuned for numerous classes of problems, the tuner consistently picks these two variants of the activity-based heuristic, although I implemented more than twenty different decision heuristics.

One of the most robust phase selection heuristics is the minimal-work heuristic [6]. After picking a variable for case-splitting, Spear picks the phase of the variable so as to minimize the amount of work that needs to be done, by picking the phase that satisfies a longer list of watched clauses. Although this heuristic does not perform well on Calysto's VCs, the tuner frequently identified it as the best heuristic on a number of other classes of problems.

Interestingly, the best order for variable elimination seems to be highly dependent on the properties and structure of the problem class. The tuner picked a different ordering heuristic for each class of problems. This might imply that the heuristics for ordering the variables for elimination are still immature, and that more research might be needed.

The heuristics for deletion of learned clauses seem to behave in a similar way: The tuner identified six out of seventeen clause sorting heuristics as good matches for various problem classes. The biggest surprise was the best configuration the tuner found for graph coloring problems — the clauses that most frequently participate in conflicts are deleted first. That finding contradicts all common explanations of how and why clause deletion should work.

5.3.2 Prefetching

According to the profiling I have done, Spear spends 60–80% of its time just waiting for the clauses to be fetched into the CPU cache. To hide this cache latency, I redesigned several algorithms in Spear, so as to be more predictable.

Afterward, I inserted memory prefetch instructions[28] at key places.

For example, the conflict analysis algorithm in MiniSAT [56] uses a first-in-first-out algorithm, while a first-in-last-out algorithm produces the same result and has a more predictable memory access pattern.

Inserting memory prefetches can also easily hurt the performance if there are too many outstanding prefetches or if the fetched memory locations are actually rarely used. For these reasons, it is important to properly guard the prefetch instructions and execute them only when there is a high probability that the fetched memory location will actually be needed. The amount of guarding has to be balanced with the cost of branch mispredictions, which can easily wipe out the advantage of prefetching.

With these techniques, I managed to speedup Spear ~30% on an AMD Athlon 64 X2 Dual Core 4600+ processor, which I used for development. The speedups on other processors were less remarkable, most likely due to different cache architectures and different costs of prefetches and mispredicted branches.

[28] Using the GCC `__builtin_prefetch` built-in function.

Chapter 6

Prototype and Experiments

This chapter begins by explaining how the structure-preserving symbolic execution (Chapter 3), structural abstraction (Chapter 4), and a custom-made bit-vector arithmetic decision procedure (Chapter 5) are architected together in the CALYSTO static checker. Each component has been carefully designed with the others in mind, resulting in a harmonious, balanced, and well-performing system.

The second part of the chapter presents the experimental results obtained on a wide variety of real-world applications. Despite the high cost of interprocedural path-sensitivity and bit-precision, experimental results show that CALYSTO achieves scalability comparable to a less-precise static checker, Saturn [135], which was especially designed to be highly scalable. The experimental results show that CALYSTO performs well on a wide variety of real-world applications, but also show that the symbolic execution and VC-checking components are nicely balanced (roughly 50% of time spent in each). Intuitively, that balance suggests that the architecture is well-conceived.

6.1 Architecture

The high-level architecture of CALYSTO is shown in Fig. 6.1. CALYSTO is designed as a compiler pass, in the spirit of Hoare's "verifying compiler" grand challenge [75]: it accepts the compiler's intermediate representation, in SSA form [43], performs various verification checks, issues bug reports and warnings, and then passes semantically unmodified SSA on to the compiler back-end. Designing a static checker in this manner has the obvious advantage of language

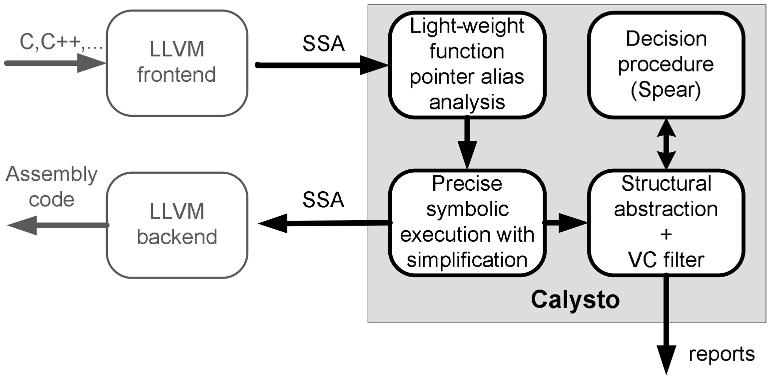


Figure 6.1: High-Level Calysto Architecture.

independence, but also helps to check for errors in the compiler front-end (and any other compiler passes that precede the Calysto pass). Supporting a different programming language requires only a different front-end and, if required, a name demangler to improve the legibility of bug reports.

Internally, the Calysto system consists of three stages, supported by the decision procedure, Spear. The first stage is a lightweight function pointer alias analysis. It constructs (a sound approximation of) the call graph, including indirect calls through function pointers. The function pointer alias analysis is sound, flow-sensitive, and context-insensitive. This pass requires negligible resources, yet is precise enough in practice.

The next stage is the symbolic execution presented in Chapter 3. Calysto symbolically executes functions in the analyzed program, computing symbolic definitions for each modified variable and memory location. These symbolic definitions are used to create VCs. The symbolic execution machinery allows generating VCs for any assertion at any point in the program. Calysto currently supports user-supplied assertions (written as Boolean expressions in whatever programming language the compiler front-end is parsing), but in the spirit of static checking also automatically generates VCs to check that each pointer dereference cannot be `NULL`.

The last stage has a dual purpose: (1) it performs structural abstraction and refinement, as described in Chapter 4, and (2) it also filters away all VCs corresponding to the same property within the same function. For instance, if a pointer that can be `NULL` is dereferenced at many different places within the same function, and that function can be called from many different contexts, Calysto will emit only one bug report per context. Such a filtering approach avoids overloading the programmer with reports that correspond to the same issue.

The third stage also emits a detailed graphical trace for each falsified VC; if the falsified VC depends on any global variables, the trace is given all the way from the root of the call graph (the *main* function). Although traces are detailed, reading them still requires an intimate understanding of the underlying property checking engine — Calysto's simplifier occasionally simplifies such large parts of formulas that it is very hard to build a complete mental model of the trace. Improving the usability of Calysto is left for future work.

The actual validity-checking of VCs is done by Spear, explained in detail in the previous chapter. In short, Spear is a sound and complete decision procedure that supports Boolean logic, bit-vector operations, and bit-accurate arithmetic. Unlike other static checkers, which use general-purpose SAT solvers or theorem provers, Spear is custom-designed for the software VCs generated by Calysto, optimizing performance.

6.2 Experimental Results

This section presents the experimental results of the combination of techniques presented in earlier chapters and prototyped in Calysto and Spear. More precisely, after introducing the benchmarks used for experiments, I give the experimental results of both Saturn and Calysto, and discuss my experiences with both static checkers.

Benchmark	LOC (total)	LOC (code)	Modules
Bftpd 1.8	4532	3306	1
Bftpd 1.9.2	4602	3368	1
HyperSAT 1.7	9123	6022	1
Spin 4.3.0	28394	20481	1
OpenSSH 4.6p1	81908	45304	11
INN 2.4.3	122727	71102	46
NTP 4.2.4p2-RC5	185865	74230	10
NTP 4.2.5p66	192019	74277	9
BIND 9.4.1p1	393318	184204	26
OpenLDAP 2.4.4a	374266	223595	27
TOTAL	1406754	685408	133

Table 6.1: Benchmarks Used for Experiments. The second and third columns show the number of lines of code before and after preprocessing ("LOC(code)" does not include comments, empty lines, and pragma-disabled code). The fourth column gives the number of compilation units produced by LLVM's front-end.

6.2.1 Benchmarks

To evaluate Calysto, I checked a number of publicly available, real-world applications: the OpenSSH remote access server and client, the InterNetNews (INN) system, the Network Time Protocol (NTP) system, the Berkeley Internet Name Domain (BIND) DNS system, and the OpenLDAP Lightweight Directory Access Protocol system. Those benchmarks are the largest open-source benchmarks that I could successfully compile with both LLVM's front-end and with the Saturn [135] static checker, which I used for comparison. I also used some smaller applications where I was able to get particularly prompt and precise feedback from the developers: the Bftpd FTP server, the HyperSAT Boolean satisfiability solver, and the Spin explicit-state model-checker. Table 6.2.1 lists the benchmarks.

I checked the automatically generated assertions that dereferenced pointers cannot be `NULL`. This is an excellent property to use to evaluate a static checker because: (i) pointers are often passed through a long sequence of calls, which necessitates interprocedural analysis, (ii) pointer manipulation in programs depends on both data- and control-flow of the program, exercising all the components of a static checker, and (iii) pointers are dereferenced very frequently in code — the number of produced VCs is probably larger than what would be generated by any other property (proper locking, for instance), and the sheer number of VCs pushes static checkers to their limits.

Initially, I started sending raw reports to developers, but quickly found that developers were very unwilling to separate out real bugs from false positives (inadvertently validating my research goal of minimizing false positives!). I began filtering the reports myself, omitting all reports that I could prove infeasible. All remaining reports were sent to the developers. At that point, I ran into an unexpected problem: the developers would either take a very defensive stance, claiming that a particular bug was either irrelevant or very improbable, or would take a very cautious stance, fixing everything in the code just to be safe, without much thought about whether a bug was feasible or not. To be as rigorous as possible in my reporting of experiments, I have defined a bug strictly as follows:

- Only a dereference of a pointer which is either uninitialized or `NULL` is considered a bug.
- There must exist a feasible path from the point where the pointer was initially defined to the point where it was dereferenced. For Calysto's evaluation, I also required that if any globals are included in the trace, the trace must be given all the way from the main function (root of the call graph). For Saturn's evaluation, I waived this constraint, because Saturn does not produce precise enough interprocedural traces.
- Every feasible `NULL` pointer dereference was considered a bug, no matter how improbable or irrelevant it might be.

- Many applications contained pointer checks. Usually, such checks exit if the pointer is `NULL` and print/log an appropriate message. In this case, the `NULL` pointer is never dereferenced, so failed pointer checks were not counted as bugs. In other words, only dereferences that would cause a segmentation fault were considered bugs.

- If a pointer ptr was guaranteed not to be `NULL`, then I also assumed that pointers with offset $ptr + i$ can never be `NULL` either. The likelihood of an integral overflow ($ptr > 0 \wedge ptr + i = 0$) is extremely remote, and software developers do not take such reports seriously. Checking that the offset is within allowed bounds is a different property, not to be confused with `NULL`-pointer checking.

For all reports that I could not prove to be false, I asked for a feasibility confirmation from the developers. Reports that neither I nor developers could prove to be feasible or infeasible are classified as unknown.

6.2.2 Saturn's Results

Saturn [135], an inspiration for Calysto, was designed for scalability from the start. Despite Calysto's more precise, more expensive analysis, can it match Saturn's proven scalability?

This section presents the results of running Saturn v1.1.[29] on the selected

[29] I am comparing against the most recent, most up-to-date version of Saturn available. An earlier version of Saturn reported low false error rates while checking for `NULL` pointer dereferences, although for a much less stringent notion of what constitutes a true bug [53]: in that paper, inconsistencies in whether a pointer is checked for `NULL` on different paths were included as real bugs; I am using the stricter definition described earlier. Unfortunately, the developers of Saturn have told me that they believe that the current version of Saturn is no longer as effective at identifying `NULL` pointer dereferences as the earlier version, that the previous version no longer exists, that they could not re-create it, and that they are not planning to update their `NULL` analysis to where they have confidence in it once again [2]. Thus, the only apples-to-apples comparison I can make, with the same definition of bugs, the same benchmarks, and the same machines, is with the current version of Saturn, which might not represent Saturn in the best possible light. Fortunately, the central point of the

benchmarks. I used only Saturn's `NULL` pointer analysis, which finds possible `NULL` pointer dereferences, and the best known parameter settings. Saturn's tutorial recommends using a 60 second timeout per function, so all the experiments presented in tabular form were obtained using the 60 second timeout. Experimental results for Saturn are presented in Table 6.2.

6.2.3 Calysto's Results

For Calysto, I used the default options, with a 10 second timeout per VC and limiting the number of VCs per function to 500, effectively setting the timeout per function to 5000 seconds. With a 5000 second timeout, Saturn produced the same results on Bftpd and Ntp and ran out of 16 GB of memory on all other benchmarks. Experimental results for Calysto are presented in Table 6.3 on page 115.

6.2.4 Discussion

Saturn's traces were significantly harder to interpret because the tool does not produce the complete trace and because I do not have the same level of familiarity with Saturn as I do with Calysto. Interestingly, there is very little overlap between the bugs reported by the two tools. Calysto tends to report either violations of C library properties, which Saturn frequently misses (presumably because of incomplete descriptions of C library functions), or very long traces, sometimes spanning through 10–15 functions, which Saturn misses due to lack of interprocedural path-sensitivity. On the other hand, most of bugs that Calysto missed were due to assertion violations (once an assertion is violated, all the code after it becomes unreachable) and loop handling — if a loop can never be executed only once, then the conjunction of the control-flow context and an

comparison is whether Calysto achieves comparable scalability to Saturn, despite performing analyses that are, by design, more precise and therefore presumably more expensive. Perhaps an earlier version of Saturn would have had precision closer to that of Calysto, but that question is moot.

Benchmark	LOC	Saturn v1.1				
		Reports	Bugs	Unkn.	FP Rate	Time [s]
Bftpd 1.8	3306	3	3	0	0%	129.51
Bftpd 1.9.2	3368	3	3	0	0%	105.17
HYPERSAT 1.7	6022	4	0	0	100%	647.21
Spin 4.3.0	20481	15	6	2	54%	2129.04
OpenSSH 4.6p1	45304	14	0	1	100%	3707.81
INN 2.4.3	71102	288	*12	34	96%	9879.69
NTP 4.2.4p2-RC5	74230	8	0	0	100%	326.44
NTP 4.2.5p66	74277	10	0	0	100%	319.33
BIND 9.4.1p1	184204	951	0	0	100%	14984.72
OpenLDAP 2.4.4a	223595	163	*14	50	88%	8098.48
TOTAL	685408	1459	38	87	97%	40226.40

Table 6.2: `NULL` Pointer Dereference Checking Results. "LOC" indicates the number of lines of true (after preprocessing) code. "Reports" is the total number of warnings produced on the benchmark. "Bugs" is the number of true bugs found. Starred (*) bug numbers represent my best-effort estimation when I could not get confirmations from developers. Bug numbers without the star have been confirmed by the developers of the corresponding benchmark. "Unknown" shows the number of reports that could not be proved either feasible or infeasible. "FP Rate" gives the false positive rate, calculated as $1 - \#Bugs/(\#Reports - \#Unknowns)$. "Time" is the total runtime in seconds. Experiments were on a dual Opteron 2.8 GHz with 16 GB RAM.

Benchmark	LOC	CALYSTO v1.5				
		Reports	Bugs	Unkn.	FP Rate	Time [s]
Bftpd 1.8	3306	12	11	0	9%	3.14
Bftpd 1.9.2	3368	5	4	0	20%	2.86
HYPERSAT 1.7	6022	0	0	0	0%	14.57
Spin 4.3.0	20481	0	0	0	0%	6858.10
OpenSSH 4.6p1	45304	4	1	0	75%	8995.64
INN 2.4.3	71102	10	*6	1	34%	1312.33
NTP 4.2.4p2-RC5	74230	30	26	0	14%	558.16
NTP 4.2.5p66	74277	13	4	3	56%	493.39
BIND 9.4.1p1	184204	5	*2	3	0%	♯2436.88
OpenLDAP 2.4.4a	223595	20	15	2	27%	200.02
TOTAL	685408	99	69	9	23%	♯20875.09

Table 6.3: CALYSTO `NULL` Pointer Dereference Checking Results. Starred (*) bug numbers represent my best-effort estimation when I could not get confirmations from developers. Bug numbers without the star have been confirmed by the developers of the corresponding benchmark. ♯ indicates that on BIND, CALYSTO's runtime does not include instances on which CALYSTO failed to complete — it ran out of memory on 8 compilation units (taking an additional 6263.57 sec), and timed out in one day on one compilation unit. Experiments were on a dual Opteron 2.8 GHz with 16 GB RAM.

assumption that the loop test has failed is false, implying that all the code after the loop is unreachable (and therefore unchecked).

After I reported Bftpd, Spin, and NTP bugs, the developers immediately fixed all of them in the next release. Thanks to prompt responses from the Bftpd and NTP developers, I managed to check the new versions that fixed all the bugs found in the previous version. In the new versions, CALYSTO found new bugs, which have also been fixed in the meantime.

The most frequent causes of Saturn's false positives were: lack of interprocedural path-sensitivity, incomplete specifications of C library functions (for instance, passing a `NULL` pointer to the *free* function is allowed), and specific code patterns that seemed like inconsistencies to Saturn. Saturn's results on BIND are especially interesting. Bind's code is among the highest quality code of all open-source applications I have seen so far — almost every single pointer is checked before dereferencing and complex data structures are checked for consistency before usage. This ubiquitous checking apparently confused Saturn's inconsistency analysis because every single report was provably false. The major sources of CALYSTO's false positives were: missing specifications of external functions, broken cycles in the call graph, and C type unsafety.

The runtimes of both static checkers are comparable. Saturn is faster on some; CALYSTO, on others. Bind was particularly problematic for CALYSTO — it ran out of memory while analyzing 8 and timed out (1 day) on 1 compilation unit. The timeout was caused by a performance bug (failing to re-use certain cached results) in CALYSTO's interprocedural analysis.

I also analyzed how much time the two checkers spend in theorem-prover calls. The results are in Tables 6.4 and 6.5. CALYSTO spends almost 50% of its time in theorem prover calls, even with a small timeout (10 s) and a fast bit-vector arithmetic prover. The amount of time spent in the theorem prover calls is unsurprising, given how difficult the computed VCs are. Despite using a slower theorem prover [78], Saturn spends a much smaller fraction of its time in theorem prover calls. As mentioned earlier, Saturn performs expensive simplification using BDDs during static analysis, whereas I use a fast and incomplete expression simplifier and maximally-shared graphs. Also, because of Saturn's approximate interprocedural analysis, Saturn's VCs are likely much simpler. The different tool design shows up in the different time proportions, but both tools end up being usably fast.

Benchmark	Calysto v1.5 Total time [s]	Spear [s]	Percentage
Bftpd 1.8	3.14	1.25	39.8%
Bftpd 1.9.2	2.86	0.88	30.7%
HyperSAT 1.7	14.57	0.10	0.6%
Spin 4.3.0	6858.10	473.50	6.9%
OpenSSH 4.6p1	8995.64	8167.36	90.7%
INN 2.4.3	1312.33	14.77	1.1%
NTP 4.2.4p2-RC5	558.16	56.38	10.1%
NTP 4.2.5p66	493.39	58.03	11.7%
BIND 9.4.1p1	$^{\sharp}$2436.88	980.48	40.2%
OpenLDAP 2.4.4a	200.02	181.90	90.9%
TOTAL	$^{\sharp}$20875.09	9934.65	47.5%

Table 6.4: Calysto Total Runtime Split. The Spear column shows the time spent in the theorem prover, with the next column showing the percentage of the total runtime.

	Saturn v1.1		
Benchmark	Total time [s]	MiniSAT [s]	Percentage
Bftpd 1.8	129.51	17	13.1%
Bftpd 1.9.2	105.17	8	7.6%
HyperSAT 1.7	647.21	232	35.8%
Spin 4.3.0	2129.04	457	21.4%
OpenSSH 4.6p1	3707.81	988	26.6%
INN 2.4.3	9879.69	2104	21.2%
NTP 4.2.4p2-RC5	326.44	82	25.1%
NTP 4.2.5p66	319.33	80	25.0%
BIND 9.4.1p1	14984.72	1141	7.6%
OpenLDAP 2.4.4a	8098.48	949	11.7%
TOTAL	40226.40	6058	15.0%

Table 6.5: Saturn Total Runtime Split. Saturn's theorem prover is the SAT solver MiniSAT.

Chapter 7

Related Work

This chapter provides a survey of closely related literature, and puts the work presented in this thesis in the context of previous research.

7.1 Extended Static Checking

The work in this thesis has been largely motivated by the work on extended static checking. That line of work is represented by several extended static checkers: ESC/Modula-3 [49], ESC/Java [63], and Boogie [90]. The focus of that body of work is on the intraprocedural analysis. The interprocedural analysis is achieved through interface invariants (pre- and post-conditions) that must be provided by the user. This approach is particularly interesting for object oriented languages (Java,C#,...), in which quantification over object interfaces can be used to prove complex properties about the architecture and intended usage of the interfaces. The work presented in this thesis focuses on the explicit interprocedural analysis that requires no interface invariants. The only requirement is that the properties to be checked are specified. These properties can be specified once per language/library and reused for other applications written in the same language and/or based on the same library.

Another approach to software static analysis is to design a light-weight analysis and then use various heuristics to rank the probability that a bug found by the analysis is really a bug. Such an approach was used in MC [71, 58], which is a relatively imprecise, but very scalable C/C++ static checker. In contrast, I designed Calysto to be very precise. By exploiting the code structure at several levels of granularity, I showed that even a very precise checker can scale to several hundred thousand lines of code. My hope is that further research will make CALYSTO-like approaches competitive to MC in terms of scalability.

Saturn [135, 134] is the project most similar in spirit to CALYSTO because it relies on a SAT solver, requires no user-provided invariants, and performs interprocedural analysis through function summarization. However, there are many important differences:

1. CALYSTO exploits the code structure at multiple levels of granularity: Symbolic execution preserves the dataflow dependencies in the original code and also exploits the structure of the CFG for faster construction of symbolic expressions. Structural abstraction relies upon the high-level code structure (functions) to perform abstraction and refinement. Finally, SPEAR, with PARAMILS's help, exploits the fine-grained structural properties of the VCs to achieve superior performance. Saturn, on the other hand, makes no effort to maintain or exploit the structure.

2. The summaries in Saturn are represented as tuples of Finite State Machine (FSM) transition relations, input predicates, output predicates, and a set of modified objects. In that respect, the summaries are almost equivalent in functionality to the summaries computed by SLAM [13]. The FSM relation is specialized for a specific property being verified. In the case of kernel locks [135], the analysis allocates only a single bit per lock. The summaries computed by the interprocedural analysis in Chapter 4 are significantly more expressive (cf. Fig. 2.1 on page 14) and represent all the effects of the function.

3. Besides summarization, Saturn makes no other attempts to avoid or mitigate the potentially exponential cost of context-sensitivity. For simple properties, like lock-checking, the overhead of context-sensitivity could be low enough that the brute-force approach works well, but it is very unlikely that it will scale to more general properties.

4. Saturn's intraprocedural analysis differs from the one presented in Chapter

3 in that it relies on BDDs to simplify the guards, rather than on structure sharing. I decided to omit the BDD-based simplification step because SAT solvers are extremely efficient at detection of conflicts and propagation of resulting implications. Calysto performs extensive simplification of the VCs during the construction of the summary graph, but all those simplifications are very cheap to perform. Furthermore, Saturn makes no attempt to reuse the structure present in the CFG for more efficient generation of VCs.

Astrée [22, 41] is a precise and fairly scalable static checker intended for safety-critical software. Such software is typically implemented as clean, heavily restricted code (see discussion in Section 1.2.2 on page 5), and that significantly simplifies the requirements on the static checker. Calysto is designed to be a general-purpose checker, and as such has to deal with type-unsafety, unstructured code, and dynamically allocated memory. The focus of the Astrée project is also different from the work presented in this thesis. Astrée is based on the framework of abstract interpretation [39], while Calysto uses almost no abstraction, and is completely based on structure exploitation.

The choice of SSA/GSA form as the starting point for computation of VCs sets Calysto apart from the other mentioned tools. For example, Boogie renames variables on the AST, creates fresh definitions in predecessors of the basic block that contains the use, and applies the weakest precondition predicate transformer to compute the VCs. Being based on SSA/GSA forms, Calysto enjoys the known benefits of single assignment forms (e.g., language independence, single point of definition of local variables, easy manipulation and transformation of the intermediate form, and so on.).

7.2 Computation of Verification Conditions

The related work on the computation of VCs can be classified into two subfields: symbolic execution and improving the quality of the VCs.

Probably the closest approach to the intraprocedural analysis presented in Chapter 3, is that of Kölbl and Pixley [83]. They considered symbolic execution in the context of equivalence checking of higher-level hardware descriptions in C++. They assume that hardware designers are aware that the code is going to be compared to Register Transfer Level (RTL) code, so most of the code is very low-level, and pointers are used sparingly[30]. The authors use a heuristic scheduling algorithm for symbolic execution and merging of different paths. My approach offers an alternative with an optimal merging strategy — all alternative definitions that reach a use are merged by the proposed restriction operator for gating path expressions. In addition, write merges are scheduled at a minimal set of join points, which is efficiently computed by the means of the iterated dominance frontier. Kölbl and Pixley keep the definitions of abstract memory locations in a list together with corresponding guards, minimized with a variant of BDDs. This is exactly the same approach as the one used in Saturn. Their internal representation is similar to the maximally-shared graphs in Calysto. Their results, although obtained on much smaller benchmarks than used in this thesis, are similar to mine. Namely, they report that, in practice, the size of computed intraprocedural summaries is linear with the number of instructions. The authors do not address interprocedural analysis in the paper.

Leino [87] presented an approach to the generation of efficient VCs based on the weakest precondition predicate transformer. The weakest precondition can handle only structured code. That work was later extended to unstructured programs [18] through explicit encoding of the CFG. Their introduction of fresh variables to encode the CFG transitions seems similar to the introduction of fresh variables with the purpose of structuring the graph [59]. The analysis presented in this thesis handles unstructured code directly without introducing redundant variables.

[30] While I was working on my masters thesis, I wrote a number of such models in SystemC. In my experience, those models abstract the bit-level details, and do not significantly differ from the RTL descriptions. Most often, those models are finite-state, meaning that there is no dynamic memory allocation.

Another line of relevant work is focused on the generation of high-quality VCs. Flanagan and Saxe proposed a passifying and renaming transformation [64] for the generation of VCs. Their transformation produces quadratic-sized VCs in the worst case, and the authors claim the size is linear in practice. Later, Leino [87] shed more light on the passifying component of the transformation. The renaming component is similar to Tseitin's transform [128], which introduces fresh variables to avoid exponential blowup of a circuit-to-CNF translation. The SSA form is based on the same principle. Introduction of fresh variables, both in Tseitin's transform and the VC generation, effectively flattens the logical formula. This flattening hides the structure of the problem. The approach proposed in this thesis also uses Tseitin's transform for encoding maximally-shared graphs to CNF, however structural abstraction and refinement are done before the flattening, while the VCs are still in the form of maximally-shared graphs. As described in Section 5.1.1, Spear's expression simplifier also works on maximally-shared graphs. In other words, the approach proposed in this thesis delays the flattening so as to exploit as much structure available in the original problem (and reflected in maximally-shared graphs) as possible.

Another difference is that Leino's approach encodes the variables required for trace reconstruction directly into the VC [89], while my approach keeps the code locations and variable names in summary graph nodes. If a satisfying assignment is found, it can be easily mapped back to the source code. This avoids the introduction of additional variables.

Lahiri, in his thesis [84], explored small model properties and the application of positive equality for more efficient solving of the VCs coming from processor verification. Both approaches can be combined with the approach presented in this thesis.

7.3 Interprocedural Analysis

Reps et al. [109] presented a polynomial-time algorithm for interprocedural, finite, distributive, subset problems (IFDS). Some of the analyses that fit in that class are: constant propagation, pointer analysis, reaching definitions, possibly-uninitialized variables, etc. Unfortunately, the summaries computed in Chapter 3 are neither distributive, nor subset problems. For instance, aliasing of pointers passed to a function makes the summary non-distributive. As the focus of this thesis is on getting as precise and as scalable analysis as possible, abstracting the summaries to the level required by the IFDS framework seemed inappropriate.

SLAM [13] is based on predicate abstraction, and the summaries are transition relations with respect to the set of chosen predicates. In SLAM, summaries are represented as BDDs. The predicate abstraction suffices for proving control-flow intensive properties, and it seems to be especially suitable for device drivers. Calysto doesn't abstract much (currently only loops, recursive data structures, and exceptional control-flow). So, encoding the transition relation as BDDs would result in serious blowup (especially on multipliers).

Structural abstraction abstracts functions in the cheapest way possible — by avoiding computation, or more precisely, by avoiding expansion of summary operators. Other abstractions, like predicate abstraction [13], typically perform relatively expensive recomputation of the abstractions in each abstraction-checking-refinement cycle.

This lazy approach to interpretation of function summaries resembles the intuition behind lazy proof explication [62], a technique used to bridge between different theories in a theorem prover. The shared intuition is to abstract away expensive reasoning — expanding a function summary or solving a sub-theory query — as unconstrained variables, and then constrain them lazily, only as needed to refute solutions to the abstracted problem. The specifics of what to abstract and how to refine, of course, are different, since I am solving a different problem.

Taghdiri's specification inference [126] computes a coarse abstraction of each function, abstracting the side-effects of function calls in a similar way as structural abstraction. If a counterexample is found, her refinement analyzes each function that participates in the counterexample, attempting to prove that there exists an intraprocedural path consistent with the trace. If there is no such path, the proof is mined for additional predicates which are then used to refine the abstraction of the function. Compared to structural abstraction, specification inference is is significantly more expensive because it cannot be made completely incremental — once the trace is found infeasible, all the reasoning has to be discarded, and started anew in the next iteration. Structural abstraction and refinement were carefully designed to be completely incremental and avoid discarding learned facts after each abstraction-checking-refinement iteration.

Shortly after publication of my work on structural abstraction [8], Anand et al. [5] published work that is similar to structural abstraction, but in the context of software testing. The main idea presented in both works is to use summarization to avoid the worst-case cost of interprocedural analysis and let the decision procedure inline or expand only what it needs. However, the approaches to the problem are different: Anand et al. approach the problem from the software testing perspective, and I approach the problem from the extended static checking perspective. These two approaches, when taken together, strongly suggest that letting the decision procedure handle interprocedural analysis might be the right way to go, and also together provide more insight into this rather important problem than either approach alone. The rest of this section provides a detailed comparison.

One of the key assumptions Anand et al. make is that the code is deterministic — unlike structural abstraction, which initially abstracts side-effects with unconstrained variables, their approach relies upon uninterpreted functions. Although such an approach gives more opportunities for elimination of common subexpressions that involve uninterpreted functions[31], it is not neces-

[31] Calysto can also merge two summary operators corresponding to the same side-effect of the same function, if and only if the summarized function is deterministic and the operands of

sarily sound: two different calls to the same non-deterministic function with the same parameters can produce two different results.[32]

Three important notions (local path constraint, definition predicate, and calling-context predicate) that Anand et al. introduce also have their counterparts in the approach proposed in this thesis. The local path constraint is similar to the intraprocedural control-flow context that the symbolic execution presented in Chapter 3 computes, the only difference being that my structure-preserving symbolic execution computes the control-flow context for *all* the paths in a function, while the local path constraint presents a conjunction of predicates that hold on a *single* path in the symbolic execution tree. The definition predicate asserts equality between the summary operator and the corresponding abstract memory location in the calling context; instead of constructing such an equality, I rather expand the summary operator in a lazy manner, inlining the corresponding symbolic expression. Finally, their calling-context predicate is similar to the control-flow context, with the difference that Calysto computes it in the summarized form for all the paths.

Anand et al. incompletely characterize my earlier work on structural abstraction [8]. Their paper contrasts the demand-driven compositional symbolic execution to structural abstraction, claiming that inlining in structural abstraction leads to combinatorial explosion. The truth is that both their approach, as well as structural abstraction, have worst-case exponential cost — only the dynamics of the approaches are different:

- In structural abstraction, the decision procedure (in my work Spear) controls which summary operator needs to be expanded through the process of structural refinement (Section 4.2.3 on page 81). The decision procedure can and does learn about the newly expanded summary operators,

both summary operators are structurally equivalent, as described in Section 4.2.4 on page 83.

[32] According to my experience, non-deterministic functions are actually quite frequent in real code. The most frequent sources of non-determinism are: calls to external functions, dependencies on the actual pointer values, dependencies on time/network/files, and calls to `random` function.

attempting to avoid unnecessary expansions. Even if a summary operator (corresponding to a single side-effect) is expanded, the nested summary operators are not. Those nested operators might or might not be expanded in the subsequent iterations of the abstraction-checking-refinement loop.

- In demand-driven compositional symbolic execution, the exponential behavior is hidden in quantifiers — the modern SMT decision procedures, like Z3 [47], handle quantifiers by an incomplete heuristic quantifier instantiation (e.g., [46]). Depending on the underlying theory and implemented heuristics, such instantiation can incur exponential blowup, or even worse, fail to terminate[33].

7.4 Decision Procedures for Bit-Vector Arithmetic

The Nelson-Oppen [100] framework is probably the most frequently used approach for proving the validity of VCs. The framework is a method for combining decision procedures. The theories have to meet certain requirements (see [94] for an overview), in order to be combined with other theories. Fortunately, most theories used in practice, like Presburger arithmetic, the theory of uninterpreted functions, and the theory of arrays, meet the requirements. De Moura and Bjørner [47] combined the Nelson-Oppen framework with the theory of bit-vectors in the Z3 automated theorem prover. Their implementation uses one SAT solver for case-splitting and another one as the bit-vector arithmetic theory solver. The theory solver does bit-blasting, in a similar way to Spear. The difference is that their theory solver only propagates the bit-vector theory facts, and does not infer conflicts, which are entirely handled by the core case-splitting SAT solver. The Nelson-Oppen framework requires such a design. Spear, on the other hand, handles both fact propagation and case-splitting

[33] Z3 uses a heuristical fail-safe bound on the number of instantiations, so it always terminates. However, other SMT provers might handle instantiation in a less safe way.

with a single, highly-tuned, SAT solver. In general, the Nelson-Oppen framework framework has certain advantages over the Spear-like approaches: (1) It is easier to combine the theory of bit-vectors with other theories, like the theory of uninterpreted functions. (2) It is easier to incorporate complex term-rewriting techniques, which are very important for certain classes of problems, like equivalence of implementations of cryptographic algorithms. In spite of the greater flexibility of the Nelson-Oppen framework, it seems that Spear-like approach is faster on instances with complex predicate structure[34].

Barrett, Dill, and Levitt [19] proposed a decision procedure for subset of bit-vector arithmetic operations based on Shostak's method[35] for combining theories [117]. Their decision procedure uses term rewriting to avoid bit-blasting as much as possible. Spear is based on similar principles, but relies upon a SAT solver, rather than a framework for combining theories. They implemented their approach in the SVC automated theorem prover. SVC has not been actively developed for many years now, so a direct comparison to the modern bit-vector arithmetic decision procedures, like Z3 and Spear, would be meaningless.

Bryant et al. [25] recently suggested an abstraction for bit-vector arithmetic based on an alternation of under- and over-approximations. At this point, it is unclear how effective that abstraction is, because their decision procedure is not publicly available and did not participate at the SMT competition. Z3 solves the hardest examples presented in [25] in 10–14 seconds, while Bryant et al. report that their technique requires 1000+ seconds. Of course, this is not an apples-to-apples comparison, because the experiments were not done on the same machine. On the same examples, Spear requires 100–1000+ seconds, depending on the parameter configuration used. The alternation between under- and over-approximations, if proven effective, could be easily implemented in Spear.

[34] Spear won the bit-vector division of the SMT 2007 competition, ahead of Z3.

[35] Shostak's method for combining theories has been a source of controversy for a long time. A number of authors pointed out the errors in Shostak's method (e.g., [111, 116, 37]). According to Detlefs et al. [51, page 386], the consensus in the community is that Shostak's method is a refinement of the Nelson-Oppen method.

Ganesh and Dill [65] proposed an abstraction-refinement technique for reasoning about the non-extensional theory of arrays [124] in combination with a bit-vector decision procedure based on bit-blasting. They implemented their approach in the STP decision procedure, which relies upon an expression simplifier that is very similar to the one in SVC. Their technique for handling arrays could be incorporated in Spear, which would help Calysto to handle arrays more precisely. STP seems to be particularly optimized for instances that are a long sequence of conjuncts, because it performs Gaussian elimination as a simplification step. Testing tools, which analyze a single path at time, tend to generate such a long sequence of conjuncts, but extended static checking tools, like Calysto, generate instances with very complex propositional structure. Gaussian elimination is ineffective for such complex instances. If needed, Gaussian elimination could be easily added as a preprocessing step in Spear. Unlike Spear, STP does not support all standard bit-vector arithmetic operators (for instance, variable shifts).

Babić and Musuvathi [11] proposed two techniques for handling modular arithmetic constraints: The first one is for linear constraints and is based on translation to integer linear programming. My later experiments showed that the leading integer linear programming solvers, like CPLEX[36], perform extremely poorly on such problems. More precisely, CPLEX failed to return after 2 weeks of continuous running on a problem that Spear solves in several seconds. It is still an open question how well an SMT solver would perform on such a translation. The second proposed technique is for non-linear constraints and is based on Newton's p-adic lifting. Newton's p-adic lifting is somewhat complex because it requires computation of Jacobians, a square matrix of partial derivatives. A simpler way to emulate p-adic lifting is simply to force a SAT solver to case-split on the least significant bits first and proceed towards the most significant bits. According to my experiments, this heuristic works very well on hand-crafted examples with many non-linear operations and very simple propositional structure. Unfortunately, such a heuristic p-adic lifting em-

[36] **http://www.ilog.com/**

ulation does not perform well on the real-world instances, like those generated by CALYSTO, that have very complex propositional structure. Getting p-adic lifting (either the Jacobian-based version or the heuristic emulation) to work well on the real-world instances is still an open problem of an immense practical importance.

7.5 Automatic Tuning

There are almost no publications on automated parameter optimization for decision procedures for formal verification. Seshia [114] explored using support vector machine classification to choose between two encodings of difference logic into Boolean SAT. The learned classifier was able to choose the better encoding in most instances he tested, resulting in a hybrid encoding that mostly dominated the two pure encodings. The only other work I am aware of is that of Andrei Voronkov — he used a less systematic approach for tuning the strategies of his Vampire first-order theorem prover. However, he has not published that work.

There is, however, a fair amount of previous work on optimizing SAT solvers for particular applications. For example, Shtrichman [118] considered the influence of variable and phase decision heuristics (especially static ordering), restriction of the set of variables for case splitting, and symmetric replication of conflict clauses on solving BMC problems. He evaluated seven strategies on the Grasp SAT solver, and found that static ordering does perform fairly well, although no parameter combination was a clear winner. Later, Shacham and Zarpas [115] showed that Shtrichman's conclusions do not apply to zChaff's less greedy VSIDS heuristic on their set of benchmarks, claiming that Shtrichman's conclusions were either benchmark- or engine-dependent. Shacham and Zarpas evaluated four different decision strategies on IBM BMC instances, and found that static ordering performs worse than VSIDS-based strategies. Lu at al. [92] exploited signal correlations to design a number of ATPG-specific techniques for SAT solving. Their technique showed roughly an order of magnitude improvement on a small set of ATPG benchmarks.

The application of machine learning techniques for improving the scalability of decision procedures has been more thoroughly researched. For instance, Grumberg et al. [69] used machine learning to learn good BDD variable orderings from a number of sample orderings. The underlying idea — to use AI-techniques to improve performance — is the same, but the underlying technology used and the goals are very different. The underlying technology that Grumberg et al. use is machine learning, while the automatic tuner used for Spear optimization is a local search engine that attempts to quickly discard bad parameter configurations. The purpose of automatic tuning of Spear was to recognize the common (structural) properties of a class of instances and find suitable search parameters and heuristics that can effectively take advantage of those properties. In contrast, Grumberg et al. dynamically try to come up with a good BDDs-ordering for each individual problem. Greumberg et al.'s work was later extended by [27], but the underlying tuning technology and goals remained the same: According to my understanding, Carbin's work [27] chooses sample BDD-orderings in each iteration by reusing the information learned from the previous iterations.

In the context of FOL theorem proving, machine learning is frequently used to detect a small set of axioms that are likely to be sufficient for proving that a given conjecture is a theorem (e.g., [125, 48]). SMT decision procedures and SAT solvers actually use similar heuristic techniques for selecting the variables for case-splitting (e.g., [97]) and for quantifier instantiation (e.g., [46]).

The automated parameter optimization tool used in my study has been recently introduced by Hutter et al. [79]; however, that work was more focused on theoretical properties of the algorithm and did not consider an application to a state-of-the-art solver for real-world problems. That work and the study presented here complement each other and also address two different communities. Very broadly, automated parameter optimization can be seen as as a stochastic optimization problem that can be solved using a range of generic and specific methods [121, 3, 21]. However, these are either limited to algorithms with continuous parameters or algorithms with a small number of discrete parameters.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Possibly the most important contribution of this thesis is that it draws the attention of the software analysis research community to, in my opinion, a somewhat neglected research direction — exploitation of structure that is so pervasive in all human-made systems. Humans need structure to solve complex problems. Recognizing that structure and exploiting it is essential if precise software analysis tools are to scale to multi-million line programs.

The main hypothesis of this thesis — that the structure of programs can be exploited for achieving a highly scalable and precise analysis — has been proven. First, I showed how structure-preserving symbolic execution can compute summaries and VCs that reflect the dataflow dependencies in the original code. The presented symbolic execution also exploits the graph properties of CFGs for fast identification of live definitions. Second, I proposed structural abstraction that exploits the natural abstraction boundaries in programs (functions) for abstraction, and structural refinement that uses the dataflow dependencies (preserved by my symbolic execution) for fast refinement. Third, I showed that at a finer level of granularity, automatic tuners can effectively exploit the properties of a specific class of instances, and adjust the search parameters accordingly, even when the combinatorial search space spanned by the parameters is enormous ($> 10^{18}$). Finally, I provided a description of the architecture of my experimental extended static checker, Calysto, that implements the techniques presented in this thesis. Calysto's results on a number of real-world benchmarks show that the proposed combination of techniques is both very scalable and sufficiently precise. Furthermore, the results encourage further research.

8.2 Future Work

While this thesis has proposed novel and effective solutions for interprocedural software analysis and bit-vector decision procedures, there are still many challenges ahead. In this section, I discuss areas that, in my opinion, need more research.

Although Calysto is very precise in terms of interprocedural path-sensitivity and handling of machine operations (bit-precise), further improvements are needed in several areas:

- Currently, Calysto knows nothing about concurrency. Recent work on iterative context-bounding by Musuvathi and Qadeer [99] and later work by Bouajjani et al. [23] could be used in combination with structural abstraction: partial order reduction [67] performed directly on the source could generate a sequence of different interleavings of the source (or the intermediate form), which could then in turn be passed to a tool based on structural abstraction, like Calysto. However, it is very unlikely that this would be efficient. A more efficient approach would require a tighter integration of structural abstraction and analysis of different interleavings.

- Calysto, like ESC/Java, handles loops by unrolling them and assuming that the loop test has failed (see Section 3.1.3). The major drawback of that approach is decreased code coverage. One promising, but not sufficiently researched direction is a combination of abstract interpretation techniques and extended static checking, as proposed by Leino and Logozzo [88]. It is unclear how well such a combination would perform with structural abstraction on non-Java code. Another possibility is to delegate handling of loops to the decision procedure, by incorporating widening and narrowing operators directly into the decision procedure.

- Arrays could be easily handled more precisely by modifying the proposed structure-preserving symbolic execution to use array read and write operators instead of computing the symbolic definitions of abstract memory locations. However, it is very possible that the combination of bit-vector arithmetic and the theory of arrays would make VCs much harder and negatively impact the scalability of the entire system. The tradeoffs would need to be evaluated experimentally.

- Structural abstraction could be significantly improved as well. First, by better and more precise refinement heuristics, and second, by checking each VC before it is lifted to the calling context (see 4.2.4).

Decision procedures, especially for bit-vector arithmetic, are a dynamic research area. SPEAR shows that SAT-solver-based decision procedures for bit-vector arithmetic can be a good choice for problems that have complex propositional structure. However, further improvements are needed. In my opinion, the most important research directions are the following:

- One of the downsides of SPEAR's architecture is that it is hard to add support for new theories. The most important theory that is currently missing is the theory of non-extensional arrays [124]. Other theories that would be immensely useful for other domains are mixed integer linear programming (e.g., [119]), and the theory of floating-point. Nelson-Oppen framework [100] offers more flexibility than SPEAR-architecture. However, more research is needed on incorporating the theory of bit-vectors into that framework.

- Adding support for quantifiers to SPEAR would also be very useful, especially for computing bit-precise loop invariants. One interesting direction would be to use a Quantified Boolean Formula (QBF) solver instead of a SAT solver for SPEAR's core.

- Calysto frequently generates a very long sequence of VCs. Often, adjacent VCs in that sequence share some common subexpressions. Solving these VCs individually wastes computation because the decision procedure has to re-learn facts about those shared subexpressions over and over again. Solving them all together exceeds the resources. Since VCs are constructed on-the-fly, through structural abstraction and refinement, it is almost impossible to guess which expressions are going to be shared ahead of time. I describe this problem in more detail in my previous work [7], and offer an approximative solution. However, much more work is needed to eliminate this source of redundancy.

- Automatic tuning techniques were exceptionally effective at optimizing Spear. As Spear is being tuned for more and more different classes of problems, automatic classification becomes the next logical step. The SATzilla [136] classifier is very promising, but further research is needed.

Bibliography

[1] The economic impact of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[2] Saturn-discuss mailing list archives, August 2007. https://mailman.stanford.edu/pipermail/saturn-discuss/.

[3] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, Jan–Feb 2006.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1986.

[5] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*, pages 367–381. Springer, April 2008.

[6] Domagoj Babić, Jesse Bingham, and Alan J. Hu. B-Cubing: New Possibilities for Efficient SAT-Solving. *IEEE Transactions on Computers*, 55(11):1315–1324, 2006.

[7] Domagoj Babić and Alan J. Hu. Exploiting Shared Structure in Software Verification Conditions. In Karen Yorav, editor, *Proceedings of Haifa Ver-*

ification Conference (HVC'07), volume 4899 of *Lecture Notes in Computer Science (LNCS)*, pages 169–184. Springer, 2007.

[8] Domagoj Babić and Alan J. Hu. Structural Abstraction of Software Verification Conditions. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 371–383. Springer, 2007.

[9] Domagoj Babić and Alan J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the* 30^{th} *international conference on Software engineering (ICSE'08)*, pages 211–220, New York, NY, USA, 2008. ACM.

[10] Domagoj Babić and Frank Hutter. Spear Theorem Prover. In *Proceedings of the 2008 SAT Race*, 2008.

[11] Domagoj Babić and Madanlal Musuvathi. Modular Arithmetic Decision Procedure. Technical Report MSR-TR-2005-114, Microsoft Research Redmond, 2005.

[12] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Proceedings of the* 3^{rd} *International Symposium on Formal Methods for Components and Objects (FMCO'04)*, volume 3657 of *Lecture Notes in Computer Science (LNCS)*, pages 1–22. Springer, November 2005.

[13] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming language design and implementation (PLDI'01)*, volume 36 of *SIGPLAN Notices*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[14] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proceedings of the* 7^{th} *International Workshop on Model Checking of Software (SPIN'00)*, volume 1885 of *Lecture*

Notes in Computer Science (LNCS), pages 113–130, London, UK, 2000. Springer-Verlag.

[15] Thomas Ball and Sriram K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[16] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the* 29^{th} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[17] Robert A. Ballance, Arthur B. MacCabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI'90)*, volume 25 of *SIGPLAN Notices*, pages 257–271, New York, NY, USA, 1990. ACM Press.

[18] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *The* 6^{th} *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE'05)*, pages 82–87, New York, NY, USA, 2005. ACM Press.

[19] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the* 35^{th} *Design Automation Conference (DAC'98)*, pages 522–527, New York, NY, USA, June 1998. ACM Press.

[20] A. Bhalla, I. Lynce, J.T. de Sousa, and J. Marques-Silva. Heuristic backtracking algorithms for SAT. In *Proceedings of the* 4^{th} *International Workshop on Microprocessor Test and Verification (MTV'03)*, pages 69–74, Austin, Texas, USA, May 2003.

[21] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pages 11–18, San Francisco, California, USA, 2002. Morgan Kaufmann Publishers Inc.

[22] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, volume 38 of *SIGPLAN Notices*, pages 196–207, New York, NY, USA, 2003. ACM Press.

[23] Ahmed Bouajjani, Sverine Fratani, and Shaz Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 207–220. Springer, 2007.

[24] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[25] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In Orna Grumberg and Michael Huth, editors, *Proceedings of the* 13^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science (LNCS)*, pages 358–372. Springer, April 2007.

[26] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software — Practice & Experience*, 30(7):775–802, 2000.

[27] Michael Carbin. Learning Effective BDD Variable Orders for BDD-Based Program Analysis, May 2006. Computer Science Honors Thesis.

[28] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI'90)*, volume 25 of *SIGPLAN Notices*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[29] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the* 6^{th} *International Conference on Compiler Construction (CC'96)*, volume 1060 of *Lecture Notes in Computer Science (LNCS)*, pages 253–267, London, UK, 1996. Springer-Verlag.

[30] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[31] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[32] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the* 12^{th} *International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 154–169, London, UK, 2000. Springer-Verlag.

[33] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, Massachusetts, USA, 1999.

[34] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the* 40^{th} *conference on Design automation (DAC'03)*, pages 368–371, New York, NY, USA, 2003. ACM Press.

[35] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions Software Engineering*, 2(3):215–222, 1976.

[36] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the* 8^{th} *European software engineering conference held jointly with* 9^{th} *ACM SIGSOFT international symposium on Foundations of software engineering (ESEC'01/FSE'01)*, volume 26 of *SIGSOFT Software Engineering Notes*, pages 142–151, New York, NY, USA, 2001. ACM Press.

[37] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354(2):187–210, 2006.

[38] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the* 17^{th} *International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science (LNCS)*, pages 449–461, Berlin, Germany, 2005. Springer.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the* 4^{th} *ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'77)*, pages 238–252, New York, NY, USA, 1977. ACM.

[40] Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In *Proceedings of the* 11^{th} *International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science (LNCS)*, pages 159–178, London, UK, 2002. Springer-Verlag.

[41] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyser. In *Proceedings of the* 14^{th} *European Symposium on Programming (ESOP'05)*,

volume 3444 of *Lecture Notes in Computer Science (LNCS)*, pages 21–30. Springer-Verlag, April 2005.

[42] David W. Currie, Alan J. Hu, and Sreeranga Rajan. Automatic formal verification of DSP software. In *Proceedings of the* 37^{th} *conference on Design automation (DAC'00)*, pages 130–135, New York, NY, USA, 2000. ACM.

[43] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[44] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[45] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[46] Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In Frank Pfenning, editor, *Proceedings of the* 21^{st} *International Conference on Automated Deduction (CADE'07)*, volume 4603 of *Lecture Notes in Computer Science (LNCS)*, pages 183–198. Springer, 2007.

[47] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the* 14^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*, pages 337–340. Springer, 2008.

[48] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999.

[49] David L. Detlefs. An overview of the extended static checking system. In *Proceedings of the* 1^{st} *Workshop on Formal Methods in Software Practice (FMSP'96)*, pages 1–9, San Diego, California, USA, January 1996. ACM.

[50] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-RR-159, Compaq Systems Research Center, Palo Alto, California, USA, December 1998. Now available from HP Labs.

[51] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.

[52] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[53] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, volume 42 of *SIGPLAN Notices*, pages 435–445, New York, NY, USA, 2007. ACM Press.

[54] Laura K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):643–669, 1990.

[55] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In Fahiem Bacchus and Toby Walsh, editors, *Proceedings of the* 8^{th} *International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science (LNCS)*, pages 61–75. Springer, June 2005.

[56] Niklas Eén and Niklas Sörensson. An extensible SAT solver. In *Proceedings of the* 6^{th} *International Conference on theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Com-*

puter Science (LNCS), pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer.

[57] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the* 4^{th} *conference on Symposium on Operating System Design & Implementation (OSDI'00)*, pages 1–1, Berkeley, California, USA, October 2000. USENIX Association.

[58] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the* 18^{th} *ACM symposium on Operating systems principles (SOSP'01)*, volume 35 of *SIGOPS Operating Systems Review*, pages 57–72, New York, NY, USA, 2001. ACM.

[59] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In Henri E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages (ICCL'94)*, pages 229–240. IEEE Computer Society, May 1994.

[60] Karem A. Sakallah Fadi A. Aloul, Igor L. Markov. Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability. In *Proceedings of the* 40^{th} *Design Automation Conference (DAC'03)*, pages 836–839. ACM Press, June 2003.

[61] Xiushan Feng and Alan J. Hu. Automatic formal verification for scheduled VLIW code. In *Proceedings of the ACM SIGPLAN Joint Conference: Languages, Compilers, and Tools for Embedded Systems, and Software and Compilers for Embedded Systems (LCTES'02/SCOPES'02)*, volume 37 of *SIGPLAN Notices*, pages 85–92, New York, NY, USA, 2002. ACM Press.

[62] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Warren A. Hunt Jr. and

Fabio Somenzi, editors, *Proceedings of the* 15^{th} *International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science (LNCS)*, pages 355–367. Springer, 2003.

[63] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, volume 37 of *SIGPLAN Notices*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[64] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the* 28^{th} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'01)*, volume 36 of *SIGPLAN Notices*, pages 193–205, New York, NY, USA, 2001. ACM Press.

[65] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, pages 519–531. Springer, 2007.

[66] Ian P. Gent, Holger H. Hoos, Patrick Prosser, and Toby Walsh. Morphing: combining structure and randomness. In *Proceedings of the* 16^{th} *National Conference on Artificial Intelligence and the* 11^{th} *Conference on Innovative Applications of Artificial Intelligence (AAAI'99/IAAI'99)*, pages 654–660, Menlo Park, California, USA, 1999. American Association for Artificial Intelligence.

[67] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 1996.

[68] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. In *Proceedings of the* 14^{th} *National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226. American Association for Artificial Intelligence, 1997.

[69] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. Learning to order bdd variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116, 2003.

[70] H. H. Guild. Some cellular logic arrays for non-restoring binary division. *The Radio and Elect. Engr.*, 39:345–348, 1970.

[71] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, volume 37 of *SIGPLAN Notices*, pages 69–82, New York, NY, USA, 2002. ACM Press.

[72] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[73] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2002.

[74] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with Blast. In *Proceedings of the* 10^{th} *International Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *Lecture Notes in Computer Science (LNCS)*, pages 235–239. Springer-Verlag, January 2003.

[75] C. Antony R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of ACM*, 50(1):63–69, 2003.

[76] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[77] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2004.

[78] Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu. Boosting Verification by Automatic Tuning of Decision Procedures. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, Washington, DC, USA, 2007. IEEE Computer Society.

[79] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the* 22^{nd} *Conference on Artifical Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press, July 2007.

[80] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Proceedings of the* 18^{th} *International Workshop on Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science (LNCS)*, pages 160–174. Springer, 2004.

[81] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the* 22^{nd} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'95)*, pages 393–407, New York, NY, USA, 1995. ACM.

[82] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[83] Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[84] Shuvendu K. Lahiri. *Unbounded System Verification Using Decision Procedure and Predicate Abstraction.* PhD thesis, Carnegie Mellon University, 2004.

[85] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[86] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (GCO'04)*, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.

[87] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[88] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the* 3^{rd} *Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *Lecture Notes in Computer Science (LNCS)*, pages 119–134, November 2005.

[89] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, 2005.

[90] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Proceedings of the* 15^{th} *European Symposium on Programming (ESOP'06), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'06)*, volume 3924 of *Lecture Notes in Computer Science (LNCS)*, pages 115–130. Springer-Verlag, March 2006.

[91] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[92] Feng Lu, Li-C. Wang, K.-T. (Tim) Cheng, John Moondanos, and Ziyad Hanna. A signal correlation guided ATPG solver and its applications for solving difficult industrial cases. In *Proceedings of the* 40^{th} *Design Automation Conference (DAC'03)*, pages 436–441, New York, NY, USA, 2003. ACM Press.

[93] Andrea De Lucia. Program slicing: Methods and applications. In *Proceedings of the* 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Los Alamitos, California, USA, November 2001. IEEE Computer Society Press.

[94] Zohar Manna and Calogero G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science (LNCS)*, pages 381–422. Springer, 2003.

[95] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

[96] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[97] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the* 38^{th} *Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.

[98] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1997.

[99] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Im-*

plementation (PLDI'07), volume 42 of *SIGPLAN Notices*, pages 446–455, New York, NY, USA, 2007. ACM Press.

[100] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[101] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 1999.

[102] Carl D. Offner. Notes on graph algorithms used in optimizing compilers, 1995. Available from http://www.cs.umb.edu/~offner/files/flow_graph.pdf.

[103] J. Patarin and L. Goubin. Trapdoor One-Way Permutations and Multivariate Polynomials. In *Proceedings of* 1^{st} *International Information and Communications Security Conference*, volume 1334 of *Lecture Notes in Computer Science (LNCS)*, pages 356–368, 1997.

[104] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the* 16^{th} *Eastern Joint Computer Conference*, pages 133–138, New York, 1959. Spartan Books.

[105] Helena Ramalhinho-Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353, Norwell, Massachusetts, USA, 2002. Kluwer Academic Publishers.

[106] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[107] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):455–490, 2002.

[108] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[109] Thomas W. Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the* 22^{nd} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, New York, NY, USA, 1995. ACM Press.

[110] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the* 15^{th} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'88)*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[111] Harald Rueβ and Natarajan Shankar. Deconstructing shostak. In *Proceedings of the* 16^{th} *Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, page 19, Washington, DC, USA, 2001. IEEE Computer Society.

[112] David A. Schmidt and Bernhard Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Proceedings of the* 5^{th} *International Symposium on Static Analysis (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 351–380, London, UK, 1998. Springer-Verlag.

[113] Philippe Schnoebelen. The complexity of temporal logic model checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Selected Papers from the* 4^{th} *Workshop on Advances in Modal Logics (AiML'02)*, pages 393–436, Toulouse, France, 2003. King's College Publication. Invited paper.

[114] Sanjit A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 2005.

[115] Ohad Shacham and Emmanuel Zarpas. Tuning the VSIDS Decision Heuristic for Bounded Model Checking. In *Proceedings of th* 4^{th} *International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV'03)*, pages 75–79. IEEE Computer Society, May 2003.

[116] Natarajan Shankar and Harald Rueβ. Combining Shostak Theories. In *Proceedings of the* 13^{th} *International Conference on Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18, London, UK, 2002. Springer-Verlag.

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

[118] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the* 12^{th} *International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 480–494. Springer, 2000.

[119] Gerard Sierksma. *Linear and Integer Programming: Theory and Practice*. Marcel Dekker, Inc., New York, NY, USA, 1996.

[120] João P. Marques Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the* 9^{th} *Portuguese Conference on Artificial Intelligence (EPIA'99)*, volume 1695 of *Lecture Notes in Computer Science (LNCS)*, pages 62–74. Springer, 1999.

[121] James C. Spall. *Introduction to Stochastic Search and Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[122] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the* 22^{nd} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'95)*, pages 62–73, New York, NY, USA, 1995. ACM.

[123] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the* 23^{rd} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[124] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the* 16^{th} *Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, page 29, Washington, DC, USA, 2001. IEEE Computer Society.

[125] Geoff Sutcliffe. Machine learning for automated reasoning. MLAR system description.

[126] Mana Taghdiri. Inferring specifications to detect errors in code. In *Proceedings of the* 19^{th} *IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 144–153, Washington, DC, USA, 2004. IEEE Computer Society.

[127] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[128] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[129] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI'95)*, volume 30 of *SIGPLAN Notices*, pages 47–55, New York, NY, USA, 1995. ACM.

[130] J. P. Warners and H. van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.

[131] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[132] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI'04)*, volume 39 of *SIGPLAN Notices*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[133] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proceedings of the* 9^{th} *International Conference on Compiler Construction (CC'00), held as part of the European Joint Conferences on the Theory and Practice of Software (ETAPS'00)*, volume 1781 of *Lecture Notes in Computer Science (LNCS)*, pages 1–17, London, UK, April 2000. Springer-Verlag.

[134] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the* 10^{th} *European software engineering conference held jointly with* 13^{th} *ACM SIGSOFT international symposium on Foundations of software engineering (ESEC'05/FSE'05)*, volume 30 of *SIGSOFT Software Engineering Notes*, pages 115–125, New York, NY, USA, 2005. ACM Press.

[135] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the* 32^{nd} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'05)*, volume 40 of *SIGPLAN Notices*, pages 351–363, New York, NY, USA, 2005. ACM Press.

[136] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In *Proceedings of the* 13^{th} *International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[137] Emanuel Zarpas. Benchmarking SAT Solvers for Bounded Model Checking. In *Proceedings of the* 8^{th} *International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science (LNCS)*, pages 340–354. Springer, 2005.

[138] Jian Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proceedings of the Quality Software,* 4^{th} *International Conference (QSIC'04)*, pages 87–92, Washington, DC, USA, 2004. IEEE Computer Society.