View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KIT

Efficient Verification of Programs with Complex Data Structures Using SMT Solvers

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tianhai Liu

aus Shandong, China

Tag der mündlichen Prüfung: 11.07.2018

1. Referent: Prof. Dr. rer. nat. Bernhard Beckert
(Karlsruher Instituts für Technologie, Deutschland)

2. Referent: Prof. Dr. Shmuel Tyszberowicz
(The Academic College of Tel Aviv-Yafo, Israel)

Abstract

Complex data structures such as list, tree, and graph are mainly located on the heap. Verifying programs with complex data structures against the properties that constrain the configurations of the objects on the heap is particularly important for safety-critical software systems with extensive heap manipulations. Erroneous heap manipulations may cause loss or unauthorized access to data, violate software security, and may eventually cause a system to crash. Program verification techniques like bounded and deductive program verification, in general, are capable of verifying a program against a property of complex data structures. However, there is always a trade-off between the scope of their analyses, and their level of automation. *Bounded program verification* techniques are fully automatic, but they only analyze a program for a small scope. *Deductive program verification*, on the other hand, have no restriction on the scope, but they often require users to provide auxiliary specifications such as loop invariants and method contracts for sub-routines.

In this thesis, we present a verification infrastructure combing the benefits of different program verification techniques. To verify a program with respect to a specification—specifying a *property* that is expected to be satisfied by the program, the envisioned verification process is as follow:

1. **Bounded program verification.** For gaining fast and initial confidence in the correctness of the program regarding the expected property, we first check whether the property holds for a small scope without providing any auxiliary specification. If a counterexample to the correctness of the program is found, no further analysis is required. Otherwise, the property holds—but only for the scope; thus a deductive verification—an unbounded program verification, is still needed.
2. **Abstraction.** Construct a semantic slice of the program with respect to the property. The semantic slice is an abstract program which contains those statements of the original program that are relevant to the property.

The remainder of the original program (i.e., the irrelevant statements) is replaced by abstractions. Proving the correctness of slices requires less auxiliary specifications as slices have less details.

3. **Bounded verification of auxiliary specifications.** Before continuing with the deductive verification, we check whether the user-provided auxiliary specifications hold in the slice using bounded program verification. If a counterexample to the correctness of the specifications is found, an inspection of the specification is needed. Otherwise, go to step 4.
4. **Deductive program verification.** Prove the correctness of the slice with the bounded-verified auxiliary specifications. If the slice (i.e., the abstract program) is proved, the original program satisfies the expected property as well (by the construction of the slice), and the analysis terminates. Otherwise, a counterexample to the slice is found, and thus go to step 5.
5. **Refinement.** Check whether the counterexample is valid for the original program, i.e., it is also a counterexample to the original program. If it is valid, a fault has been discovered, and the analysis terminates. Otherwise, we refine the slice so that the invalid counterexample is eliminated. The process then starts over at step 3.

This verification process is an instantiation of counterexample-guided abstraction refinement framework and forms the basis of our verification infrastructure. Though it still relies on user-provided specifications for deductive program verification, it holds promise for efficiency. It allows only the necessary parts of the program for the expected property to be analyzed in deductive program verification, so it eases the burden of manually discovering useful annotations. It guarantees fast and initial confidence in the correctness of program and auxiliary specifications, hence avoids unnecessary attempts and facilitates users to inspect the failed proofs in deductive program verification.

Our infrastructure repeatedly uses bounded program verification. To improve its efficiency, we provide novel approaches for bounded program verification. We provide an SMT-based encoding for bounded program verification by exploiting the recent advances of satisfiability modulo theory solvers. The encoding supports programs with complex data structures and specifications with arbitrarily nested quantifiers and reachability expressions, and is used in the rest approaches presented in this thesis. Besides, we provide a calculus to compute suitable scopes for gaining a high code coverage at bounded program verification. Finally, we study the effect of using various constraint solving techniques on the time efficiency of symbol execution—a technique used in program verification systems.

Acknowledgments

Through my Ph.D. studies, I have gained a deeper understanding of software engineering and a new recognition of myself. This learning career in Karlsruher Institut für Technologie (KIT) has been incredible, but a challenging journey for me. At the end of this trip, I would like to appreciate the people who have guided, assisted, and accompanied me.

I cannot finish this thesis without my supervisor, Prof. Dr. Bernhard Beckert. No words can adequately express my gratitude to him. In my most difficult period, He gave me the opportunity to join his group and to pursue this work. He guided the direction of my research and taught me to analyze problems at a higher level. I am very grateful for his unconditional support and help throughout my doctoral studies. His global vision, quintessential academic accomplishments, and meticulous academic style profoundly affect my future work and life.

I was delighted to meet Prof. Dr. Shmuel Tyszberowicz in my life journey. He is not only an excellent supervisor in my doctoral studies but also a best friend in my life. When I repeatedly intended to terminate my studies, he always gave me wise advice and encouraged me to keep going. Without him, I would have been faltering on a rough road for a very long distance. I thank Shmuel Tyszberowicz wholeheartedly for his unlimited support, for helping me to find my way in my studies and my life, and for all his encouragement during the writing of this thesis.

I am remarkably grateful to Dr. Mana Taghdiri for all her omnipresent care, especially in the early stage of my studies. She opened the door for me to the area Automatic Software Analysis and taught me the essential characters that are required by an academic researcher. She gave me lots of advice and cared about the work and life.

As a member of Institut für Theoretische Informatik (ITI) I feel immensely proud. I thank my colleagues, Aboubakr Achraf El Ghazi, Alexander Weigl,

Carsten Sinz, Christoph Gladisch, Daniel Grahl, David Farago, Mattias Ulbrich, Michael Kirsten, Mihai Herda, Sarah Grebing, Simon Greiner, Stephan Falke, and Thorsten Bormer for many useful discussions and comments. They let me feel the warmth of home while living in a foreign country. I express my sincere thanks to them. Thanks to all kinds of encouragement and help they give, their friendship will always be my solace.

Finally, I thank my parents, Xuemei and Guodong Liu, for giving me life and making it vibrant and beautiful. I thank my mother the value of meticulousness and perseverance and my father for teaching me to broaden my horizons and mind.

I thank my wife Kun and my children Marvin and Hannah without whom this thesis would take even more time. My wife has made a tremendous contribution to the family and blocked lots of trivia for me so that I can save time to finish writing. Children can understand me when I cannot accompany them on holidays. I dedicate this thesis to all of them.

Contents

Part I Introduction

1 Introduction 3
- 1.1 Motivation 3
- 1.2 State of the Art and Challenges 7
- 1.3 Contributions 10
- 1.4 Outline 15
- 1.5 Selected Publications 16

2 Foundations 17
- 2.1 Satisfiability Modulo Theories 17
- 2.2 Java Modeling Language 19
- 2.3 Bounded Program Verification 21
- 2.4 Deductive Program Verification 22

Part II Efficient Bounded Program Verification

3 SMT-based Encoding for Bounded Program Verification 27
- 3.1 Construction of a Verification Graph 29
  - 3.1.1 Source Code Transformations 29
  - 3.1.2 Verification Graph 30
- 3.2 Encoding Types 33
- 3.3 Encoding Control-flow 35
- 3.4 Encoding Data-flow 36
- 3.5 Handling Runtime Exception 40
- 3.6 Encoding JML Expressions 41
- 3.7 Evaluation 42

3.8 Related Work 46
3.9 Conclusion 48

4 Computing Loop Bounds based on Class Bounds for Bounded Program Verification 49
4.1 Our Calculus 52
4.1.1 Encoding Loop Control-flow 54
4.1.2 Encoding Loop Data-flow 56
4.1.3 Checking Formulas 59
4.2 Evaluation 60
4.3 Related Work 63
4.4 Discussion 65
4.5 Conclusion 66

Part III Efficient Deductive Program Verification

5 Verification-based Program Slicing for Deductive Program Verification 71
5.1 Motivating Examples 73
5.2 Our Technique 76
5.2.1 Bounded Program Verification 77
5.2.2 Extracting Relevant Code 78
5.2.3 Constructing Abstractions 79
5.2.4 Constructing Semantic Slice 80
5.2.5 Handling Runtime Exceptions 82
5.3 Evaluation 83
5.4 Related Work 87
5.5 Conclusion 88

6 Counterexample-Guided Abstraction Refinement for Deductive Program Verification 91
6.1 Our algorithm 94
6.1.1 Initial Abstractions 94
6.1.2 Checking Auxiliary Specifications 94
6.1.3 Proving 98
6.1.4 Checking Validity of Counterexamples and Refinement 99
6.2 Evaluation 100
6.3 Related Work 103
6.4 Conclusion 105

Part IV Efficient Bounded Symbolic Execution

7 **Bounded Symbolic Execution Using Incremental Constraint Solving** 109
7.1 Background 111
7.2 Our Approach 112
7.2.1 Construction of the Decision Graph 113
7.2.2 Path Condition Generation 114
7.2.3 Path Exploration 116
7.3 Evaluation 116
7.3.1 Experimental Setup 116
7.3.2 Objectives of Analysis 119
7.3.3 Results and Analysis 121
7.3.4 Threats to Validity 127
7.4 Related Work 128
7.5 Conclusion 130

Part V Conclusion

8 **Related Works** 135

9 **Conclusion and Future Works** 139

References 143

Publications 159

List of Figures

1.1 Structure of our infrastructure 5

3.1 Abstract syntax of the supported Java programs 29
3.2 Source code transformations 31
3.3 Construction of a verification graph 32
3.4 SMT encoding of Java type system 35
3.5 Control-flow formulas 36
3.6 Rules for translating program statements 38
3.7 Data-flow formulas 40
3.8 Formulas for handing runtime exceptions 41

4.1 A cyclic verification graph 55
4.2 Control-flow formulas for loop bounds computation 56
4.3 Data-flow formulas for loop bounds computation 58
4.4 Formulas for implicit state transitions 58
4.5 The formula to ensure N_l is the sharp bound for the loop l 59

5.1 Semantic slicing a program manipulating integers 74
5.2 Semantic slicing a program manipulating arrays 75
5.3 Semantic slicing a program manipulating a singly linked list . . 76
5.4 Rules for constructing the formula map 77
5.5 The constructed three kinds of abstractions. 81
5.6 Rules for constructing semantic slices 81

6.1 Structure of CEGAR algorithm 93
6.2 An acyclic verification graph 97
6.3 Formulas for checking loop invariants 98

7.1 A decision graph 113

7.2 Path conditions generated using *stack-based* approaches 115
7.3 Path conditions generated using *cache-based* approaches 120
7.4 Speedup of symbolic execution in constraint solving 122
7.5 Speed of solving path conditions 122
7.6 Speed of exploring paths 123
7.7 Time breakdown of different techniques 125
7.8 Source code transformations cost 126
7.9 Impacts of using different solvers on symbolic execution 127

List of Tables

3.1 Results of bounded program verification 46

4.1 Results of computing loop sharp bounds 62

5.1 Results of deductive program verification with abstractions . . . 85

6.1 Results of deductive program verification with abstraction refinements 102

List of Code Snippets

1 Invalid Copy Semantics 44
2 Invalid Memory Access 44
3 Total pure abstraction. 81
4 Partial pure abstraction 81
5 Impure abstraction 81
6 Code with injected guards 83

Part I

Introduction

CHAPTER 1

Introduction

Software failures in safety-critical systems (e.g., medical, automotive, aviation, and nuclear engineering) may cause severe damage and cost lives. Forty years ago, for example, Therac-25 (a software-controlled radiation therapy machine) emitted more than 100 times radiations to its patients and that malfunction resulted in severe injuries and three patients' deaths [Dalal and Chhillar, 2013, 2012; Wallace and Kuhn, 2001]. An extensive investigation revealed that Therac-25 calculated a wrong number of radiations due to unauthorized access of data. Software engineering has made tremendous progress and achievements, while software failures still remain a serious problem. For example, in mid-2017, Fiat Chrysler Automobiles NV (FCA) claimed to recall more than 1.25 million trucks worldwide to address a software error linked to reports of one death and two injuries. We have to make greater efforts on the road to success in software quality assurance.

1.1 Motivation

Complex data structures have been widely used to represent the complex data relations in software systems with object-oriented design. Unlike simple data structures (also known as primitive data types) such as `int` and `boolean` that are stored on the stack, complex data structures such as `list`, `tree`, `graph`, and user-defined classes are generally built upon primitive data types and reference-based data types, and mainly located on the heap. We focus on verifying programs with complex data structures against the properties that constrain the configurations of the objects on the heap. The properties of interest can be divided into two categories in program verification, i.e.,

the properties of complex data structures and the properties of functions. A property $P(T)$ of a complex data structure T (a class) has to be satisfied by all instances of T whenever a function of T returns, even though the function may not manipulate T. That is, the property $P(T)$ has to be preserved by any function of T. A property of a class `Set` could be that, for example, each object of `Set` contains distinct elements. On the other hand, a property $P(F)$ of a function F is only expected to be satisfied by F when F returns. A property of a function inserting elements to a set could be, for example, the set contains new elements after the function returns, and the old elements are still in the set. Unless specifically stated in the text, we use *properties of complex data structures* to refer these two kinds properties, since a function, in any case, has to satisfy the properties of its enclosing class.

Analyzing properties of complex data structures that constrain the configurations of the objects on the heap is particularly important for safety-critical software systems with extensive heap manipulations. Erroneous heap manipulations may cause loss or unauthorized access to data, violate software security, and eventually cause a system to crash. The properties of interest do not have to be complex, but their corresponding specifications typically involve arbitrarily nested quantifiers and reachability expressions. In spite of all advances in software analysis, analyzing these properties is still a challenging problem.

Traditional program verification techniques like *deductive program verification* are capable of proving the correctness of a program with respect to a specification—specifying a property that is expected to be satisfied by the program. However, it is expensive to extend their application in practice, since these techniques typically require user-provided annotations (also known as *auxiliary specifications*) to construct a proof. Constructing useful annotations is a tedious and error-prone effort, and renovating a failed proof needs more efforts, e.g., to inspect the code and the annotations. *Bounded program verification* techniques, on the other hand, exhaustively check a program regarding a small scope and guarantee a fast and initial confidence in the correctness of program if the bounded verification succeeds. It will be useful if we can bring the advantages of bounded program verification to deductive program verification. Compared to deductively verifying a program from scratch, proving a bounded-verified program will reduce efforts in deductive verification. Unfortunately, existing bounded program verification techniques have issues in scalability and code coverage. That is, they are only able to analyze a program with a low code coverage on a domain with a few objects.

This thesis presents a verification infrastructure combing the benefits of different program verification techniques. The infrastructure aims to verify a program against a given specification (property). Figure 1.1 shows the structure of our infrastructure. To verify a program P with respect to a property Q, the envisioned verification process is as follow:

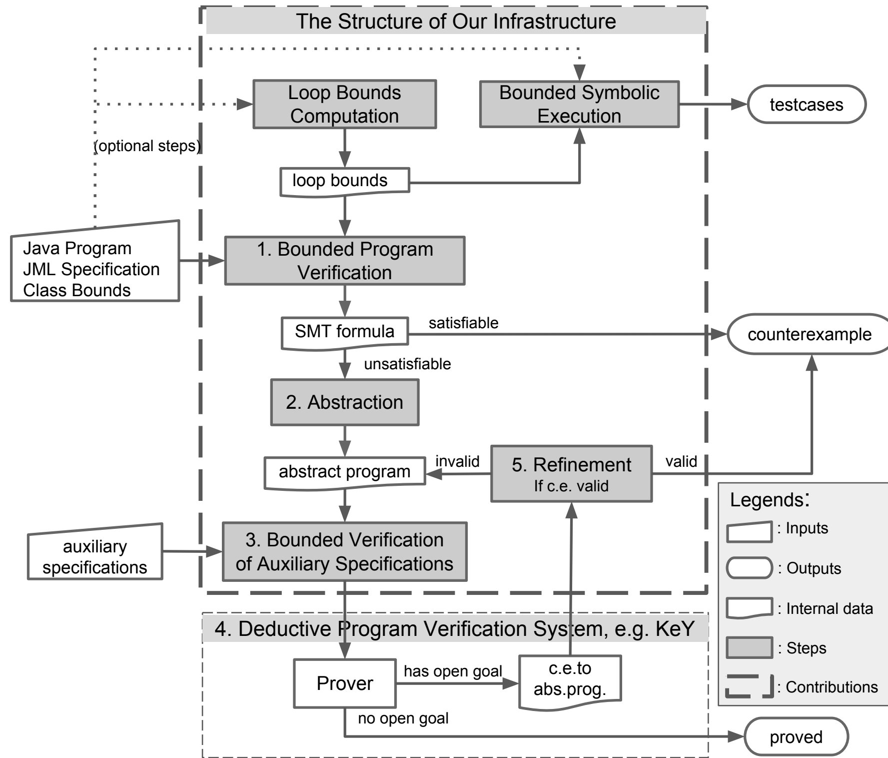


Fig. 1.1: Structure of our infrastructure. The highlighted boxes are the components of our infrastructure, and refer to the principle contributions of the thesis as well. The numbers at highlighted boxes indicate the order of envisioned verification process.

1. **Bounded program verification.** First, check if the property Q holds for a small scope B that limits the number of objects on the heap and the number of loop iterations. If Q does not hold, a counterexample to the correctness of P is found and the analysis terminates. Otherwise, the property Q holds—but only with respect to the scope B; thus a deductive verification—an unbounded program verification, is still needed.
2. **Abstraction.** Construct an abstract program (a semantic slice) $A(P)$ with respect to the property Q based on the proof of invalidity of bounded program verification. In the slice $A(P)$, the statements of the program P that are irrelevant to the property Q are replaced by abstractions. The rest of the program remain unchanged.
3. **Bounded verification of auxiliary specifications.** Before continuing with the deductive verification, check whether the user-provided auxiliary specifications I (e.g., loop invariants) hold in $A(P)$ using bounded pro-

gram verification. If a counterexample to the correctness of I is found (since $A(P)$ has been bounded verified), the inspection of I is needed. Otherwise, go to step 4.

4. **Deductive program verification.** Prove the correctness of the abstract program $A(P)$ with bounded-verified auxiliary specifications I for the property Q. If $A(P)$ is verified, the analysis terminates, and the original program P satisfies the property Q as well (by the construction of the abstract program $A(P)$). Otherwise, a counterexample c to $A(P)$ is found, and thus go to step 5.
5. **Refinement.** Check whether c is valid for the original program P using bounded program verification, i.e., it is also a counterexample to the original program. Meanwhile, a larger scope $L(B)$ for bounded program verification has been computed based on the counterexample c. If c is valid, a fault has been discovered, and the analysis terminates. Otherwise, we refine the slice $A(P)$ so that the invalid counterexample c is eliminated. The process then starts over at step 3.

This verification process is an instantiation of Counterexample-Guided Abstraction Refinement (CEGAR) [Clarke et al., 2000] framework and forms the basis of our verification infrastructure. The construction of abstractions is guided by the bounded proof provided by bounded program verification, and the refinement of abstractions is guided by the counterexample detected by deductive program verification. The novelty of our instantiation is that it constructs (and refines) abstractions at the statement level and handles properties of complex data structures. Though the infrastructure still relies on user-provided specifications for deductive program verification, it holds promise for efficiency. It allows only the parts of the program that are necessary for checking the desired property to be analyzed in deductive program verification, so it eases the burden of manually discovering useful annotations. Besides, it guarantees fast and initial confidence in the correctness of program and auxiliary specifications, hence avoids unnecessary attempts and facilitates users to inspect the failed proofs in deductive program verification.

To consolidate the efficiency of the infrastructure, we exploit the recent advances of Satisfiability Modulo Theory (SMT) solvers [Barrett et al., 2017] (e.g., Yices [Dutertre, 2014], CVC4 [Deters et al., 2014], MathSAT5 [Sebastiani and Trentin, 2015], and Z3 [de Moura and Bjørner, 2008]) and provide novel approaches for the bounded program verification that has been heavily used in our infrastructure. We provide an SMT-based encoding for bounded program verification. The encoding supports Java programs with complex data structures and Java Modeling Language (JML) [Leavens et al., 2006] specifications with arbitrarily nested quantifiers and reachability expressions. We use the encoding to translate programs and specifications into a quantified bit-vector formula (QBVF) [Wintersteiger et al., 2013] regarding a scope of analysis, and check the satisfiability of the formula using an SMT solver. QBVF allows to specify logical constraints that are structurally closer to the

program and the specifications, thus facilitates understanding the models to a satisfiable formula. The SMT solver can perform high-level simplifications on QBVF before reasoning the formula in a basic logic, hence significantly improves the scalability of bounded program verification. Besides, we provide an SMT-based calculus to compute suitable loop bounds based on class bounds for gaining a good code coverage (and heap coverage) in bounded program verification.

To improve the time efficiency of symbolic execution—a means of analyzing programs that has been widely used in program verification and testing, we provide an empirical study of symbolic execution using various constraint solving techniques. The results of the study recommend to use incremental SMT solvers to reduce the time cost of symbolic execution. An incremental SMT solver can (re-)use the intermediate lemmas that it has learned in previously constraint solving as opposite to a common SMT solver that learns lemmas from scratch each time it is invoked.

Though the overall thesis presents a verification infrastructure combing the benefits of different program verification techniques, we design the components of the infrastructure (i.e., the highlighted boxes in Fig. 1.1) as stand-alone analyses so that they can be used in various contexts. We have implemented the components in prototype tools and performed various experiments to evaluate their benefits compared to alternatives. Overall, our tools provide superior results.

1.2 State of the Art and Challenges

Bounded Program Verification

Bounded program verification techniques (also known as static scope-bounded checking) have become an increasingly attractive choice for gaining confidence in the correctness of software. Existing techniques (e.g., Jalloy [Vaziri-Farahani, 2004], JForge [Dennis et al., 2006], TACO[Galeotti et al., 2013], Miniatur[Dolby et al., 2007], Karun[Taghdiri, 2008], and MemSAT[Torlak et al., 2010]) statically check functional properties of an unrolled program (in which loops and recursions are unrolled based on *loop bounds*) with respect to a bounded domain (in which the number of objects of each class is limited based on *class bounds*). They encode the unrolled program and the property of interest as a *propositional logic* formula and check the satisfiability of the formula using a Boolean satisfiability (SAT) solver. These techniques provide an attractive trade-off between automation and completeness. They exhaustively analyze an unrolled program and thus guarantee to find any bug regarding the analyzed property within the domain, but defects outside the domain will be missed. Although these techniques have been successfully used to find bugs in various programs, the encoding using propositional logic limits their

scalability: they can check code with respect to only a few objects and loop unrollings, especially when the code contains integer expressions and arrays.

Moreover, existing bounded program verification techniques typically require users to provide class bounds and loop bounds as separate parameters. These two kinds of bounds, however, have to be well matched for gaining a good code and heap coverage. Many techniques have been proposed to compute loop bounds, including the techniques used in the analysis of worst case execution time (WCET), e.g., [Gulavani and Gulwani, 2008; Ermedahl et al., 2007; Michiel et al., 2008; Thesing, 2004; Cullmann and Martin, 2007], and bounded model checking techniques, e.g., [Milicevic and Kugler, 2011]. These techniques, however, either do not support programs with complex data structures and structural loops, or requires users to provide annotations for the loops before computation, e.g., loop invariants and loop structure patterns. Further, they only compute the approximate loop bounds resulting in the loops are unrolled more times than necessary and thus lots of code will not be covered in the analysis.

With the improvement of the technique for solving satisfiability problems, the extended static checker for Java (ESC/Java [Flanagan et al., 2013] and ESC/Java2 [Cok and Kiniry, 2004]), translate a program and the specifications into a *first-order* logic formula, and check the satisfiability of the formula using SMT solvers such as Z3 and Yices. In particular, they translate the program under analysis into Dijkstra's guarded commands [Dijkstra, 1975], encoding the intended property as an `assert` command. They then compute weakest preconditions to generate verification conditions as predicates in a first-order logic, and use several SMT solvers to resolve the constraints. Failed proofs are turned into error messages and returned to the end user. These techniques, however, use undecidable logics due to quantification over infinite types. They sometimes report a counterexample that might, with more effort, have been shown to be invalid and the solver may not terminate with a conclusive outcome. It would be helpful if the encoding of the bounded program verification supports programs with complex data structures, generates decidable satisfiability formulas, and gains the advances of SMT solvers as well.

Deductive Program Verification

The power of deductive program verification has increased in the last decades. In contrast to bounded program verification that checks for errors in a program regarding a small scope, deductive program verification systems (e.g., KeY [Ahrendt et al., 2016], VCC [Cohen et al., 2009], and PVS [Owre et al., 1992]) aim to *prove* the correctness of a program. The KeY system, for example, proves the correctness of a Java program for intended properties by performing symbolic execution [King, 1976] on the program. The symbolic execution of simple statements is easy, whereas intensive auxiliary specifications are required when loops or recursions are met. Writing necessary specifications

is a creative activity, thus is a difficult and error-prone effort [see Bormer, 2014, Chapter 5] even for an expert of KeY. It is a considerable burden to provide a good auxiliary specification, such that the specification should be weak enough for its annotated sub-routine and strong enough for its invoking procedures. A strong auxiliary annotation increases the difficulty to implement its sub-routine and it is error-prone to write such a specification. On the other hand, a too weak auxiliary specification may cause the proof to fail and hereafter, a costly inspection on the failed proof is indispensable. Therefore, it is beneficial if we can reduce the efforts from the verification engineers in the deductive program verification.

Counterexample-Guided Abstract Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) framework iteratively refines abstract models of a system using counterexamples. It was first introduced by Kurshan [Kurshan, 1994], and then appeared in a number of analysis techniques (e.g., [Balarin and Sangiovanni-Vincentelli, 1993; Lind-Nielsen and Andersen, 1999; Clarke et al., 2000, 2003] that focus on checking finite state systems. From this century CEGAR framework has been widely used in program verification (e.g., [Ball et al., 2004; Chaki et al., 2004; Clarke et al., 2005; Taghdiri and Jackson, 2007; Beyer et al., 2007; Gupta et al., 2011; Abdulla et al., 2016]). These CEGAR instances construct abstractions of the program under analysis, and refine the abstractions based on the spurious counterexamples detected in the program verification. The on-demand iterative nature of CEGAR framework guarantees that only as much information about the program will be analyzed as is necessary to check the property of interest. Proving abstract programs may require less auxiliary specifications in deductive program verification, since abstract programs have less details. It will be beneficial if we apply CEGAR framework on deductive program verification to ease the burden of manually discovering useful auxiliary specifications. Existing CEGAR instances, however, construct abstractions mostly at the predicate level (e.g., [Abdulla et al., 2016]) and rarely at the function level (e.g., [Taghdiri and Jackson, 2007]). To check a program for some property, they typically abstract the program as a Boolean program using a given set of predicates, model checks the Boolean program, and discover additional predicates to refine the Boolean program. That is, only the predicates of the code are replaced by abstractions if they are irrelevant to a property of interest. The rest of the code (e.g., assignment statements) remains unchanged, even if it is not relevant to the property at all. It is very likely that such coarse abstracted programs are still very hard to be annotated by verification engineers. Moreover, they mostly do not handle the properties of complex data structures that our infrastructure targets. They have been used to handle properties of a finite state machine or to check access violation errors such as null-pointer dereference and array access out of bound.

1.3 Contributions

The thesis presents an efficient verification infrastructure to verify a program with complex data structures against the properties that constrain the configurations of program objects on the heap. In particular, we provide an efficient bounded program verification to reduce the efforts of the verification engineers in deductive program verification. We design the components of the infrastructure as stand-alone analyses that can be used in a variety of contexts. For gaining efficient analyses, we provide novel approaches for each component by exploiting recent advances of SMT solvers. The concrete contributions of this thesis are as follows:

1. **An SMT-based encoding of programs and specifications for bounded program verification.**

   Existing bounded program verification techniques typically translate object-oriented programs and their specifications into a propositional logic formula, and solve it using an SAT solver. Although they have been successfully used to find bugs in various programs, such a direct translation to SAT (also known as bit-blasting) limits their scalability: they can check code regarding a few objects and loop unrollings—loops are unrolled in the code, especially when the code also contains integer expressions and array objects.

   We provide a novel approach to bounded program verification that exploits recent advances in SMT solvers to provide better scalability. We introduce a translation of object-oriented programs and their specifications to quantified bit-vector formulas (QBVF), which can be solved efficiently using recent SMT solvers. Compared to SAT-based bounded program verification techniques that only perform Boolean-level simplifications (e.g., shared expression detection and symmetry breaking), an SMT solver performs high-level reasoning and simplifications (such as heuristic quantifier instantiation and template-based model finding), then flattens the formula and analyzes it in a quantifier-free SMT logic, and uses bit-blasting only as a last resort. This significantly improves the scalability of the solver. Our SMT-based encoding supports Java programs with complex data structures and JML specifications with arbitrary nested quantifiers and reachability expressions.

   We have implemented our approach in the prototype tool InspectJ and compared it to JForge, an SAT-based bounded program verification tool, using a large-scale implemented shortest-path algorithm. The results show that InspectJ can analyze code for more objects and loop unrollings than JForge. The average speedup that InspectJ obtained is $\sim$25x.

   Chapter 3 presents the original work that has been published in [Liu et al., 2012], and the contributions that have not been published yet. The unpublished work consists of:

- A verification graph to represent the control- and data-flow of a program. It is essentially a labeled control-flow graph annotated by specifications. Compared to the computation graph (used in [Liu et al., 2012]) that is designed for a program without loops, the verification graph also supports the program with loops. Besides, it contains nodes representing the desired properties at certain control point and represents the program and specifications in a single graph. Hence, it facilitates the generation of formulas and the understanding of the models to the satisfiable formulas. The graph is presented in Section 3.1.
- An alternative approach to handle runtime exceptions. Compared to the approach that has been published in [Liu et al., 2012], this new approach reduces the number of SMT variables in the target formula, thus makes the formulas easier for SMT solving. The new approach is presented in Section 3.5.
- The rules that are used to translate the program into SMT formulas. They are presented in Fig. 3.6.

2. **A calculus to compute suitable scopes for bounded program verification.**

   Bounded program verification techniques statically analyze a program regarding a small scope. They typically require users to provide two kinds of bounds to designate a scope: *class bounds* that limit the number of objects of each class on the heap, and *loop bounds* that limit (by loop unrolling) the number of iterations of each loop. These two kinds of bounds, however, are not independent and their intricate relations are scattered in the code and specifications. Exhaustively enumerating all inputs to the program to compute the exact loop bounds based on class bounds, however, is not practical. Therefore, it is a challenging problem to designate a suitable scope for which all code of the unrolled program and objects of the bounded domain will be covered in the bounded verification.

   Our calculus computes suitable scopes for bounded program verification by computing exact loop bounds (if they exist) based on class bounds. We encode a loop of the program under analysis as an SMT formula, and check the satisfiability of the formula using an SMT solver that can handle optimization problems. If the formula is satisfiable, we get the loop bound from the model of the formula. Otherwise, the analyzed loop is either unreachable from the entry point of the analyzed program, or there are inputs to the program for which the loop does not terminate. We can distinguish these two kinds of cases by analyzing the proof of invalidity generated by the solver. Our calculus supports arbitrarily complex structural loops.

   We have implemented our calculus in the prototype tool BoundJ and compared it with an alternative approach that is used in bounded model checking. In contrast to the alternative approach that either computes

imprecise loop bounds or does not terminate, the results show that BoundJ computes exact loop bounds for the given class bounds.

Chapter 4 presents the original work that has been published in [Liu et al., 2017], and some discussions that have not been published yet. In the discussions, we propose some alternative approaches to compute loop bounds based on class bounds, and investigate the possibility to use these approaches in bounded program verification. These discussions include:

- A discussion on computing loop bounds by exhaustively enumerating all inputs to the analyzed program based on class bounds. For the benchmark used in our experiments, we show the size of configurations of the objects on the heap before calling the analyzed methods. The discussion is presented in Section 4.2.
- A discussion on computing loop bounds using bounded model checking (BMC) techniques. It is presented in Section 4.4.

In addition, we also investigate the related work to detect non-terminating loops in scope-bounded code analysis in Section 4.3.

3. **A verification-based program slicing technique for deductive program verification.**

Deductive program verification systems, e.g., KeY system, typically require experienced verification engineers to write auxiliary specifications such as loop invariants and method contracts of sub-routines. The engineers have to first identify the program slices that are relevant to the property of interest. However, it is challenging for them to find such program slices for a *partial* property—constraining parts of program behaviors, since usually only parts of the implementation are relevant to a partial property. It would be very helpful if we can automatically find the slices in deductive program verification.

We provide a verification-based program slicing technique to construct a *semantic slice* (an abstract program) for a partial property. In the slice, the parts of the program that are irrelevant to the partial property are replaced by an abstraction, whereas the rest of the program (i.e., the relevant parts) remains unchanged. In contrast to verifying the whole program against a partial property, verifying a slice requires less auxiliary specifications, and the correctness of the slice implies the correctness of the original program (by the construction of abstractions). As a result, our program slicing technique eases the burden of manually discovering the relevant slices for a partial property and expedites the progress of deductive program verification—less proof steps are needed for the slice.

The novelty of our technique stems from using bounded program verification techniques to guide the construction of slices. Bounded program verification techniques, requiring no auxiliary specification, translate the analyzed program and its *negated* specification into a satisfiability problem—an SMT formula consisting of a set of SMT constraints, and try

to find a solution to that problem. If a solution to the problem is found, then that is a counterexample to the correctness of the original program, and no further analysis is required. If no solution is found (i.e., the formula is unsatisfiable), the partial property holds for user-provided bounds, and we obtain an *unsatisfiable core* (unsat core) by exploiting the recent advances of SMT solvers. An unsat core is a subset of SMT constraints that is unsatisfiable on the information gained during the unsatisfiability proof. Therefore, the statements of the program are irrelevant to the property of interest when their target SMT constraints (by the *bijective* function that translates statements to SMT constraints) are not in the unsat core. The rest of the program is relevant to the property (by minimization of the unsat core).

We have implemented our technique in the prototype tool AbstractJ and evaluated the benefits of using our technique to prove some programs that are interesting to deductive program verification The results show that on average the verification using our technique requires only $1/2$ of the auxiliary specifications and calculus rules compared to proving it as usual in deductive program verification.

Chapter 5 presents the original work that has been published in [Liu et al., 2016], and further contributions that are not published yet. The unpublished work consists of:

- An example to show the benefits of using our technique for proving programs with linked data structures. Proving the correctness of programs with linked data structures, in general, is very difficult since it typically requires verification engineers to provide non-trivial lemmas for reasoning about sets of linked objects, e.g., what objects can be reached from a source object following particular class fields. The example is presented in Section 5.1.

4. **An algorithm instantiating counterexample-guided abstraction refinement framework for deductive program verification.**

The Counterexample-Guided Abstraction Refinement (CEGAR) methodology provides an automatic framework that gradually refines abstract models of a system and holds promise for scalability. Starting with an initial abstract program with abstractions, it iteratively refines the abstractions based on spurious counterexamples. Many code analysis techniques have instantiated the CEGAR framework for gaining a scalable analysis. All instances of this framework, however, construct abstractions of the code at the predicate level. That is, abstractions replace only the predicates of the code that are irrelevant to the property of interest. The rest of the code (e.g., assignment statements) remains unchanged, even though it is not relevant to the property at all. Therefore, the abstract program does not provide precise relevance between the program and the intended property. Besides, they focus on checking properties of a finite state machine or

access violation errors such as null-pointer dereferences and array access out of bound.

We provide a CEGAR algorithm for deductive program verification. The algorithm forms the basis of the envisioned process using our infrastructure. The novelty of our algorithm is that it handles properties of complex data structures and construct abstractions at the statement level. In particular, we use our verification-based program slicing technique to construct and refine abstractions. The on-demand iterative nature of the algorithm guarantees that only the necessary parts of the program to check the property are analyzed in deductive program verification. Therefore, our algorithm eases the burden of discovering useful auxiliary specifications for deductive program verification.

We have implemented our algorithm in the prototype tool RefineJ and evaluated the benefits of using our algorithm in deductive program verification. Though in general the more refinements, the more auxiliary specifications and calculus rules are needed in deductive program verification. The results show using our algorithm in practice requires fewer specifications and rules than proving a program as usual. Out of the total 21 verification tasks using our algorithm, 19 tasks require less auxiliary specifications, and 13 tasks require fewer calculus rules compared with proving as usual in KeY.

Chapter 6 presents the original work that has been published in [Liu et al., 2016], and two contributions that are not published yet. The unpublished work consists of:

- An SMT-based encoding of auxiliary specifications to check whether the abstract program fulfills the auxiliary specifications. Bounded program verification is fully automatic, and thus facilitates users to amend their specifications before the deductive program verification. It guarantees fast and initial confidence in the correctness of user-provided specifications for deductive verification. Therefore, we reduce the scope of inspection on a failed proof and avoid unnecessary attempts in deductive program verification. This encoding is presented in Section 6.1.2.
- An experiment to evaluate the benefits of using our algorithm, and particularly the benefits of using our refinement techniques in deductive program verification. It is presented in Section 6.2.

5. **An efficient symbolic execution technique using incremental SMT solvers.**

   Symbolic execution as a means of analyzing programs has been widely used in program verification and testing. One important obstacle for expanding the application of symbolic execution in practice is its high cost of path condition solving. We present an efficient symbolic execution technique by exploiting the advances of incremental SMT solvers. Incremental

SMT solvers learn lemmas during constraint solving, and these lemmas can be later (re)used to solve similar, *but not identical*, constraints.

To the best of our knowledge, *no* research has studied incremental SMT solving in symbolic execution. Thus it is particularly important to evaluate this technique for symbolic execution. We have implemented our technique in the prototype tool SymbolicJ and compared it with other incremental constraint solving alternatives on a huge set of benchmarks (400 programs analyzed, and each of them has 100 to 20000 lines of code) and a variety of criteria such as the number of solved path conditions, the speed of path exploration, and the number of feasible paths explored. Overall, the results show that our technique provides the best efficiency. In particular, the speedup obtained using incremental SMT solving is of $\sim$14.4x to an alternative incremental symbolic execution technique.

Chapter 7 presents the original work that has been published in [Liu et al., 2014], and additional contributions that are not published yet. The unpublished work consists of:

- A path exploration algorithm that has been used in our technique. It is presented in Section 7.2.3.
- Evaluating the impacts of using different SMT solvers to the performance of symbolic execution techniques. It is possible that the experimental results are specific to an SMT solver. To address this threat we performed an empirical study using three solvers: CVC4, MathSAT5, and Z3, the modern SMT solvers that are widely used in software and hardware analysis. Although these solvers show different time costs for the same constraint problems, the results still show our technique is more efficient than the alternatives. This work is presented in Section 7.3.3.

1.4 Outline

The thesis is divided into five parts. After introducing the context of the topic in Part I, the thesis presents its contributions in the subsequent three parts, where Part II is about bounded program verification, Part III about deductive program verification, and Part IV about bounded symbolic execution. Part V concludes the thesis.

Part I shows the topic of interest and state of art in Chapter 1 and the technical foundations of its fundamental techniques in Chapter 2.

Part II begins with an SMT-based encoding for bounded program verification in Chapter 3. The encoding aims to improve the scalability of bounded program verification and has been used by the remaining contributions of the thesis. Chapter 4 presents a calculus computing loop bounds based on class bounds to increase the object coverage and code coverage in bounded program verification.

Part III discusses several possibilities to bring the advantages of bounded program verification techniques to deductive program verification. As the first chapter of Part III, Chapter 5 presents a program slicing technique that constructs abstractions of a program for gaining efficient deductive program verification. This chapter provides the motivation and the abstraction technique for Chapter 6, which presents an algorithm instantiating counterexample-guided abstraction refinement framework for deductive program verification.

Part IV provides an empirical study on incremental symbolic execution techniques in Chapter 7.

Part V presents the related work in Chapter 8 and then ends with conclusions in Chapter 9. Nevertheless, each chapter also contains its relevant related work and conclusion for the area that it concentrates.

1.5 Selected Publications

Unless specifically stated in the text, most of the contributions in this thesis have been published in the following *selected* references.

1. **Tianhai Liu**, Michael Nagel, and Mana Taghdiri. Bounded Program Verification Using an SMT Solver: A Case Study. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST), Canada, Proceedings.* pages 101-110. IEEE, 2012.
2. **Tianhai Liu**, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference (HVC), Israel, Proceedings.* volume 8855 of *LNCS*, pages 284–299. Springer, 2014.
3. **Tianhai Liu**, Shmuel S. Tyszberowicz, Mihai Herda, Bernhard Beckert, Daniel Grahl, and Mana Taghdiri. Computing Specification-Sensitive Abstractions for Program Verification. In Martin Fränzle and Deepak Kapur and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium (SETTA), China, Proceedings.* volume 9984 of *LNCS*, pages 101–117. Springer, 2016.
4. **Tianhai Liu**, Shmuel S. Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. Computing Exact Loop Bounds for Bounded Program Verification. In Kim G. Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Third International Symposium (SETTA), China, Proceedings.* volume 10606 of *LNCS*, pages 147–163. Springer, 2017.

CHAPTER 2

Foundations

2.1 Satisfiability Modulo Theories

A Boolean Satisfiability (SAT) problem is a decision problem that can be expressed as a formula in *propositional logic*. An SAT formula (or an SAT instance), that is represented either in Conjunctive Normal Form (CNF) or Disjunctive Normal Form (DNF), consists of atoms (e.g., `v`, and $\neg$`v`), connectives ($\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$), and parentheses. An SAT formula is *tautology* if it evaluates to $true$ for all interpretations, *contradictory* (or unsatisfiable) if always $false$, or *satisfiable* if $true$ for at least one interpretation.

A Satisfiability Modulo Theory (SMT) instance, in principle, is an SAT instance with various theories that are expressed in classical *first-order logic* (FOL) with equality. The theories of, for example, *integers*, *bit-vectors*, and *arrays* are used to encode program variables and fields in SMT-based program verification techniques. Each theory defines specific predicates and functions, e.g., the theory of integers defines various arithmetic functions which facilitate the modeling of arithmetic operations in the program code. SMT is a generalization of SAT with more expressive power than SAT. In contrast to SAT that treats atoms as bitwise-level variables, SMT does it in word-level. The recent SMT solvers typically first resolve an SMT formula using the theories involved in the formula, before flattening the formula in propositional logic. SMT allows logical constraints that are structurally closer to the original program and specification, and can be significantly simplified via high-level reasoning. For example, a Java conditional expression `0!=1` can be represented using two 4-bits bit-vectors of `[0,0,0,0]` and `[0,0,0,1]`, and translated into an SAT formula using 8 Boolean variables as shown in Formula 2.1. Resolving

the SAT formula needs to interpret the Boolean variables, while for an SMT formula it compares two numbers directly using the theory of integers.

$$
\begin{aligned}
&(x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg x_3) \wedge \\
&(\neg y_0 \wedge \neg y_1 \wedge \neg y_2 \wedge \neg y_3) \wedge \\
&((x_0 \wedge \neg y_0) \vee (x_1 \wedge \neg y_1) \vee (x_2 \wedge \neg y_2) \vee (x_3 \wedge \neg y_3))
\end{aligned}
\tag{2.1}
$$

SMT-LIB [Barrett et al., 2017] is a description language for SMT. It has been proposed in 2003 and since then has been gaining force as a standard SMT format and been supported by various SMT solvers such as Z3, Yices, CVC4, MathSAT5, Boolector [Brummayer and Biere, 2009], veriT [Bouton et al., 2009], SMTInterpol [Christ et al., 2012], AProVE [Giesl et al., 2017], and STP [Ganesh and Dill, 2007].

The SMT logic is a many-sorted FOL. It is explained using the SMT-LIB 2.6 syntax where expressions are given in prefix notation. Here, we describe the basic theories and the operators. Further SMT constructs will be explained once they occur in the thesis. In circumstances where there is no ambiguity, we use SMT to represent the SMT-LIB language in the thesis.

Core Theory.

In the Core theory, the unique sort `Bool` denotes the set $\{true, false\}$ of Boolean values. The instances of the `Bool` sort are connected using the basic Boolean operators `and`, `or`, `not`, `=>` (implies), and `ite` (if-then-else). Besides, the keywords for universal (`forall`) and existential (`exists`) quantifiers facilitate expressing complex formulas. The Core theory is implicitly used by other theories; that is, its sort, operators and keywords are used by others.

Theory of Integers.

In the theory of Integers, the unique sort `Int` represents the mathematical numerals. Besides the classical mathematical operations, e.g., `+`, `-`, and `*`, the numerical relational operators `>`, `<`, `>=`, and `<=` can be used to encode the arithmetic operations and conditional expressions of Java programs.

Theory of Bit-Vectors.

In the theory of Bit-Vectors, a sort is defined as `(_ BitVec m)` where m is a numeral greater than 0 denoting the size of the bit-vectors (instances of the m-bit sort). The expression `(_ bv`n m`)` denotes an instance (a numerical n) of the m-bit sort. Bit-vectors of different sizes have different sorts in SMT. This theory can be used to encode the precise semantics of unsigned and of signed two-complements arithmetic. It supports a large number of logical and arithmetic operations on bit-vectors. Examples include `bvule` (unsigned less than or equal to), `bvuge` (unsigned greater than or equal to), `bvadd` (addition), `bvashr` (arithmetic shift right), and `bvshl` (shift left).

We can obtain a conclusive result of an SMT formula in a decidable SMT logic with the theory of Bit-Vectors. The SMT logic with the theory of integers is undecidable, hence it is possible that the underlying SMT solver outputs 'unknown'.

SMT Commands.

The SMT command `(declare-fun f (S`$_1$` .. S`$_{n-1}$`) S`$_n$`)` declares an uninterpreted function $f : S_1 \times .. \times S_{n-1} \rightarrow S_n$. Constants are functions that take no arguments. Basic formulas are combined using the Boolean operators. Multiple SMT problems often share similar sets of declarations. An SMT solver typically maintains an assertion stack to take advantage of such similarities. SMT provides the commands `push` and `pop` to manipulate such stack. Each stack frame stores an *assertion set*, which includes locally-scoped declarations of functions, sorts, and logical formulas. The command `(assert F)` adds a formula `F` in the current assertion set. The command `(check-sat)` checks if all assertions sets in the stack are satisfiable. If the formula is satisfiable, the SMT command `(get-model)` returns a satisfiable model that contains the values of the variables, the interpretations of functions and predicates in the analyzed formula. Otherwise, using the SMT command `(get-unsat-core)` we can obtain the proof of unsatisfiability, an unsatisfiable SMT formula that consists of a subset of assertions of original formula.

2.2 Java Modeling Language

Java Modeling Language (JML) is a behavioral interface specification language for Java. It has been widely used in numerous software analysis tools. Examples of tools for program verification are KeY, JForge, ESC/Java2, TACO, and Sireum/Kiasan for Java [Deng et al., 2012]. Following the design by contract paradigm [Meyer, 1997], JML allows one to formally describe the expected behaviors of a Java module and to statically or dynamically check whether an implementation fulfills the method contracts.

JML provides a rich set of specification facilities. Classical logical operators, side-effect free Java expressions, and first-order constructs can be used in specifications. Besides, JML provides additional clauses and keywords to strengthen its expressiveness. Following is a short description of JML clauses and keywords used in the thesis. Further JML constructs will be explained when they occur.

A basic JML specification can be categorized by the two clauses **`requires`** and **`ensures`** for the pre- and post-state of a method, respectively. The pre-state of a method is the program state after passing parameters to the method and before executing the method's body. The post-state of a method is the state of the program just before the method normally returns or throws an exception. The **`requires`** clause denotes a method's precondition that has

to be satisfied in the pre-state of the method. The **ensures** clause specifies a method's postcondition, that has to hold in the post-state of the method. Preconditions and postconditions may be violated when the specified method is executing, while their evaluations must be done at the pre-state and the post-state of the method, respectively. JML provides another clause, the **invariant**, to denote a class invariant that has to be satisfied by all instances of the class at both pre-state and post-state of the method[1]. In the thesis we suppose the class **invariant** has been substituted by a pair of **requires** and **ensures** clauses for simplicity. JML annotations are marked by `//@` or `/*@` (closed by `*/`) in Java comments.

A JML formula consists of side-effect free expressions using standard logical operators, and universal (**\forall**) and existential (**\exists**) quantification. An Boolean-valued expression of the form `(\forall T v;D(v);F(v))` evaluates to **true** if for *all* instances of type `T` that satisfy the domain restriction `D`, the formula (or relation) `F` is **true**. The universal quantification is distinct from existential quantification `(\exists T v;D(v);F(v))`, which only constrains that the formula (or relation) `F` holds for at least one instance of the domain `D`. Note that, the quantifier range in the thesis includes only those objects that have been created in the current heap state, in contrast to that the range of JML quantifiers extends over all objects of the given variable type, including objects that are not yet created [see Leavens et al., 2006, Chapter 12 page 113]. To describe the properties of complex data structures, JML provides a reflexive transitive clause `\reach(T x, Field f)` that denotes the smallest set that contains `x`, and all instances of type `T` that are reachable through the field `f`. Since this function only supports one field, we introduced a Boolean-valued function `\reach(Object x, Object y, FieldSet fs)` that returns **true** when `y` is in the reflexive transitive closure of `x` over any field in the `fs` union of fields.

The JML specifications should avoid throwing exceptions. For example, the expression `x.f>0` will trigger `NullPointerException` if `x` is **null**. To avoid exceptions in JML specifications, we suppose that all Java member fields, formal parameters, and return values, by convention, are non-null in JML if the **nullable** modifier is not used. For other cases such as array access operations and arithmetic computation, it is JML user's obligation to avoid exceptions in the specifications. Otherwise, the verification results will be erroneous. When an expression throws an exception in a **requires** (or **ensures**) clause, it stops evaluating the rest parts of the clause, and immediately evaluates the **requires** (or **ensures**) clause to be **false**, hence the analyzed program will be trivially considered to be correct (or wrong). For example, the JML precondition `/*@requires x/0==x/0 || true;@*/` is **false** since the sub-expression `x/0` will throw an `ArithmeticException` of divide-by-zero. Note that JML

[1] Class invariants do not have to be preserved by the methods that are declared with the **helper** modifier.

adapts Java short-circuit evaluation, that is why the above expression is **`false`** while the JML precondition `/*@`**`requires true`**` || x/0==x/0;@*/` is **`true`**.

In addition to the properties specified in **`ensures`** clauses, there are two kinds of properties that are important for the correctness of the analyzed method: *whether the method terminates* and *whether the method throws exceptions*. These two kinds of properties are difficult (or impossible) to be specified using **`ensures`** clauses. Therefore, JML provides specific constructs for them. The **`diverges false`** clause specifies that its annotated method has to terminate, whereas **`diverges true`** disables the termination checking. If a method terminates and throws an exception, it terminates exceptionally, otherwise it terminates normally. In the thesis, we only check if the intended property holds when its annotated method terminates normally. That is, no exception has been thrown is the prerequisite to check the **`ensures`** clause. We handle exceptions runtime exceptions (e.g., `NullPointerException`, and `ArrayIndexOutOfBounds`) as built-in properties of the analyzed method. When an exception is thrown, a built-in property is violated, and a fault is found. This property can be specified using the keyword **`normal_behavior`**.

Finally, JML allows specifying the heap locations that are allowed to be updated in a method. Only the heap locations (represented by a set of fields) of the **`assignable`** clause are allowed to be modified by a method, even if they are just temporary modifications that fall back to original values after the method invocation. The clause **`assignable \nothing`** denotes that the annotated method does not modify heap locations, but it is allowed to allocate objects; **`assignable \everything`** enables the method to modify any heap location and to allocate objects. Compared to **`assignable \nothing`** that allows object allocation, JML* [Weiß, 2011], a modified version of JML used in KeY, provides the **`assignable \strictly_nothing`** that does not allow to modify heap locations or to allocate objects. In the thesis, we also support the clause **`assignable \strictly_nothing`** for gaining a more accurate specification.

2.3 Bounded Program Verification

Bounded program verification (also known as static scope-bounded checking) has become an increasingly attractive choice for gaining confidence in the correctness of software. Existing bounded program verification techniques (e.g., JForge, Jalloy, MiniAtur, TACO, MemSAT [Torlak et al., 2010], and Karun [Taghdiri, 2008]) statically and exhaustively check functional properties of a program (in which loops and recursions are unrolled) with respect to a bounded domain (in which the number of objects of each class is bounded). They encode a bounded program and a property of interest into a logical formula with respect to a bounded domain and check the satisfiability of the formula using an SAT/SMT solver. They provide an attractive trade-off between automation and completeness. They automatically exhaustively ana-

lyze a program based on the bounds and thus guarantee to find within the bounds any bug with respect to an intended property, but defects outside bounds may be missed.

A bounded program verification process typically consists of i) the code transformation that unrolls loops and recursions, ii) the translation of the program and specification into a logical formula, and iii) the formula resolution using a solver. The existing techniques typically require users to provide both the *loop bounds* and the *class bounds* for the code transformation and translation. Loop bounds contain the number of unrollings for each loop in the code transformation, and class bounds contain the maximal number of objects of each class in the program translation. Specifically for the primitive type, e.g., the Java `int`, the class bound denotes the number of bits to represent an integer as a string of bits.

Bounded program verification is modular—it checks program methods in isolation, against specifications—and focuses on data-related properties of data-structure-rich programs—those that manipulate the configurations of the objects on the heap. Such properties can also be verified by full verification engines such as deductive software verification systems (e.g., KeY), SMT-based proof engines (e.g., Boogie [Barnett et al., 2005]), and shape analyses (e.g., TVLA [Lev-Ami and Sagiv, 2000]). Although such approaches provide proofs, they either require users to add extensive annotations (e.g., loop invariants), or do not support arbitrary data structures. Bounded program verification techniques do not require auxiliary specifications.

Bounded program verification has similarities to bounded model checkers (e.g., CBMC [Clarke et al., 2004], SMT-CBMC [Armando et al., 2009], LLBMC [Sinz et al., 2010], ESBMC [Cordeiro et al., 2012], VeriSoft [Godefroid, 2005]). They both perform a fully automatic and exhaustive analysis with respect to the given bounds and require little intermediate annotations from the user. However, model checkers focused on temporal safety properties of entire programs, though recently some model checkers support modular analysis and specifications.

2.4 Deductive Program Verification

Deductive program verification systems (e.g., KeY, VCC, and PVS) are very effective to faithfully prove the correctness of programs with respect to non-trivial properties. Typically, they translate a program and a specification into a logical formula (proof obligation) and apply various deduction rules on the formula to obtain a proof. They often require users to private auxiliary specifications (e.g., method contracts and loop invariants).

The KeY system accommodates a range of techniques to prove or disprove that a program satisfies given properties, e.g., interactive theorem proving, abstract interpretation, modular specification, and counterexample generation. Program verification using KeY is usually done in auto-active style: the

user interacts with the system only through provided auxiliary specifications. The symbolic execution of simple statements is easy, whereas intensive auxiliary specifications are required when loops or recursions are met. Writing necessary annotations is a creative activity, thus is a difficult and error-prone effort even for an expert of KeY.

The KeY system uses Java Dynamic Logic (JavaDL) for reasoning Java programs. The Java program and its intended specification are translated into a JavaDL *sequent* (i.e., a JavaDL formula). JavaDL extends FOL with (i) a type system whose signatures are based on Java's (partial order) type system, (ii) a program p, that describes the state changes of legal sequence of Java statements, and (iii) the modalities $\langle p \rangle\, \phi$ (diamond) and $[p]\, \phi$ (box), in which ϕ is a JavaDL formula. Intuitively, $\langle p \rangle\, \phi$ denotes total correctness, i.e., p terminates and the formula ϕ holds in the final state, and $[p]\, \phi$ means partial correctness: if p terminates then formula ϕ holds in the final state.

The common basis of KeY is a rule-based symbolic execution engine. It symbolically executes a sequence of statements following the natural control flow and replaces each assignment statement by JavaDL expressions that can effectively represent the state changes effected by the statement. This replacement activity named as *updates* in KeY system. Considering, for example, the following JavaDL Formula (2.2) that has been translated from the program `int foo(int x){x = x+1; return x;}` with the pre- and postcondition `x>0`.

$$\begin{aligned} \forall\ int\ x; (x > 0 \rightarrow \\ [x = x + 1;]\, x > 0) \end{aligned} \tag{2.2}$$

In the *updates* phase, various KeY proof rules are used to substitute the program statements. The Quantify Elimination rule, for example, eliminates the quantifiers by replacing the quantified variable with a free variable in the context. Applying the rule on Formula 2.2, the sequent becomes Formula (2.3), in which x_0 is a free variable.

$$\begin{aligned} &(x_0 > 0 \rightarrow \\ &[x_0 = x_0 + 1;]\, x_0 > 0) \end{aligned} \tag{2.3}$$

Symbolic execution explores the statement $x_0 = x_0 + 1$ in Formula (2.3) and updates the sequent using the Assignment rule. A new sequent is generated, as shown in Formula (2.4).

$$\begin{aligned} &(x_0 > 0 \rightarrow \\ &x_1 = x_0 + 1; [\,]\, x_0 > 0) \end{aligned} \tag{2.4}$$

Finally, the empty box operator [ ] is discarded in Formula (2.4) and the final sequent is generated, as shown in Formula (2.5) that can be easily proved (i.e., the proving goal to prove the sequent can be closed), hence the method `foo` preservers the property `x>0`.

$$x_0 > 0 \rightarrow x_0 + 1 > 0 \tag{2.5}$$

If the proving goal to prove a sequent can not be closed, i.e., a program with provided auxiliary specifications are not verified, KeY tries to generate a counterexample for the sequent in the goal. KeY translates the sequent into an SMT formula with size-bounded SMT sorts. Users of KeY can specify the size of the SMT sorts. Using heaps, location sets, and sequences would require a very large size for these SMT sorts, thus in a preprocessing step, semantic blasting is performed before the translation to SMT. During this step the occurrences of some function and predicate calls are replaced with . Then the translation is handed over to an SMT solver that will tries to find a model for it. If a model is found, KeY presents it in a readable form to facilitate users in the inspection of the failed proof.

KeY accepts JML* [Weiß, 2011], a modified version of JML, as the specification language for Java programs. JML* implements most, but not all, JML features and adds a few more. One example of the extension is the interpretation of JML quantifiers. Like our interpretation of quantifiers 2.2, JML* quantifiers range over the objects that have been created before evaluating the quantifiers.

Part II

Efficient Bounded Program Verification

CHAPTER 3

SMT-based Encoding for Bounded Program Verification

In this chapter we present an SMT-based encoding of programs and specifications for bounded program verification. Bounded program verification techniques (e.g., Jalloy [Vaziri-Farahani, 2004], JForge [Dennis et al., 2006], TACO[Galeotti et al., 2013], Miniatur[Dolby et al., 2007], Karun[Taghdiri, 2008], and MemSAT[Torlak et al., 2010]) typically translate object-oriented programs and their specifications into a propositional logic formula, and check the satisfiability of the formula using an SAT solver. In the translation, the bit-blasting technique [Kroening and Strichman, 2016] is used to translate the intermediate representation of programs and specifications into a Boolean formula, as shown in Section 2.1. With the increase of the size of inputs (e.g., bit-width), bit-blasting will generate a huge amount of terms in the target formula so that it is impossible to be handled by an off-the-shelf SAT solver. Besides, bit-blasting disperses the high-level information (e.g., bit-vector level) thus the target formula may contain terms that are unnecessary for SAT solving. Therefore, scalability is one of the issues for these bounded program verification techniques using such a direct translation to SAT.

Our encoding exploits recent advances in SMT solvers to gain better scalability in bounded program verification. We encode a Java program and its JML specifications as a Quantified Bit-Vector Formula (QBVF) and check the satisfiability of the formula using an SMT solver. QBVF allows logical constraints that are structurally closer to the original program and specification, and high-level simplifications before being flattened in a basic logic. That is, an SMT solver that supports the theory of bit-vectors (see Section 2.1) performs high-level reasoning and simplifications (such as heuristic quantifier

instantiation and template-based model finding), then flattens the formula and analyzes it in a quantifier-free SMT logic, and uses bit-blasting only as a last resort. The high-level preprocessing improves the scalability of the solver significantly. The novelty of our encoding is to exploit the theory of bit-vectors of recent SMT solvers for gaining efficient bounded program verification. We use decidable theories of SMT logic (e.g., the theory of bit-vectors and the core theory) in our encoding so we can always obtain conclusive answers in bounded program verification. To our knowledge, this is the *first* attempt to use an SMT solver to overcome the efficiency issue in bounded program verification.

The process of translating a program and its specifications into an SMT formula is divided into two stages. First, a labeled control-flow graph, called *verification graph*, is constructed from the program and its specifications. Second, the SMT formula is extracted from the graph. Given a program p that is annotated by a **`requires`** clause req and an **`ensures`** clause ens, and a group of class and loop bounds, let pre and $post$ be the formulas encoding req and ens, respectively; f and ex be the formulas encoding normal and exceptional executions of p, respectively, we produce the Formula (3.1).

$$pre \wedge f \wedge (\neg post \vee ex) \tag{3.1}$$

If Formula (3.1) is satisfiable, its model represents a counterexample to the correctness of the program regarding the property of interest: a pre-state that satisfies req, but its post-state either violates ens or causes an exception. Otherwise, the correctness of the program p is guaranteed. Recall that in bounded program verification, the loops of the analyzed program are unrolled and the heap size is limited. The number of loop unrollings for a loop l is given by the loop bound. The maximal number of objects of a class c is given by the class bound. Therefore, we have obtained an initial confidence in the correctness of the program. To obtain more confidence we can increase the bounds for the bounded program verification. [1]

We aim to provide a lightweight bounded program verification approach, thus not all Java features are supported. We support a basic subset of Java that does not include strings, real numbers, and concurrency. We support a class hierarchy definition without interfaces and abstract classes. Figure 3.1 shows the grammar that is supported by our approach.

The rest of the chapter is organized as follows: Section 3.1 presents the construction of a verification graph and the following five sections (Section 3.2 to Section 3.6) present the extraction of an SMT formula from the verification graph; Section 3.7 shows the experiments to evaluate the benefits of using our approach in bounded program verification; Section 3.8 presents the related works and this chapter ends with conclusions in Section 3.9.

[1] By Daniel Jackson's *small-scope hypothesis* [Jackson, 2012], exhaustive checking within small bounds can achieve good code coverage and kill most of the mutants.

```
Prog ::= ClassDcl*

ClassDcl ::= class Class [extends Class]{FieldDcl* ProcDcl*}
FieldDcl ::= Access Type Field
ProcDcl  ::= Access (void | Type) Proc(ParDcl*){Stmt}
ParDcl   ::= Type Var

Stmt  ::= Var = Expr | Var[Expr] = Expr
        | Expr.Field = Expr | [var =] Proc(Expr*)
        | Var = new Class(Expr*) |new Class()[Expr]
        | if (Expr) Stmt [else Stmt]
        | while (Expr) Stmt | Return [Var]
        | continue | break | Stmt; Stmt

Expr  ::= Const | Var | Var[Expr] | Expr.Field
        | Expr instanceof Class | (Class) Expr
        | Expr BinOp Expr | !Expr

BinOp ::= + | - | * | / | >> | << | >>> | & | | | < | >
        | == | != | && | ||

Const ::= null | true |false | 0 | 1 | -1 |...
Type  ::= Class | boolean | int | Class[] | boolean[] | int[]
Access::= public | protected | private

Proc, Var, Class, Field ::= Identifier
```

Fig. 3.1: Abstract syntax of the supported Java programs, supposing that the expressions have no side-effect.

3.1 Construction of a Verification Graph

The construction of a verification graph—essentially a labeled control-flow graph—is divided into two stages. First, the analyzed program is transformed into a program with annotation statements constructed during the code transformations. Second, the verification graph is constructed from the transformed program.

3.1.1 Source Code Transformations

To facilitate code transformation we use the JML $assume$ and $assert$ annotation statements. These statements are enclosed in annotations and evaluated at the program points where they appear—textual points in the code. The JML $assume\ Q$; statement (Q is a Boolean expression) denotes that the predicate Q always holds during program verification. An $assert\ Q$; statement, on the other hand, denotes that the predicate Q is expected to be satisfied by the

analyzed program, thus it has to be checked in the program verification. If Q evaluates to $false$, the property specified by Q is violated, and the verification fails; otherwise the property holds.

We unroll loops with respect to the user-provided *loop bounds*. Consider a loop l with the loop condition `cond` has the loop bound x ($x > 0$). We unroll the loop x times and add a statement **`assume`** `!(cond);` to ensure the loop iterating at most x times. Similar to loop unrolling, a recursive method can be unrolled by inlining the method at the method invocation statements and by appending **`assume false`**`;` statements. Moreover, any object allocation statement is decomposed. That is, an allocation statement `x = new Class(`e_1`, .., `e_n`);` is broken into two consecutive statements `x = new Class; x.init(`e_1`, .., `e_n`);` for object allocation and initialization, respectively.

If called methods are annotated by method contracts, their associated method invocation statements are substituted by the method contracts. In particular, for a called method m that is annotated by a precondition pre and a postcondition $post$, we add an **`assert`** `pre;` statement before the method invocation statement, and an **`assume`** `post;` statement after the method invocation. Otherwise, the called method will be inlined at the method invocations. To facilitate the construction of the labeled control-flow graph, we replace the pre- and post-conditions of the analyzed method (i.e., the top-level method in the verification) by two annotation statements. We replace the precondition by adding an **`assume`** `pre;` statement immediately after initializing the method arguments, and replace the postcondition by adding an **`assume`** `!(post);` (i.e., negate the postcondition) after the `return` statement of the code. We handle loop invariants differently and elaborate on the details in Chapter 6.

Figure 3.2 shows the source code transformations performed on the `setAll` method in Fig. 3.2(a). The `setAll` method traverses, starting with the receiver, all entries of a singly linked list, and updates the `data` field of each entry to the value of the `d` argument. The **`ensures`** clause (Fig. 3.2(a) Line 5) denotes that all entries of the list contain a reference to `d` after `setAll` returns. Figure 3.2(b) shows the transformed code with annotation statements.

3.1.2 Verification Graph

We construct a verification graph from the transformed program (using the source code transformations presented in Section 3.1.1). A verification graph is essentially a labeled control-flow graph. If it is constructed from a program without loops, it is acyclic, otherwise cyclic. It has a single entry node and a single exit node. A path through the graph from entry to exit represents an execution of the program. Passing through a node in the path represents the control reaches the program point that the node denotes; traversing an edge represents either evaluation of the predicate of a conditional or annotation statement, or the execution of a statement such as an assignment, return, or an object allocation. Definition 3.1 presents the properties of a verification graph.

```
1  class Entry {
2  /*@ nullable @*/ Data data;
3  /*@ nullable @*/ Entry next;
4
5  /*@ ensures(\forall Entry o;
6    @ \reach(this, o, next);
7    @ o.data == d);
8    @*/
9    void setAll(Data d){
10     Entry e = this;
11     while (e != null){
12       e.data = d;
13       e = e.next;
14     }
15   }
16 }
17 class Data{}
```

(a) Original code

```
1  class Entry {
2  /*@ nullable @*/Data data;
3  /*@ nullable @*/Entry next;
4
5    void setAll(Data d){
6      Entry e = this;
7      if (e != null){
8        e.data = d;
9        e = e.next;
10       //@ assume e == null;
11     }
12   /*@ assume !(\forall Entry o;
13     @ \reach(this, o, next);
14     @ o.data == d);
15     @*/
16   }
17 }
```

(b) Transformed code

Fig. 3.2: Source code transformations. Given a code in Fig. 3.2(a) and loop bound 1, we unroll the loop once and add an **assume** statement at Line 10 in the transformed code in Fig. 3.2(b) to ensure the loop to iterate at most one time. The JML **ensures** clause in the original code is replaced by the JML **assume** statement at Line 12 in the transformed code. For simplicity, the transformed code has no **assume true;** since the precondition is trivial $true$.

Definition 3.1. *A verification graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite set of graph nodes $\mathcal{V}$ with a set of graph edges $\mathcal{E}$ of 2-subsets of $\mathcal{V}$. It has the following properties:*

- *$\mathcal{G}$ is a directed graph;*
- *$\mathcal{G}$ is node-labeled, i.e., nodes are labeled by a sequence of numbers;*
- *$\mathcal{G}$ has a single entry node v_{entry} and a single exit node v_{exit}, and any node $v \in \mathcal{V}$ is on a path from node v_{entry} to node v_{exit}.*

Supposing $\mathcal{G}$ is constructed from a transformed program P without loops, $\mathcal{G}$ has the relations to P:

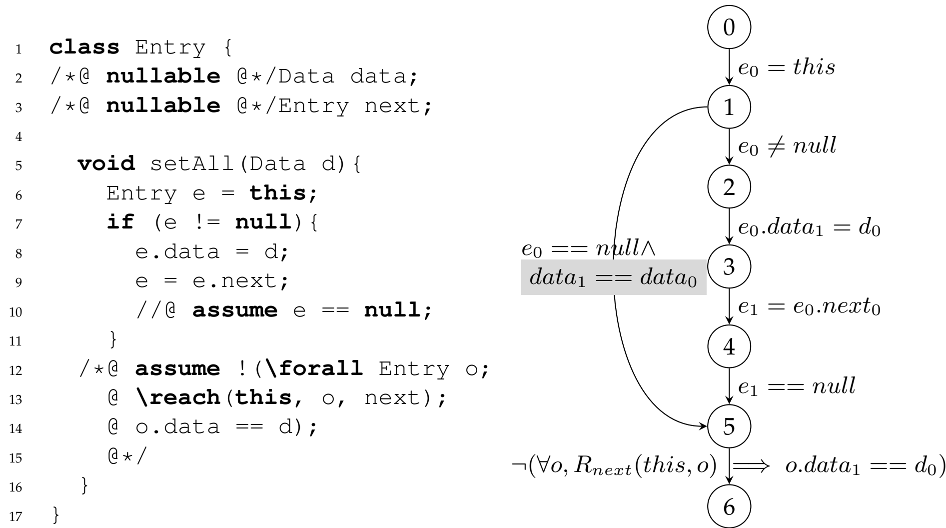
- *A node $v \in \mathcal{V}$ denotes a program point (i.e., textual point in the code) of P;*
- *An edge $e \in \mathcal{E}$ denotes either a predicate test (i.e., to evaluate the predicate of a branch or annotation statement) or an elementary statement (e.g., an assignment statement) of P.*

Figure 3.3 shows the construction of a verification graph from the transformed code in Fig. 3.3(a). Figure 3.3(b) presents the constructed verification graph. The numbers in the circles label the nodes. Edges are annotated by the

branch conditions (or predicates of annotation statements) or by elementary statements.

We rename variables (and fields) so that they are assigned at most once along each path of the graph. The initial variables (and fields) are named using the index 0, and the index increases when the variables or fields are updated. Algorithm 1 presents the basic algorithm for renaming variables and fields in an acyclic verification graph.

The process of renaming may introduce unspecified variables at the merging nodes. A merging node has multiple incoming edges in a verification graph. For example, the node 5 in Fig. 3.3(b) is a merging node that has the incoming edges $E_{1,5}$ and $E_{4,5}$. The field $data$ is renamed to $data_0$ and $data_1$ on the paths $[0, 1, 5]$ and $[0, 1, 2, 3, 4, 5]$, respectively. Hence, $data_1$ is unspecified on the path $[0, 1, 5, 6]$. We add constraints (the highlighted expression) to ensure that $data_1$ equals to $data_0$ when edge $E_{1,5}$ is traversed. Though the variable e is another unspecified variable on node 5, no additional constraint is needed since e is not used on the path following node 5. The edge $E_{5,6}$ represents the negation of the postcondition, thus any path of this graph represents the execution of the program that violates the postcondition.



(a) The transformed code (b) A verification graph

Fig. 3.3: Construction of a verification graph. A verification graph of the transformed code in Fig. 3.3(a) is shown in Fig. 3.3(b). The code in Fig. 3.3(a) is identical to the one in Fig. 3.2(b). We replicate it for the purpose of visualization.

Algorithm 1 Renaming variables (and fields) in an acyclic verification graph.

```
1: G(V, E) := A verification graph with nodes V and edges E.

2: function TRAVERSE(e ∈ E)
3:     for p ∈ the incoming edges of the source node of e do
4:         if p.isTraversed == false then
5:             TRAVERSE(p)
6:         end if
7:     end for

8:     identifiers := Variable ↦ Identifier
9:     identifiers ← the identifiers of variables immediately before the statement of e

10:    for v ∈ the variables used by the statement of e do
11:        rename v to identifiers.get(v)
12:    end for
13:    for v ∈ the variables defined by the statement of e do
14:        rename v to ← identifiers.get(v) with suffix + 1;
15:    end for
16:    e.isTraversed = true
17: end function
```

Compared to a global-state encoding, the verification graph represents a program state implicitly, as a collection of independent variables and fields. That is, each update to a variable (or a field) triggers renaming that variable (or field) only, without causing the whole global state of the program to be renamed. Furthermore, using verification graphs allows one to encode the control- and data-flow constraints separately, which prevents deeply-nested formulas and helps produce more readable counterexamples.

The verification graph is a modified version of the *computation graph* [Vaziri-Farahani, 2004]. It inherits the benefits of the computation graph such as implicit program state encoding. In contrast to the computation graph that only represents the control- and data-flow of the program, the verification graph also represents the specifications, e.g., method contracts and loop invariants.

3.2 Encoding Types

We encode Java types using the bit-vectors theory (see Section 2.1). Handling the complete Java's type system requires encoding `Object` class. If the analyzed program and its specifications do not use `Object` class or the member methods of `Object` class, a formula encoding `Object` class will impose an unnecessary overhead on the SMT solver. Therefore, we do not encode the `Object` class unless the analyzed code or specifications explicitly use the `Object` class or the member methods of `Object` class.

We treat a Java class whose declaration does not contain the Java `extends` or `implements` keywords as a top-level class. A top-level class `A` is encoded using a bit-vector sort S_A in the form of `(_ BitVec m)`, where $m = \lceil log(n+1) \rceil$ and n is the class bound of `A`. The instances of the sort S_A can be represented using their integer values as follows:

$$null_A,\ \underbrace{1, 2, \ldots, last_A}_{\texttt{valid range of A}}, \ldots, n, \ldots, 2^m - 1.$$

We use an SMT constant $null_A$ to denote the `null` value of class `A` and an SMT variable $last_A$ to denote the most recently allocated object of `A`.[2] All bit-vectors whose integer values belong to `A`'s *valid range* of $[1, .., last_A]$ represent the allocated objects of `A`. For example, we constrain the receiver object to be allocated using the following SMT assertion, in which `bvule` is the *unsigned less than* operation for bit-vectors. The SMT `(assert expr)` command constrains that the Boolean expression `expr` evaluates to $true$ in the SMT solving.

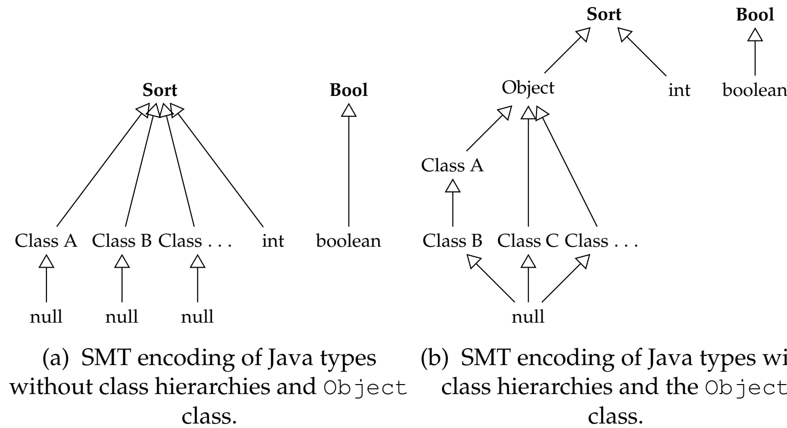
```
(assert (and (not (= this null_A))(bvule this last_A)))
```

Figure 3.4(a) illustrates the SMT encoding of Java types when the analyzed program does not involve class hierarchies and `Object` class. Instead of encoding the Java constant `null` as a single value that is compatible with all classes, our encoding ensures that each top-level class has a distinct SMT `null` value.

We encode a superclass and its subclasses using a single SMT sort and constrain the subtyping relation in the encoding. Consider a class hierarchy where classes `A` and `B` extend class `C`. Given class bounds m, n, and w (all instances of the subclasses are instances of the superclass, thus $m + n \leq w$) for `A`, `B`, and `C`, respectively, we encode all three classes using the bit-vector sort S_C of size $c = \lceil log(w+1) \rceil$. This allows us to treat instances of a subclass as an instance of its superclass. All three classes share one `null` value. An SMT variable $last$ of is used to represent the last allocated object of each type. Instances of subclasses are represented by the values of non-overlapping sub-ranges. In our example, the allocated objects of types `A`, `B`, and `C` are given by the sub-ranges $[1, .., last_A]$, $[m+1, .., last_B]$, and $[m+n+1, .., last_C]$, respectively, where $last$ constants are defined in the following SMT formula:

```
(declare-fun last_A () (_ BitVec c))
(declare-fun last_B () (_ BitVec c))
(declare-fun last_C () (_ BitVec c))
(assert (bvule last_A m))
```

[2] Using the identifiers of SMT variables directly in the text impedes the readability, since SMT variables may have long names. For ease of convenience, we use, for example, $null_A$ in the text and `null_A` in the SMT formulas. Both $null_A$ and `null_A` denotes the same variable. We keep using this naming method in the rest of the thesis.



(a) SMT encoding of Java types without class hierarchies and `Object` class.

(b) SMT encoding of Java types with class hierarchies and the `Object` class.

Fig. 3.4: Two kinds of SMT encoding of Java types. The instances (i.e., bit-vectors with fixed-size) of the built-in meta-sort **Sort** encode Java types and the SMT built-in sort **Bool** encodes Java `boolean` type. Each class in Fig. 3.4(a) has a distinct null value.

```
(assert (or (and (bvule last_B (bvadd m n))
                 (bvugt last_B m))
            (= idxB (_ bv0 c))))
(assert (or (and (bvule last_C w)
                 (bvugt last_C (bvadd m n)))
            (= idxC (_ bv0 c))))
```

The valid ranges of `A` and `B` are $[1, .., last_A]$ and $[1, m+1, .., last_B]$, respectively. The valid range of `C` includes also the valid ranges for `A` and `B`, and is defined as $[1, .., last_A, m+1, .., last_B, m+n+1, .., last_C]$.

The Java expression `(o instanceof T)` evaluates to $true$ if `o` is not `null`, and is in the valid range of `T`. Typecasting an object `obj` to a class T is allowed if `obj == null` or `(obj instanceof T)` holds. Overridden methods and fields are resolved via a sequence of nested tests on the actual object type using Java `instanceof` keyword.

Similar to the encoding of reference types, we encode primitive types also using bit-vectors. That is, we encode Java `int` as a bit-vector. The size of the bit-vector is the user-provided class bound of `int`. For simplicity, we encode Java `boolean` type as the SMT built-in **`Bool`** sort. The integer operations are modeled using the SMT bit-vector operations.

3.3 Encoding Control-flow

We encode the control-flow of the analyzed program as an SMT formula that captures the partial order of the edges in the verification graph traversal.

Definition 3.2 provides the encoding of control-flow based on the verification graph. It constrains that (i) if a control-flow formula is satisfied, at least one of the entry edges has to be traversed, (ii) if one of the incoming edges of node j is traversed, at least one of j's outgoing edges will be traversed, and (iii) the control-flow formula is a conjunction of the implications. The control-flow formula alone does not prevent unfeasible paths (e.g., both outgoing edges of a branch node can be taken). We encode the data-flow of the verification graph to guarantee that exactly one of the outgoing edges is traversed.

Definition 3.2. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a verification graph, $E_{i,j}$ $(i, j \in \mathcal{V}, E_{i,j} \in \mathcal{E})$ be an edge from node i to node j, $a \in \mathcal{V}$ be an entry node, thus the control-flow is encoded as a formula:*

$$\left(\bigvee_{\substack{a,b \in \mathcal{V}, \\ a \neq b}} E_{a,b} \right) \wedge \bigwedge_{\substack{i,j \in \mathcal{V}, \\ i \neq j}} \left(E_{i,j} \implies \bigvee_{\substack{j,k \in \mathcal{V}, \\ j \neq k}} E_{j,k} \right)$$

Figure 3.5 presents the SMT formula encoding the control-flow of the verification graph shown in Fig. 3.3(b). We use an SMT Boolean variable `E_i_j` to represent the edge from node `i` to node `j`. The data-flow formula is presented in Fig. 3.7.

```
(assert E_0_1)
(assert (=> E_0_1 (or E_1_2 E_1_5)))
(assert (=> E_1_2 E_2_3))
(assert (=> E_2_3 E_3_4))
(assert (=> E_3_4 E_4_5))
(assert (=> E_4_5 E_5_6))
(assert (=> E_1_5 E_5_6))
```

Fig. 3.5: Control-flow formulas for the verification graph in Fig. 3.3(b).

3.4 Encoding Data-flow

We encode the data-flow of a verification graph as a formula to capture program behaviors. That is, if an edge of a verification graph is traversed, its associated state transitions are performed or branch conditions are evaluated. Definition 3.3 presents the data-flow encoding of a verification graph.

Definition 3.3. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a verification graph, $E_{i,j}$ $(i, j \in \mathcal{V}, E_{i,j} \in \mathcal{E})$ be an edge from node i to node j, then the data-flow is encoded as a formula:*

$$\bigwedge_{\substack{i,j \in \mathcal{V}, \\ i \neq j}} (E_{i,j} \implies \mathcal{T}(stmt, i, j))$$

where $stmt$ denotes a program statement associated with edge $E_{i,j}$ and $\mathcal{T}$ is a translation function for $stmt$.

The translation function $\mathcal{T}(stmt, i, j)$ translates a program statement $stmt$ into an SMT formula, where i and j are the source and target nodes of the directed edge $E_{i,j}$, respectively. In the rest of this chapter we also use i and j to denote the program states at nodes i and j, respectively, for the ease of convenience. In a verification graph any program variable (and class field) is assigned only once on a single path. If a variable `v` is updated during the state transition from state i to state j, we denote `v` in the states i and j as $\mathtt{v}_i$ and $\mathtt{v}_j$, respectively. Figure 3.6 presents the translation rules for the translation function $\mathcal{T}(stmt, i, j)$. In particular, translation rules R_1 – R_7 are used for the encoding of program statements, R_8 – R_{18} for expressions, and R_{19} – R_{22} for variables and fields. The translation rules for the expressions such as `(o instanceof T)`, $\mathtt{o}_1$ `= (T)`$\mathtt{o}_2$, and `reach(o$_1$, o$_2$, f)` will be explained later in the chapter.

Fields

Fields are translated into uninterpreted functions over bit-vectors. A field `f` of type `B` declared in a class `A` is encoded as follows:

```
(declare-fun f ((_ BitVec m)) (_ BitVec n)),
```

where `(_ BitVec m)` and `(_ BitVec n)` denote the SMT sorts for `A` and `B`, respectively, and `m` and `n` are the length of two sorts.

To ensure that the configurations of the objects on the heap are valid, the encoding of fields has to fulfill the following properties:

- If `B` is a reference type, then for any object `o` of class `A` in the pre-state of the analyzed program, `o.f` refers to `null` or to an object of `B`;
- If field `f` of an object `o` of class `A` is updated, `f` fields of the other objects of `A` remain unchanged.

We encode the first property using an SMT assertion as follows:

```
(assert (forall (o (_ BitVec m)) (=>
                (and (not (= o null_A)) (bvule o last_A))
                (bvule (f o) last_B))))
```

Both `last_A` in SMT formula and $last_A$ in the text denote the same variable. As stated in Section 3.2, we use this kind of naming method in the text to avoid terms with long names.

We encode the second property by maintaining the mappings from the domain to the range of SMT uninterpreted functions. Each update to a field requires a new function to represent the result. R_3 is the translation rule for assigning a field. It denotes that, at state `j`, field `f` replicates its values at state `i` except for the object `o`. Read accesses to the fields are encoded as computing images of the uninterpreted functions. R_2 is the translation rule for using a field.

$\mathbf{R}_1$: $\mathcal{T}(\texttt{v = e;}, i, j)$	:=	`(=` $\mathcal{E}(\texttt{v}, j)$ $\mathcal{E}(\texttt{e}, i)$`)`
$\mathbf{R}_2$: $\mathcal{T}(\texttt{v = o.f;}, i, j)$	:=	`(=` $\mathcal{E}(\texttt{v}, j)$ `(`$\mathcal{E}(\texttt{f}, i)$ $\mathcal{E}(\texttt{o}, i)$`))`

$\mathbf{R}_3$: $\mathcal{T}(\texttt{o.f = e;}, i, j)$	:=	`(forall ((x T(o))) (ite (= x o)`
		`(= (`$\mathcal{E}(\texttt{f}, j)$ `x)` $\mathcal{E}(\texttt{e}, i)$`)`
		`(= (`$\mathcal{E}(\texttt{f}, j)$ `x) (`$\mathcal{E}(\texttt{f}, i)$ `x)))`

$\mathbf{R}_4$: $\mathcal{T}(\texttt{v = a[k];}, i, j)$	:=	`(=` $\mathcal{E}(\texttt{v}, j)$ `(`$\mathcal{E}(\texttt{h}, i)$ $\mathcal{E}(\texttt{a}, i)$ $\mathcal{E}(\texttt{k}, i)$`))`

$\mathbf{R}_5$: $\mathcal{T}(\texttt{a[k] = e;}, i, j)$	:=	`(forall ((x int)) (ite (= x k)`
		`(= (`$\mathcal{E}(\texttt{f}, j)$ $\mathcal{E}(\texttt{a}, i)$ `x)` $\mathcal{E}(\texttt{e}, i)$`)`
		`(= (`$\mathcal{E}(\texttt{f}, j)$ `a x) (`$\mathcal{E}(\texttt{f}, i)$ `a x)))`

$\mathbf{R}_6$: $\mathcal{T}(\texttt{T o = new T;}, i, j)$	:=	`(and (bvult` $\mathcal{E}(last_T, i)$ `(_ bv`s `m))` `(=` $\mathcal{E}(\texttt{o}, j)$ `(bvadd` $\mathcal{E}(last_T, i)$ `(_ bv1 m)))`

$\mathbf{R}_7$: $\mathcal{T}(\texttt{T[] a = new T[k];}, i, j)$	:=	`(and (bvult` $\mathcal{E}(last_{T[]}, i)$ `(_ bv`s `m))`
		`(=` $\mathcal{E}(\texttt{a}, j)$ `(bvadd` $\mathcal{E}(last_{T[]}, i)$ `(_ bv1 m)))`
		`(forall ((x int)) (=> (and`
		`(bvsge x(_ bv0 m))(bvslt x(_ bv`k `m)))`
		`(= (f a x)` $null_T$`))))`

$\mathbf{R}_8$: $\mathcal{E}(\texttt{e}_1\ \texttt{==}\ \texttt{e}_2, i)$	:=	`(=` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_9$: $\mathcal{E}(\texttt{e}_1\ \texttt{!=}\ \texttt{e}_2, i)$	:=	`(not (=` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`))`
$\mathbf{R}_{10}$: $\mathcal{E}(\texttt{e}_1 > \texttt{e}_2, i)$	:=	`(bvsgt` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{11}$: $\mathcal{E}(\texttt{e}_1 < \texttt{e}_2, i)$	:=	`(bvslt` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{12}$: $\mathcal{E}(\texttt{e}_1 \texttt{>=} \texttt{e}_2, i)$	:=	`(bvsge` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{13}$: $\mathcal{E}(\texttt{e}_1 \texttt{<=} \texttt{e}_2, i)$	:=	`(bvsle` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{14}$: $\mathcal{E}(\texttt{e}_1 + \texttt{e}_2, i)$	:=	`(bvadd` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{15}$: $\mathcal{E}(\texttt{e}_1 - \texttt{e}_2, i)$	:=	`(bvsub` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{16}$: $\mathcal{E}(\texttt{e}_1 \times \texttt{e}_2, i)$	:=	`(bvmul` $\mathcal{E}(\texttt{e}_1, i)$ $\mathcal{E}(\texttt{e}_2, i)$`)`
$\mathbf{R}_{17}$: $\mathcal{E}(\texttt{e.f}, i)$	:=	`(`$\mathcal{E}(\texttt{f}, i)$ $\mathcal{E}(\texttt{e}, i)$`)`
$\mathbf{R}_{18}$: $\mathcal{E}(\texttt{a[e]}, i)$	:=	`(`$\mathcal{E}(\texttt{f}, i)$ $\mathcal{E}(\texttt{a}, i)$`)` $\mathcal{E}(\texttt{e}, i)$`)`
$\mathbf{R}_{19}$: $\mathcal{E}(\texttt{o}, i)$	:=	$\texttt{o}_i$
$\mathbf{R}_{20}$: $\mathcal{E}(\texttt{a}, i)$	:=	$\texttt{a}_i$
$\mathbf{R}_{21}$: $\mathcal{E}(\texttt{v}, i)$	:=	$\texttt{v}_i$
$\mathbf{R}_{22}$: $\mathcal{E}(\texttt{f}, i)$	:=	$\texttt{f}_i$

Fig. 3.6: Translation rules for the translation from program statements to a data-flow formula. The translation function $\mathcal{T}$ is used for program statements, and $\mathcal{E}$ for expressions. The i and j in $\mathcal{T}(\texttt{stmt}, i, j)$ denote, respectively, the program states before and after the execution of the statement $stmt$. A variable v at state i is denoted as v_i. o and a correspondingly represent an object and an array. The SMT function `T(o)` returns the SMT sort of `o`. The SMT variable $last_T$ (or $last_{T[]}$) denotes the most recent allocated instance of `T` (or `T[]`) immediately before the execution of object (array) allocation statement. Furthermore, f stands for a member field or the array accessor. Note that these rules are based on the assumption that the expressions have no side-effect, e.g., they do not raise an exception. The exception handling is explained in Section 3.5.

Arrays

An array refers to its elements using the array accessor field. We encode the array accessor field using an uninterpreted function with two arguments. An array accessor field is declared as follows:

$$\texttt{(}\textbf{declare-fun}\ \texttt{RefA (S}_{A[]}\ \texttt{S}_{int}\texttt{) S}_{A}\texttt{),}$$

where $\texttt{S}_A$, $\texttt{S}_{int}$, and $\texttt{S}_{A[]}$ are the corresponding SMT sorts for class `A`, `int`, and array type `A[]`. Though it is more convenient to encode the array accessor field using the theory of arrays, our encoding using uninterpreted functions is still suitable for the SMT solvers that do not support the theory of arrays. R_4 and R_5 are the translation rules for program statements that read and write arrays, respectively. Using R_5, an assignment statement `a[i] = e`, for example, is translated into an SMT assertion as follows:

```
(assert (forall ((x int)) (ite (= x i)
                               (= (RefA_1 a x) e)
                               (= (RefA_1 a x) (RefA a x)))))
```

This assertion ensures that $RefA_1$ is a copy $RefA$ in which the indexes a and i are mapped to the element e they have identical elements on the rest indexes.

Allocation

Recall that a bit-vector $last_A$ denotes the most recent allocated object of class `A`. It also denotes the number of allocated objects of `A`. For the allocation statement, we use a new bit-vector with the same sort of $last_A$ to denote the object to be allocated. The integer value of the new bit-vector is larger than the value of $last_A$, while it does not exceed the bound of class `A`. Encoding an allocation statement `A a = new A`, for example, results a formula as follows:

$$last_{A,k} < B_A \wedge last_{A,k+1} = last_{A,k} + 1.$$

$last_{A,k}$ and $last_{A,k+1}$ respectively represent the number of allocated `A` objects before and after the allocation, and B_A represents the class bound of `A`. R_6 is the translation rule for object allocation statement. It guarantees that the expression $last_{A,k+1}$ does not overflow and that the number of allocated objects ($last_{A,k+1}$) does not exceed the class bound of A.

The array length is initialized upon the array allocation and remains unchanged. We encode the array length using an uninterpreted function mapping from array to integers. To encode the array allocation statement `A[] a = new A[length]`, we declare and initialize the length of the array as follows:

```
(declare-fun lengthA[] (S_A[]) S_int)
(assert (= (lengthA[] a) length))
```

S_A[] and S_int are the SMT sorts for the type A[] and int, respectively. When an array is allocated, it is filled up by the default elements. In particular, an integer array will be filled up with zeros, an Boolean array by false, and an reference-type array by null. R_7 is the translation rule for array allocation.

Figure 3.7 shows the SMT formula encoding the data-flow of the verification graph in Fig. 3.3(b). The SMT (reach_next this o) function encodes the reachability relation. It evaluates to $true$ if the object o is in the reflexive transitive closure of the receiver object (this) over the next relation (field). Details of this function are to be explained in Section 3.6.

```
(assert (=> E_0_1 (= e_0 this)))
(assert (=> E_1_2 (not (= e_0 null_Entry))))
(assert (=> E_2_3 (forall ((o Entry)) (ite (= o e_0)
             (= (data_1 o) d_0) (= (data_1 o) (data_0 o))))))
(assert (=> E_3_4 (= e_1 (next_0 e_0))))
(assert (=> E_4_5 (= e_1 null_Entry)))
(assert (=> E_1_5 (= e_0 null_Entry)))
(assert (=> E_1_5(forall ((o Entry))(=(data_1 o)(data_0 o)))))
(assert (=> E_5_6 (not (forall ((o Entry))
              (=> (reach_next this o) (= (data_1 o) d_0))))))
```

Fig. 3.7: Data-flow formulas of the verification graph in Fig. 3.3(b).

3.5 Handling Runtime Exception

We check two kinds of runtime exceptions: null-pointer dereference and array index out of bounds. We use a global Boolean variable exc that is initialized to false and updated at any location of the code when an exception may be thrown, i.e., at the statements that read/write fields or arrays. That is, when an exception is thrown, exc is replicated and the new exc evaluates to true, and false otherwise. When a program terminates normally, i.e., no exception has been thrown, all copies (but with distinct names) of exc have to be false.[3] In particular, to check whether an expression o.f raises a null-pointer dereference exception, we add a condition $o \neq null$ before evaluating the expression. Handling exceptions of an array access expression a[k], for example, requires two conditions: $a \neq null$ and $0 <= k \wedge k < a.length$. Figure 3.8 shows the formula to handle the null-pointer exception. The first four assertions encode the control- and data-flow, and the last assertion will be satisfied when an exception has been thrown. To reduce the overhead of SMT solvers, we statically resolve the exception conditions by propagating constants along the paths in the verification graph. In Fig. 3.3(b), e_0 is assigned

[3] An constraint specifies that at most one of the exc variables evaluates to true.

by the receiver object at node 1. Therefore we do not need to check exceptions on the nodes 2 and 3.

```
(assert (=> E_2_3 (or E_3_4 exc_0)))
(assert (=> E_2_3 (= exc_0 (= e_0 null_Entry))))
(assert (=> E_3_4 (or E_4_5 exc_1)))
(assert (=> E_3_4 (= exc_1 (= e_0 null_Entry))))
(assert (=> E_5_6 (not (and (not exc_0) (not exc_1)))))
```

Fig. 3.8: Control- and data-flow formula for handing runtime exceptions. The formula is translated from the verification graph in Fig. 3.3(b).

3.6 Encoding JML Expressions

Many JML constructs, e.g., logical operators, arithmetic operators, and conditional operators, use Java semantics thus encoding them using bit-vectors is straightforward.

We encode the JML universal **`\forall`** and existential **`\exists`** quantifiers using SMT `forall` and `exists` quantifiers, respectively. The range of values for a quantified variable of a reference type may include references to the objects that are not constructed by the analyzed program [see Leavens et al., 2006, Chapter 12 page 113]. Those objects that are created in the annotations, e.g., `Object o = new Object();`, are in the range. Actually, an object created in the annotations does not exist on the program's heap. Recall that out encoding is based on the the assumption that the Java and JML expressions have no side-effect. In our encoding, the JML quantifier range includes only the objects on the heap that the program manipulates. Let $\texttt{S}_A$ be the SMT sort of class `A` and $\texttt{last}_A$ be the number of T objects on the heap, $\mathcal{E}$ be the translation function for JML expressions, the JML expression `(\forall A o; R(o); Q(o))` is translated into a formula as follows:

$$\texttt{(forall (o S}_A\texttt{) (=> (and (not (= o null)) (bvule o last}_A\texttt{)) (=> } \mathcal{E}\texttt{(R(o)) } \mathcal{E}\texttt{(Q(o)))))}.$$

Reachability

The JML reachability construct `\reach(Object x, Field f)` only supports one field (see Section 2.2). To express properties of complex data structures, e.g., a tree with the fields `left` and `right`, we introduced into JML a ternary Boolean-valued function `\reach(Object x, Object y, FieldSet fs)` that returns **`true`** when `y` is in the reflexive transitive closure of `x` over any field in the union `fs` of fields.

Transitive closure (the reachability construct) over arbitrary domains can not be axiomatized in pure first-order logic [Lev-Ami et al., 2009]. For finite domains, however, an axiomatization has been given by Claessen [Claessen, 2008]. Inspired by his approach, we introduce the following axioms to compute the transitive closure of a (homogeneous) field `f` of type `A` declared in a class `A`. For a more concise syntax, we pretend that `f` is a function of type $A \rightarrow A$.

```
(assert (forall ((x A) (y A)) (= (= (f x) y) (= (P x y) 1))))
(assert (forall ((x A) (y A) (z A)) (=> (and (> (P x y) 0)
                              (> (P y z) 0)) (> (P x z) 0))))
(assert (forall ((x A) (y A)) (=> (> (P x y) 1)
 (exists (w A) (and (= (P x w)1) (= (P x y)(+ 1 (P w y))))))))
```

The auxiliary function $P : A \times A \rightarrow int$ is defined to represent the smallest number of steps required to reach from one object to another (via f). Therefore, **`\reach`**`(x, y, f)` evaluates to $true$ iff $x = y$ or $(P\ x\ y) > 0$. The first constraint sets $P(x, y)$ to 1 if and only if $x.f = y$. The second constraint ensures transitivity, and the third constraint defines a partial order over all the objects reachable from a single source. The third constraint is crucial for soundness when f is cyclic. Converting the axioms to use uninterpreted functions and bit-vectors is straightforward. It should be noted that the maximum value of P is the number of objects of A. Therefore, in converting to bit-vectors, the int type used in the declaration of P above can use the same number of bits as needed to declare A. Furthermore, the addition operator must be constrained not to overflow.

3.7 Evaluation

We have implemented our technique in the prototype tool *InspectJ* that uses the Jimple, 3-address intermediate representation of Java Bytecode provided by the Soot optimization framework [Lam et al., 2011], to preprocess Java Bytecode, the Common JML Tools package (ISU) [Leavens et al., 2006] to parse JML specifications, and Z3 [de Moura and Bjørner, 2008] as the underlying SMT solver. Furthermore, we have compared InspectJ against JForge [Dennis et al., 2006], a well-known SAT-based bounded verification tool that can handle Java programs with complex data structures and has been used in several other projects (e.g., [Shao et al., 2009, 2010; Galeotti et al., 2013]). All the experiments are performed on an Intel Xeon 2.53 GHz with 16 GB of RAM using Linux 64bit. We used SAT4J 2.3.0 and Z3 3.2.

Object of Analysis

We have checked a large-scale implementation of Dijkstra's shortest path algorithm that forms the basis of several optimized routing algorithms in graphs

with millions of nodes. This algorithm computes single-source shortest paths in graphs with non-negative edge weights. The optimized version [Delling, 2009] that we target makes heavy use of a priority queue backed by a binary heap. It is is written in C++ and has been manually ported to Java code using mostly syntactic conversions. The target code [Nagal, 2013] consists of 7 classes with a total of 37 methods and 346 Java source lines, excluding whitespace and specifications. To our knowledge, all previous verification of Dijkstra algorithm were performed on either a very abstract, or a very basic implementation [Mange and Kuhn, 2007; Klasen et al., 2010]. Our target code optimizes the memory layout and cache effects through sophisticated interconnections of data structures. The specifications of the analyzed methods consist of 27 lines and mostly constrain the internal integrity of the binary heap data structure. Properties include, for example, the *min-heap property* that the value of each node is greater than or equal to the value of its parent, thus the minimum-value element is the root. JML specifications are converted to JFSL [Yessenov, 2009] that JForge expects. This conversion required only simple syntactic changes. Apart from different specification languages, both JForge and InspectJ operate on the same inputs. We set both tools to inline called methods in all the experiments, and increased the bounds until both tools timed out.

Results of Bug Detection

We have checked 10 out of a total of 19 `public` methods. The remaining 9 methods are not expected to fulfill the desired properties and they are called by the analyzed methods. Both InspectJ and JForge revealed three previously-unknown bugs in the Java implementation of the binary heap data structure, two of which represented the same problem in two different methods. All bugs required small bounds (maximum 3 for class bounds and loop bounds).

The first bug (repeated twice) was introduced when the C++ code was ported to Java. Assigning a `struct` in C++ copies all of its fields by value, while assigning an object in Java only copies it by reference. The bug and its fix are shown in Listing 1. The JML specification that caught the bug is also given. It constrains the keys of the `elems` array to match the keys of the `heap` array. The code at lines 5–6 modifies the single object `heap[index1]` (`key` and `val` are integer values) due to Java copy-by-reference. However, the original C++ code [Delling, 2009] modifies a fresh copy of `heap[index1]`. This unintended modification to `heap` causes inconsistencies between `heap` and `elems` which causes the specification to fail.

The second bug was present in the original C++ code that was already heavily tested. It involved a memory location that was freed, but under certain conditions was referenced again. This results in undefined behavior in C++ but went undetected as it usually worked. InspectJ detected this as a null pointer exception since the Java code marks freed locations by `null`. The bug and its fix are shown in Listing 2. Line 2 removes the last element of

```
1  /*@ ensures (\forall int i; 0 <= i && i < heap.length ==>
2    @ elems[heap[i].value].key == heap[i].key)
3    @*/
4  // VERSION WITH BUG
5  heap[index2] = heap[index1];
6  heap[index2].key = k;
7
8  // VERSION WITHOUT BUG
9  heap[index2].key = heap[index1].key;
10 heap[index2].value = heap[index1].value;
11 heap[index2].key = k;
```

Listing 1: Invalid Copy Semantics

the heap, but Line 3 accesses the first element without checking whether the heap has become empty (`heap[0]` is a dummy element). Swapping Lines 2 and 3 produces the intended behavior and fixes the bug in this case. We have already reported the problem, and it has been fixed by the original developers.

```
1  // VERSION WITH BUG
2  dropHeap();
3  x = heap[1];
4  ....
5
6  // VERSION WITHOUT BUG
7  x = heap[1];
8  dropHeap();
9  ....
```

Listing 2: Invalid Memory Access

Runtime Evaluation

We compared the time cost of InspectJ with JForge. All the bugs previously described were fixed prior to this comparison. The evaluation results are given in Table 3.1. The *Bits*, *Objs*, and *Loop* columns contain the bounds on the size of integer, objects of each class, and loop iterations, respectively. The time cost of each tool is given in seconds and is split into the time spent in the encoding phase (denoted by *Encode*), in which the code and its specifications are translated into SAT or SMT formulas, and in the solving phase, which is performed by the underlying solver. By default, JForge uses an old version of SAT4J which we replaced with its most recent version 2.3.0. Furthermore, for a fair comparison, we also used Z3 (in SAT solving mode) as the back-end

solver for JForge. The *Total* column gives the sum of the encoding time and the best solving time. Any runtime beyond our threshold of 600 seconds is denoted by *TO* (timeout).

The methods `insert`, `decreaseKey`, and `deleteMin` provide the main functionality of the binary heap. They are used to insert, update, and delete heap elements, respectively. The method `minElement` returns the root element of the heap. Verifying these methods returns `UNSAT`, meaning that the solver cannot find a counterexample, and thus the specifications hold for the given bounds. We also experimented with some satisfiable cases by underspecifying the `run` method, against some specifications denoting partial properties (and the built-in exception checking which is present in both tools). The `run` method is the functional entry point of the Dijkstra code. Calling this method typically involves calling all the above methods multiple times.

As shown in Table 3.1, when checking some methods with respect to very small bounds (see first rows for `deleteMin`, `insert`, and `minElement`), InspectJ is slower than JForge. This is caused by the slow startup of the Soot and ISU libraries used in InspectJ encoding phase, which becomes a bottleneck if total runtime is low. However, in all other cases InspectJ is significantly faster, and is capable of checking bounds that JForge cannot.

An interesting case is `minElement` for which the runtime of InspectJ is independent of the bounds. This is because InspectJ only makes a few passes over the input program and specification to generate the SMT formulas and the SMT solver can deduce unsatisfiability of the formula using high-level simplifications. Increasing the bounds beyond 10 in this case causes an internal error in JForge due to the big sizes of the formulas. JForge encoding generates Boolean formulas through bit-blasting. It incorporates low-level optimization such as symmetry breaking and sharing detection, and thus depends on the analyzed bounds.

During the experiments, we noticed that InspectJ covers some paths in the code that JForge does not. For example, `deleteMin` removes the root of the heap and restructures the resulting tree to remain balanced. Covering all distinct cases requires at least five tree nodes, but JForge times out in that scope and thus cannot analyze certain paths of the code.

Our tool scales much better it takes less time to create a more compact formula. As a result, one can increase the scope far beyond what is possible with JForge. In certain cases, for example the `minElement` method, our tool shows constant runtime independent of the analyzed bounds. Because `minElement` is a pure method, the SMT solver can quickly find a contradiction due to the equality feature of the theory of Core. An SAT solver cannot employ this high level semantic.

Although more scalable than JForge, InspectJ still does not deliver required scalability in all cases. The `run` method, for example, cannot be checked beyond two loop iterations due to its complexity (nested loops with various method calls).

Table 3.1: Results of Bounded Program Verification

Method	Bits	Objs	Loop	JForge					InspectJ			
				Encode	SAT4J	Z3	Total	Result	Result	Encode	Z3	Total
decreaseKey	3	3	3	0.6	TO	61.8	62.4	unsat	unsat	1.5	0.4	1.9
	4	4	4	0.7	TO	82.5	83.2	unsat	unsat	1.5	8.7	10.3
	5	5	5	1.8	TO	TO	TO	-	unsat	1.5	31.3	32.8
	6	6	6	8.7	TO	TO	TO	-	unsat	1.5	117.1	118.6
	7	7	5	63.5	TO	TO	TO	-	unsat	1.5	357.6	359.1
	7	7	6	66.0	TO	TO	TO	-	unsat	1.6	507.5	509.1
deleteMin	3	3	3	0.5	3.4	0.6	1.1	unsat	unsat	1.7	0.2	1.9
	4	4	4	1.5	414.8	36.4	37.9	unsat	unsat	1.7	3.4	5.0
	5	5	5	4.8	TO	TO	TO	-	unsat	1.7	52.5	54.2
	6	6	6	29.5	TO	TO	TO	-	unsat	1.7	133.4	135.1
insert	3	3	3	0.5	1.6	0.5	1.0	unsat	unsat	1.6	0.4	1.9
	4	4	4	0.8	69.8	14.8	15.6	unsat	unsat	1.6	5.4	7.0
	5	5	5	2.1	TO	409.8	411.9	unsat	unsat	1.6	86.8	88.4
	6	6	6	11.3	TO	TO	TO	-	unsat	1.6	110.0	111.6
	7	7	6	71.2	TO	TO	TO	-	unsat	1.6	311.4	313.0
minElement	4	4	4	0.5	0.3	0.2	0.7	unsat	unsat	1.4	0.0	1.4
	6	6	6	6.4	19.5	6.9	13.3	unsat	unsat	1.4	0.0	1.4
	7	7	7	49.5	70.0	16.6	66.1	unsat	unsat	1.4	0.0	1.4
	8	8	8	TO	-	-	TO	-	unsat	1.4	0.0	1.4
	10	10	10	TO	-	-	TO	-	unsat	1.4	0.0	1.4
	11	11	11	FAIL	-	-	-	-	unsat	1.4	0.0	1.4
run	3	3	1	9.6	1.5	2.2	11.8	sat	sat	3.2	0.7	3.9
	4	4	1	16.7	9.5	4.3	21.0	sat	sat	3.2	6.9	10.0
	7	7	1	371.1	TO	299.0	TO	-	sat	3.2	0.2	3.4
	10	10	1	TO	-	-	TO	-	sat	3.2	2.4	5.6
	3	3	2	TO	-	-	TO	-	sat	5.0	52.7	57.7

3.8 Related Work

SAT-based Encoding

Many bounded program verification approaches (e.g., Jalloy [Vaziri-Farahani, 2004], JForge [Dennis et al., 2006], TACO[Galeotti et al., 2013], Miniatur[Dolby et al., 2007], Karun[Taghdiri, 2008], and MemSAT[Torlak et al., 2010]) have been developed to check programs with complex data structures. Jalloy [Vaziri-Farahani, 2004] analyzes Java programs by translating the code into an Alloy [Jackson, 2012] formula and checking it against a property, also expressed in Alloy, using an SAT solver. Alloy is a first-order relational logic with transitive closure that makes it well-suited for specifying complex data structure properties, e.g., the properties of linked lists and trees. Similar to our technique, its translation to SAT requires bounding the number of loop iterations and the program's heap size. Consequently, Jalloy analysis provides a bounded verification: it exhaustively checks a program within the analyzed bounds, but cannot guarantee anything beyond that. Other techniques (e.g., JForge and TACO) have been developed to improve the translation of Java programs into Alloy formulas, thus resulting in more compact SAT formulas

that can be handled more efficiently by existing SAT solvers. The technique presented in [Dennis et al., 2006] applied Alloy-based program verification to check the JML specifications of KOA [Kiniry et al., 2006], a platform for e-voting experimentation. Unlike our technique, all these techniques use propositional logic (via a relational logic) with only local simplifications at Boolean level. We, on the other hand, translates the code and specifications into an SMT logic to allow high-level simplifications.

SMT-based Encoding

The extended static checker for Java (ESC/Java [Flanagan et al., 2013] and ESC/Java2 [Cok and Kiniry, 2004]) checks Java programs with respect to functional properties. ESC/Java handles loops by unrolling them a finite number of times, and replaces procedures calls with user-provided specifications. Users express both the property and the intermediate annotations in JML. ESC/Java translates the given program to Dijkstra's guarded commands [Dijkstra, 1975], encoding the property as assert commands. It then computes weakest preconditions to generate verification conditions as predicates in a first-order logic, and uses several theorem provers (e.g., Z3 [de Moura and Bjørner, 2008] and Yices [Dutertre, 2014]) to prove the verification conditions. Failed proofs are turned into error messages and returned to the user. These techniques, however, either have undecidable logic due to quantification over infinite types, or require user-interactions to produce sufficiently precise invariants due to their loop invariant inference schemes. Therefore, it sometimes reports a counterexample that might, with more effort, have been shown to be invalid and the solver may not terminate with a conclusive outcome.

JML Encoding

In our encoding, a JML quantifier ranges over the objects of a given class on the heap that the analyzed program manipulates. This interpretation has been used in various program verification tools, e.g., WHY3 platform [Bobot and Paskevich, 2011], TACO [Galeotti et al., 2013], PVS [van den Berg and Jacobs, 2001], KeY [Ahrendt et al., 2016], and Isabelle [Klein and Nipkow, 2006]. Such an interpretation encodes the Java semantics more precise compared to a classical one [see Leavens et al., 2006, Chapter 12 page 113] that quantifies also the objects that are constructed by the specifications. It is possible to obtain our quantifier semantics in JML by introducing predicates to omit the objects created in the annotations. The approach presented in [Beckert and Platzer, 2006] applied the range predicate `\created(o)` that specifies whether the object `o` has been created on the heap or not. Another alternative proposed in [Ahrendt et al., 2009] avoided the existential quantifiers by introducing an instance variable of Boolean type to Java classes to indicate each object created or not.

3.9 Conclusion

We have presented an SMT-based encoding of object-oriented programs and specifications into bounded program verification. Our approach translates a Java program and a JML specification into a Quantified Bit-Vector Formula (QBVF), and solves it using an SMT solver. The novelty of our approach is exploiting the theory of quantified bit-vectors of recent SMT solvers, which allows logical constraints that are structurally closer to the original program and specification as well as, and high-level simplifications before being flattened into a basic logic. We aim to provide a lightweight bounded program verification approach, thus not all Java features are supported. Our encoding supports a basic subset of Java that does not include strings, real numbers, and concurrency. We support a class hierarchy definition without interfaces and abstract classes.

We have implemented our approach in the prototype tool InspectJ and compared it to JForge—a compatible SAT-based engine, on a large-scale implementation of the Dijkstra's shortest path algorithm. The results were encouraging; we found 3 previously-unknown bugs, and witnessed significantly better scalability over JForge. Checking programs with respect to bigger bounds allowed us to cover some execution paths that JForge could not cover. Thus it seems a viable approach to verify Java programs by translating the source code into SMT formulas and solving them, as it increases performance compared to other approaches and allows users to check more complex, high-level specifications within a greater scope.

Bounded program verification checks the correctness of a program with respect to user-provided class bounds and loop bounds. These two kinds of bounds, however, are not independent—their intricate relations are spread among the code and specifications, and have to be chosen carefully. When the bounds are not well chosen, it causes dead code or unused objects on the heap, thus resulting in a bad code coverage or heap coverage. In Chapter 4 we will investigate the relations between the bounds and present a calculus to compute the exact loop bounds based on class bounds for further improving the efficiency of bounded program verification technique.

CHAPTER 4

Computing Loop Bounds based on Class Bounds for Bounded Program Verification

Bounded program verification techniques (e.g., Jalloy [Vaziri-Farahani, 2004], JForge [Dennis et al., 2006], TACO[Galeotti et al., 2013], Miniatur[Dolby et al., 2007], Karun[Taghdiri, 2008], MemSAT[Torlak et al., 2010], and InspectJ presented in Chapter 3) check the correctness of a program against a property for a small scope. They use (i) *loop bounds*—the bounds on the number of loop iterations, and (ii) *class bounds*—the bounds on the number of class instances on the heap, as the determining factors of the scope of analysis. [1] These techniques typically require the users to provide these bounds. These two kinds of bounds, however, are not independent—their intricate relations are spread among the code and specifications, and have to be chosen carefully. Consider a loop bound $B_l \in \mathbb{Z}^*$ ($\mathbb{Z}^*$ denotes non-negative integers) and a class bound $C_T \in \mathbb{Z}^*$ are provided to a program that has a loop l and a class T. The program will be transformed into a new program by unrolling the loop B_l times.[2] The new program (called the *unrolled* problem) is to be verified in bounded program verification with respect to the class bound C_T. When the bounds are not well chosen, either the unrolled program has dead code, e.g., the parts of program executions that violate the class bound, or the heap has unused objects, i.e., the objects that are not used in any execution, thus resulting in a bad code or heap coverage in bounded program verification. In the worst cases, (i) when a loop is unrolled too many times, the dead code

[1] The integer values are not uncommon to be limited in bounded program verification. In the thesis the range of integers is bounded regarding the number of bits in the two's complement integer representation.

[2] Figure 3.2 presents an example of loop unrolling.

impedes the performance of the underlying solver so that the verification process may fail due to the solver being overloaded and the reason to the failed verification is unclear for the users; (ii) when a loop is unrolled too few times so that none of the program executions that reach the loop will be valid in the unrolled program, thus any property concerning the loop will vacuously hold in all runs.

Before we continue with this chapter we explain, for the ease of understanding, the concepts (e.g., loop termination and loop sharp upper bounds) involved in this chapter in Definition 4.1 and Definition 4.2.

Definition 4.1. *Let P be a Java program, l be a loop in P, X be the inputs to P that are consistent with class bounds and preconditions, $head_l : X \mapsto Boolean$ be a function denoting whether the first statement of l is executed for an input, $tail_l : X \mapsto Boolean$ be a function denoting whether a statement immediately succeeding l is to be executed for an input in X. Then:*

- ***Reachable loop.*** *Loop l is reachable if*

$$\exists x \in X, head_l(x),$$

otherwise is unreachable.
- ***Total termination.*** *Loop l total-terminates and l is a finite loop (i.e., l terminates for all inputs) if*

$$\forall x \in X, reach_l(x) \implies tail_l(x),$$

otherwise l non-terminates and l is an infinite loop (i.e., does not terminate for any input).
- ***Partial termination.*** *Loop l partial-terminates and l is a terminating loop (i.e., l terminates for partial inputs) if*

$$\exists x \in X, reach_l(x) \wedge tail_l(x),$$

otherwise l is unreachable or l non-terminates.
- ***Terminating inputs / executions.*** *The executions of P that cause the termination of l are named terminating executions, and their corresponding inputs are terminating inputs.*
- ***Non-terminating inputs / executions.*** *The executions of P that cause the non-termination of l are named non-terminating executions, and their corresponding inputs are non-terminating inputs.*

Definition 4.2. *Let P be a Java program, l be a loop in P, X be the inputs to P that lead l to terminate, $iter_l : X \mapsto \mathbb{Z}^*$ be a function with domain X and non-negative integers $\mathbb{Z}^*$ as range denoting the number of iterations of l for an input. Then:*

- ***Loop upper bound.*** *UB_l is the loop upper bound of the loop l if*

$$\forall x \in X, iter_l(x) \leq UB_l.$$

- ***Loop sharp upper bound.*** *$UB_l^\sharp$ is the loop sharp upper bound of the loop l if*

$$\forall x \in X, iter_l(x) \leq UB_l \land \exists x \in X, UB_l^\sharp = iter_l(x).$$

The loop (sharp) lower bounds are defined analogously, with replacing the relational operator $\leq$ by $\geq$. In this chapter the loop lower and sharp lower bounds of loop l are denoted as LB_l and $LB_l^\sharp$, respectively.

Several techniques have been proposed to compute loop upper bounds, including worst-case execution time (WCET) analysis techniques ([Thesing, 2004; Ermedahl et al., 2007; Cullmann and Martin, 2007; Gulavani and Gulwani, 2008; Michiel et al., 2008]), pattern-based loop bounds computation techniques (e.g., [Gulwani et al., 2009]), and bounded model checking techniques (e.g., [Milicevic and Kugler, 2011]). These techniques, however, either handle only loops with simple structures, e.g., no conditional statements or branching statements, or require user-provided annotations on loops, e.g., loop invariants or loop variants. They cannot compute loop sharp upper bounds for loops with arbitrary structures, and none of them compute loop sharp lower bounds.

BMC-based Loop Bound Computation. A simple alternative to compute loop upper bounds can be used in NBIS [Günther and Weissenbacher, 2014]—an incremental bounded model checker. Starting from an initial upper bound, it unrolls a loop and performs sanity-checks whether the loop condition still holds after the last iteration. If so, it unrolls the loop for a new upper bound candidate (e.g., new bound = old bound x 2), and does sanity checking again. This approach may work in the presence of concrete inputs; however, it is imprecise with only class bounds—unknown but finite number of objects on the heap. It may compute upper bounds that are higher than the sharp upper bound, thus many unreachable paths arise in the unrolled program, and the verification may fail. The reason of the failure, however, is unclear for the user. To overcome this potential failure, the user may restart verification with smaller class bounds. Thus the confidence in the correctness of the code is reduced, as the number of relevant objects is smaller. Moreover, this approach does not compute bounds when the loops partial-non-terminate. A loop partial-non-terminates when it does not terminate for at least one input while it may terminate for other inputs. This is inconsistent with bounded program verification techniques, which do analyze the terminating executions (see Definition 4.1) of a method and ignore non-terminating runs.

In this chapter we provide an SMT-based calculus to compute loop sharp bounds for bounded program verification. It computes the loop sharp upper and loop sharp lower bounds based only on class bounds. The calculus can be used as a preprocessing phase in bounded program verification. It exhaustively checks all possible runs of a loop and computes the sharp bounds for the analyzed loop. Our calculus can therefore provide the user with an insight on what loop bounds to consider in bounded program verification, and

enhances the confidence in the program correctness with respect to the class bounds. When a method contract exists, we consider only the precondition and ignore the postcondition. In addition to a numerical bound, we also output a concrete inputs at the pre-state of the method call and an execution trace that witnesses each computed bound, which guarantees that the computed bound is feasible. We produce loop sharp bounds even for a non-terminating loop (see Definition 4.1), provided that the loop has at least one terminating execution. This is consistent with bounded program verification. Besides, we can detect unreachable loops by analyzing the results of bound computation.

We compute loop *sharp upper* and loop *sharp lower* bounds for a Java program annotated in JML [Leavens et al., 2006]. We encode a program, its precondition, and its sub-routines' specifications (if they exist) as an SMT formula. Each loop is encoded as a recursive uninterpreted function that takes the loop iterations as inputs. The formula is solved for the sharp upper and lower bounds for each loop. This is achieved by calling an SMT solver that is able to solve optimization problems. Given a formula f and an optimization objective o, the SMT solver finds a model for f to achieve the goal o. Several off-the-shelf SMT solvers have been extended to solve optimization problems, e.g., Z3 [de Moura and Bjørner, 2008] with νZ [Bjørner et al., 2015] and SYMBA [Li et al., 2014], MathSAT5 [Cimatti et al., 2013] with OptiMathSAT [Sebastiani and Trentin, 2015]. Besides, some SMT-based algorithms for solving optimization problems have been developed, e.g., the authors of [Ma et al., 2012] integrated an SMT solver with a classical incremental solving algorithm to solve generic optimization problems, and an algorithm in [Sebastiani and Tomasi, 2012] aims to solve linear arithmetic problems. Our calculus explores the advances of SMT solvers. The logic used in our encoding is undecidable, i.e., it is possible for the underlying solver to output "*unknown*".

We have implemented our calculus in the prototype tool *BoundJ*, and compared BoundJ with our Java implementation of BMC-based Loop Bound Computation called *UnrollJ*. Our experiments reveal that in the cases that UnrollJ computes approximate loop bounds BoundJ gave sharp bounds. On the other hand, UnrollJ has a good scalability. It can produces loop bounds with the increased class bounds, while BoundJ returns "*unknown*" for large class bounds. However, for the small class bounds generally used in bounded program verification, the solver returns definite answers.

4.1 Our Calculus

The process of the calculus is divided into two stages. First, the analyzed program with loops is translated into an SMT formula. Second, the SMT formula is solved using an SMT solver and the outcomes of the solver contain the value of loop sharp bound for the loop of interest. Except to unroll loops, all code transformations presented in Chapter 3 are performed on the program in the first stage. Our calculus does not require any user-provided specifications or

annotations; the user only provides bounds on the number of objects of each class. Nevertheless, if the precondition of the analyzed method is provided, our analysis will take them into account.

Given a transformed program p, a set C of class bounds, and optionally a **requires** clause req for the analyzed method of p, let pre be the formula encoding req, N_l be the number of evaluating loop conditions (= 1 + the number of loop iterations), and f be the formula encoding the terminating executions of p, we produce the Formula 4.1 in the encoding stage.

$$pre \wedge f \wedge N_l > 0 \tag{4.1}$$

When the formula is satisfiable, a model to the formula represents an execution trace: a pre-state that satisfies req, loop l is reached and iterates $N_l{-}1$ times, and both loop l and method m terminate. When the formula is unsatisfiable, it means that for the set C of class bounds the l loop either does not terminate for all inputs or l is an unreachable loop.[3] We can distinguish these potentials by analyzing the outcomes of the SMT solver. To compute the loop upper bound for loop l, we use the SMT (`maximize` N_l) command to instruct the SMT solver to find a model where N_l is the largest compared to the values in other models. The $\text{LB}_l^\sharp$ is computed similarly to sharp upper bound computation, using the command `minimize` instead of `maximize`.

We use integers to encode loop iterations. Compared to bit-vectors whose values remain in a static range, integers are convenient to represent the number of loop iterations, since (i) a loop may not terminate, and (ii) even for terminating loops, the number of iterations is not known and thus cannot be bounded apriori. It is possible for the solver to output "*unknown*" because our logic is undecidable due to quantifying over integers, and then our analysis terminates with no conclusive outcome.

Given a transformed program p, a set C of class bounds, and optionally a **requires** clause req for the analyzed method of p, we (i) compute the *loop sharp upper bound* ($\text{UB}_l^\sharp$) and the *loop sharp lower bound* ($\text{LB}_l^\sharp$) for each *top-level* loop l that is not nested in any loop in the transformed program, (ii) unroll each top-level loop based on its $\text{UB}_l^\sharp$, and (iii) compute loop sharp bounds for the remaining top-level loops (if they exist) in the unrolled program. The process terminates when no loop exists in the unrolled program. In order to analyze only valid executions, we consider the whole code in each computation. The output of each computation contains a sharp bound, a witnessing pre-state of the analyzed method, and an execution trace representing a valid execution of the program.

[3] The unsatisfiability of the Formula 4.1 could be caused by a corner case in which the specifications are over-specified (e.g., the precondition req is trivial $false$.)

4.1.1 Encoding Loop Control-flow

We use the *verification graph* (see Definition 3.1) to represent the control- and data-flow of programs with loops. In contrast to our bounded program verification approach (see Chapter 3) that unrolls loops and thus the graph is acyclic, we preserve loops, and the graph is cyclic. For ease of convenience, we name some specific edges and nodes in a loop and present the naming in Definition 4.3:

Definition 4.3. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a cyclic verification graph that is constructed from a program with a loop l, nodes $\mathcal{V}_l \in \mathcal{V}$ be the control points in loop l, and node $y \in \mathcal{V}_l$ represents the control points immediately before evaluating the loop condition, thus:*

- ***Entry nodes / edges.*** *Node $x \in \mathcal{V}$ is the entry node of l if*

$$x \in \mathcal{V} \backslash \mathcal{V}_l \wedge \exists E_{x,y} \in \mathcal{E}.$$

  Edge $E_{x,y}$ is an entry edge of l if x is the entry node of l.
- ***Exit nodes / edges.*** *Node $x \in \mathcal{V}$ is an exit node of l if*

$$x \in \mathcal{V} \backslash \mathcal{V}_l, \exists E_{y,x} \in \mathcal{E}.$$

  Edge $E_{y,x}$ is an exit edge of l if x is an exit node of l.
- ***Head nodes / edges.*** *Node $x \in \mathcal{V}$ is a head node of l if*

$$x \in \mathcal{V}_l, \exists E_{y,x} \in \mathcal{E}.$$

  Edge $E_{y,x}$ is a head edge of l if x is a head node of l.
- ***Tail nodes / edges.*** *Node $x \in \mathcal{V}$ is a tail node of l if*

$$x \in \mathcal{V}_l, \exists E_{x,y} \in \mathcal{E}.$$

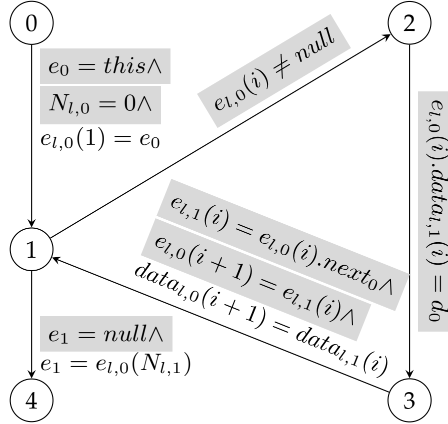
  Edge $E_{x,y}$ is a tail edge of l if x is a tail node of l.

In Fig. 4.1 we present the cyclic verification graph constructed from a Java `setAll` method that sets the `data` field of all list elements to the input value, provided that the input value is not `null`. In the graph, the nodes 0, 4, 2, and 3 are called entry, exit, head, and tail nodes, respectively, and the edges $E_{0,1}$, $E_{1,4}$, $E_{1,2}$ and $E_{3,1}$ are called entry, exit, head, and tail edges, respectively.

Bounded program verification techniques typically encode (acyclic) control-flow using simple Boolean variables. Our approach, on the other hand, encodes (cyclic) control-flow using uninterpreted functions in the SMT logic. More precisely, similarly to our previous approach presented in Chapter 3, when an edge from node x to node y does not belong to a loop, we encode it using a Boolean variable $E_{x,y}$, whose value denotes whether the edge is traversed or not. A edge traversed denotes either a statement has been executed, or a predicate evaluates to $true$ and a branching occurred. When an

```
1  class Entry {
2    /*@nullable*/ Data data;
3    /*@nullable*/ Entry next;
4
5    void setAll(Data d){
6      Entry e = this;
7      while (e != null){
8        e.data = d;
9        e = e.next;
10     }
11   }
12 }
13 class Data{}
```

(a) A Java program



(b) Verification graph with loops

Fig. 4.1: Construction of a cyclic verification graph. Figure 4.1(b) shows the verification graph that is constructed from the Java `setAll` method in Fig. 4.1(a). Variables and fields that are updated in the loop are renamed with the loop ID and numbers. The variable $N_{l,0}$ represents the number of times that the loop condition has been checked, and it is renamed to $N_{l,1}$ when loop condition evaluates to $false$.

edge belongs to a loop, the encoding must clarify in which loop iterations the edge is traversed. Therefore, **(i)** when an edge from x to y belongs to a *top-level* loop l (i.e., l is not nested in any loops), we encode it using a Boolean-valued, uninterpreted function $E_{x,y} : \mathbb{N} \rightarrow Boolean$ ($\mathbb{N}$ denotes natural numbers). The expression $E_{x,y}(i_l)$ evaluates to $true$ if the edge is traversed in the $(i_l)^{th}$ iteration of l. The exit edge of l is traversed once the loop condition is not fulfilled for the $(N_l)^{th}$ iteration of the loop. We encode the exit edge of l as an expression $E_{x,y}(N_l)$, where $N_l > 0$ and $N_l = 1 + K_l$, where K_l is the number of iterations of the loop l in one run. **(ii)** When a loop l_2 is nested in a loop l_1, the iterations of l_2 depend on the iterations of l_1. We encode the edge $E_{x,y}$ that belongs to l_2 using a Boolean-valued, uninterpreted function $E_{x,y} : \mathbb{N} \times \mathbb{N} \rightarrow Boolean$. The expression $E_{x,y}(i_{l_1}, i_{l_2})$ evaluates to $true$ if the edge is traversed in the $(i_{l_2})^{th}$ iteration of l_2 while in the $(i_{l_1})^{th}$ iteration of l_1. We encode the exit edge of l_2 as an expression $E_{x,y}(i_{l_1}, N_{l_2})$, and $N_{l_2} - 1$ is the number of iterations of l_2 in the $(i_{l_1})^{th}$ iteration of l_1. That is, the edge variables encoding the control-flow of inner loop will get an additional parameter corresponding to the iteration number of the outer loop. It works the same way for any depth of nesting.

Definition 4.4 presents the rules to encode control-flow of a top-level loop. In particular, if i is the node that represents the control points immediately before evaluating the loop condition, then (i) if an entry edge of the loop

is traversed, either the first iteration starts or the loop exits before the first iteration, (ii) if a tail edge at the ith iteration of the loop is traversed, then either the $(i+1)^{th}$ iteration starts or the loop exits before that iteration, and (iii) if an edge the belongs to the loop is traversed, one of the outgoing edges of its target node will be traversed as well.

Definition 4.4. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a cyclic verification graph, l be a top-level loop in $\mathcal{G}$, nodes $\mathcal{V}_l \in \mathcal{V}$ be the nodes belonging to loop l, l also has entry edge E_{entry}, exit edge E_{exit}, head edge E_{head}, and tail edge E_{tail}, the control-flow of the loop l is encoded as a conjunction of the following formulas:*

$$\begin{aligned} E_{entry} &\implies E_{head}(1) \vee E_{exit}(1) \\ \forall i_l \in \mathbb{N}, E_{tail}(i_l) &\implies E_{head}(i_l+1) \vee E_{exit}(i_l+1) \\ \bigwedge_{\substack{x,y \in \mathcal{V}_l, \\ x \neq y}} (\forall i_l \in \mathbb{N}, E_{x,y}(i_l) &\implies \bigvee_{\substack{y,z \in \mathcal{V}_l, \\ y \neq z}} E_{y,z}(i_l)) \end{aligned}$$

Figure 4.2 presents the formula encoding the control-flow of the verification graph of Fig. 4.1(b). For readability, we express the formula using first-order logic constructs. For example, the term $E_{0,1}$ denotes the SMT variable `E_0_1` and $E_{1,2}(i_l)$ denotes the SMT uninterpreted function `(E_1_2 i_l)`. We keep using this simplification in the following sections when it does not cause ambiguity. In this example, $E_{0,1}$ is a Boolean variable, while $E_{1,2}$, $E_{2,3}$, $E_{3,1}$, and $E_{1,4}$ are Boolean-valued functions. For each loop l in the verification graph with loops, we introduce a variable $N_l \in \mathbb{N}$ to represent the number of times that the loop condition has been checked for a loop l. In particularly, we use two variables $N_{l,0}$ and $N_{l,1}$ to represent the values before and after executions of loop l.

$$\begin{aligned} &E_{0,1} \\ E_{0,1} &\implies E_{1,2}(1) \vee (E_{1,4}(N_{l,1}) \wedge N_{l,1} = 1) \\ \forall i_l \in \mathbb{N}, E_{1,2}(i_l) &\implies E_{2,3}(i_l) \\ \forall i_l \in \mathbb{N}, E_{2,3}(i_l) &\implies E_{3,1}(i_l) \\ \forall i_l \in \mathbb{N}, E_{3,1}(i_l) &\implies E_{1,2}(i_l+1) \vee (E_{1,4}(N_{l,1}) \wedge N_{l,1} = i_l + 1) \end{aligned}$$

Fig. 4.2: Control-flow formulas for the verification graph in Fig. 4.1.

4.1.2 Encoding Loop Data-flow

We use the type encoding shown in Section 3.4 to encode Java classes and integers. In particular, we encode classes that are involved in the analyzed code using SMT bit-vectors. If the class bound for a Java class T is n, we

encode T as a bit-vector of size $\lceil log(n+1) \rceil$. In the following description, we use S_T to represent the SMT sort of a class T and express the data-flow formula using first-order logic constructs to make reading easier.

In an acyclic verification graph, all variables and fields of the program are renamed so that they are assigned at most once along each path of the graph. Since our computation graphs can be cyclic, renaming cannot be achieved by enumerating all paths. We rename variables and fields of the program assuming that each loop constructs a separate naming context (similar to a called method, for example). This separates the naming of variables (fields) in one loop from the others, which makes it easier to support complex loop structures. More precisely, renaming variables (fields) involves the following steps.

1. Starting from the innermost loop l, we give an initial name to any variable (field) that may be updated by l, and then perform renaming within the body of l as for an acyclic computation graph.
2. We collapse the cycle (loop) l of the computation graph into a single node, denoting the initial and the final names of the variables updated in l.
3. We repeat step 1, considering the collapsed loops.

Hence, any time a collapsing node is visited, adequate conditions are produced to ensure that the variables (fields) of the current context hold the same values as the initial/final variables (fields) of the collapsed loop. In Fig. 4.1(b), d_0, $N_{l,0}$, $N_{l,1}$, e_0, e_1, and $next_0$ belong to the outer context, whereas $e_{l,0}$, $e_{l,1}$, $data_{l,0}$, and $data_{l,1}$ belong to the loop context. Since the loop does not update the $next_0$ field and the constants, e.g., $this$ and $null$, both contexts share them.

Similar to our previous approach that is described in Section 3.4, data accesses outside loops are encoded as follows: A variable v of type T is encoded as an SMT variable $v : S_T$, and a field f of type B declared in a class A is encoded as a function $f : S_A \rightarrow S_B$. However, if a variable or a field are updated within a loop, one needs to know the updates performed in each loop iteration. A variable v_l of type T that may be modified within a loop l is encoded as a function $v_l : \mathbb{N} \rightarrow S_T$, where $\mathbb{N}$ denotes natural numbers and $v_l(i_l)$ denotes the value of v in the $(i_l)^{th}$ iteration of l. Similarly, a field f of type B declared in a class of type A that may be modified within a loop l, is encoded as a function $f_l : \mathbb{N} \times S_A \rightarrow S_B$, where $f_l(i_l, o)$ denotes the value of $o.f$ in the $(i_l)^{th}$ iteration of the loop. The highlighted expressions in Fig. 4.1(b) represents the state transitions or predicate tests. Figure 4.3 presents the data-flow formulas for the code in Fig. 4.1(b). The first 2 formulas correspond to the edges outside the loop and the last 3 ones encode the data-flow in each loop iteration.

As discussed in Section 3.1, unspecified variables or fields may exist when the verification graph has merging nodes. Suppose two paths p_1 and p_2 merge on a node (called *merging node*), and a variable v is updated (and renamed) to v_i on path p_1 and it remains unchanged on path p_2. When the path p_2 is traversed, there is no state transition that transforms v to v_i and then v_i

$$
\begin{aligned}
E_{0,1} &\implies e_0 = this \land N_{l,0} = 0 \\
E_{1,4}(N_{l,1}) &\implies e_1 = null_{Entry} \\
\forall i_l \in \mathbb{N}, E_{1,2}(i_l) &\implies e_{l,0}(i_l) \neq null_{Entry} \\
\forall i_l \in \mathbb{N}, o : T, E_{2,3}(i_l) &\implies (o = e_{l,0}(i_l) \implies data_{l,1}(i_l, o) = d_0) \land \\
&\qquad (o \neq e_{l,0}(i_l) \implies data_{l,1}(i_l, o) = data_{l,0}(i_l, o)) \\
\forall i_l \in \mathbb{N}, E_{3,1}(i_l) &\implies e_{l,1}(i_l) = next_0(e_{l,0}(i_l))
\end{aligned}
$$

Fig. 4.3: Data-flow formulas for the verification graph of Fig. 4.1.

is unspecified. When the merging node belongs to a loop, e.g., node 1 in Fig. 4.1(b), all variables (and fields) that may be updated in the loop will be left unspecified on each iteration.

We provide formulas for the merging edges to mimic some implicit state transitions and thus ensure that the variables on the merging nodes have the correct value. The unhighlighted expressions in Fig. 4.1(b) represents these constraints and they are translated into SMT formulas in Fig. 4.4. These constraints mimic, before the first iteration a state transition from [e_0, $data_0$] to [$e_{l,0}(1)$, $data_{l,0}(1)$], after the last iteration a transition from [$e_{l,0}(N_{l,1})$] to [e_1], and in each new iteration $i+1$ a state transition from [$e_{l,1}(i)$, $data_{l,1}(i)$] to [$e_{l,0}(i+1)$, $data_{l,0}(i+1)$].

$$
\begin{aligned}
E_{0,1} &\implies e_{l,0}(1) = e_0 \\
\forall o : T, E_{0,1} &\implies data_{l,0}(1, o) = data_0(o) \\
E_{1,4}(N_{l,1}) &\implies e_1 = e_{l,0}(N_{l,1}) \\
\forall i_l \in \mathbb{N}, E_{3,1}(i_l) &\implies e_{l,0}(i_l + 1) = e_{l,1}(i_l) \\
\forall i_l \in \mathbb{N}, o : T, E_{3,1}(i_l) &\implies data_{l,0}(i_l + 1, o) = data_{l,1}(i_l, o)
\end{aligned}
$$

Fig. 4.4: Formulas to specify the variables (and fields) of Fig. 4.1.

If a loop l_2 is nested in a loop l_1, the iterations of l_2 depend on the iterations of l_1. Therefore, we encode those variables that are updated in the inner loop l_2 by adding an additional iteration parameter to the SMT functions that represent those variables. That is, if a variable v of type T is modified within l_2, we declare an SMT function $v_{l_1,l_2} : \mathbb{N} \times \mathbb{N} \rightarrow S_T$, where $v_{l_1,l_2}(i_1, i_2)$ denotes the value of v in the $(i_2)^{th}$ iteration of l_2 while in the $(i_1)^{th}$ iteration of l_1. Updated fields are encoded in a similar way by adding an additional parameter.

4.1.3 Checking Formulas

We ensure the SMT variable N_l to represent the number of evaluating loop conditions using the formula $N_l{>}0 \implies E_{exit}(N_l)$. The formula denotes that if the loop l is reached then the loop condition evaluates to $false$ before the exact N_l^{th} iteration. To ensure N_l is the sharp bound on the number of loop iterations, we trigger the solver to find a model where N_l has the maximal assignment. We check the satisfiability of a formula F that conjuncts the control-flow formula, the data-flow formula, and the formula presented in Fig. 4.5. If the formula F is satisfiable, N_l will be interpreted by the solver: (i) $N_l{=}0$ denotes l is not reachable, and (ii) $N_l{>}0$ denotes the loop l is reachable and its sharp upper bound $\text{UB}_l^\sharp{=}N_l{-}1$. If formula F is unsatisfiable and solver returns $N_l{=}\infty$[4], thus loop l non-terminates regarding the provided class bounds, otherwise either the user-provided class bounds are not large enough or the user-provided precondition evaluates to $false$. We can distinguish these potentials by analyzing the proof of invalidity provided by the solver.

```
(push)
(assert (=> (> N_{l,1} 0) (E_{1,4} N_{l,1})))
(maximize N_{l,1})
(check-sat)
(get-model)
(pop)
```

Fig. 4.5: The formula to ensure N_l is the sharp bound for the loop l of Fig. 4.1(b). Recall that the SMT `maximize` command instructs the underlying solver to produce a model that maximizes the value of $\mathtt{N}_{l,1}$.

Due to either method invocations or that a loop is nested in another loop, multiple occurrences of a loop exist in the verification graph. Algorithm 2 is used to compute the loop sharp bounds for each loop occurrence in the graph. The two functions, `computeLoopSharpUpperBound` and `computeLoopSharpLowerBound`, take a cyclic verification graph and a top-level loop as inputs and compute the loop sharp upper and lower bounds for the loop, respectively. The function `unrollLoops` transforms the verification graph to a new graph by unrolling all top-level loops in the old verification graph. When the algorithm terminates, all loops of the verification graph have been unrolled with respect to the computed sharp upper bounds and bounded program verification techniques can directly extract the formula from the graph.

[4] Different SMT solvers may represent the infinite number using different symbols. The symbol ∞ is used in Z3 solver.

Algorithm 2 Compute Loop Sharp Upper/Lower Bounds.

```
1: G(V, E) := A cyclic verification graph with nodes V and edges E.

2: function COMPUTELOOPSHARPBOUNDS(G)
3:     loopSharpBounds ← ∅

4:     while G has loops do
5:         for each loop l ∈ G do
6:             if l is a top-level loop then
7:                 UB♯_l ← COMPUTELOOPSHARPUPPERBOUND(G, l)
8:                 LB♯_l ← COMPUTELOOPSHARPLOWERBOUND(G, l)
9:                 loopSharpBounds ← loopSharpBounds ∪ (l ↦ {UB♯_l, LB♯_l})
10:            end if
11:        end for
12:        G ← UNROLLLOOPS(G, loopSharpBounds)
13:    end while

14:    return loopSharpBounds
15: end function
```

4.2 Evaluation

We have implemented our technique in a prototype tool *BoundJ* that uses Z3 as the underlying SMT solver. We have also considered an approach used in NBIS [Günther and Weissenbacher, 2014] to evaluate the precision of our approach. Since NBIS targets C code, and does not consider specifications or class bounds, we implemented its approach in a prototype tool (*UnrollJ*) that targets Java and accepts the same inputs that BoundJ does. We have used a collection of benchmarks, selected from the literature in program verification, e.g., KeY and InspectJ, OpenJDK, and TPDB (Termination Programs Data Base) [Frohn, 2014]. All the experiments have been performed on an Intel Core 2.50 GHz with 4 GB of RAM using Linux 64bit.

The results are shown in Table 4.1. The *Method* column shows the names of the entry methods of the analyzed programs. In total 6 programs have been analyzed. To increase the complexity of the specifications, we also added method contracts for the sub-routines (if exist). The method `deleteMin` of the class `BinaryHeap` (4 methods, 109LOC) effectively extracts the minimum element in a *min heap*[5] and restores the data structure properties of min heap. The `removeDups` method of the class `KeYList` (4 methods, 33LOC) removes the duplicate elements from a queue and preserves the remaining elements to be singly linked. The `add` method (3 methods, 39LOC) is the classical List implementation in JDK 1.7. All those methods have complex preconditions, i.e., quantifiers have been involved. The `add` method also uses

[5] A min heap is a binary heap where the values that are stored in the children nodes are greater than the value stored in the parent node.

JML reachability expressions to constrain the configurations of objects on the heap. The `copy` method copies elements from a list to another list when the elements contain data that fulfills some requirements .

In addition to these data-structure-based benchmarks, we have also used benchmarks that involve only primitive types. Such benchmarks are typical for the loop bound computation and the non-termination detection communities. Those methods (`fibonacci` and `gauss`) do not have any specification and there are around 10LOC in average in each method. We have selected these benchmarks for the following two reasons: (i) The number of iterations for the loops in those two methods is non-linearly distributed. Therefore, computing their loop bounds is particularly challenging in many existing approaches; (ii) To validate that our approach indeed computes loop bounds for the methods that contain at least one terminating path. For each analyzed program, we have exploited each tool to compute ∼4 loop sharp upper bounds for different class bounds, and in total 25 computations for loop sharp upper bounds have been done using each tool. Besides, we also used BoundJ to compute the loop sharp lower bounds. Thus in total we have done 75 loop bounds computations.

The *Class Bound* column shows the bounds on the number of objects of each class and on the size of the integer bit-width. For a bound n, the analyses of `deleteMin`, `removeDups`, and `copy` methods have to explore data spaces of size $(n+1)^{11} * 2^{9n}$, $(n+1)^9 * 2^n$, and $(n+1)^5 * 2^n$, respectively.[6] The columns $\text{LB}^\sharp$ and $\text{UB}^\sharp$ represent the computed loop sharp lower bounds and sharp upper bounds, respectively. When a method contains more than one loop, the bounds are shown as a sequence of numbers separated by commas. The symbol X denotes that the loop is not reachable from the method. The question mark ? means that no conclusive answer has been achieved after the timeout limit of 20 minutes. We observe that:

1. BoundJ computed *sharp* loop lower and upper bounds for all data structure-rich methods. A careful inspection of the code reveals that all computed loop bounds are exact the sharp bounds.
2. UnrollJ does not always compute precise loop bounds as BoundJ does. Since UnrollJ does not consider the whole code in loop bounds computation, on average its computed loop bounds are 4.2 times (median 4, maximum 13) greater than the ones BoundJ computed. In addition, UnrollJ failed to compute the loop bounds for the non-terminating methods `copy` and `fibonacci` because of timeout, while BoundJ still produced the loop sharp upper bounds for all program executions that terminate.
3. UnrollJ can compute loop upper bounds with increased class bounds, while BoundJ timeouts for 2 (out of 25) cases. For the rest 23 cases, BoundJ always produces loop *sharp* bounds based on the user-provided class bounds.

[6] The numbers are calculated based on the number of accessed classes, fields, and parameters. Computations are given soon.

Method	Class Bound	$LB^{\sharp}$ (*BoundJ*)	$UB^{\sharp}$ (*BoundJ*)	$UB^{\sharp}$ (*UnrollJ*)
BinaryHeap. deleteMin	3	X	X	3
	4	1	1	4
	5	1	1	5
	6	1	2	6
KeYList. removeDup	4	5, 0	5, 0	7, 1
	5	5, 0	6, 0	15, 2
	6	5, 0	7, 1	31, 3
	7	5, 0	8, 2	63, 4
	8	5, 0	9, 3	127, 5
OurList. copy	3	0, 0	3, 1	?, ?
	4	0, 0	4, 1	?, ?
	5	0, 0	5, 2	?, ?
	6	0, 0	6, 2	?, ?
OpenJDKList. add	3	1, 1	1, 2	1, 2
	4	1, 1	3, 4	3, 4
	5	1, 1	7, 8	7, 8
	7	1, 1	21, ?	21, 32
	10	1, 1	?, ?	175, 256
NonTerm. fibonacci	3	0	9	?
	4	0	10	?
	5	0	10	?
	6	0	10	?
NonTerm. Gauss	6	0	63	63
	7	0	?	127
	9	0	?	508

Table 4.1: Results of computing sharp loop bounds using BoundJ and UnrollJ. $LB^{\sharp}$ denotes loop sharp lower bound and $UB^{\sharp}$ denotes loop sharp upper bound.

Complexity of Our Benchmark

Given the class bound 3 for each type that is involved in the analyzed program, there are at most 4 values (`null` and 3 objects) for each class, for the domain of the fields, and for the range of the fields of non-primitive types in the analyzed program. For the range of the fields of primitive types, i.e., `int`, 2^3 potential assignments exist as the bit-width is 3. Each array has up to 2^3 elements. Consider the case that the methods of the class `BinaryHeap` are analyzed using the scope 3. There are 3 classes used and are total of nine fields involved in the implementation: the `BinaryHeap` has two array fields, and the `BinaryHeapElement` and `BinaryHeapIndexKey` each has two fields of primitive types. The configurations of all inputs consist of the assignments of all fields, that is, the number of the configurations is $(4*4*2^3)^2*(4*2^3)^7 = 2^{49}$. Given a scope n for the class `BinaryHeap`, the number of configurations of inputs is $(n+1)^{11}*2^{9n}$. The OpenJDK `List` has 3 classes (`LinkedList`, `Entry` and `Value`), 4 fields of non-primitive types (`header`, `prev`, `next`,

and `elem`), and 1 field of primitive type (`size`). Given a class bound n for OpenJDK `List`, the number of configurations of inputs is $(n+1)^9 * 2^n$. The size is so large that it is impractical to enumerate them. Our approach employs the advances of an SMT solver to explore the space much more sufficiently.

4.3 Related Work

Computing Loop Sharp Upper Bounds

Worst case execution time (WCET) analysis of programs crucially depends on the number of loop iterations. State-or-the-art WCET analysis tools, e.g., [Gulavani and Gulwani, 2008; Ermedahl et al., 2007; Michiel et al., 2008; Thesing, 2004; Cullmann and Martin, 2007], either reply on user-provided program assertions describing loop iterations (e.g., [Gulavani and Gulwani, 2008]) or compute an interval of loop bounds. The techniques that are described in [Blanc et al., 2010; Gulavani and Gulwani, 2008; Shkaravska et al., 2010], for example, instrument each loop with an iteration counter and symbolically execute them with well-chosen loop inputs. The number of iterations of each loop is then represented as a function based on the loop inputs. To compute more precise loop upper bounds, the technique presented in [Lokuciejewski et al., 2009] uses a combination of abstract interpretation, inter-procedural program slicing, and inter-procedural data-flow analysis. All these approaches, however, focus on numerical loops; some of them (e.g., [Blanc et al., 2010; Cullmann and Martin, 2007]) even require well-structured loops with no branches inside them. Our approach can work on complex loops and target data-structure-rich programs.

To compute upper bounds for complex loops in C++ code, SPEED [Gulwani et al., 2009] generates computational complexity bound functions that contain well-implemented abstract data structures. For loops that access data structures, it generates symbolic bound expressions in terms of numerical properties of the data structures and user-defined quantitative functions which describe the effects of object member methods on the numerical properties of that data structure (such as the length of a list or the depth of a tree). Finally it generates symbolic bounds depending on the generation of loop invariants by using heuristic pattern matching. SPEED, however, requires user-provided specifications and does not support complex heap configurations (e.g., transitive reachability) of the analyzed method, and in some cases outputs only an approximate loop bound functions.

Similar to our approach, the bounded model checking approach [Milicevic and Kugler, 2011] encodes loops recursively using SMT theory of lists. In the encoding, each program state is represented as an element of a list, that models an unbounded search path. In order to find a path to an error state, it constraints the first element to be a valid initial state, and two consecutive elements stand for a state transition, finally the error state is represented

by the last element. This approach, however, aims at model checking safety properties of programs, and cannot be used for loop bounds computation.

The incremental bounded model checker NBIS [Günther and Weissenbacher, 2014] can be adapted to compute loop upper bounds. It instruments every loop in a given C/C++ code with an unrolling assertion that checks whether the loop can iterate beyond the current loop bound. The formula that encodes the code (up to the end of each loop) along with the unrolling assertions is checked for satisfiability. A model to the formula denotes an execution trace in which a loop iterates more times than its current bound. That loop is then unrolled according to the newly-found bound and the process starts over. However, such an approach does not terminate in case of non-terminating loops, or does not return the sharp upper bounds, since the execution trace corresponding to a model stops the execution when exiting the loop with a new bound, hence the code and any constraint following the loop are ignored. Consider a case where each iteration of the loop allocates one instance of class `A` and the code after the loop allocates two objects of type `A`. If $bound(A) = 5$, no valid execution of the code (with respect to the class bound) can iterate the loop more than three times, whereas the computed upper bound will be 5 when ignoring the code after the loop. Preventing the trace from stopping is not possible, since it requires an encoding that does not depend on the loops being unrolled (since the needed number of unrolling is unknown prior to invoking the satisfiability procedure).

An alternative is to check whether the trace is valid with respect to the class bounds by executing (symbolically or dynamically) the whole code. Invalidity of the current instance, however, does not necessarily mean that the newly-found loop bound is impossible; it may still be that another satisfying instance can be valid and gives a higher loop bound. Thus, in the worst case, such a validity check requires enumerating all possible satisfying instances, which makes the approach impractical.

Detecting Non-terminating Loops

Various techniques (e.g., [Velroyen and Rümmer, 2008; Brockschmidt et al., 2011]) have been developed to detect non-terminating programs with no concern for bounded domains. Similar to our approach, JForge [Dennis et al., 2006]—a bounded program verification system, checks non-termination with respect to bounded domains. Its primary goal is to check a given method with respect to user-provided specifications using an SAT solver. JForge detects what program statements were involved in checking the specification. If the program is verified in the bounded domain, JForge analyzes the unsatisfiable core returned from the solver and maps the core back to statements of the analyzed method. Statements that were not involved are marked as “missed” statements. If the missed-statements include the whole loop body but the loop condition is not missed, JForge warns the user about non-termination. This

approach only supports simple loops and cannot handle arbitrary ones. Besides, since its non-termination detection is a by-product of bounded program verification, it requires the user to provide a carefully considered postcondition that is fulfilled by the analyzed method and describes the behaviors of the loop in question, thus its non-termination detection cannot be invoked directly.

Bounded model checkers such as Java PathFinder [Visser et al., 2003] and JPF-SE [Anand et al., 2007b] backtrack the symbolic execution path when the current abstract state is equal to one of the states in the path. Thus they can be used to detect non-terminating loops. Unlike our approach that requires type bounds, these checkers require a bound on the size of inputs and/or the search depth.

4.4 Discussion

We discuss two alternative approaches to compute loop upper bounds for the loops that always terminate.

For loops that always terminate, one complementary approach for CBMC is to (symbolically) check the feasibility of the extracted pre-state on the program. If the pre-state is feasible, we can get the number m of the loop iterations and start over the computation of loop upper bounds of CBMC with unrolling the loop m times. Otherwise, we instrument the negation of the pre-state into the BMC instance and try to get another pre-state by invoking the satisfiability procedure, and check its feasibility. In the worst case this approach exhaustively searches all configurations of inputs based on the bounded domain, which may cause tens of thousands of invocations of the satisfiability procedure, and the size of the BMC instance will increase dramatically. Another approach for handling loops that always terminate is to first unroll the loops based on the loop bounds N calculated by CBMC, then to check whether a feasible pre-state exists to pass through the Nth loop iteration and to exit from the loop. If such a pre-state does not exist, we check whether a feasible pre-state exists to pass through the $(N-1)^{th}$ loop iteration and then exit from the loop. If the pre-state exists, we give the number of the latest checked loop iteration as the loop upper bound. However, this approach may cause lots of invocations of the satisfiability procedure if nested loops occur in the method. Both approaches depend on the condition that the loops always terminate, otherwise, CBMC will have infinite loop unrolling and cannot calculate the loop bounds.

Besides the problems that have been mentioned in Section 4.3, all above approaches reduce the necessary confidence in the correctness of the analyzed code. The overestimated loop upper bounds result in many unreachable paths after loop unrolling. Hence, the formulas that are translated from the unreachable paths may overload the underlying solver and cause the verification process to fail. It might be due to the computed upper bound of the loop that

is too high or that the user-provided class bounds are too high. When the verification engineer uses smaller class bounds than those in the previous run to recompute the loop upper bounds, the new loop upper bounds still may be too high since the computation ignores the code and specifications following the loop under consideration. If the engineer arbitrarily selects smaller loop upper bounds, not all inputs concerning the class bounds are completely analyzed, thus the correctness of the code is not guaranteed for the class bounds. Consequently, although this iterative approach is successful for terminating loops in the absence of class bounds and specifications, it is not applicable in the context of bounded program verification since the class bounds are necessary and the confidence cannot be guaranteed for the class bounds.

4.5 Conclusion

We have presented a calculus for computing *sharp* upper and lower loop bounds based on class bounds. Such an analysis is particularly useful for bounded program verification in which the user has to provide bounds on both the number of objects of each class and the number of loop iterations. Determining feasible loop bounds with respect to class bounds is still a critical problem for bounded program verification. When the bounds are not well chosen, either the unrolled program has dead code—the parts of program executions that violate the class bound, or the heap has unused objects—the objects that are not used in any execution, thus resulting in a bad code or heap coverage in bounded program verification.

Our calculus provides the user with an insight on what loop bounds to consider and enhances the confidence in the correctness of the analyzed programs. We focus on programs with complex data structures and support arbitrary configurations of the objects on the heap. We translate Java code and JML specifications (excluding the postconditions of the entry method) into an SMT formula and solve it using an SMT solver that can solve optimization problems. If the formula is satisfied, a model to the formula represents an execution trace that satisfies the preconditions, reaches the loop under analysis, and leads the method to terminate. The loop sharp bounds can be obtained from the model. Otherwise, it denotes that the loop does not terminate, or the provided class bounds (or specifications) are inconsistent.

We have implemented our calculus in the prototype tool BoundJ and compared with an alternative approach that incrementally unrolls a loop and sanity-checks whether the loop condition still holds after the last iteration. Experiments show that our technique indeed produces the sharp bounds, whereas not necessarily the sharp ones. Our approach can assist the bounded program verification engineers to obtain more confidence in the verification process. Although our analysis is not guaranteed to produce a conclusive outcome (due to the undecidability of our logic), our experiments show that

in practice the unknown outcome occurs for higher input bounds and not for the small bounds that are typically used in bounded program verification.

Part III

Efficient Deductive Program Verification

CHAPTER 5

Verification-based Program Slicing for Deductive Program Verification

Deductive program verification systems, e.g., KeY, typically require verification engineers to write auxiliary specifications, e.g., loop invariants and method contracts of called methods, with respect to the specification. To discover useful annotations that are fulfilled by the called methods and also meet the requirements of the calling methods is, unfortunately, a complicated and error-prone effort. To ease the burden, verification engineers routinely decompose a complex specification (the *whole* property) into a conjunction of less complex specifications (each of them specifies a *partial* property—a part of the whole property), and then prove the partial properties separately instead of having a single verification concerning to the whole property. Then, usually, only parts of the implementation are relevant for proving a partial property and only partial and less complex auxiliary specifications are needed.

Before writing auxiliary specifications, the verification engineers have to identify the slices of the implementation relevant to the partial properties. However, it is a considerable burden to manually discover such slices in programs with complex data structures regarding properties that constrain the configurations of program objects on the heap. Usually, this kind of programs have sophisticated interconnections of data structures, and the relationship between the implementation and the property is obscure. When the slices are not well identified, more effort from the verification engineers are needed in the process of deductive program verification. For example, lack of annotations for the relevant program parts may cause the proof to fail, and hereafter, a costly inspection on the failed proof is indispensable. Besides, annotations of the irrelevant parts require more proof steps and may overload the deductive

verification systems. It would be very helpful if we can automatically find such slices for deductive program verification.

The main contribution of this chapter is a verification-based program slicing technique to construct a program slice for deductive program verification. Given a program annotated by a specification and the scope of analysis (designated by class / loop bounds), we construct a *semantic slice* (see Definition 5.1) with respect to the slicing criteria, i.e., the specification and the scope of analysis. In the slice, the parts of the program that are irrelevant to the slicing criteria are replaced by abstractions (i.e., they are not completely removed), whereas the rest of the program (i.e., the relevant parts) remains unchanged. Verifying slices requires less auxiliary specifications (as the abstractions have less details), and their correctness—by construction—implies the correctness of the original program concerning the specification. As a result, our program slicing technique is capable of liberating the verification engineers from finding the relevant program slices manually, and expediting the progress of deductive verification: less proof steps are needed.

Definition 5.1. *Let P be a program, Q be a specification for P, then*

- ***Relevant to Q.*** *A statement in P is relevant to Q if its behavior affects the evaluation results of Q, vice versa.*
- ***Irrelevant to Q.*** *A statement of P is irrelevant to Q if its behavior does not affect the evaluation results of Q, vice versa.*

*Thus, a **semantic slice** that is constructed regarding the slicing criteria Q consists of all relevant statements to Q and the abstractions that overestimate the behavior of the irrelevant statements to Q.*

The core idea is to use the bounded program verification technique presented in Chapter 3 to guide the construction of slices. The bounded program verification technique does not require auxiliary specifications. It translates the analyzed program and its *negated* specification into an SMT formula F with respect to the class / loop bounds, and solves F using an SMT solver. If a model to F is found (i.e., F is satisfiable), then that is a counterexample to the correctness of the original program as well, and no further analysis is required. If no model is found (i.e., F is unsatisfiable), the partial property holds for the scope of analysis. The computation is based on the *unsatisfiable core* (unsat core)—a sub-formula F_{core} of F that is still unsatisfiable. The unsat core is produced by an SMT solver that can provide the proof of invalidity. If a program statement is translated into SMT formulas f and f are not in the unsat core, then the statement is irrelevant to the property. For each irrelevant statement, we construct an abstraction that overestimates its behaviors. Finally, we generate a semantic slice by replacing irrelevant statements using the constructed abstractions.

The semantic slice is generated based on a particular bounded proof constructed by bounded program verification. Therefore it (i) may be too abstract

and thus deductive verification is not possible, and (ii) may exclude unnecessary, yet helpful details, hence deductive verification may require more effort. To handle the first problem, we provide an algorithm to refine the abstractions as presented in Chapter 6. To handle the second problem, we optimize the semantic slices to include those helpful details for the deductive verification. We have implemented our technique in a prototype tool and evaluated the benefits of using our technique in deductive program verification. The results show in practice the constructed slices are sufficiently precise.

5.1 Motivating Examples

To demonstrate outcomes of our program slicing technique we use three examples. In these examples, we use programs with different data structures. These data structures are in turn integers, arrays, and a singly linked list. To simplify verification, we use **`diverges true`** to disable loop termination checking and **`assignable \everything`** to allow arbitrary heap manipulations in the analyzed program, but the program has to fulfill its **`ensures`** clause.

Figure 5.1(a) shows an example to construct abstractions of a program manipulating integers. The `numberOfPrime` method computes the number of prime numbers between two given integers `x` and `y` (exclusive). The JML **`\result`** construct refers to the value returned by the method. The first line denotes that if the number of prime numbers is larger than 0, then `x < y`. Carefully inspecting the code, a verification engineer will notice that the **`ensures`** clause becomes false only when `x >= y`. In that case, the outer loop (Fig. 5.1(a), Lines 6–17) is never executed and the variable `size` remains equal to 0. However, using traditional static slicing techniques, all program statements (Fig. 5.1(a), Lines 5–21) will be relevant with respect to the variables `x`, `y`, and `size` at the `return` statement. Thus loop invariants are required for the two loops. We choose 3 as the class bound[1], that is there are at most 3 objects of each class, and the bit-width of the integer is 3. Based on the class bound, we compute the loop sharp upper bounds using our technique presented in Chapter 4. With these two kinds of bounds, our technique generates an abstract program as shown in Fig. 5.1(b). The two `native` methods at lines 24 and 26 are the constructed abstractions. Their annotations denote that they return unspecified values of the appropriate type, hence they overestimate the behavior of the irrelevant statements to the desired property. In the abstract program, the outer loop body (Fig. 5.1(a), Lines 7–16) and the branch (Fig. 5.1(a), Lines 18–20) are replaced by abstractions. Thus, it becomes easier to write loop invariants for the outer loop, and no loop invariant is needed for the inner loop. KeY proved the original program with

[1] In bounded program verification the range of integers is bounded regarding the number of bits in the two's complement integer representation. Thus the class bound 3 is the smallest bound we can choose in order to represent the integer $2 \in \left[-2^2, 2^2 - 1\right]$ in the loop branching statement at Line 8 in Fig. 5.1(a).

```
1  /*@ ensures \result>0 ==> x<y;
2    @ diverges true;
3    @ assignable \everything;*/
4  int numberOfPrime(int x,int y){
5    int size = 0;
6    for(int i=x; i<y; i++){
7      boolean isPrime = true;
8      for(int j=2; j<i; j++){
9        if(i%j==0){
10         isPrime = false;
11         break;
12       }
13     }
14     if(isPrime){
15       size++;
16     }
17   }
18   if(size > 0){
19     int[] a = new int[y-x];
20   }
21   return size;
22 }
```

(a) Original program

```
1  /*@ ensures \result>0 ==> x<y;
2    @ diverges true;
3    @ assignable \everything;*/
4  int numberOfPrime(int x,int y){
5    int size = 0;
6    for(int i=x; i<y; i++){
7      size = pure_int();
8
9
10
11
12
13
14
15
16
17   }
18   pure_allocArrayInt();
19
20
21   return size;
22 }
23  //@ assignable \strictly_nothing;
24  native int pure_int();
25  //@ assignable \nothing;
26  native int[] pure_intArray();
```

(b) Abstract program

Fig. 5.1: Semantic slicing a program manipulating integers. The `numberOfPrimes` method computes the number of prime numbers between two integers. The empty lines in the abstract program are left deliberately for an intuitive comparison.

26 auxiliary specifications using 5802 proof rules (counted as the number of JML constructs and logical connectors), the abstract program with provided 8 auxiliary specifications using 646 proof rules.

Figure 5.2 shows example of constructing abstractions for a program manipulating arrays. It provides a map data type implemented using associative arrays. Keys and values are recorded in separate arrays, `keys` and `values`, respectively, and have the same index in the arrays. The method `put(k,v)` invokes the method `getIndexOf` to check whether `k` already exists in the map. If it exists, the old value is replaced by `v`; otherwise, the methods `addKey` and `addValue` reallocate the arrays `keys` and `values`, and add `k` and `v` to the new arrays. The **`ensures`** clause guarantees that the value `v` is in this map (using the **`\exists`** quantifier). By default, referenced variables are not null, thus we use the **`nullable`** clause that enables also the `null` value. The constructed abstract program is shown in Fig. 5.2(b), where the method invocation statements (Fig. 5.2(a), Line 11 and Line 15, respectively) are replaced by abstractions, thus no loop invariants are needed for the

```
1  class Key {}
2  class Value {}
3  class Map {
4   /*@ nullable */ Key[] keys;
5   /*@ nullable */ Value[] values;
6   /*@ ensures(\exists int i;0<=i&&
7     @ i<values.length;values[i]==v);
8     @ diverges true;
9     @ assignable \everything;      */
10  void put(Key k, Value v){
11   int pos = getIndexOf(k);
12   if (pos>=0){
13    values[pos] = v;
14   } else {
15    addKey(k);
16    addValue(v);
17   }
18  }
19  int getIndexOf(Key k){
20   int r = -1;
21   for(int i=0;i<keys.length;i++){
22    if (keys[i] == k){
23     r = i;
24    }
25   }
26   return r;
27  }
28  void addKey(Key k){
29   Key[] oldKs = keys;
30   keys = new Key[keys.length+1];
31   keys[keys.length - 1] = k;
32   for (int i=0;i<oldKs.length;i++){
33    keys[i] = oldKs[i];
34   }
35  }
36  void addValue(Value v){
37   Value[] oldVs = values;
38   values=new Value[values.length+1];
39   values[values.length - 1] = v;
40   for (int i=0;i<oldVs.length;i++){
41    values[i] = oldVs[i];
42   }
43  }
44 }
```

(a) Original program

```
1  class Key {}
2  class Value {}
3  class Map {
4   /*@ nullable */ Key[] keys;
5   /*@ nullable */ Value[] values;
6   /*@ ensures (\exists int i;0<=i&&
7     @ i<values.length;values[i]==v);
8     @ diverges true;
9     @ assignable \everything; */
10  void put(Key k, Value v){
11   int pos = pure_int(k);
12   if(pure_boolean()){
13    values[pos] = v;
14   } else {
15    impure_keys(k);
16    addValue(v);
17   }
18  }
19  void addValue(Value v){
20   Value[] oldVs = values;
21   values=new Value[values.length+1];
22   values[values.length - 1] = v;
23   for (int i=0;i<oldVs.length;i++){
24    values[i] = pure_Value();
25   }
26  }
27  //@ assignable \strictly_nothing;
28  native int pure_int();
29  //@ assignable \strictly_nothing;
30  native boolean pure_boolean();
31  //@ assignable this.keys;
32  native void impure_keys();
33  //@ assignable \strictly_nothing;
34  native/*@nullable*/Value pure_Value();
35 }
```

(b) Abstract program

Fig. 5.2: Semantic slicing a program manipulating arrays. The `put` method puts a key and a value to a map.

methods `getIndexOf` and of the `addKey`. For the loop (Fig. 5.2(b), Line 23) of `addValue` method, the required loop invariant does not need to constrain all details of the loop body. It is very likely that manually discovering these facts requires non-trivial efforts. KeY has proved the original program anno-

tated needed only 14 auxiliary specifications and 14684 proof rules, while the abstract program with 4 auxiliary specifications using 3879 proof rules.

```
1  class Node {
2   /*@ nullable */ Node next;
3   /*@ requires this.next!=null;
4     @ ensures this.next!=null;
5     @ diverges true;
6     @ assignable \everything;*/
7   void add(Node p){
8    Node e = this;
9    while (e.next != null){
10    e = e.next;
11   }
12   e.next = p;
13  }
14 }
```

(a) Original program

```
1  class Node {
2   /*@ nullable */ Node next;
3   /*@ requires this.next!=null;
4     @ ensures this.next!=null;
5     @ diverges true;
6     @ assignable \everything;*/
7   void add(Node p){
8    Node e = pure_Node();
9    e.next = p;
10  }
11  /*@ assignable \strictly_nothing; */
12  native/*@nullable*/Node pure_Node();
13 }
```

(b) Abstract program

Fig. 5.3: Semantic slicing a program manipulating a singly linked list. The `add` method puts a node to the end of a list.

Figure 5.3 shows an example that constructs abstractions of a program manipulating a singly linked list. The method `add(p)` in Fig. 5.3(a) appends a node `p` at the end of the list. The method is expected to fulfill the property that appending a node to a non-empty list still results in a non-empty list. The abstract program is shown in Fig. 5.3(b), where all program statements except the last one (Fig. 5.3, Line 12) are replaced by an abstraction. KeY has proved the abstract program without any auxiliary specifications using 50 proof rules. We did not manage to prove the original program with a reasonable effort. That is because deductive program verification in general requires non-trivial lemmas or reasoning about sets to specify linked data structures, e.g., what objects can be reached from a source object following particular fields.

5.2 Our Technique

The process of constructing a semantic slice from a program with respect to a property is divided into four stages. First, the analyzed program is verified using bounded program verification. Second, the parts of the code which are relevant to the property are extracted from the outcomes of the bounded verification. Third, the abstractions are constructed from the relevant code parts. Fourth, the semantic slice is constructed using the abstractions.

5.2.1 Bounded Program Verification

The bounded program verification approach (see Chapter 3) translates a program P into an SMT formula F, i.e., a set of SMT assertions and declarations, and solves F using an SMT solver. In the source code transformation phase (see Section 3.1.1), program P is transformed into an unrolled program $P^\dagger$ by unrolling the loops (and by inlining the called methods) of P with respect to the loop bounds. Thus, each program statement of P appears at least once in program $P^\dagger$. We use $stmt_{loc}$ to denote a statement at the program point loc of P and $stmt^\dagger_{loc}$ to denote each of $stmt_{loc}$'s appearances in $P^\dagger$. After the code transformation, each $stmt^\dagger_{loc}$ is translated into at least one SMT assertion, e.g., a branch statement is translated into two SMT assertions. Therefore, there exists a *many-to-one* relationship between the SMT assertions and the original program statements.

We construct a *formula map* $M := \{F \mapsto S\}$ to represent the many-to-one relationship between the set F of SMT assertions and the original program statements S. Figure 5.4 presents the principle rules used for constructing the formula map.

R$_1$: $\mathcal{T}(\texttt{v = e;}, i, j)$	$\rightarrow$	$M := M \cup \{f_{stmt^\dagger_{loc}} \mapsto stmt_{loc}\}$
R$_2$: $\mathcal{T}(\texttt{v = o.f;}, i, j)$	$\rightarrow$	$M := M \cup \{f_{stmt^\dagger_{loc}} \mapsto stmt_{loc}\}$
R$_3$: $\mathcal{T}(\texttt{v = a[k];}, i, j)$	$\rightarrow$	$M := M \cup \{f_{stmt^\dagger_{loc}} \mapsto stmt_{loc}\}$
R$_4$: $\mathcal{T}(\texttt{T o = new T;}, i, j)$	$\rightarrow$	$M := M \cup \{f_{stmt^\dagger_{loc}} \mapsto stmt_{loc}\}$
R$_5$: $\mathcal{T}(\texttt{T[] a = new T[k];}, i, j)$	$\rightarrow$	$M := M \cup \{f_{stmt^\dagger_{loc}} \mapsto stmt_{loc}\}$
R$_6$: $\mathcal{T}(\texttt{o.f = e;}, i, j)$	$\rightarrow$	$M := M \cup$ $\{(\implies E_{i,j}\ (= (\mathcal{E}(\texttt{f}, j)\ o)\ \mathcal{E}(\texttt{e}, i))) \mapsto stmt_{loc}\}$
R$_7$: $\mathcal{T}(\texttt{a[k] = e;}, i, j)$	$\rightarrow$	$M := M \cup$ $\{(\implies E_{i,j}\ (= (\mathcal{E}(\texttt{f}, j)\ \mathcal{E}(\texttt{a}, i)\ o)\ \mathcal{E}(\texttt{e}, i))) \mapsto stmt_{loc}\}$

Fig. 5.4: Construction rules for the formula map M. The functions $\mathcal{T}$ and $\mathcal{E}$ (see Section 3.4) are respectively used to translate program statements and expressions into SMT formulas. The $stmt_{loc}$ denotes a program statement at the program point loc in the original program, while $stmt^\dagger_{loc}$ denotes each of $stmt_{loc}$'s appearances in the unrolled program. The symbols i and j in $\mathcal{T}(stmt^\dagger_{loc}, i, j)$ denote the program states before and after the execution of the statement $stmt^\dagger_{loc}$, respectively. The $f_{stmt^\dagger_{loc}}$ denotes the SMT formula that is translated from statement $stmt^\dagger_{loc}$ using the translation rules shown in Fig. 3.6. The arrow $\rightarrow$ is read as: when the translation on its left-hand side ends, the formula map is updated as shown on its right-hand side.

Basically, for each assignment statement $stmt^{\dagger}_{loc}$ of $P^{\dagger}$ that does not modify class fields and arrays, as shown in R_1– R_5, there exists a mapping $f_{stmt^{\dagger}_{loc}} \mapsto stmt_{loc}$ in the formula map M, where $stmt_{loc} \in S$ denotes the original statement of $stmt^{\dagger}_{loc}$, and $f_{stmt^{\dagger}_{loc}} \in F$ represents the SMT formula that is translated from $stmt^{\dagger}_{loc}$ using the translation rules shown in Fig. 3.6. For the assignment statements that change the field values or the array values, as shown in R_6– R_7, the formula map M contains only the entities whose keys encode the changed values. For example, the statement `o.f = e` is translated into a formula with two clauses, $o.f' = e$ and $\forall T\, x, x \neq o \implies x.f' = x.f$, where T represents the type of o and f' denotes the field f after the statement execution. Thus, only the former clause is used to update the formula map.

The branch statements are handled a bit different. The branch statement `if (cond) stmt; else stmt;` is translated into two SMT formulas, $E_{true} \implies \mathcal{E}(cond)$ and $E_{false} \implies \neg\mathcal{E}(cond)$, where E_{true} and E_{false} are edge variables. For each formula, the formula map contains an entity that links the formula to the branch statement.

5.2.2 Extracting Relevant Code

When the target SMT formula F—a set of SMT assertions and declarations—is unsatisfiable, an SMT solver capable of generating proofs is used to find a proof of invalidity,[2] i.e., an *unsatisfiable core* (unsat core) $F_{core} \subseteq F$. More concisely, given an unsatisfiable SMT formula with SMT declarations and assertions, if there exists an SMT formula F_{core} such that $F_{core} \subseteq F$ and F_{core} is unsatisfiable, then F_{core} is called an unsat core of the original formula F. To the best of our knowledge, the unsat core provided by the solver is not guaranteed to be minimal. That is, removing an assertion from F_{core} does not guarantee F_{core} becomes satisfiable.

To facilitate the extraction of relevant code, we minimize the unsat core F_{core} to ensure it is local minimal, i.e., removing any assertion from F_{core} renders it satisfiable. Algorithm 3 minimizes an unsat core by exhaustively checking the assertions. of the unsat core. If the unsat core remains unsatisfiable when deactivating (negating) the constraint of an assertion, the assertion is not necessary for the unsat core. The algorithm traverses the verification graph of $P^{\dagger}$ in the depth-first order, and for each statement $stmt^{\dagger}_{loc}$ it meets deactivates all the assertions that are linked to $stmt_{loc}$ in the formula map M.

A local minimal unsat core F_{core} has the following properties:

- Removing any assertion from F_{core} causes F_{core} to be satisfiable.
- F remains unsatisfiable when removing any assertion $f \in (F \backslash F_{core})$ from F.

[2] We use the *clause selector* technique [Cimatti et al., 2007] to enable the SMT solver to reuse the lemmas learned in previous solving.

Algorithm 3 Minimize an unsat core

1: $\mathcal{C}$: unsatisfiable SMT constraints;

2: **function** TRAVERSE($\mathcal{C}$)
3: $\mathcal{C}^+ \leftarrow \emptyset$ // local minimal unsat core.
4: $\mathcal{C}^- \leftarrow \emptyset$ // unnecessary constraints.

5: **for** $c \in \mathcal{C}$ **do**
6: **if** $c \notin \mathcal{C}^-$ **then**
7: **if** $(\mathcal{C} - c) \setminus \mathcal{C}^-$ is UNSAT **then**
8: $\mathcal{C}^+ \leftarrow$ UNSATCORE($(\mathcal{C} - c) \setminus \mathcal{C}^-$)
9: $\mathcal{C}^- \leftarrow \mathcal{C}^- \cup ((\mathcal{C} - c) \setminus \mathcal{C}^+)$
10: **end if**
11: **end if**
12: **end for**

13: **return** $\mathcal{C}^+$
14: **end function**

Supposing the unrolled program $P^\dagger$ is translated into a set $F_{P^\dagger}$ of SMT assertions, and the **requires** and **ensures** clauses of the analyzed program are translated into the SMT assertions, f_{req} and f_{ens}, respectively, $F = (F_{P^\dagger} \cup \{f_{req}\} \cup \{f_{ens}\})$, and F_{core} is local minimal. We discuss the following three cases, and the third one is used for extracting relevant code.

- If $f_{req} \notin F_{core}$, $f_{ens} \notin F_{core}$, and $F_{P^\dagger} \cap F_{core} \neq \emptyset$, then there is no valid execution for the unrolled program $P^\dagger$: the provided class or loop bounds might be not large enough.
- If $f_{req} \in F_{core}$, $f_{ens} \in F_{core}$, and $F_{P^\dagger} \cap F_{core} = \emptyset$, then either the pre-/post-condition conflict with each other, or the precondition evaluates to $false$, or the postcondition evaluates to $true$ (the postcondition is negated in bounded program verification).
- If $f_{ens} \in F_{core}$ and $F_{P^\dagger} \cap F_{core} \neq \emptyset$, then the program statements $F_{core} \times M$ are relevant to the property, where M is the formula map. The rest of the program is irrelevant to the property.

5.2.3 Constructing Abstractions

The original program statements that are relevant to the property of interest are called *mustHave* statements, and the remaining statements are called *mayHave* statements: they are not necessary for checking the property in bounded program verification, but may be helpful in the deductive program verification. We use S^+ to denote the set of *mustHave* statements, S^- to denote the set of *mayHave* statements, and $S^+ = S \backslash S^-$, where S denotes all original program statements.

We construct abstractions for the *mayHave* statements. According to the behavior of mayHave statements, we construct three kinds of abstractions (methods with annotations) as shown in Fig. 5.5. In particular, for each *mayHave* statement $stmt_{loc} \in S^{-}$, we construct an abstraction as follows:

- If $stmt_{loc}$ has no side effects and does not create objects, we construct a `totalPure_T` method with **`assignable \strictly_nothing`** (Listing 3), where `T` represents an appropriate type that will be talked soon.
- If $stmt_{loc}$ has no side effects but creates objects, e.g., the statement `T o = new T();` creates a `T` instance while the `class T{}` has no fields, we construct a `partialPure_T` method with **`assignable \nothing`** (Listing 4).
- If $stmt_{loc}$ has side effects, e.g., the method call `this.func(e)` changes fields or global variables in the called (or nested called) method, e.g., the method declaration `void func(e){this.f=e;e.g[0]=null;}`, we construct an `impure_T` method annotated by **`assignable`** `FieldSet` (Listing 5), where `FieldSet` denotes a collection of fields (or arrays) that may be changed by the called method, e.g., **`assignable this`**`.f, e.g[*];`. We compute `FieldSet` by propagating variables on the verification graph and construct a chain of field dereferences.

These three kinds of methods and their annotations are auto-generated, and their method calls will return unspecified, yet *distinct*, values of an appropriate type.

5.2.4 Constructing Semantic Slice

We construct a semantic slice using the abstractions. In particular, for each *mayHave* statement $stmt^{-} \in S^{-}$, if $stmt^{-}$ is a branch statement, we replace its branch condition expression by the constructed abstraction, and if $stmt^{-}$ is an assignment statement, we replace its right-hand side expression by its abstraction. Figure 5.5 shows the rules used to transform the code to a semantic slice.

Optimizing Semantic Slice

The constructed abstract programs contain less details and in general the efforts of verification engineers in writing auxiliary specifications are reduced using them. However, they may exclude unnecessary, yet helpful details, hence deductive verification may require more effort in the process of proving. Typically, a deductive program verification system, e.g., KeY, symbolically executes a program and applies various calculus rules to make a proof. During symbolic execution, the program states of the original program are very likely have more details than those of the abstract program. Therefore, the symbolic execution paths which are invalid for the concrete programs can be valid

```
1  /*@ assignable \strictly_nothing; @*/
2  native /*@ nullable @*/ T totalPure_T();
```

Listing 3: Total pure abstraction.

```
1  /*@ assignable \nothing; @*/
2  native /*@ nullable @*/ T partialPure_T();
```

Listing 4: Partial pure abstraction

```
1  /*@ assignable FieldSet; @*/
2  native T impure_T();
```

Listing 5: Impure abstraction

Fig. 5.5: The constructed three kinds of abstractions. `T` represents an appropriate type required by the original statement, and the pure method returns an unspecified value of `T` which includes `null` as well. The Java keyword `native` is used to avoid implementations of the methods. The JML **`assignable \nothing`** clause denotes that the annotated method has not modified heap locations, but may have allocated objects; The **`assignable \strictly_nothing`** denotes the annotated method neither changed heap locations nor created objects; **`assignable`** `FieldSet` denotes the a collection `FieldSet` of fields may have been changed by the annotated method.

for the abstract programs. Furthermore, symbolic execution of an abstract

$\mathbf{R}_1$: $\mathcal{A}$⟦`T v = e;`⟧	$\rightarrow$	`T v = totalPure_T();`
$\mathbf{R}_2$: $\mathcal{A}$⟦`v = e;`⟧	$\rightarrow$	`v = totalPure_T();`
$\mathbf{R}_3$: $\mathcal{A}$⟦`e.f = e;`⟧	$\rightarrow$	`e.f = totalPure_T();`
$\mathbf{R}_4$: $\mathcal{A}$⟦**`if`** `(e)`⟧	$\rightarrow$	**`if`** `(totalPure_T())`
$\mathbf{R}_5$: $\mathcal{A}$⟦**`while`** `(e)`⟧	$\rightarrow$	**`while`** `(totalPure_T())`
$\mathbf{R}_6$: $\mathcal{A}$⟦**`return`** `e;`⟧	$\rightarrow$	**`return`** `totalPure_T();`
$\mathbf{R}_7$: $\mathcal{A}$⟦`T o = new T(e);`⟧	$\rightarrow$	`T o = partialPure_T();`
$\mathbf{R}_8$: $\mathcal{A}$⟦`T a = new T[k];`⟧	$\rightarrow$	`T a = partialPure_T();`
$\mathbf{R}_9$: $\mathcal{A}$⟦`v = invokeFun(e);`⟧	$\rightarrow$	`v = impure_T(e);`

Fig. 5.6: The rules used to process the *mayHave* statements using the abstractions shown in Fig. 5.5. The transformation is denoted by $\mathcal{A}$. These rules are based on the assumption that the expression `e` is side-effect free. The arrow $\rightarrow$ should be read as: the *mayHave* statement on the left-hand side is transformed into the statement on the right-hand side.

statement may require more rules due to the applications of *case split* proof rule. Therefore, we provide optimizations of abstractions as follow.

Collapsing a group of consecutive mayHave *statements.* When possible, we abstract a group S^- of *mayHave* statements into one single statement, thus reducing the number of abstract statements. This is available for any two nodes m and n in the verification graph, where m dominates [3] n, n post-dominates [4] m, and *all* the statements in S^- whose edges are in the paths from m to n are *mayHave* statements. The new abstract statement invokes the `impure_T` method shown in Listing 5.

Keeping helpful details of the code. We treat the statements that are unnecessary for bounded program verification, yet helpful for the deductive program verification, as *mustHave* statements in the abstract program. An assignment statement with a right-hand expression such as object allocation, numbers, and constants, is considered to be *mustHave* if another *mustHave* statement uses its defined variable.

5.2.5 Handling Runtime Exceptions

Typically, whenever a *functional* property—provided by **`ensures`** clause—is proved to hold for the analyzed program, the *built-in* property that no runtime exception is thrown holds as well. Thus, when more than one *functional* property to be verified, the proof steps for checking runtime exceptions have to be redone. We handle the built-in property and the functional property separately to ease the verification progress, and assist users to concentrate on discovering useful auxiliary specifications for *functional* properties. We first check whether the built-in property holds in the analyzed program, and then verify whether the program fulfills the functional property with the assumption that no exception is thrown in the program. In the latter verification a statement `v = o.f`, for example, is translated into $o \neq null \wedge o' = o.f$, rather than $(o \neq null \implies o' = o.f) \vee (o = null \implies exc)$, where exc denotes that a runtime exception is thrown. In the former verification, we suppose the functional specifications are trivially satisfied, e.g., **`requires true;`** and **`ensures true;`**. We inject guards into the code, such that if a guard passes an exception is thrown. Furthermore, we treat the possible exception types separately. That is, we aim to provide a single semantic slice for each exception type.

For handling runtime exceptions, the code in Fig. 5.1(a) is transformed into the code in Listing 6. We insert a guard (Lines 20–23) which sets to true the flag *NASE* in the class *RTE* if a `NegativeArraySizeException` is about to be thrown (Line 24). Thus, when the program in Listing 6 preserves the

[3] In a cyclic verification graph, a node m dominates a node n if every path from the entry node to node n has to pass through node m.

[4] In a cyclic verification graph, a node m post-dominates a node n if every path from node n to the exit node has to pass through node m.

value of this exception flag, no `NegativeArraySizeException` is thrown in the original program, as the guard is checking the statement at line 20. All program parts not relevant to whether the exception is thrown are abstracted. In our approach, when there is no runtime exception and the functional properties have been fulfilled by the analyzed abstract programs, the original program is also verified. The decomposition of handling runtime exceptions from functional properties breaks a complex specification into a conjunction of partial specifications, thus eases the burden of program verification.

```
1  /*@ requires RTE.NASE = false;
2    @ ensures RTE.NASE = false;
3    @ diverges true;
4    @ assignable \everything;*/
5  int numberOfPrime(int x, int y) {
6    int size = 0;
7    for(int i=x; i<y; i++){
8      boolean isPrime = true;
9      for(int j=2; j<i; j++){
10       if(i%j==0){
11         isPrime = false;
12         break;
13       }
14     }
15     if(isPrime){
16       size++;
17     }
18   }
19   if (size>0){
20     if (y-x < 0) {
21       RTE.NASE = true;
22       return;
23     }
24     this.a = new int[y-x];
25   }
26   return size;
27 }
```

Listing 6: Code with injected guards

5.3 Evaluation

The verification-based program slicing technique that we have presented (i) liberates verification engineers from finding the relevant program slices manually, (ii) reduces the proof complexity especially for partial properties, for which most of the program slices are irrelevant, and (iii) assists the developers to understand their program behaviors according to the properties of interest.

We have implemented our technique in the prototype tool *AbstractJ*. We use KeY as the deductive program verification tool. Recall that the KeY system performs symbolic execution of sequential Java programs, using various proof rules. Program verification with KeY is usually done in auto-active style: the user interacts with the system only through provided auxiliary specifications, while the proof result is obtained automatically. The number of rule applications is our primary measure of proof complexity. We have used 5 benchmark programs, all taken from the related program verification literature and the KeY repository. Each program has 2 to 6 partial properties to be verified. We have also considered two other approaches to evaluate the effectiveness of our approach (*abstraction*) in program verification. One approach, *baseline*, proves the original programs using KeY as usual. The other approach, *highlight*, is similar to the *abstraction* approach, but it only highlights the relevant program statements and retains the irrelevant statements rather than abstracting them. We have completed 21 verification tasks using each approach, and in total we have completed 63 ($= 21 * 3$) verification tasks in our experiments. We have written the auxiliary specifications as compact as possible and measured the auxiliary specifications as the number of the operands of JML expressions, JML constructs, and logical connectors, e.g., **`loop_invariant`**, **`assignable`**, **`forall`**, `&&`, etc.[5] We used the SMT solver Z3 to compute the unsat cores. All experiments have been performed on an Intel Core i5-2520M CPU with 2.50 GHz running on a 64-bit Linux.

To evaluate the effect of the *abstraction* approach on reducing the complexity of programs, we have compared the number of Java statements of the original and abstract programs. The results are shown in Table 5.1. The column *method* shows the Java class and its entry method to be verified; the verified properties are listed in the column *properties* where the *nullPointer*, *indexBounds*, and *negSize* represent the runtime exceptions `NullPointer`, `ArrayIndexOutOfBounds`, and `NegativeArraySize`, respectively. The other properties are functional ones; following is a explanation.

- `List.merge(list)`. When the `merge` method returns normally, the receiver list contains less or equal amount of elements than before invoking the method (denoted by *leElems*), and all elements in the receiver list were in the old receiver list or exist in the `list` parameter (denoted by *subset*).
- `Map.put(key,value)`. When the `put` method returns normally, the property *oldKey* denotes that the receiver map contains `value` if it has the `key`, the *sameValues* property denotes that the `put` method did not modify the map entries whose keys are distinct from the input `key` parameter, and the property *kvMatched* means that wherever the `key` is stored in

[5] Different engineers may write different auxiliary specifications for the same programs. We have asked an experienced KeY engineer to prove the original programs and a relatively inexperienced KeY user to prove the abstract programs. They carefully inspected and ensured that the specifications are compact enough concerning the requirement specifications.

Table 5.1: Results of deductive program verification with abstractions

method	properties	bounds	origin stmts	baseline		highlight		abstraction		
				specs	rules	specs	rules	stmts	specs	rules
List. merge(list)	nullPointer	(3,3,3)	27	22	3578	12	3046	4	0	196
	indexBounds	(3,3,3)	43	59	4641	46	4434	33	46	3717
	negSize	(3,3,3)	31	13	4316	14	2723	16	6	1188
	leElems	(3,3,3)	22	14	2962	14	2962	13	6	1715
	subset	(3,3,3)	22	82	6299	56	5715	15	52	4404
Map. put(key,value)	nullPointer	(4,4,4)	32	28	4485	14	3780	9	0	512
	indexBounds	(4,4,4)	48	61	6154	54	5557	48	54	5488
	negSize	(4,4,4)	32	17	4084	12	3753	16	0	654
	oldKey	(4,4,4)	26	30	4295	30	4295	11	22	1725
	sameValues	(4,4,4)	26	27	9823	34	8494	12	26	4647
	kvMatched	(4,4,4)	26	50	7327	50	7327	26	50	8814
LRS. doLRS()	nullPointer	(4,4,3)	39	11	3022	8	2818	9	0	673
	indexBounds	(4,4,3)	43	44	5006	14	4545	30	14	4502
	posLen	(4,4,3)	26	32	4155	14	2908	17	10	1255
Set. intersect(set)	nullPointer	(3,4,4)	48	23	10937	18	10226	22	4	4124
	negSize	(3,4,4)	38	17	14555	14	9963	7	0	899
	indexBounds	(3,4,4)	58	57	19715	33	12287	51	33	6714
	emptySet	(3,4,4)	33	94	64807	46	13557	14	21	3001
	subset	(3,4,4)	33	142	RO	60	136225	16	52	11211
Graph. remove(nodes)	sameNodes	(3,3,3)	54	78	RO	60	14985	6	0	923
	sameEdges	(3,3,3)	54	119	RO	83	RO	18	67	12334

`keys` array, the corresponding value in the associated `values` array is the input `value`.

- `LRS.doLRS()`: The method `doLRS` searches the longest repeated sub-string from an arbitrary string. Property `posLen` denotes that if such a sub-string is found, the length of the sub-string should be greater than 0.
- `Set.intersect(set)`. The method `intersect` returns the intersections of two sets. Property `emptySet` denotes that the returned set is empty if any input set is empty, and property `subset` denotes that the returned set is subset of any of input sets.
- `Graph.remove(nodes)`. The method `remove` recursively removes `nodes` from a graph. Properties `sameNodes` and `sameEdges` denote that any node not being in `nodes` and any edge whose nodes are not in `nodes` will remain unchanged when the `remove` method returns normally.

The *bounds* column shows the class and loop bounds that we used in the bounded program verification tool. In particular, the bounds for each run of bounded verification is denoted by a ternary (x, y, z) where x, y, and z denote the class bounds, the size of the integer (i.e., bitwidth), and the loop bounds. For each experiment, the abstract program has been generated in less than 10 seconds. The *origin stmts* column displays the number of the original program statements. The injected guard statements (see Section 5.2.5) are treated as original statements when handling runtime exceptions. The column *stmts*

shows the number of program statements that have been generated by the *abstraction* approach. On average, 49.5% (median 50%, maximum 85.2%) of statements in the original programs have been abstracted by the *abstraction* approach. For 2 properties (`indexBounds`, and `kvMatched` the method `put`) the approach *abstraction* seems to has no effect since the abstract and original programs have same lines of code. A careful inspection reveals that one single concrete statement is abstracted. From the results it can be seen, the abstract programs contain less, yet enough details for partial properties. The more partial the verified property, the fewer details the abstract programs have. Conservatively speaking, even in the case where the abstract programs are identical to the original programs, the *abstraction* approach assists verification engineers at exploring the relevant statements—all program statements that have not been abstracted are necessary for the properties under consideration. The *highlight* approach shows to the verification engineers the relevant program statements, while the *abstraction* approach provides additional benefits: (i) automatic generation of auxiliary specifications for the irrelevant program statements, and (ii) *possible* reduction of proof complexity for partial properties. Besides, the *abstraction* can increase users confidence in the correctness of their programs, before starting deductive verification.

For a fair comparison of the amount of manually written auxiliary specifications, the *highlight* approach reused the auxiliary specifications that have been written manually in the *abstraction* approach (shown in the column *specs* of the column *abstraction* in Table 5.1). The *abstraction* approach generates specifications for the unnecessary program slices, however, the *highlight* approach does not generate specifications and thus the verification engineers need to write specifications for the unnecessary slices. On average, 37.2% (median 26.7%) of specifications for the highlighted programs have been automatically generated by the *abstraction* approach.

All properties in the table have been proved using the *abstraction* approach for the chosen bounds. When using the approaches *highlight* and *baseline*, several properties are improvable. The column *rules* provides the number of rule applications. Any rule application beyond our threshold of 2000000[6] is denoted by *RO*. For 18 properties that have been proved by all approaches, the *abstraction* approach needed only 50.1% (median 55.2%) of the rules required by the *highlight* approach. It is not guaranteed that the *abstraction* approach requires fewer rule applications than the other two approaches for arbitrary properties. Besides the reasons explained in Section 5.2.3, KeY creates branches for each abstract statement, to check its pre-/post-conditions.[7] The *abstraction* approach requires fewer rules than other approaches, assuming they use same auxiliary specifications, only when the rule cost introduced by the abstract statements is lower than the cost of symbolic execution of the irrelevant

---

[6] The time cost and memory consumption grow exponentially based on the rule applications. It required ∼30 min and more than 4 GB memory for 2000000 rules.

[7] The trivial pre-/post-conditions of each abstract statement requires ∼20-100 rules.

original statements. In other words, the more partial the verified property, the less complex is the proof of the abstract programs. The property *kvMatched* is an example for a less partial property.

Although we used small bounds for AbstractJ in the experiments, all properties have been proved using our approach. On the other hand, the verification engineers are free to provide larger bounds for AbstractJ. Given the same input formula, Z3 may find an unsat core that is different from the core found by other SMT solvers. AbstractJ may generate different abstract programs using other SMT solver, but the abstract programs will still expose to the verification engineers the relevant program parts for the desired property.

5.4 Related Work

Several methods have been proposed to split the program under analysis for particular concerns. Traditional program slicing techniques (e.g., static/dynamic slicing) generate a group of accessible statements (a slice) concerning variables of interest at particular locations. Due to the complexity of the specification expressions and various complex data structures in the analyzed programs, it is challenging to find specification-sensitive slices correctly.

Conditioned slicing techniques [Fox et al., 2004; da Cruz et al., 2010; Chebaro et al., 2012; Comuzzi and Hart, 1996; Chung et al., 2001; Barros et al., 2012] have been widely applied to simplify programs with respect to the specifications. Comuzzi et al. [Comuzzi and Hart, 1996] introduced predicates as a slicing criterion; the slice contains the statements affecting the predicates. That idea has been extended by introducing preconditions [Chung et al., 2001], symbolic execution [Barros et al., 2012], and program verification [da Cruz et al., 2010] into conditioned slicing techniques. Typically, conditioned slicing produces a program slice based on the specification using the symbolic execution with the inputs generated by a solver. The pre-/post-conditions (generally formulas of first-order logic) are expressed in terms of the (input) variables at program locations of interest. However, intensive human interaction is required to guide the symbolic execution by choosing a suitable criterion. Moreover, when a conditioned slice is not proved, it is not clear whether the original program is incorrect or the slice overestimates the program. For program comprehension, GamaSlicer [da Cruz et al., 2010] verifies the program with respect to specifications before generating semantic-based slices. Nevertheless, it may not terminate with a conclusive result since it targets an undecidable logic. Our approach ensures that the soundness of the proof depends only on the deductive verification. Besides, we do not remove the statements that are not in the slice but abstract them. This way it is guaranteed that if the abstracted program fulfills the specification so does the original program. Finally, these approaches cannot handle programs with complex data structures and specifications.

The following three approaches tried to improve the verification process using bounded analysis. Bormer et al. [Bormer, 2014] claim that verifying programs using the bounded model checker LLBMC [Merz et al., 2012] facilitates proving with VCC [Cohen et al., 2009]. Annotations written in VCC's specification language are translated into assertions that can be checked by LLBMC. Unlike our approach, it still requires the user to write auxiliary specifications for the unnecessary statements. We believe that our approach can complement Bormer et al.'s approach. The authors of [Ghazi et al., 2014] try to verify Alloy programs using deductive verification, after the Alloy analyzer [Jackson, 2012]—based on bounded analysis, fails in finding a counterexample in bounds dictated by the machine. Donaldson et al. [Donaldson et al., 2011] combine k-induction and inductive invariant method to facilitate program verification using significantly weaker annotations. These above approaches do not claim to reduce the overhead of writing specifications. However, the k-induction often allows using weaker loop invariants that are required by the inductive invariant approach. Unlike our approach that reduces annotation overhead for called methods and loops, Kroening's approach only reduces writing loop invariants overhead. Our approach can reduce the burden of specifications not only for loops. Our approach helps software developers to understand their code with respect to the requirement specification and provides abstract programs that contains less details.

5.5 Conclusion

Deductive program verification systems typically require experienced verification engineers to write auxiliary specifications, e.g., loop invariant and method contracts. To discover useful auxiliary specifications that are fulfilled by the sub-routines and also meet the requirements of the calling procedures, the engineers have to identify the parts of the program that are relevant to the intended property. It is very likely that such program parts are challenging to find regarding a property (called a *partial property*) that is only relevant to small parts of the program.

We provide a verification-based program slicing technique to construct a semantic slice (an abstract program) with respect to a partial property. In the abstract program, the program parts that are irrelevant to the partial property are replaced by an abstraction (i.e., they are not completely removed), whereas the rest of the program (i.e., the relevant parts) remains unchanged. In contrast to verifying the whole program, verifying slices requires less auxiliary specifications (as the abstractions have less details), and their correctness—by their construction—implies the correctness of the original program concerning the partial property. Therefore, our technique liberates the verification engineers from identifying the relevant slice and eases the deductive verification progress: fewer proof steps are needed. The construction of the slice is based on the bounded program verification. If a property holds for the

analyzed program in the bounded verification, we extract the relevant code to the property using the *unsatisfiable core* (unsat core) provided by the SMT solver. We construct the abstractions for the relevant code and finally build the semantic slice by transforming the irrelevant code into abstract statements with the abstractions.

We have implemented our technique in a prototype tool, AbstractJ, and performed several experiments to evaluate the benefits of using our technique. The results show that 50% of the user's workload in writing auxiliary specifications were taken off using our technique compared to proving programs as usual.

If the abstract program is not proved, the analysis terminates with no conclusive answer since the slice may be too abstract thus deductive verification is impossible. To find the root causes of the failed proof is, unfortunately, unusually difficult since the scope of inspection on the failed proof is larger than usual. To ease the burden of inspecting the failed proof, we provide an algorithm that uses counterexamples to refine the abstractions. We present the algorithm in the next chapter (Chapter 6).

CHAPTER 6

Counterexample-Guided Abstraction Refinement for Deductive Program Verification

An abstraction-based deductive program verification has been explored in the previous chapter, in this chapter we present its refinement techniques. The verification-based program slicing technique presented in Chapter 5 constructs a semantic slice of the analyzed program with respect to the desired specifications (and also the scope of analysis). If the slice with required auxiliary specifications is proved in deductive program verification, the specifications are fulfilled by the original program as well. Otherwise, the correctness of the original program is vague. The deductive verification may fail in two major cases: i) the user-provided annotations are not consistent with the program or the desired specifications, i.e., the annotations do not hold for the program or do not satisfies the specifications, and ii) the slice may be too abstract and thus deductive program verification is not possible. The burden of inspecting the failed proof is considerable in both cases. Besides, each abstraction of the slice corresponds to a collection of implementations that are irrelevant to the specifications, and then in the second case it is obscure to manually renovate the failed proof, i.e., to only reveal the relevant program parts that have been replaced by abstractions.

Counterexample-Guided Abstraction Refinement (CEGAR) [Kurshan, 1994] methodology provides an automatic framework that gradually refines abstract models of a system. Many code analysis techniques, e.g., [Lind-Nielsen and Andersen, 1999; Clarke et al., 2000, 2003; Ball et al., 2004; Chaki et al., 2004; Clarke et al., 2005; Beyer et al., 2007; Gupta et al., 2011; Abdulla et al., 2016], have instantiated the CEGAR framework for gaining an efficient code analysis. All instances of this framework, however, construct (and refine)

abstractions of the code at the predicate level. That is, the predicates of the code that are irrelevant to the property of interest are replaced by abstractions. The rests of the code (e.g., assignment statements) remains unchanged, even though they are not relevant to the property at all. It is very likely that writing annotations for such coarse abstracted programs are very hard. Moreover, they do not handle the properties of complex data structures that our algorithm targets. They have been used to handle properties of a finite state machine or to check access violation errors such as null-pointer dereference and array access out of bound.

We present an algorithm instantiating the CEGAR framework for deductive program verification. Starting with an initial abstract program, we iteratively refine the abstractions of the program based on its counterexamples that are found by the deductive verification. The novelty of our algorithm is that it handles properties of complex data structures and refines the abstractions at the statement level. That is, for each abstraction that replaces a collection S of program statements in the abstract program, in the new abstract program the abstraction will be replaced by: i) the statements $S^+ \subseteq S$ that are relevant to the specifications concerning the counterexamples, and ii) new abstractions that are constructed for the rest of S (i.e., the irrelevant parts $S^- = S \backslash S^+$). The on-demand iterative nature of the algorithm guarantees that only as much information about the program will be analyzed as is necessary to check the property of interest. Therefore, our algorithm eases the burden of manually renovating the failed proofs in deductive program verification. The core idea is using bounded program verification to guide the refinement of the abstractions. Figure 6.1 shows the structure of our algorithm. The fundamental algorithm is as follows.

1. **Initial abstractions.** To verify a program P with respect to a property Q, construct an abstract program $A(P)$ that overestimates P by replacing the parts of P that are irrelevant to Q by an abstraction.
2. **Checking auxiliary specifications.** Check whether the provided auxiliary specifications I hold in $A(P)$ using bounded program verification. If a counterexample is found, the inspection of I is needed. Otherwise, go to step 3.
3. **Proving.** Prove the abstract program $A(P)$ with bounded-verified auxiliary specifications I in deductive program verification. If $A(P)$ is proved, the overall analysis terminates, and program P satisfies the property Q as well (by the construction of the abstract program). Otherwise, a counterexample c to $A(P)$ is found, and thus go to step 4.
4. **Checking validity of counterexamples.** Check whether the counterexample c is valid for the original program P. If c is valid, a fault of P has been discovered, and the analysis terminates. Otherwise, go to step 5.
5. **Refinement.** Refine $A(P)$ so that the spurious counterexample c is eliminated, and thus go to step 2.

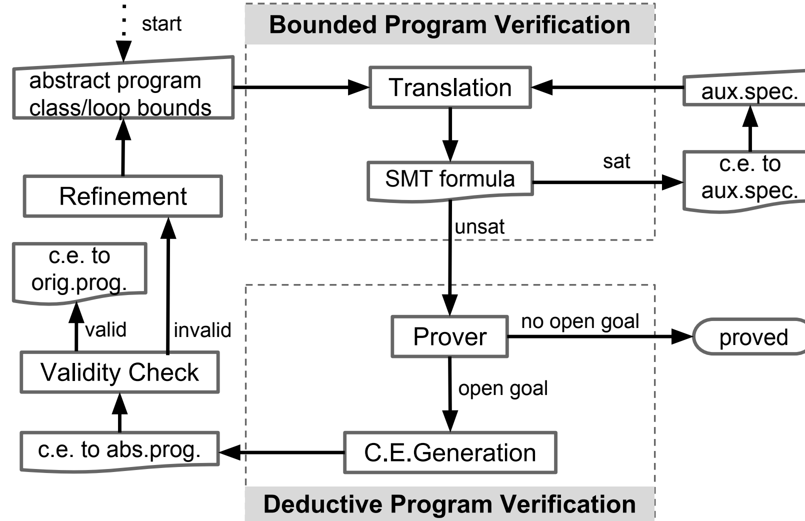


Fig. 6.1: Structure of our algorithm.

This algorithm forms the basis of all CEGAR-based techniques. Our algorithm, however, differs from all the existing ones in that the abstract program is not necessary a finite-state program. That is, $A(P)$ is an abstract program where loops and recursions are allowed. Bounded program verification is fully automatic and thus is heavily used to facilitates the verification engineers to renovate the program and the annotations. We use bounded program verification to check whether the property Q holds for the program P at the step 2, and to check whether the auxiliary specifications I hold for the program P and also fulfill the property Q at the step 5. Therefore, we guarantee an initial confidence in the correctness of the abstract program and auxiliary specifications with respect to the desired property, and thus avoid unnecessary attempts in deductive verification and reduces the scope of inspection on a failed proof.

Our algorithm ensures that it is totally dependent the deductive program verification to guarantee the correctness of the analyzed program. The bounded program verification techniques improve the process of deductive program verification, but they are not responsible for providing the certification of the final program quality. Besides, bounded program verification techniques are usually not capable of checking the properties such as the termination of the analyzed program (see Definition 4.1). [1] For example, the refinements of the abstractions are guided by the counterexamples provided by the deductive program verification. When a counterexample causes the

[1] The calculus presented in Chapter 4 is capable of checking the termination of the analyzed program regarding the class bounds, however, its logic is undecidable and thus it does not guarantee to give conclusive answers.

analyzed program non-terminating, the algorithm can not check the validity of the counterexample due to the statically loop unrolling. Therefore, the premise of our algorithm is that the analyzed program total-terminates, i.e., the program terminates for all inputs.

6.1 Our algorithm

We describe the steps of our algorithm in the section. We mainly describe how we check auxiliary specifications using bounded program verification, check validity of the counterexamples, and refine abstractions based on the counterexamples. Besides, we show the counterexample generation in the KeY system. Details of programs and specifications that our algorithm supports can be found in Chapter 3.

6.1.1 Initial Abstractions

The initial abstractions substitute the parts of the analyzed program that are irrelevant to the desired property. Our algorithm does not rely on a specific program abstraction technique to construct the initial abstractions. The abstractions of the code can be constructed at different levels, e.g., at the function level [Taghdiri and Jackson, 2007], at the predicate level [Abdulla et al., 2016], or the statement level (see Chapter 5). However, to obtain an efficient CEGAR instantiation, we use the verification-based program slicing technique to construct the initial abstractions at the statement level since the technique also supports the predicate and function levels and thus constructs more accurate abstractions.

Given a program P annotated by a specification (a property Q), the verification-based program slicing technique constructs an abstract program $A_B(P)$ from program P with respect to the scope of analysis B (designated by a collection of class/loop bounds). In the construction of the program $A_B(P)$, the parts of program P (e.g., the method calls, the predicates, and the expressions on the right-hand side of the assignment statements) are transformed into abstractions if they are irrelevant to the property Q, whereas the rest of the program (i.e., the relevant parts) is trivially replicated in $A_B(P)$. Recall that the abstractions are constructed based on a particular bounded proof provided by bounded program verification. The program $A_B(P)$ may exclude the code that becomes relevant when the scope of the analysis is larger than B, and thus it is not possible to be proved using deductive program verification. In the following sections, we present the techniques to recompute the scope of analysis and then to renovate failed proofs by refining the abstractions.

6.1.2 Checking Auxiliary Specifications

Before continuing with the deductive verification, we check whether the user-provided auxiliary specifications hold for their annotated program constructs

and also satisfy the expectations from their contexts regarding the specification of the analyzed method. Given an abstract program $A(P)$, a specification Q that annotates $A(P)$, and a user-provided auxiliary specification q that annotates a program construct p (e.g., a loop or a method call in $A(P)$), we produce an SMT formula and check the satisfiability of the formula using an SMT solver. Recall that the program $A(P)$ has been verified with respect to the specification Q using bounded program verification beforehand, [2] If a model to the formula is found (i.e., the formula is satisfiable), then the specification q is not consistent with program $A(P)$ or specification Q. That is, the specification q either does not hold for its annotated program construct p: q may overestimate the behavior of p, or does not fulfill the expectations from the specification Q: q may underestimate the behavior of p. This two possible cases can be trivially distinguished by analyzing the model to the formula. If no model is found (i.e., the formula is unsatisfiable), then the specification q is verified with respect to the chosen scope of analysis.

We use the SMT encoding presented in Chapter 3 to encode the program and its specifications. We have handled checking method calls with method contracts and elaborated on the details in Section 3.1.1. In this section we will present the details of the SMT encoding for checking loop invariants in bounded program verification.

Deductive program verification techniques typically translate a program and its specifications into a logical formula in particular logic, e.g., Java Dynamic Logic (JavaDL; see Section 2.4), and apply various proof rules on the formula to construct the verification conditions using the symbolic execution. Unlike bounded program verification techniques that construct a finite-state program by unrolling loops (and recursions) regarding the loop bounds and thus provide a scope-bounded proof, deductive program verification techniques require loop invariants and provide a proof that is valid for any loop iterations.

A loop invariant denotes a property that holds for its annotated loop and also fulfills the expectations of the program construct where it is included. For example, we add a loop invariant for the abstract program [3] of the Fig. 6.2(a). This loop invariant (Line 7) has to be satisfied by its annotated loop (Line 11) at some specific program points (will be talked soon) and also has to satisfy the expectations from the `numberOfPrimes` method regarding to its postcondition (Line 1). If any of these two requirements are not met, the deductive verification fails and the verification engineers have to find a reason to it—inspecting (and renovating) failed proofs is a complicated and error-prone effort.

[2] The verification-based program slicing technique constructs the abstractions based on a bounded proof of the analyzed program that is provided by the bounded program verification.

[3] The abstract program of Fig. 6.2(a) is constructed from the program of Fig. 5.1(a).

Checking whether a loop invariant holds for its annotated loop

When the symbolic execution reaches a loop in deductive program verification, the loop invariant rule shown in Definition 6.1 is used for the loop. The rule denotes that a loop can be replaced by its loop invariant (and its negated loop condition) if the loop invariant is satisfied at the following program points.

1. on entry into the loop,
2. immediately before each execution of the loop body,
3. immediately after each execution of the loop body,
4. on exiting from the loop.

When the loop condition expression has no side-effect, we omit the loop invariant checking at two kinds of program points. The program point (2) is skipped since the loop invariant obviously holds if it holds at program points (1) or (3). Besides, we exclude the program point (3) when it is on the last loop iteration since the loop invariant will be checked on exiting from the loop.

Definition 6.1. *A loop invariant inv_l holds for a* `while` *loop l if*

$$\frac{\{Cond \wedge inv_l\} \quad \texttt{stmts} \quad \{inv_l\}}{\{inv_l\} \quad \texttt{while(Cond) stmts} \quad \{\neg Cond \wedge inv_l\}}$$

We treat the loop invariant as a property that has to be satisfied at three kinds of program points, those that are denoted by i), iii), and iv) above. Given a loop invariant inv_l for a loop l, we add JML **`assert`** $\neg inv_l$`;` annotation statements (see Section 3.1.1) at these program points in the abstract program and construct an acyclic verification graph from the abstract program. Figure 6.2(b) shows an acyclic verification for an abstract program. the first four and the last one highlighted expressions are used to check whether the loop invariant is satisfied by the loop. Note that the second and third highlighted expressions can be omitted since the loop condition expression is side-effect free. We keep them in the graph to facilitate illustration.

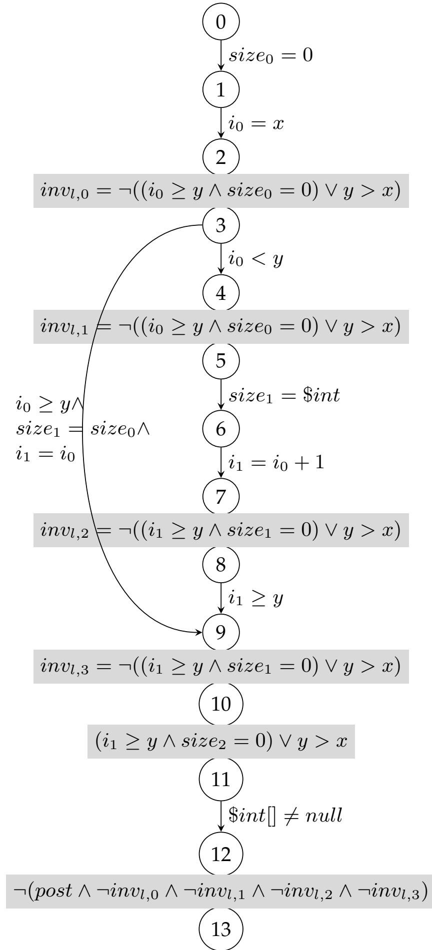
The JML **`assignable`** `FieldSet;` clause denotes the variables or the fields that may been changed by its annotated program construct. To check whether the loop satisfies the **`assignable`** clause, we trivially check whether the variables or the fields referred by `FieldSet` have been renamed by the loop in the verification graph. In the verification graph of Fig. 6.2(b) we check whether the `size` variable is renamed on the paths [3, 9] or [3, 4, 5, 6, 7, 8, 9] regarding the **`assignable`** `size;` clause (Fig 6.2(a), Line 9). Figure 6.3 shows the formula fragment for checking the loop invariant. Except the second to last formula, the formulas are for checking whether the loop invariant holds for the loop.

Checking whether a loop invariant fulfills the expectations of its context

If a loop satisfies its loop invariant, the loop invariant specifies a condition on the program state after the execution of the loop. In deductive program

```
1  /*@ ensures \result>0 ==> y>x;
2    @ diverges true;
3    @ assignable \everything;
4    @*/
5  int numberOfPrime(int x,int y){
6   int size = 0;
7   /*@ loop_invariant
8     @ (i>=y&&size==0)||y>x;
9     @ assignable size;
10    @*/
11  for(int i=x; i<y; i++){
12   size = pure_int();
13  }
14  pure_allocArrayInt();
15  return size;
16 }
17 /*@ assignable
18   @ \strictly_nothing;
19   @*/
20 native int pure_int();
21 //@ assignable \nothing;
22 native int[] pure_intArray();
```

(a) Abstract program



(b) Acyclic verification graph

Fig. 6.2: The acyclic verification graph (loop bound is 1) for the abstract program with a loop invariant. If the loop invariant is violated, the Boolean variable inv_l evaluates to $true$, otherwise $false$. The variables $\$int$ and $\$int[]$ denote the unspecified results of the methods `pure_int` (Fig. 6.2(a), Line 21) and `pure_intArray` (Fig. 6.2(a), Line 22), respectively. For readability, in the last highlighted expression we use $post$ to represent the postcondition of the `numberOfPrime` method.

verification the symbolic execution of the code following the loop continue with this program state, and the concrete program behavior of the loop will be ignored. Therefore, the concrete program behavior of the loop is replaced

$$
\begin{aligned}
E_{2,3} &\implies E_{3,4} \vee E_{3,9} \vee inv_{l,0} \\
E_{4,5} &\implies E_{5,6} \vee inv_{l,1} \\
E_{7,8} &\implies E_{8,9} \vee inv_{l,2} \\
E_{9,10} &\implies E_{10,11} \vee inv_{l,3} \\
E_{2,3} &\implies inv_{l,0} = \neg((i_1 \geq y \wedge size_1 = 0) \vee y > x) \\
E_{4,5} &\implies inv_{l,1} = \neg((i_1 \geq y \wedge size_1 = 0) \vee y > x) \\
E_{7,8} &\implies inv_{l,2} = \neg(i_1 \geq y \wedge size_1 = 0) \vee y > x) \\
E_{9,10} &\implies inv_{l,3} = \neg((i_1 \geq y \wedge size_1 = 0) \vee y > x) \\
E_{10,11} &\implies (i_1 \geq y \wedge size_2 = 0) \vee y > x \\
E_{12,13} &\implies \neg((size_2 > 0 \implies y > x) \wedge \neg inv_{l,0} \wedge \neg inv_{l,1} \wedge \neg inv_{l,2} \wedge \neg inv_{l,3})
\end{aligned}
$$

Fig. 6.3: Formula fragment for checking the loop invariant of Fig. 6.2(a) using bounded program verification. If the loop invariant for loop l is violated at a program point i, the Boolean variable $inv_{l,i}$ evaluates to $true$, otherwise $false$. The second to last formula is for checking whether the loop invariant satisfies the expectations from its context regarding the postcondition of the `numberOfPrimes` method. Except the second to last formula, the formulas are for checking whether the loop invariant holds for the loop.

by the loop invariant. It may happen that a method with a loop satisfies its method contracts, but the deductive verification fails to prove that since the loop invariant of the loop does not provide sufficient property. To check whether a loop invariant satisfies the method contracts, we add a JML **`assume`** inv_l`;` annotation statement immediately after the last **`assert`** statement of loop l. To avoid using the concrete loop behavior in bounded program verification, in the expression of loop invariant condition the variables (and the fields) that are annotated by the **`assignable`** clause are renamed, i.e., get unspecified values. In the verification graph of Fig. 6.2, the fifth highlighted expression is used to check whether the loop invariant fulfills the postcondition (Fig. 6.2(a), Line 7) of `numberOfPrime` method. The variable `size` is renamed in the **`assume`** statement it is annotated in the **`assignable`** `size;` clause (Fig 6.2(a), Line 9). The second to last formula in Fig. 6.3 is for checking whether the loop invariant satisfies the expectations from its context regarding the postcondition of the `numberOfPrimes` method.

6.1.3 Proving

The KeY system (see Section 2.4) is used as the deductive program verification system in our algorithm. Recall that KeY translates the generated abstract program and the property of interest into formulas in JavaDL and applies various proving rules on the formulas in the symbolic execution [King, 1976]. In contrast to proof obligation generators, which usually produce very large

first-order logic problems from programs/specifications and try to discharge them using fully automated off-the-shelf theorem provers, all reasoning is completely integrated into the KeY prover.

When a proof goal cannot be closed, KeY tries to generate a counterexample to the JavaDL formula in that goal. The formula is negated and translated into SMT formula. The JavaDL constructs, e.g., heaps, locations sets, and sequences, are translated into SMT definitions using semantic blasting [Herda, 2014]. Bounded SMT sorts are used in the SMT formula to guarantee decidability of the formula. The SMT formula is handed over to an SMT solver which tries to find a model for it. If the SMT solver succeeds in finding a model, the counterexample generator presents it in a readable form, showing the value of each constant and the contents of all heaps, location sets and sequences.

Let $A(P)$ be an abstract program, m be the analyzed method of $A(P)$, ce be a counterexample generated by the deductive program verification for the program $A(P)$, thus ce provides following information:

- A program state $pre_{m,ce}$ before invoking method m, i.e., the pre-state of m. This state contains:
  - a collection $args(pre_{m,ce})$ of arguments of m,
  - a collection $consts(pre_{m,ce})$ of constants,
  - a collection $objs(pre_{m,ce})$ of program objects on the heap,
  - a configuration of the objects on the heap.

  That is, for each object on the heap, its fields refer to other objects or constants, e.g., `null` and constant numbers.
- A program state $post_{ce}$ for which the proof fails, i.e., a property q is violated at state $post_{ce}$. The state contains:
  - a collection $consts(post_{ce})$ of constants,
  - a collection $objs(post_{ce})$ of program objects on the heap,
  - a configuration of the objects on the heap.
- A symbolic execution path $path(A(P), ce)$. It contains a sequence of program points that lead the program $A(P)$ to the program point q_{loc} where q evaluates to $false$ and the proof fails.

For each counterexample, KeY generates a test case for the analyzed program. When executing the program with the test case as the input, the above information can be obtained from the trace of the program execution. When the analyzed program does not terminate for the test case, our algorithm requires that the analyzed program has to total-terminate (see Definition 4.1).

6.1.4 Checking Validity of Counterexamples and Refinement

To check the validity of a counterexample, larger class bounds are required since the abstract program, the specification that annotates the program, and the auxiliary specifications are consistent (verified) for the old bounds. We compute new class bounds as shown in Definition 6.2.

Definition 6.2. *Let* $A(P)$ *be an abstract program,* ce *be a counterexample to* $A(P)$*,* C *be a class that is involved in* $A(P)$*,* $post_{ce}$ *be the program state where the proof fails,* $objs(post_{ce})$ *be a collection of program objects on the heap at state* $post_{ce}$*, thus the new class bound for* C *is the number of* C *objects in* $objs(post_{ce})$*.*

New loop bounds can be obtained directly by tracing the execution of the analyzed program with the generated test case. For other loops, the calculus presented in Chapter 4 is used to compute the new loop upper bounds based on the new class bounds. However, the calculus may return the 'unknown' due to its undecidable logic. If the calculus failed to compute a loop bound, we heuristically provide a loop bound. Instead of negating the loop condition on the last iteration (e.g., as shown in Fig. 3.2), to compute new loop unrolls, we transform a loop

`while(cond){stmts;}` to

`if(cond){stmts; if(!cond)var=var;}`,

where `var` is a variable that is modifiable in `stmts`. This transformation prevents unrolling the loops that are irrelevant for the program correctness. Our technique generates a new SMT formula that is the conjunction of the translation of the counterexample and the translation of the original program for the new bounds. When the formula is satisfiable, then either the counterexample is valid, or the loop requires further iterations if the loop condition is still `true` after traversing the last iteration. In the latter case, we double the loop bounds and repeat the validity check.

If the formula is unsatisfiable, we find the statements with respect to the counterexample using the verification-based program slicing technique as shown in Chapter 5. In the unrolled program, we highlight a *mayHave* statement as a *mustHave* statement (see Section 5.2.3) when the statement is in the newly found statements.

6.2 Evaluation

In this chapter we presented an algorithm instantiating the CEGAR framework for deductive program verification. The algorithm eases the burden of manually discovering useful auxiliary specifications and inspecting the failed proofs. In contrast to the program slicing technique presented in Chapter 5, our algorithm iteratively refines the abstractions and guarantees that only as much information about the program will be analyzed as is necessary to check the desired property.

We have implemented our algorithm in the prototype tool *RefineJ*. We use AbstractJ, our verification-based program slicing tool shown in the previous chapter—as the abstraction constructor and KeY as the deductive program verification system.

We have used benchmarks that are used in the experiment of AbstractJ (see Section 5.3). In this section, we investigate the cases when refinements are needed for the initial abstract programs, and evaluate the benefits of using our algorithm in the deductive verification. In total, we have proved 16 programs that were not proved before. We used the same environment as testing AbstractJ. That is, the SMT solver Z3 is used to compute the unsat cores, and all experiments have been performed on an Intel Core i5-2520M CPU with 2.50 GHz running on a 64-bit Linux. In this environment, each run of RefineJ can be completed within 10 seconds. For a fair evaluation of the benefits of using RefineJ in the deductive verification, when the auxiliary specifications that have been used in the experiment of AbstractJ are feasible for the refined programs, they are reused. When the existing auxiliary specifications are not feasible, we have written the auxiliary specifications as compact as possible. We use *abstraction* to denote the approach used in AbstractJ, *refinement* the refinement approach presented in this chapter, and *baseline* to denote proving the original programs as usual.

We pose the following research questions.
RQ1. How much effort required to refine the abstract programs?
RQ2. How *baseline* and the CEGAR instantiation compares?

RQ1. How much effort required to refine the abstract programs?

To evaluate the effort to refine abstract programs, we first find the cases when the abstract programs generated by the *abstraction* approach are not proved. The *abstraction* approach requires class bounds and loop bounds. We have carefully chosen the class bounds to guarantee that the specifications do not trivially evaluate to $false$ due to lacking necessary program objects. Based on the class bounds, we have computed the loop sharp upper bounds using the BoundJ tool (see Chapter 4). Therefore, the *abstraction* and *refinement* approaches have a complete code coverage in their analyses. The chosen (and computed) bounds are presented in Table 6.1. The *bounds* columns show the class and loop bounds that we used in *abstraction* and *refinement*. In particular, the bounds used in each verification task is denoted by a ternary tuple (x, y, z) where x, y, and z denote the class bounds, the size of the integer (i.e., bitwidth), and the loop bounds. The columns *methods* and *properties* denote the entry method of the analyzed program and its desired property, respectively. If the initial abstract programs generated by the *abstraction* approach are not proved, we apply refinements on the abstract programs. Otherwise, no refinements are required, and we denote these cases using the symbol X.

We observed that, the initial abstract program are needed to be refined in most of the verification tasks. From the result, ∼76.2% verification tasks require the process of abstraction refinements. In particular, out of the total 21 initial abstract programs, 16 programs require refinements to complete

Table 6.1: Results of deductive program verification with abstraction refinements

methods	properties	origin	baseline		abstraction				refinement			
		stmts	specs	rules	bounds	stmts	specs	rules	bounds	stmts	specs	rules
List. merge(list)	nullPointer	27	22	3578	(1,1,0)	4	0	196	(2,2,1)	4	0	196
	indexBounds	43	59	4641	(1,1,0)	8	0	195	(1,2,1)	21	13	2342
									(1,3,3)	26	37	2768
									(3,3,3)	33	46	3717
	negSize	31	13	4316	(1,1,0)	2	0	140	(1,2,1)	8	0	276
									(2,2,1)	16	6	1188
	leElems	22	14	2962	(3,2,1)	11	2	1809	(3,3,3)	13	6	1715
	subset	22	82	6299	(3,2,1)	13	10	1872	(3,3,3)	15	52	4404
Map. put(key,value)	nullPointer	32	28	4485	(1,2,1)	30	12	1779	(2,3,3)	30	4	512
	indexBounds	48	61	6154	(1,2,1)	23	6	1080	(2,3,3)	48	54	5488
	negSize	32	17	4084	(1,2,1)	15	0	1435	(2,3,3)	16	0	654
	oldKey	26	30	4295	(1,2,1)	11	0	582	(2,3,3)	11	22	1725
	sameValues	26	27	9823	(1,2,1)	10	0	582	(2,3,3)	12	26	4647
	kvMatched	26	50	7327	(1,2,1)	10	0	582	(2,3,3)	26	50	8814
LRS. doLRS()	nullPointer	39	11	3022	(1,3,3)	9	0	673	X	X	X	X
	indexBounds	43	44	5006	(1,3,3)	30	14	4502	X	X	X	X
	posLen	26	32	4155	(1,3,3)	15	0	RO	(3,3,3)	17	10	1255
Set. intersect(set)	nullPointer	48	48	10937	(1,2,1)	10	0	480	(3,2,1)	22	4	4124
	negSize	38	38	14555	(1,2,1)	7	0	456	(3,2,1)	7	0	899
	indexBounds	58	58	19715	(3,2,1)	25	16	RO	(3,3,3)	51	33	6714
	emptySet	33	94	64807	(3,2,1)	14	21	3001	X	X	X	X
	subset	33	142	RO	(3,2,1)	16	52	11211	X	X	X	X
Graph. remove(nodes)	sameNodes	54	78	RO	(3,2,1)	6	0	923	X	X	X	X
	sameEdges	54	119	RO	(3,2,1)	13	20	6060	(3,3,3)	18	67	12334

the verification tasks. [4] Besides, using *refinement* approach helps a lot with renovating failed proofs: with only a few of refinements the verification tasks can be completed. Out of the total 16 unproved initial abstract programs, only 3 programs require maximal 3 times of refinements, and the remaining 13 programs need the refinement only once.

In 19 (of 21) verification tasks, the code size in the abstract programs increases as applications of the refinement process increase and each refinement require verification engineers to provide more auxiliary specifications with respect to the newly revealed code. This observation meets our expectations of the refinements since we gradually refine the abstractions. There are also two exceptional cases due to the optimizations on the abstractions. (i) In verifying the method `List.merge(list)` against the `nullPointer` property, the initial and refined abstract programs have same code size. A careful inspection reveals that the *refinement* approach replaced the abstracted statements by the original ones. (ii) In verifying the method `Map.put(key, value)` against the `nullPointer` property, it requires less auxiliary specifications after the abstraction refinement. An inspection on the code reveals that in the

[4] The remaining 5 programs were trivially proved without refinements. It shows that our verification-based program slicing technique can reveal all the relevant program parts to the intended properties even with very small bounds.

initial abstract program a loop contains an allocation statement that requires a larger class bound, while in the refined abstract program that the allocation statement has been replaced by an abstraction since a larger class bound is computed and used in the refinement.

RQ2. How the CEGAR instantiation and the *baseline* approach compares?

For ease of comparison, the column *baseline* in Table 6.1 shows the same data as the column *baseline* in Table 5.1, i.e., the number of required auxiliary specifications and the number of used proof rules in KeY. We copy the data here for the purpose of comparison. From the results, the CEGAR instantiation gradually requires more auxiliary specifications in deductive program verification. Even counting the total number of required auxiliary specification, it still requires less specifications than using the *baseline* approach in some cases: 19 (of 21) verification tasks require less auxiliary specifications than using *baseline* in the deductive verification and 8 (of 21) tasks require zero auxiliary specifications. Moreover, we calculated the total number proof steps required to complete each verification task using our algorithm. The results show that, out of the total 21 verification tasks, 13 tasks require less proof rules than using the *baseline* approach.

In our experiments, before proving the abstract program with auxiliary specifications in KeY, we check whether the abstract program is consistent with the auxiliary specifications and also the desired specifications. All counterexamples provided by KeY denote that either the class bounds or the loop bounds are small and the newly computed bounds are still practical for our algorithm.

6.3 Related Work

Counterexample-guided abstraction refinement (CEGAR) framework iteratively refines abstract models of a system using counterexamples. It was first introduced by Kurshan [Kurshan, 1994], and then appeared in a number of analysis techniques (e.g., [Balarin and Sangiovanni-Vincentelli, 1993; Lind-Nielsen and Andersen, 1999; Clarke et al., 2000, 2003] that focus on checking finite state systems. Clarke, et al. [Clarke et al., 2000, 2003], for example, used CEGAR framework to check programs represented by labeled Kripke structures (a form of finite state machines) against properties expressed in temporal logic. The initial abstraction partitions the variables of the Kripke structure against the given property. If a spurious counterexample is found, the abstraction is refined by partitioning one of the equivalence classes. This technique has been implemented in NuSMV [Cimatti et al., 2002] and used to check a Fujitsu IP core design.

From this century CEGAR framework has been widely used in program verification (e.g., [Ball et al., 2004; Chaki et al., 2004; Clarke et al., 2005; Taghdiri

and Jackson, 2007; Beyer et al., 2007; Gupta et al., 2011; Abdulla et al., 2016]). To the best of our knowledge, these CEGAR algorithms construct abstractions mostly at the predicate level (e.g., [Abdulla et al., 2016]) and rarely at the function level (e.g., [Taghdiri and Jackson, 2007]). To check a program for some property, they typically abstract the program as a Boolean program using a given set of predicates, model checks the Boolean program, and discover additional predicates to refine the Boolean program. Phi Diep, et al. [Abdulla et al., 2016], on the other hand, use variable slicing techniques to obtain an abstract program. To check a concurrent program, Phi Diep, et al. construct an abstraction of the concurrent program by only keeping track of a subset of variables. If a counterexample to the abstraction is spurious, Phi Diep, et al. refine the abstraction by decreasing the set of omitted variables. Variable slicing is one of the verification-guided approaches that are able to address the state-space exposing problem. In contrast with these algorithms, our algorithm constructs the abstractions of the original program at the statement level. By the observation that the more detailed the program, the more complicated the required auxiliary specifications are, our algorithm can assist verification engineers to efficiently discover the exact parts of a program that are relevant to an intended property.

CEGAR framework mostly has been used in automatic program verification techniques (e.g., [Taghdiri and Jackson, 2007]) that focus on checking finite state programs. These techniques require no user-guidance at all and hold promise for scalability using CEGAR framework. To our knowledge, the software model checker SLAM [Ball et al., 2004] is the first system that has applied the CEGAR framework to a program with an infinite number of states. SLAM checks a given C program with respect to a temporal safety property without requiring any user-provided intermediate annotations. It performs the analysis by iteratively refining a predicate abstraction of the code to eliminate spurious counterexamples. Since verifying temporal safety properties of an arbitrary piece of code is undecidable, SLAM is not guaranteed to terminate. Like SLAM, our algorithm focuses on checking infinite state programs as well. Our algorithm has been applied on KeY system, a deductive program verification system, that proves Java programs where loops and recursions are allowed. Unlike SLAM, our algorithm can handle properties of complex data structures that constrain the configurations of the objects on the heap of a program. Analyzing these kinds of properties is particularly important for safety-critical software systems with extensive heap manipulations. Besides, our verification-based program slicing technique provides a decidable logic and guarantees to construct abstract programs regarding the small scope of analysis. By the *small-scope hypothesis* [Jackson, 2012], if an abstract program does not satisfy its specifications, in many cases that will be detected and reported during the bounded verification before the run of deductive verification. Finally, our algorithm guarantees that the accuracy of the program verification result is completely dependent on the deductive program verification.

6.4 Conclusion

In this chapter we present a CEGAR algorithm for deductive program verification. The novelty of our algorithm is that it handles properties of complex data structures and constructs abstractions of a program at the statement level. That is, the program parts that are irrelevant to the property of interest are replaced by abstractions. The rest of the program (the relevant parts) remains unchanged. The on-demand iterative nature of the algorithm guarantees that only as much information about the program will be analyzed as is necessary to check the property of interest. Therefore, our algorithm eases the burden of manually discovering useful auxiliary specifications for deductive program verification. We construct and refine abstractions by exploiting the benefits of bounded program verification techniques. Bounded program verification does not require auxiliary specifications and can guarantee fast and initial confidence of the correctness of the program and provided auxiliary specifications, and thus eases the burden of inspecting the failed proofs for the deductive program verification.

We have implemented the abstraction refinement in the prototype tool RefineJ, and evaluated the benefits of using RefineJ in the deductive verification. We considered various programs that are taken from published literature in the area of program verification. We have used RefineJ to refine the abstractions of the semantic slices constructed by AbstractJ—our verification-based program slicing tool presented in Chapter 5. From the results, RefineJ gradually reveals the code that are relevant to the desired property and helps a lot with renovating failed proofs: with only a few of refinements the failed verification tasks can be completed. Besides, compared to the common way to prove programs, using RefineJ reduces the number of auxiliary specifications: out of the total 21 verification tasks, using RefineJ 19 tasks requires less auxiliary specifications and 13 tasks requires less proof rules compared to the common way.

Part IV

Efficient Bounded Symbolic Execution

CHAPTER 7

Bounded Symbolic Execution Using Incremental Constraint Solving

In the four previous chapters (Chapters 3–6), we have present SMT-based approaches to improve the efficiency of program verification. In this chapter we continue to investigate the impact of using incremental SMT solvers in symbolic execution, a well-known technique that has gained a significant momentum in recent years.

Symbolic execution as a means of analyzing programs has been widely used in program testing [Kapus and Cadar, 2017; Braione et al., 2017], verification condition generation [Ahrendt et al., 2016; Jacobs et al., 2011; Jaffar et al., 2012; Nguyen et al., 2017], and program verification [Harris et al., 2010; McMillan, 2010; Alberti et al., 2012; Jaffar et al., 2009; Pasareanu and Visser, 2004]. It typically takes a parametrized program $P(\overrightarrow{x})$ as input and constructs a symbolic Boolean expression (called *path condition*) that encodes the conditions on the inputs in order to follow one particular path through the program. When a path condition evaluates to $true$ (using constraint solving, for example), its related path is feasible, otherwise it is infeasible. To check whether a property holds for the program, Symbolic execution explores the paths for the desired property by proving that all paths to certain error nodes are infeasible (i.e., the property is violated if an error node is reachable).

Unfortunately, one important obstacle for gaining high code coverage is the high cost of path condition solving [Visser et al., 2012; Yang et al., 2013; Borges et al., 2014; Cadar et al., 2008a]. To the best of our knowledge, existing symbolic execution tools invoke the constraint solver multiple times along one single execution path and each time a new solving starts from scratch even though the constraint has been resolved previously.

Incremental constraint solving (*cache-based*) approaches have been proposed (e.g., KLEE [Cadar et al., 2008a] and GREEN [Visser et al., 2012]) to optimize path condition solving in order to improve time efficiency of symbolic execution. They only solve the "changed parts" of the constraint, that is to solve problems related to the changes between any two constraints that symbolic execution consecutively generates. For example, consider that the symbolic execution produces the constraint $pc_1 : a{>}b \ \wedge \ x{<}y$ for which the solver outputs the following solution [a=2, b=1, x=3, y=4]. To compute the solution for the next constraint $pc_2 : a{>}b \ \wedge \ x{\geq}\ y$ this approach proceeds as follows: It invokes the solver to solve only the changed part of the constraint, namely $x{\geq}\ y$, which is a simpler problem, and combines the new solution [x=4, y=3] with the already-computed solution [a=2, b=1]. The combined solution clearly satisfies pc_2. This idea works under the assumption that the symbolic execution explores similar paths in order (e.g., using depth-first search) and that not all expressions are dependent (see Definition 7.1). An optimized alternative builds on the observation that the approach discussed above could be generalized to build on the solutions of all previously visited path constraints as opposed to only the last one visited. It caches solutions of every independent expression observed in every path constraint. Considering the previous example, a global cache stores solutions to the expressions $a{>}b$, $x{<}y$, and $x{\geq}\ y$ which appeared independently in the two individual path constraints pc_1 and pc_2. Despite the overhead in memory and time consumption related to caching (to store, lookup, and combine solutions), it has been observed that this optimization is beneficial. Popular symbolic execution tools, such as KLEE, CREST [Burnim and Sen, 2008], PEX [Tillmann and de Halleux, 2008], and SPF [Pasareanu and Rungta, 2010], use similar features.

Definition 7.1. *Let $pc = (\mathcal{E}, \mathcal{V})$ be a path condition with Boolean-valued expressions $\mathcal{E}$ and variables $\mathcal{V}$, $var(e)$ ($e \in \mathcal{E}$) be the set of variables used in expression e, thus*

An expression $x \in \mathcal{E}$ and an expression $y \in \mathcal{E}$ ($x \neq y$) are dependent, if

$$\exists v \in \mathcal{V}, v \in var(x) \wedge v \in var(y),$$

otherwise x and y are independent.

Unfortunately, these *cache-based* approaches cannot help in a scenario where the paths that a symbolic execution explores become long. The longer the path explored, the less the independent expressions since the number of input variables is limited. It may be necessary to spawn a completely new search to solve a constraint even if only one of the clauses in this conjunct is new. For example, the cached solution [x=3, y=2] to the constraint $x{>}y$ will not help to solve the constraint $x{>}y \ \wedge \ x{>}3$.

In this chapter we present an incremental SMT solving (*stack-based*) approach for symbolic execution. Our approach represents the program under analysis in a decision graph with respect to the *loop bounds*. A decision graph is essentially a compact version of the acyclic verification graph (see Section 3.1.2). Unlike the verification graph, a decision graph does not label its

nodes and each edge represents a predicate test (i.e., a branch decision or an annotation statement; see Section 3.1.1) and the basic code block immediately following the branch. Thus, it is very likely that the decision graph has less nodes and edges than the verification graph representing a same program. Exploring paths through the graph in depth-first order, our approach creates a new frame on the assertion stack of an SMT solver when reaching a new edge (i.e., a branching decision), and fills the frame with an SMT assertion that is translated from a branch condition with respect to the given *class bounds*.

Our approach improves the time efficiency of symbolic execution by exploiting the recent advances of incremental SMT solvers to solve similar path conditions. An incremental SMT solver reuses the intermediate lemmas that it has learned in previously constraint solving, in contrast to a common SMT solver that learns lemmas from scratch each time when it is invoked. For the scenario where *cache-based* approaches can not help, modern incremental SMT solvers, such as CVC4 [Deters et al., 2014], MathSAT5 [Cimatti et al., 2013], Yices [Dutertre, 2014], and Z3 [de Moura and Bjørner, 2008] can help: during constraint solving these tools learn lemmas, which can be later (re)used to solve similar, *but not identical*, constraints. Unlike the *cache-based* approach that invokes a solver for both current branch condition and previous path conditions that are mutually-dependent, our approach only lets an SMT solver check the current branch condition. Using an incremental SMT solver in path exploration, however, requires to update states on program assignment statements and load states on branching points to generate fresh constraints. That is even worse for those paths traversed multiple times. Our approach represents a program state implicitly as a collection of independent variables and has path conditions constructed before path exploration. It not only reduces the cost of path exploration, but also reduces the size of path conditions by eliminating the common sub-expressions in path condition.

We have implemented our approach in the prototype tool *SymbolicJ*. To the best of our knowledge *no* existing symbolic execution tool uses incremental SMT solving for symbolic execution. There is no clear reason why incremental SMT solving support has not been explored more intensively, although more research is needed to combine these two complementary approaches. Hence, it is important to evaluate how helpful this alternative can be. We compared SymbolicJ with various implementations of cache-based approaches in reducing the time cost of constraint solving in symbolic execution. Overall, our evaluation indicate that SymbolicJ provides superior results.

7.1 Background

Symbolic execution has two components: path condition generation and path condition solving. A path condition is a symbolic boolean expression that encodes the conditions on the inputs to follow one particular path through

the program. Path condition solving serves to check path feasibility and to generate concrete inputs for P. We explain each of them below.

When symbolic execution evaluates a branch instruction in path exploration, it needs to decide which branch of the control flow to select. In a regular execution with concrete inputs the evaluation of a boolean expression is either true or false so only one branch of the conditional can be taken. In contrast, in symbolic execution the evaluation of a boolean expression is a symbolic value so both branches can be taken resulting in different paths to be explored in the program. Symbolic execution characterizes each path it explores with a *path condition* over the input variables $\overrightarrow{x}$. This condition is defined with a conjunction of boolean expressions $pc(\overrightarrow{x}) = \bigwedge_{i>0} b_i$. Each boolean expression b_i denotes a branching decision made during the execution of a distinct path in the program under test. Symbolic execution terminates when it explores all paths corresponding to the different combinations of decisions (e.g., for software testing), or terminates when it finds a feasible path that leads to violation of the desired property (e.g., for program verification). Programs with loops and recursion may result in an infinite number of paths; in those cases, one needs to define a bound on the number of iterations of loops.

Symbolic execution uses constraint solving in two cases: (i) to check path feasibility, and (ii) to generate counterexamples to the desired property (or test inputs). In the first case, symbolic execution checks if the current path is feasible by checking if its path condition is satisfiable. Exploration of one path is interrupted if the path condition becomes unsatisfiable. In the second case, symbolic execution uses a constraint solver to solve constraints associated with complete paths. The solutions to these constraints correspond to counterexamples (or test inputs) for achieving high path coverage.

7.2 Our Approach

Our approach can be used in the *verification* mode to check whether the desired property holds for a program, or in the *testing* mode to generate test inputs. Given as input a parametrized program (annotated by specifications), the class bounds, and the loop bounds, our approach explores the program in depth-first order and checks the path conditions using an incremental SMT solver. In the *verification* mode, our approach terminates either when it finds a program input that leads the program to satisfies the negated specifications (i.e., a counterexample to the specifications), or when all paths have been explored. In the *testing* mode, our approach ignores the specifications and terminates when it explores all paths corresponding to the different combinations of decisions. Our symbolic execution approach is divided into three steps. First, the decision graph is constructed from the analyzed program. Second, path conditions are constructed from the decision graph. Third, path conditions are solved using an incremental solver in path exploration.

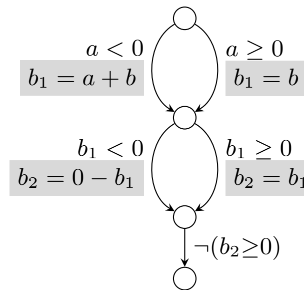
7.2.1 Construction of the Decision Graph

A central part of our approach is the construction of a decision graph—a compact version of the verification graph—to support path condition generation and path exploration. We build this graph from an acyclic verification graph which is constructed from the analyzed program as shown in Section 3.1. In addition to the code transformations used in the construction of the verification graph, our approach uses heuristics to detect loops with concrete (i.e., non-symbolic) conditionals. For example, it identifies that the following loop iterates exactly `K` times `for(int i=0; i<K; i++)`. In addition, constants are unfolded and unreachable code is removed. Using constant (and `null`-value) propagation, we avoid adding unnecessary exceptional branches. For example, there is no need to prepend an exceptional branch on `x.f=y;` when the object `x` is known to be not `null` at the field access. On the other hand, if `x` is known to be `null` on the current statement, we replace this statement by an edge that targets one error/exit node of the decision graph. Note that as the decision graph is acyclic, each variable definition is dynamically unique, i.e., we can treat them as constants. It is important to note that our approach performs all these transformations and the path condition generation (to be shown in Section 7.2.2) before path exploration.

Figure 7.1 shows a decision graph for the `add` function in Fig. 7.1(a). The code of function `add` is only for illustrative purpose; the decision graph in Fig. 7.1(b) is constructed from the `add` function, where new variables (b_1 and b_2) are introduced and each variable is defined only once similar to SSA [Rosen et al., 1988]. It is important to note that similar effect can be obtained without applying this transformation, e.g., by hashing the assignment statements.

```
1  //@ ensures \result >= 0;
2  void add(int a, int b) {
3    if (a < 0) {
4      b = a + b;
5    }
6    if (b < 0) {
7      b = -b;
8    }
9    return b;
10 }
```

(a) A simple code



(b) Decision graph

Fig. 7.1: A decision graph in (b) for the code in (a). Its edges contain the branch conditions and the basic code block (*highlighted*).

7.2.2 Path Condition Generation

It is well known that sharing of structurally equal expressions can reduce space and time requirements in constraint solving, especially when dealing with large constraints. Modern SMT solvers identify the sharing automatically, but there is cost associated with it and the mechanism to identify sharing is non-optimal for the analyzed programs. Aware of that, we construct the path conditions in a representation that facilitates the identification of the sharing. In particular, we translate the branch conditions and assignment statements on the edges of the decision graph into SMT assertions and variable definitions, respectively, using the rules presented in Fig. 3.6. In contrast to constructing a single long path condition over the program arguments for each branch, we treat the variables defined in the assignment statements as the references to the common sub-expressions (i.e., to the expressions on the right-hand side of the statements), and use them to construct many short path conditions.

Our representation of path conditions brings the information of code level to facilitate the *elimination of common sub-expressions* in SMT solving. Consider, for example, the code fragment `if(.)a=x+y;if(a+z>10){.}`. With traditional symbolic execution, the path corresponding to the traversal of the true branches is denoted by the constraint ... $x+y+z > 10$. Our approach, however, translates this constraint into ... $a_1 = x_0 + y_0 \wedge a_1 + z_0 > 10$ as it identifies that the expression denoted by a_1 can be reused in other contexts. The use of such representation increases space requirements, i.e., it increases the number of variables and conjuncts in the constraint. On the other hand, it helps the constraint solver by letting it associate information with newly defined symbols (in this case, a_1).

A constraint solver does not admit destructive state updates; symbols that have been defined in the stack cannot be reassigned. For that reason, to enable the use of incremental solving it is necessary to use a functional program representation (e.g., an SSA-like program representation) whose variables can be assigned only once. This can be obtained explicitly in constructing the decision graph, by transforming the program into a functional representation, or implicitly, by renaming symbols on-the-fly during state-space exploration. In our approach, each sub-expression that is reused triggers the definition of a new frame in the assertion stack. Symbolic execution restores state when backtracking by selectively dropping frames from the assertion stack.

Figure 7.2 shows side-by-side the SMT formulas produced with this optimization disabled (*Stack*) and enabled (*SymbolicJ*) for the `add` method presented in Fig. 7.1(a). In contrast to *Stack*, that generates fresh constraints on decision points, *SymbolicJ* reuses expressions. For example, in Fig. 7.2(b), *SymbolicJ* renames variable `b_1` in query 1 to refer to `a + b`, and uses it in queries 2 and 4. Later in this chapter we evaluate how such transformation can speedup stack-based constraint solving.

(a) *Stack*

```
(declare-fun a () Int)
(declare-fun b () Int)
(push) ; query 1
(assert (< a 0))
(check-sat) ; sat
(push) ; query 2
(assert (< (+ a b) 0))
(check-sat) ; sat
(push) ; query 3
(assert (not (>= (- 0 (+ a b)) 0)))
(check-sat) ; unsat
(pop)
(pop)
(push) ; query 4
(assert (>= (+ a b) 0))
(check-sat) ; sat
(push) ; query 5
(assert (not (>= (+ a b) 0)))
(check-sat) ; unsat
(pop)
(pop)
(pop)
(push) ; query 6
(assert (>= a 0))
(check-sat) ; sat
(push) ; query 7
(assert (< b 0))
(check-sat) ; sat
(push) ; query 8
(assert (not (>= (- 0 b) 0)))
(check-sat) ; unsat
(pop)
(pop)
(push) ; query 9
(assert (>= b 0))
(check-sat) ; sat
(push) ; query 10
(assert (not (>= b 0)))
(check-sat) ; unsat
(pop)
(pop)
(pop)
(exit)
```

(b) *SymbolicJ*

```
(declare-fun a () Int)
(push) ; query 1
(assert (< a 0))
(check-sat) ; sat
(define-fun b_1 () Int (+ a b))
(push) ; query 2
(assert (< b_1 0))
(check-sat) ; sat
(define-fun b_2 () Int (- 0 b_1))
(push) ; query 3
(assert (not (>= b_2 0)))
(check-sat) ; unsat
(pop)
(pop)
(push) ; query 4
(assert (>= b_1 0))
(check-sat) ; sat
(define-fun b_2 () Int b_1)
(push) ; query 5
(assert (not (>= b_2 0)))
(check-sat) ; unsat
(pop)
(pop)
(pop)
(push) ; query 6
(assert (>= a 0))
(check-sat) ; sat
(define-fun b_1 () Int b)
(push) ; query 7
(assert (< b_1 0))
(check-sat) ; sat
(define-fun b_2 () Int (- 0 b_1))
(push) ; query 8
(assert (not (>= b_2 0)))
(check-sat) ; unsat
(pop)
(pop)
(push) ; query 9
(assert (>= b_1 0))
(check-sat) ; sat
(define-fun b_2 () Int b_1)
(push) ; query 10
(assert (not (>= b_2 0)))
(check-sat) ; unsat
(pop)
(pop)
(pop)
(exit)
```

Fig. 7.2: The SMT-LIB scripts expressing path conditions of the Java `add` method presented in Fig. 7.1(a). They are generated using *stack-based* approaches for the verification of the `add` method. Modern SMT solvers provide an assertion stack to incrementally solve the problems that share similar sets of definitions and assertions. SMT-LIB provides `push` and `pop` commands to manipulate such stack. Each stack frame stores an *assertion set*, which includes locally-scoped functions and logical formulas. The command `(check-sat)` returns `sat` if the conjunction of all assertions sets in the stack is satisfiable, or `unsat` otherwise. The SMT comments, i.e., the texts following the semicolon mark ";", indicate what happens during exploration.

7.2.3 Path Exploration

Our technique takes as input the root of the decision graph, an initial model (i.e., a concrete input vector), and an optional time-budget for exploring the program state space. To facilitate illustration we omit the time-budget and define our path exploration as a recursive depth-first search (DFS) through the decision graph. It explores the edges of the decision graph according to the decisions from the current model. By construction, it elaborates a satisfiable stack of assertions as it drives execution towards a feasible path. Consequently, it only explores a new path when execution hits a dead-end. In that case, either the desired property holds for the explored path (using the *verification* mode), or a new test case is generated (using the *testing* mode). Then our approach backtracks exploration to the last unvisited path in the decision graph.

Algorithm 4 shows the algorithm of path exploration. We use the following functions to support our definition.

- `genTest` produces a test case for the input model;
- `pushContext` and `popContext` are wrappers for SMT-LIB commands `push` and `pop`, respectively;
- `loadDefsAndAsserts` augments the logical context of the solver with definitions and assertions passed as argument;
- `check-sat-and-get-model` returns a feasible model if the set of assertions passed as argument is satisfiable; it returns `null` otherwise;
- `hasSyntheticAsserts` indicates if the edge has the assertion introduced in the loop unrolling. It returns $true$ if the argument edge associates to the branch of the bound-hit iteration of the loop;
- `eval` checks if the decision associated to a branch is satisfied with concrete input.
- `terminates` denotes the overall symbolic execution terminates.

7.3 Evaluation

This section presents the experiments we have conducted to evaluate various techniques for symbolic execution. We aim to understand the extent to which constraint solving can be optimized. Our hypothesis is that two factors are important to determine efficiency of symbolic execution: (i) the use of incremental solving (since many path constraints from symbolic execution are similar), and (ii) the use of common sub-expressions elimination (since clause sharing plays an important role in constraint solving).

7.3.1 Experimental Setup

We considered five techniques to evaluate the effectiveness of the cache-based and stack-based approaches to incremental solving, and to investigate the

Algorithm 4 Path Exploration Algorithm

```
1: function TRAVERSE(node, model)
2:     if model == null then
3:         return
4:     end if
5:     if node.hasNoChildren then
6:         GENTEST(model)
7:         if isVerificationMode then
8:             // in the verification mode of our approach
9:             TERMINATES()
10:        else
11:            // in the testing mode of our approach
12:            return
13:        end if
14:    end if

15:    leftEdge ← node.leftEdge, rightEdge ← node.rightEdge
16:    reachLeft ← EVAL(leftEdge, model)
17:    covered ← reachLeft ? leftEdge : rightEdge
18:    uncovered ← reachLeft ? rightEdge : leftEdge

19:    // explore the covered edge
20:    PUSHCONTEXT()
21:    LOADDEFSANDASSERTS(covered)
22:    if covered.hasSyntheticAsserts ∧ !EVAL(covered, model) then
23:        model ← CHECK-SAT-AND-GET-MODEL()
24:    end if
25:    TRAVERSE(covered.targetNode, model)
26:    POPCONTEXT()

27:    // explore the uncovered edge
28:    PUSHCONTEXT()
29:    LOADDEFSANDASSERTS(uncovered)
30:    model ← CHECK-SAT-AND-GET-MODEL()
31:    TRAVERSE(uncovered.targetNode, model)
32:    POPCONTEXT()
33: end function
```

benefit of using common sub-expressions elimination in symbolic execution. All techniques have been implemented in the same infrastructure. We have implemented the infrastructure in Java in ∼20KLOC. We used InspectJ to perform source code transformation and to construct verification graphs. The infrastructure generates constraints in SMT so it can interface with any compliant solver. For example, Z3 is called directly through its programmatic interface to create corresponding Z3 expressions. The infrastructure supports both theories of integers and bit-vectors to assess the impact of various options

of incremental solving to speedup symbolic execution. The infrastructure reuses the created objects to reduce the time and also memory allocation in constraint generation. We briefly describe these techniques below and illustrate them in the rest of this section.

Baseline is the technique that does *not* use incremental solving. It produces a path condition whose conjuncts correspond to the control decisions on symbolic input variables reached along an execution path. This technique makes an independent call to a solver on each query issued from symbolic execution. Figure 7.3(a) shows an example of SMT formulas generated using this technique.

Caching refers to the *cache-based* technique that uses the *independent clauses* optimization. In the cases that symbolic variables are independent, this optimization is helpful since its individual query to a solver is potentially simpler and therefore conceptually cheaper. It incurs in overhead to partition constraints, lookup, and update the cache. Figure 7.3 shows SMT formulas produced with this optimization disabled (*Baseline* in Fig. 7.3(a)) and enabled (*Caching* in Fig. 7.3(b)) for the code shown in Fig. 7.1(a). *Caching* issues each query with the construction of variables and terminates with the destruction of the solver context. For each query, only dependent constraints reach the solver; solutions are cached to avoid redundant queries.

Partitioning optimizes *Caching* by partitioning constraints incrementally. It keeps in memory the set of partitions and corresponding variables for the previously explored constraint. When reaching a control decision, it obtains new partitions by merging all partitions that have variables in common, considering the new variables involved in the decision. Consider the path constraint $PC{=}\bigwedge C_i(V_i)$, where C_i is a partition of dependent clauses in PC involving only symbolic variables V_i. When reaching a control decision $C(V)$, a new partition will be constructed. If the symbolic variables V are not involved in PC, the new partition is equal to $C(V)$, otherwise the new partition is the union of $C_i(V_i)$ and $C(V)$, because V_i and V have shared symbolic variables. It incurs in additional overhead to merge partitions. The SMT formulas generated by this technique are the same as generated by *Caching*, e.g., the script in Fig. 7.3(b).

Stack refers to the technique that creates a new frame on the assertion stack of an SMT solver when reaching a new control decision. An example of the SMT formulas generated by *Stack* can be found in Fig. 7.2(a). In this technique, the solver context that are evolved as new assertions are added to the stack and survives across the symbolic execution of different paths. Note that learned lemmas created on a stack frame are destroyed upon a `pop` of that frame.

SymbolicJ refers to our technique (and also the tool) that optimizes *Stack* by eliminating common sub-expressions. It replaces the common sub-expressions by fresh variables. That increases expression sharing to improve the speed and memory usage of constraint generation, and also facilitates the solver to

identify the constraints associated to the symbolic variables. Besides, it builds path conditions prior to path exploration. That improves time and space efficiency of path exploration. Figure 7.2(b) contains the SMT-LIB formulas generated by *SymbolicJ*.

7.3.2 Objectives of Analysis

We used two sets of programs in our evaluation. The first set includes programs collected from the benchmark of KLEE [Cadar et al., 2008a], an open-source static symbolic execution tool for C programs. The second set includes programs automatically generated with RUGRAT [Breech et al., 2008], a grammar-based Java program generator that has been proposed to support empirical evaluation of testing and analysis techniques. [1]

The KLEE Coreutils benchmark used in [Cadar et al., 2008a] contains 96 Unix core programs (4.5 KLOC together). The tool handles C programs that our infrastructure does not support. Instead, we ran KLEE on the benchmark, collected path conditions produced by the tool, and analyzed them in order, i.e., consecutive constraints in the list reflect exploration order and are similar. We set the time budget for the symbolic execution of KLEE to 30 seconds[2] and used the default configuration for running KLEE. We confirmed, as expected, that KLEE spends most of its time budget (90%=∼27s/30s) in constraint solving.

RUGRAT produces Java programs based on weights associated to grammar production rules. We considered three options for the program size: 5, 10, and 20 KLOC. We generated a total of 300 programs, 100 programs for each program size.

The *independent* variables of our experiments are the time budget for symbolic execution, the size of the program, the form of constraint solving, and the choice of SMT solver. We used various *dependent* variables, e.g., the number of path conditions generated, the number of program states explored, and the number of test inputs generated. These variables are standard metric to assess effectiveness of symbolic execution. In principle, the higher this number is the higher the chances symbolic execution will reveal a bug.

We used several SMT solvers (e.g., Z3 [de Moura and Bjørner, 2008], CVC4 [Deters et al., 2014], and MathSAT5 [Sebastiani and Trentin, 2015]) for solving constraints and bounded depth-first search for exploring paths (i.e., loops are unrolled for a limited number). Even though the results are

[1] A practical challenge for these kinds of generators is to construct realistic programs. However, an empirical study [Hussain et al., 2012] that compared real and generated programs with 78 existing software metrics indicates that it is statistically impossible for a program analysis technique to differentiate a program written by a human from one that the tool generates.

[2] It is very time-consuming to process huge text files. In our environment, KLEE generates constraint files with the average file size ∼500 megabytes per program in 30 seconds, and the file size grows exponentially over time.

(a) *Baseline*

```
(declare-fun a () Int) ; query 1
(assert (< a 0))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 2
(declare-fun b () Int)
(assert (and (< a 0) (< (+ a b) 0)))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 3
(declare-fun b () Int)
(assert (and (< a 0) (< (+ a b) 0)
        (not (>= (- 0 (+ a b)) 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 4
(declare-fun b () Int)
(assert (and (< a 0) (>= (+ a b) 0)))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 5
(declare-fun b () Int)
(assert (and (< a 0) (>= (+ a b) 0)
        (not (>= (- 0 (+ a b)) 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 6
(assert (>= a 0))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 7
(declare-fun b () Int)
(assert (and (>= a 0) (< (+ a b) 0)))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 8
(declare-fun b () Int)
(assert (and (>= a 0) (< (+ a b) 0)
        (not (>= (- 0 (+ a b)) 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 9
(declare-fun b () Int)
(assert (and (>= a 0) (>= (+ a b) 0)))
(check-sat) ; sat
(exit)
(declare-fun a () Int) ; query 10
(declare-fun b () Int)
(assert (and (>= a 0) (>= (+ a b) 0)
        (not (>= (- 0 (+ a b)) 0))))
(check-sat) ; unsat
(exit)
```

(b) *Caching*

```
(declare-fun a () Int) ; query 1
(assert (< a 0))
(check-sat) ; sat
(get-value (a)) ; [a]:=[-1]
(exit)
(declare-fun a () Int) ; query 2
(declare-fun b () Int)
(assert (and (< a 0) (< (+ a b) 0)))
(check-sat) ; sat
(get-value (a b)) ; [a, b] := [-1, 0]
(exit)
(declare-fun a () Int) ; query 3
(declare-fun b () Int)
(assert (and (< a 0) (< (+ a b) 0)
        (not (>= (- 0 (+ a b)) 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 4
(declare-fun b () Int)
(assert (and (< a 0) (>= (+ a b) 0)))
(check-sat) ; sat
(get-value (a b)) ; [a, b] := [-1, 1]
(exit)
(declare-fun a () Int) ; query 5
(declare-fun b () Int)
(assert (and (< a 0) (>= (+ a b) 0)
        (not (>= b 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 6
(assert (>= a 0))
(check-sat) ; sat
(get-value (a)) ; [a]:=[0]
(exit)
(declare-fun b () Int) ; query 7
(assert (< b 0))
(check-sat) ; sat
(get-value (b)) ; [b]:=[-1]
                ; cache hit: [a>=0]
(exit)
(declare-fun a () Int) ; query 8
(declare-fun b () Int)
(assert (and (>= a 0) (< (- 0 b) 0)
        (not (>= (- 0 b) 0))))
(check-sat) ; unsat
(exit)
(declare-fun a () Int) ; query 9
(declare-fun b () Int)
(assert (>= b 0))
(check-sat) ; sat
(get-value (b)) ; [b]:=[0]
                ; cache hit: [a>=0]
(exit)
(declare-fun a () Int) ; query 10
(declare-fun b () Int)
(assert (and (>= a 0) (>= b 0)
        (not (>= b 0))))
(check-sat) ; unsat
(exit)
```

Fig. 7.3: The SMT-LIB scripts expressing path conditions of the Java `add` method presented in Fig. 7.1(a). They are generated using *Baseline* and *Caching* techniques to verify the `add` method against the specification. Each query terminates with the sequence of commands `check-sat` and `get-value` which indicate whether the constraint was satisfied or not. The solver context, that maintains the lemmas learned in previous computations, is destroyed with the command `exit`. Comments indicate what happens during exploration.

deterministic, we ran our scripts multiple times to confirm that environmental changes did not introduce noise in our measurements. We used an Intel Xeon E5-2670 CPU with 2.60GHz clock running on a 64-bit Linux, and set 8GB as the max heap size for one symbolic execution.

7.3.3 Results and Analysis

We pose the following research questions.

RQ1. How cache-based and stack-based techniques compare?
RQ2. What is the benefit of using common sub-expression elimination?
RQ3. Where is each technique spends most time?
RQ4. How sensible different solvers are to the techniques?

RQ1. How cache-based and stack-based approaches compare?

To answer this research question we compared the effectiveness of the techniques on the KLEE and RUGRAT benchmarks. We only considered variants without applying common sub-expression elimination in this experiment.

The KLEE benchmark

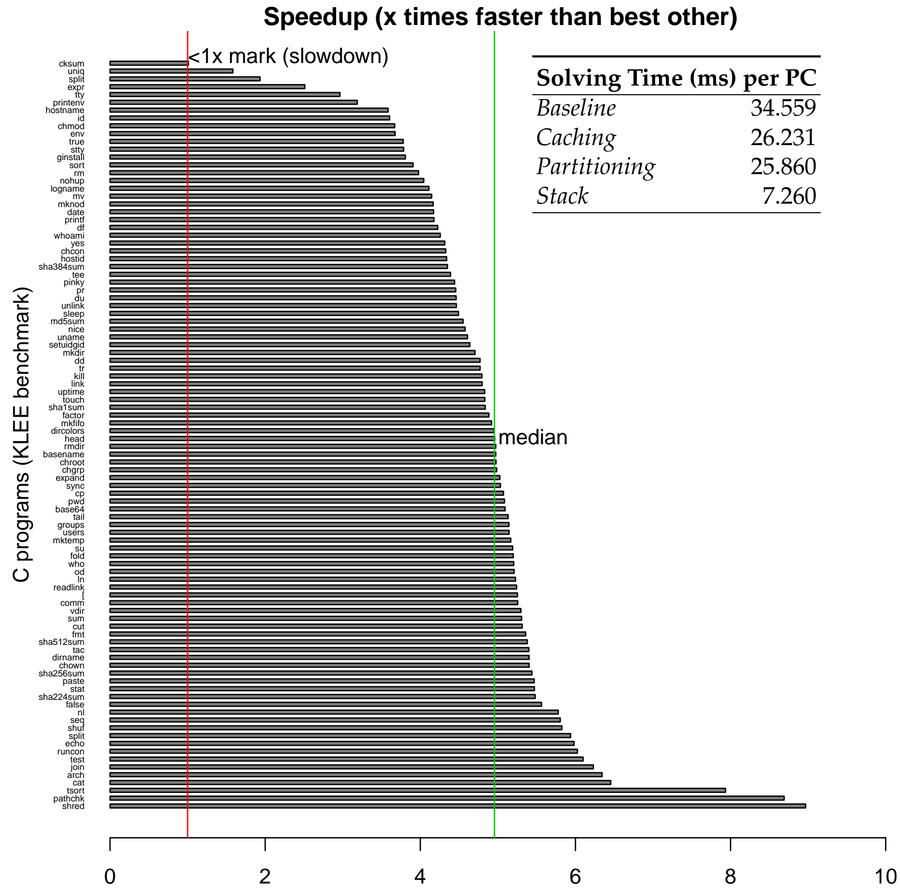
Figure 7.4 shows the speedup that the technique *Stack* obtains compared to the technique *Partitioning*. The table in the right-top corner shows the time of solving each path condition. [3] Considering the 96 analyzed programs the median speedup of *Stack* over *Partitioning* was ∼5x. In absolute terms *Stack* analyzed all path conditions of a program in 0.14s in the best case and 72.36s in the worst case, with a median cost of 6.3s and an average cost of 7.53s. For 92 of the 96 programs *Stack* has solved all path conditions of each program under 10s. 2 programs were solved under 30s and for only 2 programs it required more time: 54.9s and 72.36s.

This gain of faster constraint solving is evidenced by: when symbolic execution proceeds along one single path, the *Stack* approach pushes new clauses onto the constraint stack whenever exploration observes a new control decision. At that point exploration proceeds if the solver responds positively to the checking of satisfiability of the constraint in the stack (feasibility checking) and a new stack frame is created. In contrast, the technique *Baseline* does not save solver context; hence the solver needs to perform the search from scratch whenever reaching a control decision.

The RUGRAT benchmark

Figure 7.5 and Fig. 7.6 show results of various techniques for programs that are generated by RUGRAT. We fixed the time budget to 10 minutes for depth-first path exploration.

[3] We did *not* evaluate SymbolicJ in this experiment as that would require post-processing KLEE-generated constraints and result in unfair comparisons.



Solving Time (ms) per PC	
Baseline	34.559
Caching	26.231
Partitioning	25.860
Stack	7.260

Fig. 7.4: Speedup of stack-based constraint solving over the best other techniques using Z3 on KLEE constraint files. The table in the right-top shows the solving time per path condition using various techniques.

	5K	10K	20K
Number of path conditions / second			
Baseline	92.1	27.0	15.5
Caching	109.6	13.5	8.1
Partitioning	187.9	30.6	12.8
Stack	1319.0	518.9	298.0
SymbolicJ	3672.4	1443.9	862.0

	5K	10K	20K
Total number of path conditions (x 1000)			
Baseline	52	15	8
Caching	63	7	4
Partitioning	107	17	7
Stack	812	341	186
SymbolicJ	2156	834	485

Fig. 7.5: Speed of path condition solving on RUGRAT benchmarks in 10 minutes.

Figure 7.5 shows the average speed of solving path conditions using each technique and the total number of path conditions solved. The average speed of solving path conditions is the total path condition solving time divided by the number of path conditions within a 10m time slot. The results indicate that the use of incremental SMT solving (i.e., *Stack* and SymbolicJ) is beneficial. This is consistent with the plots from Fig. 7.6.

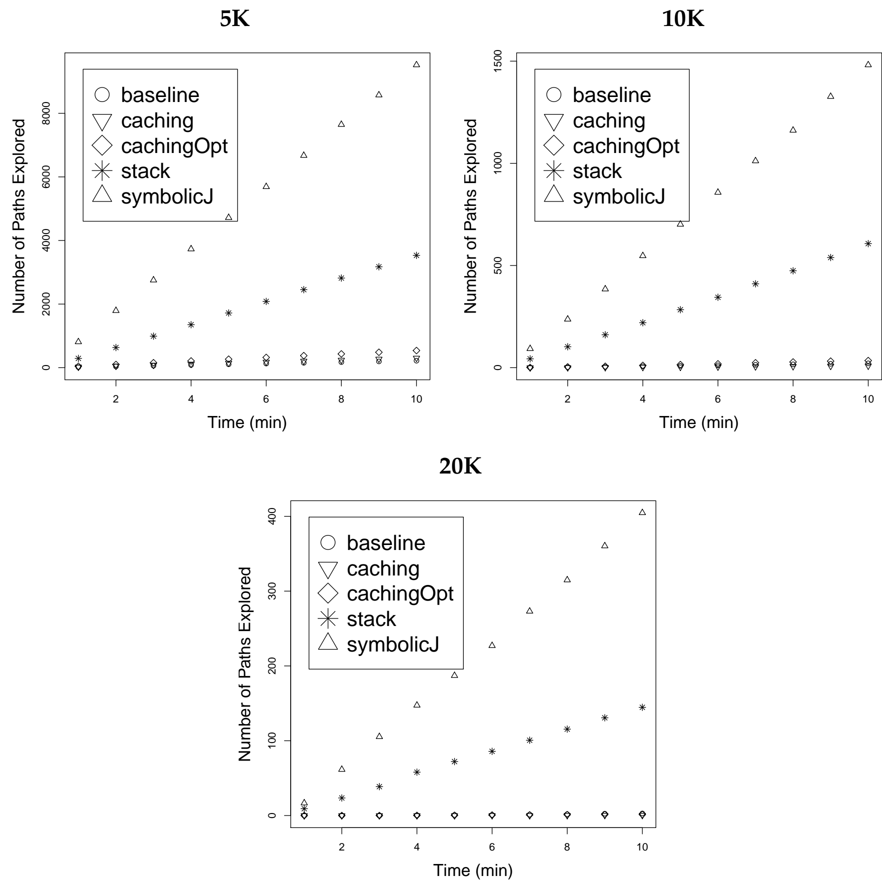


Fig. 7.6: Average number of complete paths explored (i.e., tests generated) for different options of program size using Z3. Time budget is set to 10 minutes.

Each datapoint in Fig. 7.6 indicates the number of explored complete paths, i.e., the number of test cases generated, for a pair of the technique and the point in time. These plots show progress of different techniques. Notice from the Y-axis that as the size of programs grows the number of explored complete paths decreases. We observed that as size of programs

grows, constraint solving also becomes much more expensive; this justifies the decrease in number of complete paths explored on longer programs.

All plots from Fig. 7.6 show a linear X-Y relationship, indicating that the cost of exploring one path remains nearly the same during symbolic execution. Note that results are averaged across several programs. This linear behavior is surprising. In principle, it would be justified when feasible complete paths are uniformly distributed across the exploration tree and the cost of exploring one path is constant. A close inspection of the results revealed that this indeed occurs many times although not always. However, as many subjects are considered, a linear behavior emerged in the averaged plots.

It should be noted that the constraints from the RUGRAT benchmark build on the theory of integers whereas the constraints from the KLEE benchmark build on the theory of bit-vectors. We compared the techniques using different theories and obtained some evidence that the stack-based techniques we presented are effective for the two relevant theories.

RQ2. What is the benefit of using common sub-expression elimination?

Analyzing Fig. 7.6 it can be seen that SymbolicJ performs remarkably well. In contrast to the *Stack* approach, this approach does not appear to degrade performance as the size of programs and constraints increase. The reason for this is justified. 1) On reaching each branch decision, SymbolicJ reuses the constraints constructed before the path exploration while *Stack* constructs new constraints when the variables involved in the branch condition were updated in the path leading to this branch. This is evidenced in Fig. 7.7, in which *Stack* has a notable overhead in path exploration. 2) To save search space and time, most modern SMT solvers map structure-equal expressions to a singleton to construct a compact problem. While modern solvers detect shared expressions at the formula level, SymbolicJ introduces intermediate variables as macros to shared expressions at the code level. Figure 7.7 also shows that *Stack* spends notably more time in building logical context than SymbolicJ.

RQ3. Where is each technique spends most time?

Figure 7.7 shows the time breakdown of the techniques considering 4 sources of runtime cost: path exploration, Z3 expression construction, Z3 constraint solving, and the residuals. Path exploration time includes the time of path condition generation, e.g., storing / restoring states and constructing symbolic expressions. We divide the constraint solving to *Z3 expression construction* and *Z3 solving*. The *Z3 expression construction* refers to build the logical context in Z3, that is to create and add Z3 assertions to the logical stack. The *Z3 expression construction* time includes the time of creating Z3 expressions (we used Z3's programmatic interface (i.e., API) for that). The *Z3 solving* is the phase to check satisfiability of the logical problem. The *Z3 solving* time includes actual

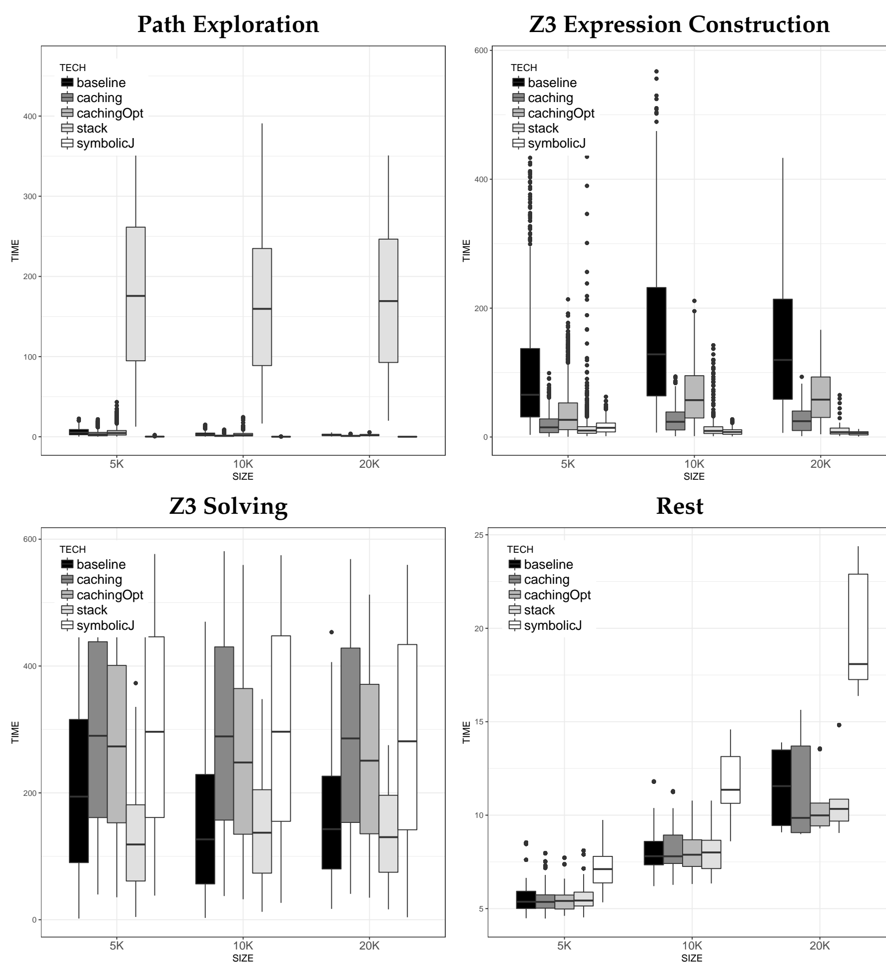


Fig. 7.7: Average time breakdown of different techniques using Z3 in 10 minutes.

solving and caching time, and the residuals include the remaining parts, for example, the time of performing code transforms.

The following observations are made:

- *Baseline* spends more time in *Z3 expression construction* compared to other techniques. This happens because *Baseline* needs reconstruct all Z3 expressions for a new query, while *Caching* reduces the amount of constraints issued to the solver and consequently also reduces this cost.
- *Stack* spends more time in path exploration compared to other techniques. This happens because *Stack* needs to update states on assignment statements and load states on decision points to generate fresh constraints,

while SymbolicJ has constraints constructed before path exploration. That is even worse for those paths traversed multiple times; *Stack* will reload the states and recompute the constraints for each traversing, while SymbolicJ has constraints constructed prior to the path exploration.
- All caching techniques and SymbolicJ spent most time on solving constraints and at least 70% of the time is spent in constraint solving in this phase.
- *Stack* spent less time in Z3 solving compared to other techniques, while it can solve more constraints than any other technique except SymbolicJ.
- SymbolicJ spent more time in residuals than other techniques. This happens because SymbolicJ has a code transformation to rename variables, while this is not done by other techniques.

Construction of Decision Graphs. We construct the decision graph before the path exploration. The construction of the decision graph requires a sequence of source code transformations. For example, unroll loops according to the loop bounds, inline methods on each method invocation statement, and renaming variables (and fields) to make sure each variable (and field) is defined at most once on each program path.

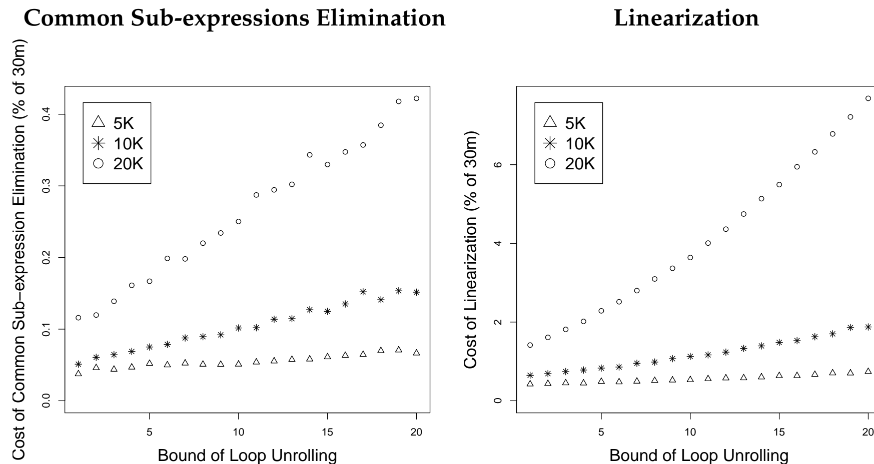


Fig. 7.8: Average percentage of code transformation cost. For example, the average cost of linearization for a 20K program configured to unroll loops at most 3 times is approximately 34s(=(1.9/100)*30*60s)

We evaluated how costly these code transformations can be relative to the other costs. We observed that the linearization procedure (i.e., inlining methods and unrolling loops) is significantly more expensive compared to renaming variables and fields. But still linearization has relatively low cost.

Figure 7.8 shows how each of these operations scales with program size and bound of loop unrollings. 60 subjects have been checked with a timeout 30 minutes. The scale of the Y-axis is the percentage of a 30m time budget. The results are averaged across all subjects considered for that size. In the worst case, linearization of 20K programs with 20 loop unrollings takes roughly 2m24s (=144s=8% of 30m). Note that the 20K linearization plot shows an exponential increase in cost. Previous works (e.g., [Jackson, 2012]) indicate that exhaustive testing within small bounds can achieve high statement and branch coverage and kill most of the mutants. Thus small bounds are often sufficient to find most errors.

RQ4. How sensible different solvers are to the techniques?

Figure 7.9 shows the number of states explored using various SMT solvers. We used the pipeline-based interfaces of the solvers since the API-based interfaces are missed for CVC4 and MathSAT5.

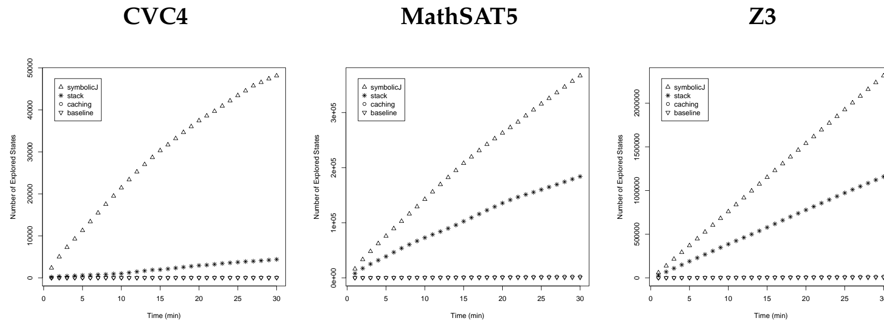


Fig. 7.9: Average number of states explored from 5K subjects using CVC4, MathSAT and Z3 in 30 min.

7.3.4 Threats to Validity

As usual, it is possible that results do not generalize much beyond our subject set. To mitigate this threat we used a set of 300 automatically-generated Java programs and a set of 96 real C programs from the GNU operating system. Besides, it is possible that results are specific to Z3. To address this threat we considered other solvers, namely CVC4 and MathSAT5. Although these solvers show different costs for different constraint problems, we observed a similar behavior on these solvers for the techniques we analyzed.

Another threat to validity is the possibility of errors in our implementation. We carefully inspected our code and the consistency of our results.

Nevertheless, additional experiments are necessary to assess generality of our results.

7.4 Related Work

Symbolic execution is a technique for systematic test-input generation that has gained significant momentum in recent years [Cadar et al., 2011]. Several tools have been proposed to support symbolic execution, including Bogor/Kiasan [Deng et al., 2006], Cloud9 [Bucur et al., 2011], (j)CUTE [Sen et al., 2005], DART [Godefroid et al., 2005], DiSE [Yang et al., 2014], DSC [Islam and Csallner, 2010], EXE [Cadar et al., 2008b], eXpress [Taneja et al., 2011], KLEE [Cadar et al., 2008a], ParSym [Siddiqui and Khurshid, 2010], PEX [Tillmann and de Halleux, 2008], SAGE [Godefroid et al., 2012], SMART [Godefroid, 2011], and SPF [Pasareanu and Rungta, 2010]. Unfortunately, symbolic execution is expensive both in time and space. We discuss most-related recent work to reduce the high cost in constraint solving and in path exploration during symbolic execution.

Time Reduction

Typically, symbolic execution spends a major part of its time in constraint solving. Cadar et al. [Cadar et al., 2008a] proposed several approaches to simplify constraints prior to calling an SMT solver during symbolic execution. The static symbolic execution tool KLEE implements *Caching* as we described. In addition, KLEE implements constraint checking with a potential solution. It is based on the assumption that a solution of subset often satisfies extra constraints. We need to investigate how this additional optimization compares with those we considered.

Visser et al. [Visser et al., 2012] proposed GREEN, an infrastructure to share results of symbolic executions across different environments. It not only reuses the previously results in a single symbolic analysis, but also reuse more generalized results such as the results from other runs of symbolic executions and the results from different users. Moreover, they reused the results for blocks of complicated code. GREEN proposes canonical representations of path conditions to enable caching across different programs. The intuition is that after partitioning constraints with respect to dependent clauses the chance of finding structurally equal symbolic constraints increases. For example, solutions to constraints produced in the symbolic execution of one program could be used to solve constraints produced from the symbolic execution for another program. The results of GREEN are encouraging to speedup constraint solving. It is important to note that we also optimized the caching algorithm by incrementally constructing the independent constraints, instead of constructing them from scratch each time.

Incremental SMT solving is an active field of research with the goal of optimizing problems that can be characterized by many similar sub-problems. For example, detecting the longest program execution trace [Li et al., 2014], solving scheduling problems [Steiner, 2010], etc. As a basic decision procedure, incremental SMT solving searches for a satisfying assignment by performing various operations (e.g., unit propagation). When internal conflicts occur, incremental SMT solvers extract and store conflict clause to prune exploration search space. More specifically, incremental solvers store learned and conflicting clauses in the assertion stack so that they can be reused upon backtracking. Recently, Audemard et al. [Audemard et al., 2013] proposed a technique to strengthen the clauses learned by the solver by extending an incremental SMT solver to execute in multiple threads. We observed that this development can directly improve symbolic execution.

Incremental SMT solving [Hooker, 1993; Whittemore et al., 2001; Wieringa, 2014] has been applied in some domain and achieved higher performance by incrementally solving sets of related problems. Some experimental evidence showed the effectiveness of the incremental facility to classical solvers. For symbolic execution, to the best of our knowledge, there was *no* existing symbolic execution tool using incremental SMT solving before our related publication [Liu et al., 2014]. We evaluated the benefits of the gain and recommend the use of incremental SMT solving in symbolic execution.

Space Reduction

Several techniques have been proposed to address the path-explosion problem in symbolic execution. Godefroid [Godefroid, 2007] proposed compositional symbolic execution. The idea is to perform symbolic execution using symbolic summaries of functions that are incrementally computed in symbolic execution. Yang et al. [Yang et al., 2014] proposed Directed Incremental Symbolic Execution (DiSE) to speedup symbolic execution by capitalizing on the changes across two code versions. Taneja et al. [Taneja et al., 2011] proposed eXpress that builds on similar ideas. Both compositional and differential symbolic execution are legitimate approaches to scale symbolic execution. SymbolicJ is *orthogonal* to these techniques. It improves symbolic execution on the apace dimension by eliminating common sub-expressions and building the path conditions prior to path exploration. It is also possible to customize SymbolicJ to perform compositional and differential symbolic execution.

Anand et al. [Anand et al., 2007a] proposed the use of an interprocedural static analysis to detect potential flows of symbolic data to statements. One of the goals was to speedup symbolic execution by avoiding unnecessary code instrumentation. Even though SymbolicJ does not require code instrumentation we plan to evaluate how SymbolicJ compares to with further optimization applied. Also, Siddiqui and Khurshid [Siddiqui and Khurshid, 2010], Staats and Păsăreanu [Staats and Pasareanu, 2010], and Bucur et al. [Bucur et al., 2011] independently proposed the use of Parallel Symbolic Execution (PSE) to

optimize the time dimension for symbolic execution. SymbolicJ *complements* parallel symbolic execution. For example, our approach can be used on each of the parallel processes to amplify the improvement of PSE.

More recently, Anand and Harrold [Anand and Harrold, 2011] proposed Heap Cloning. The idea of the technique is to use an image of the concrete heap while performing symbolic execution. One of the goals was to speedup symbolic execution by avoiding unnecessary calls to reliable library code. Park et al. [Park et al., 2012] proposed the use of large test suite bases to support constraint solving. They use DSC [Islam and Csallner, 2010], a test case generator using dynamic symbolic execution, to build path conditions and existing test inputs to solve constraints. We will evaluate how SymbolicJ compares to optimized versions of DSC that use such techniques.

7.5 Conclusion

Program verification techniques using symbolic execution generally check the model of a program for the desired property by proving that all paths to certain error nodes are infeasible (i.e., no program execution will violate the property). One important obstacle for gaining high structural coverage is the high cost of path condition solving.

We present an incremental SMT solving (*stack-based*) approach for symbolic execution. Our approach represents the program under analysis in a decision graph with respect to the loop bounds and constructs the path conditions (concerning the class bounds) before the path exploration. Thus it reduces the time of path condition generation. In the phase of path exploration, our approach reduces the time of path condition solving by eliminating common sub-expressions in the conditions and exploiting the recent advances of incremental SMT solvers. An incremental SMT solver reuses the intermediate lemmas that it has learned in previously constraint solving in contrast to a common SMT solver that learns lemmas from scratch each time it is invoked.

We have implemented our approach in the prototype tool *SymbolicJ*. To the best of our knowledge *no* existing symbolic execution tool uses incremental SMT solving for symbolic execution. Hence, it is important to evaluate how helpful our approach can be. We performed an empirical study on the efficiency of symbolic execution using different techniques. We compared SymbolicJ with various alternatives using incremental solving. We considered various options of *cache-based* incremental solving and a large set of programs; both real (96 C programs from the KLEE benchmark) and artificially-generated (300 randomly-generated programs of various sizes: 5, 10, and $20K$). Overall, results indicate that *stack-based* approaches provide superior results compared to cache-based approaches. The median speedup obtained when using the support of a modern incremental SMT solver is of ~ 5x (min.:~ 1x, avg.:~ 4.8x, max.: ~ 9x). When the optimization that eliminates common sub-expressions is enabled in *stack-based* approach (i.e., the

approach provided by SymbolicJ), results indicate that the speedup obtained is of ~2.57x compared to a basic *stack-based* approach. Note that results are restricted to the use depth-first search. More research is needed to find ways to combine caching- and stack-based approaches to improve results even further.

Part V

Conclusion

CHAPTER 8

Related Works

In the five previous chapters (Chapters 3–7) we have discussed the related works that are specific for the area they concentrate. In this chapter we will present the program verification techniques for checking the correctness of programs with complex data structures.

Bounded Program Verification

Many bounded verification approaches (e.g., Jalloy [Vaziri-Farahani, 2004], JForge [Dennis et al., 2006], TACO[Galeotti et al., 2013], Miniatur[Dolby et al., 2007], Karun[Taghdiri, 2008], and MemSAT[Torlak et al., 2010]) have been developed that target program with complex data structures. Similar to our technique in Chapter 3, these approaches are exhaustive in the analyzed domain and produce non-spurious counterexamples (with respect to the analyzed bounds). However, unlike our technique that translates the code and specifications into an SMT logic to allow high-level simplification before bit-blasting, they directly use bit-blasting to generate the propositional logic formulas from a relational logic. Bit-blasting is a formula flattening technique that flats a high-level formula by representing its word-level variables, e.g., bit vectors, using bit-wise terms. Thus, scalability is their key issue since bit-blasting may not scale when the bitwidth (size of a bit-vector) increases.

Scalability of bounded program verification can be improved by partitioning the set of all program executions based on the program's control-flow or data-flow properties, and analyzing each partition separately [Shao et al., 2009, 2010]. Another possibility is to introduce a CEGAR framework (see e.g., Karun [Taghdiri, 2008]), to iteratively analyze only the necessary parts

of the code. Such ideas are independent of the underlying solver and can be incorporated into our approach in future.

ESC/Java[Flanagan et al., 2002] and ESC/Java2[Cok and Kiniry, 2004] analyze JML specifications of Java programs where loops are bounded, but the objects in the heap are not. They support various SMT solvers and theorem provers, but due to quantification over infinite types, their target logics are undecidable. Thus the solver may not terminate with a conclusive outcome.

Deductive Program verification

Traditional program verification approaches made extensive use of the program structure in structuring the analysis. Each method would be checked against its specification, using specifications of the called methods as surrogates for their code. Many tools (e.g., KeY [Ahrendt et al., 2016], VCC [Cohen et al., 2009], and PVS [Owre et al., 1992]) have been applied successfully to substantial programs, but they suffer from an obstacle that limits their applicability. It turns out that the burden of writing useful auxiliary specifications (e.g., method contracts and loop invariants) for the called methods is considerable. First, understanding the required contributions of the called methods is a serious issue when the called methods are deeply called by the top method. Thus it is difficult to write just enough auxiliary specifications for the called methods. A too detailed specification increases the difficulty to prove the correctness of a program and it is error-prone to write such a specification. On the other hand, a too weak auxiliary specification causes the verification to fail. Second, it is unclear whether the auxiliary specification or the analyzed program is wrong when the verification fails.

Shape Analysis

Shape analysis is a static code analysis that can estimate the shape (e.g., Tree or Graph) of the data structure that is used in the analyzed program. Many shape analysis algorithms [Chase et al., 1990; Jones and Muchnick, 1979, 1982; Plevyak et al., 1993; Stransky, 1992] have been developed to infer shape invariants of programs with a particular data structure. They typically represent the memory states via shape graphs in which the nodes represent memory locations, and edges represent field relations. Each node is associated to a group of shape predicates that describe its relations with other nodes. A finite set of shape graphs is estimated via merging the nodes that satisfy similar shape predicates to a summary node. to obtain a finite set of graphs that implicitly represent the data structure properties. Sagiv et al. provide a parametric framework for shape analysis (PSA) [Sagiv et al., 2002] that generalizes these algorithms and can generate various shape analysis algorithms with different precision and complexity according to the chosen shape predicates. PSA encodes properties of memory locations using particular shape predicates and evaluates predicates using Kleene's 3-valued logic [Kleene et al., 1952].

The output can be `true` (denoted by value 1), meaning that the property is proved to be correct, `false` (denoted by value 0), meaning that the property is known to be incorrect, or `unknown` (denoted by value $1/2$), meaning that nothing can be deduced. PSA has been implemented in the three-valued logic analyzer (TVLA) [Lev-Ami and Sagiv, 2000] that can automatically generate a static-analysis algorithm from the operational semantics of a given program. TVLA requires users to provide abstract operational semantics in first-order predicate logic with transitive closure. These semantics have to be proved correct before the actual verification and contains enough details for gaining a successful verification. It uses abstract-interpretation technique [Cousot and Cousot, 1979] in shape analysis and conservatively merges shape nodes with similar behaviors to a summary node with abstract operational semantics. In contrast to traditional program verification that requires users to provide annotations, TVLA has been used to infer loop invariants in proving the insertion sort and bubble sort of an abstract datatype (ADT) of linear linked list [Lev-Ami et al., 2000] and binary search trees [Reineke, 2006]. However, the precision of these analysis depends on the chosen of shape predicates in summarizing nodes.

The pointer assertion logic engine (PALE) [Møller and Schwartzbach, 2001] is another shape analysis tool for verifying data structure invariants. PALE requires users to provide a large set of annotations for the analyzed procedures. The annotations are expressed in pointer assertion logic (PAL), a monodic second-order logic expressed over records, pointers, and Boolean, and are treated as hints for the underlying decision procedure, MONA [Klarlund et al., 2002]. When the analysis terminates, either the data structure invariants have been proved or a counterexample has been generated. PALE has been shown to perform efficiently in practice, however, it has a non-elementary worst-case complexity. Thus PALE aims to verify partial properties of a single data structure. Besides, it eagerly requires user-provided annotations and does not support arbitrary data structures.

To support composite data structures, especially nesting lists, Berdine et al. proposed a higher-order predicate-based shape analysis (HOPSA) [Berdine et al., 2007] to support a variety of complex data structures. HOPSA specifies a family of linear data structures using a higher-order inductive predicate and infers new predicates to express complex composite structures in the analysis. it has been successfully used to verify several procedures in a system library. In order to reduce the amount of annotations, Bohne [Wies et al., 2006] use a number of decision procedures to infer loop invariants that may contain transitive closure and quantifiers. However, it also requires users to provide some annotations as hints in the abstraction of each code fragment.

In contrast to the approaches like PSA that group each family of nodes into a single summary node when the nodes associate to equivalent predicates, grammar-based shape analysis (GSA) [Lee et al., 2005] associates new approximate grammars to the summary nodes of the shape graphs. That is, GSA provides better precision than PSA to improve the accuracy and main-

tains fewer shape graphs to reduce the space cost. The semantics of the shape graphs is defined as assertions to be proved via separation logic [Reynolds, 2005]. GSA has been used to verify the binomial heap construction algorithm and the Schorr-Waite tree traversing algorithm [Cormen et al., 2009]. All these techniques are useful to analyze certain linear linked lists such as singly- and doubly-linked lists. However, they are not easily extensible to arbitrary data structures.

Model Checking

Model checkers such as FSoft [Ivancic et al., 2005], CBMC[Clarke et al., 2004], and SLAM [Ball and Rajamani, 2001] were designed to check temporal safety properties. They provide a fully automatics analysis and can produce sound counterexamples. They have been successfully used in checking large programs against control properties, but they are not suitable for checking the kind of data-structure properties that we aim. Several model checkers (e.g., [Ganai and Gupta, 2006; Armando et al., 2009; Sinz et al., 2010; Cordeiro et al., 2012; Vujosevic-Janicic and Kuncak, 2012]) incorporate SMT solvers as their underlying engines. Similar to our approach, they translate a program and its property into an SMT logic that consist of bit-vectors and/or arrays. Unlike our approach, their logics are quantifier-free or does not support reachability expressions. To our knowledge, all of these model checkers focus on checking C programs (thus no object-oriented features are supported), and their translations are highly tuned for checking memory layout and finite-state-machine properties; no data-structure properties (beyond simple array accesses) can be checked. JBMC [Cordeiro et al., 2018] is the first BMC-based Java verifier. It processes Java bytecode and only checks the runtime exceptions, e.g., `NullPointerException` and `ArrayIndexOutOfBoundsException`. It can not check the properties of complex data structures that we target.

Symbolic Execution

TestEra [Khalek et al., 2011] and Korat [Milicevic et al., 2007] also check Java programs against data-structure properties with respect to a bounded heap. However, they perform the check dynamically. That is, they generate all non-isomorphic input structures that satisfy the preconditions within the given bounds, run the program on each input, and check the results against an oracle (or a postcondition). For checking code that involves a single data structure, these approaches would suffice; they would achieve the same results as bounded program verification. However, for checking code that involves several data structures, the number of possible inputs can become too large to enumerate and execute explicitly.

CHAPTER 9

Conclusion and Future Works

Verifying programs with complex data structures, e.g., lists, trees, and graphs is particularly important for safety-critical software systems with extensive heap manipulations. Erroneous heap manipulations may cause loss of data or unauthorized access to data, violate software security, and may eventually cause a system to crash. Program verification techniques such as deductive- and bounded program verification generally are capable of verifying programs with complex data structures. However, they either require the verification engineers to provide many annotations and discovering useful annotations is a complicated and error-prone effort, or they do not scale—the analysis only works for a very small scope.

This thesis presented an infrastructure to reduce the effort of verification engineers in deductive program verification, i.e., the effort to inspect failed proofs and to discover useful annotations.

- **Reducing the number of failed proofs.** Instead of proving the correctness of a program from scratch, we verify the program using bounded program verification before using deductive verification. Bounded program verification is fully automatic, and grants fast and initial confidence in the correctness of the program, and it is then convenient for the verification engineers to renovate the failed verification. Besides, if a program has bugs, it is very likely to find most of the bugs already in a small scope. Therefore, proving a bounded verified program reduces the effort of verification engineers in inspecting failed proofs.
- **Reducing the unnecessary implementation details.** Instead of writing annotations for the whole program, we provide an instantiation of the Counterexample-guided Abstraction Refinement (CEGAR) framework

to gradually reveal the necessary implementation details for the deductive verification. Guided by bounded program verification, we construct abstractions to replace those program parts whose behavior does not affect the evaluation result of the specification and gradually refine the abstractions based on the spurious counterexamples. Code with less details requires less annotations. Thus, we reduce the effort of verification engineers in discovering useful annotations.

Our infrastructure gradually reveals the program parts that are relevant to the desired property and then guarantees that only as much information about the program will be analyzed as is necessary to check the property. Thus it holds the promise of scalability. Besides, our CEGAR instantiation has a certain degree of extensibility and it does not rely on a specific bounded program verification. The deductive program verification can still benefit from a scope-bounded code analysis. As an application using Satisfiability Modulo Theories (SMT) solvers, our infrastructure exploits the recent advances of SMT solvers. With the continuous development of SMT solvers, we believe that our infrastructure will become more efficient.

Though the thesis overall presents a comprehensive infrastructure to check whether a program fulfills its desired property, the components of the infrastructure are designed as stand-alone analyses that can be used in different contexts. We provide an SMT-based bounded program verification approach (see Chapter 3) that scales better than an SAT-based bounded program verification. It allows to analyze the programs for a larger scope and makes a step forward in extending the applications of bounded program verification in practice. In addition, our calculus for computing the accurate scope of analysis (Chapter 4) improves the efficiency of existing bounded program verification techniques, thus they may find software bugs that they could not find before. We provide a verification-based program slicing technique (Chapter 5) for the construction (and the refinement) of the abstractions in our CEGAR instantiation (Chapter 6). The slicing technique and the CEGAR instantiation liberate the verification engineers from the implementations that are irrelevant to the desired property and also from the unnecessary verification tasks during software development—they do not need to verify the whole program if the updated implementation still fulfills the abstractions. Finally, we investigate the impact of using incremental SMT solvers on the speed of symbolic execution (Chapter 7). The results of our empirical study recommend the verification engineers to use incremental SMT solvers when they need to check an SMT formula that is similar, but not identical, to the previously checked formulas.

We have implemented our approaches in prototype tools and performed various experiments to evaluate the benefits of using those approaches compared to existing (or constructed) alternatives. The preliminary results show that in almost all cases our approaches provide faster and more accurate answers of analysis and are more conducive to the deductive verification than

the alternatives. However, there are still cases where our approaches do not scale with the rising scope of analysis. Existing techniques can be used to enhance the scalability of our approaches; for example, symmetry breaking, incremental control-/data-flow analysis, and summary inference. Besides, we plan to cover more language features. Currently, the infrastructure supports a basic subset of Java that does not include floating point numbers, concurrency, and lambda calculus. Although we provide an encoding of the Java Modeling Language (JML) reachability construct, this feature has only been tested in bounded program verification but has not been used in the experiments involving deductive program verification. Thus it is important to investigate the impacts of using the feature in our CEGAR instantiation. Another challenging and interesting task is to migrate our approaches into other programming languages such as C and C++. They are widely used for embedded and safety-critical systems.

The development of program verification is not an overnight process. It is a gradual process. For example, program verification gains a lot of benefits from using SMT solvers, and the development of SMT solvers is a gradual process. It has been developed for many years and only in last decade, it attracts people's attention. In addition, there is no a single solution for all problems, we combine the benefits of different techniques and improve the efficiency of the program verification. The superior results recommend the people to provide a heterogeneous solution (i.e., a program verification system combining different techniques) to program verification. Recall the motivation at the beginning of the first chapter that, more efforts are required on the road to success in software quality assurance. We have made small steps on the road in the thesis. So far, we have only made small steps in advancing the process of extending applications of deductive program verification. The bigger challenges are still ahead.

References

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Bui Phi Diep. Counter-example guided program verification. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *Proceedings of FM*, volume 9995 of *LNCS*, pages 25–42, 2016. (Cited on pages 9, 91, 94, and 104.)

Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of FM*, volume 5850 of *LNCS*, pages 612–627. Springer, 2009. (Cited on page 47.)

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*, 2016. Springer. ISBN 978-3-319-49811-9. (Cited on pages 8, 47, 109, and 136.)

Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In Nikolaj Bjørner and Andrei Voronkov, editors, *Proceedings of LPAR*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012. (Cited on page 109.)

Saswat Anand and Mary Jean Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *Proceedings of ASE*, pages 33–42. IEEE, 2011. (Cited on page 130.)

Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In Orna Grumberg and Michael Huth, editors, *Proceedings of TACAS*, volume 4424 of *LNCS*, pages 117–133. Springer, 2007a. (Cited on page 129.)

Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java pathfinder. In Orna Grumberg and Michael Huth, editors, *Proceedings of TACAS*, volume 4424 of *LNCS*, pages 134–138. Springer, 2007b. (Cited on page 65.)

Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009. (Cited on pages 22 and 138.)

Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Proceedings of SAT*, volume 7962 of *LNCS*, pages 309–317. Springer, 2013. (Cited on page 129.)

Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Costas Courcoubetis, editor, *Proceedings of CAV*, volume 697 of *LNCS*, pages 29–40. Springer, 1993. (Cited on pages 9 and 103.)

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of SPIN*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001. (Cited on page 138.)

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of iFM*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004. (Cited on pages 9, 91, 103, and 104.)

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Proceedings of FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005. (Cited on page 22.)

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard: Version 2.6. Technical report, The University of Iowa, 2017. (Cited on pages 6 and 18.)

José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. *Formal Aspects of Computing*, 24(2):217–248, 2012. (Cited on page 87.)

Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of IJCAR*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006. (Cited on page 47.)

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, *Proceedings of CAV*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007. (Cited on page 137.)

Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007. (Cited on pages 9, 91, and 104.)

Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz - an optimizing SMT solver. In Christel Baier and Cesare Tinelli, editors, *Proceedings of TACAS*, volume 9035 of *LNCS*, pages 194–199. Springer, 2015. (Cited on page 52.)

Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *Proceedings of LPAR*, volume 6355 of *LNCS*, pages 103–118. Springer, 2010. (Cited on page 63.)

François Bobot and Andrey Paskevich. Expressing polymorphic types in a many-sorted language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Proceedings of FroCoS*, volume 6989 of *LNCS*, pages 87–102. Springer, 2011. (Cited on page 47.)

Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Pasareanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of PLDI*, pages 123–132. ACM, 2014. (Cited on page 109.)

Thorsten Bormer. *Advancing Deductive Program-Level Verification for Real-World Application: Lessons Learned from an Industrial Case Study*. PhD thesis, Karlsruhe Institute of Technology, 2014. (Cited on pages 9 and 88.)

Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Proceedings of CADE*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009. (Cited on page 18.)

Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of ISSTA*, pages 90–101. ACM, 2017. (Cited on page 109.)

Ben Breech, Lori L. Pollock, and John Cavazos. RUGRAT: runtime test case generation using dynamic compilers. In *Proceedings of ISSRE*, pages 137–146. IEEE, 2008. (Cited on page 119.)

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexceptions for Java bytecode. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Proceedings of FoVeOOS*, volume 7421 of *LNCS*, pages 123–141. Springer, 2011. (Cited on page 64.)

Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009. (Cited on page 18.)

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In Christoph M. Kirsch and Gernot Heiser, editors, *Proceedings of EuroSys*, pages 183–198. ACM, 2011. (Cited on pages 128 and 129.)

Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of ASE*, pages 443–446. IEEE, 2008. (Cited on page 110.)

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *Proceedings of OSDI*, pages 209–224. USENIX, 2008a. (Cited on pages 109, 110, 119, and 128.)

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2):10:1–10:38, 2008b. (Cited on page 128.)

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of ICSE*, pages 1066–1071. ACM, 2011. (Cited on page 128.)

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004. (Cited on pages 9, 91, and 103.)

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Bernard N. Fischer, editor, *Proceedings of the PLDI*, pages 296–310. ACM, 1990. URL `2.93585`. (Cited on page 136.)

Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of SAC*, pages 1284–1291. ACM, 2012. (Cited on page 87.)

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Proceedings of SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012. (Cited on page 18.)

In Sang Chung, Wan Kwon Lee, Gwang Sik Yoon, and Yong Rae Kwon. Program slicing based on specification. In Gary B. Lamont, editor, *Proceedings of SAC*, pages 605–609. ACM, 2001. (Cited on page 87.)

Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of CAV*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002. (Cited on page 103.)

Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings of SAT*, volume 4501 of *LNCS*, pages 334–339. Springer, 2007. (Cited on page 78.)

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013. (Cited on pages 52 and 111.)

Koen Claessen. Expressing transitive closures for finite domains in pure first order logic. Chalmers University of Technology, 5 2008. (Cited on page 42.)

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000. (Cited on pages 6, 9, 91, and 103.)

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. (Cited on pages 9, 91, and 103.)

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. (Cited on pages 22 and 138.)

Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005. (Cited on pages 9, 91, and 103.)

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. (Cited on pages 8, 88, and 136.)

David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proceedings of CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004. (Cited on pages 8, 47, and 136.)

Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In Marie-Claude Gaudel and Jim Woodcock, editors, *Proceedings of FME*, volume 1051 of *LNCS*, pages 557–575. Springer, 1996. (Cited on page 87.)

Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *Proceedings of CAV*, LNCS. Springer, 2018. (Cited on page 138.)

Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012. (Cited on pages 22 and 138.)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT, 2009. ISBN 978-0-262-03384-8. (Cited on page 138.)

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Proceedings of POPL*, pages 269–282. ACM, 1979. (Cited on page 137.)

Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In Christine Rochange, editor, *Workshop on WCET Analysis*, volume 6 of *OASICS*. IBFI, 2007. (Cited on pages 8, 51, and 63.)

Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an online laboratory for program verification and analysis. In Claus Brabrand and Pierre-Etienne Moreau, editors, *Proceedings of LDTA*, page 3. ACM, 2010. (Cited on page 87.)

Sandeep Dalal and Rajender Singh Chhillar. Case studies of most common and severe types of software system failure. *Journal of Advanced Research in Computer Science and Software Engineering*, 2(8), 2012. (Cited on page 3.)

Sandeep Dalal and Rajender Singh Chhillar. Empirical study of root cause analysis of software failure. *ACM Software Engineering Note*, 38(4):1–7, 2013. (Cited on page 3.)

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. (Cited on pages 6, 42, 47, 52, 111, and 119.)

Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe, 2009. (Cited on page 43.)

Xianghua Deng, Jooyong Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of ASE*, pages 157–166. IEEE, 2006. (Cited on page 128.)

Xianghua Deng, Jooyong Lee, and Robby. Efficient and formal generalized symbolic execution. *Automated Software Engineering*, 19(3):233–301, 2012. (Cited on page 19.)

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of ISSTA*, pages 109–120. ACM, 2006. (Cited on pages 7, 27, 42, 46, 47, 49, 64, and 135.)

Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: how it works, and how to use it. In *Proceedings of FMCAD*, page 7. IEEE, 2014. (Cited on pages 6, 111, and 119.)

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. (Cited on pages 8 and 47.)

Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of FSE*, pages 195–204. ACM, 2007. (Cited on pages 7, 27, 46, 49, and 135.)

Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In Eran Yahav, editor, *Proceedings of SAS*, volume 6887 of *LNCS*, pages 351–368. Springer, 2011. (Cited on page 88.)

Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Proceedings of CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014. (Cited on pages 6, 47, and 111.)

Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Workshop on WCET Analysis*, volume 6 of *OASICS*. IFBI, 2007. (Cited on pages 8, 51, and 63.)

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of PLDI*, pages 234–245. ACM, 2002. (Cited on page 136.)

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Pldi 2002: Extended static checking for Java. *SIGPLAN Notices*, 48(4S):22–33, 2013. (Cited on pages 8 and 47.)

Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software: Practice and Experience*, 34(1):15–46, 2004. (Cited on page 87.)

Florian Frohn. Termination problems data base (TPDB) webpage. `http://termination-portal.org/wiki/Home`, 2014. Accessed: November 2017. (Cited on page 60.)

Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Transactions on Software Engineering*, 39(9): 1283–1307, 2013. (Cited on pages 7, 27, 42, 46, 47, 49, and 135.)

Malay K. Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In Soha Hassoun, editor, *Proceedings of ICCAD*, pages 794–801. ACM, 2006. (Cited on page 138.)

Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Proceedings of CAV*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007. (Cited on page 18.)

Aboubakr Achraf El Ghazi, Mattias Ulbrich, Christoph Gladisch, Shmuel S. Tyszberowicz, and Mana Taghdiri. Jkelloy: A proof assistant for relational specifications of Java programs. In Julia M. Badger and Kristin Yvonne Rozier, editors, *Proceedings of NFM*, volume 8430 of *LNCS*, pages 173–187. Springer, 2014. (Cited on page 88.)

Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017. (Cited on page 18.)

Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005. (Cited on page 22.)

Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of POPL*, pages 47–54. ACM, 2007. (Cited on page 129.)

Patrice Godefroid. Higher-order test generation. In Mary W. Hall and David A. Padua, editors, *Proceedings of PLDI*, pages 258–269. ACM, 2011. (Cited on page 128.)

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of PLDI*, pages 213–223. ACM, 2005. (Cited on page 128.)

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. (Cited on page 128.)

Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In Aarti Gupta and Sharad Malik, editors, *Proceedings of CAV*, volume 5123 of *LNCS*, pages 370–384. Springer, 2008. (Cited on pages 8, 51, and 63.)

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of POPL*, pages 127–139. ACM, 2009. (Cited on pages 51 and 63.)

Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In Neha Rungta and Oksana Tkachuk, editors, *Proceedings of SPIN*, pages 40–47. ACM, 2014. (Cited on pages 51, 60, and 64.)

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of POPL*, pages 331–344. ACM, 2011. (Cited on pages 9, 91, and 104.)

William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of POPL*, pages 71–82. ACM, 2010. (Cited on page 109.)

Mihai Herda. Generating bounded counterexamples for KeY proof obligations. Master's thesis, Karlsruher Institut für Technologie, 2014. (Cited on page 99.)

John N. Hooker. Solving the incremental satisfiability problem. *The Journal of Logic Programming*, 15(1-2):177–186, 1993. (Cited on page 129.)

Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Chen Fu, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. In Eric Bodden and Madanlal Musuvathi, editors, *Proceedings of ISSTA*, pages 1–6. ACM, 2012. (Cited on page 119.)

Mainul Islam and Christoph Csallner. Dsc+mock: a test case + mock class generator in support of coding against interfaces. In Jonathan Cook and James A. Jones, editors, *Proceedings of ISSTA*, pages 26–31. ACM, 2010. (Cited on pages 128 and 130.)

Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In Kousha Etessami

and Sriram K. Rajamani, editors, *Proceedings of CAV*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005. (Cited on page 138.)

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN 0262017156,9780262017152. (Cited on pages 28, 46, 88, 104, and 127.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings of NFM*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011. (Cited on page 109.)

Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. An interpolation method for CLP traversal. In Ian P. Gent, editor, *Proceedings of CP*, volume 5732 of *LNCS*, pages 454–469. Springer, 2009. (Cited on page 109.)

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Proceedings of CAV*, volume 7358 of *LNCS*, pages 758–766. Springer, 2012. (Cited on page 109.)

Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Proceedings of POPL*, pages 244–256. ACM, 1979. (Cited on page 136.)

Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In Richard A. DeMillo, editor, *Proceedings of POPL*, pages 66–74. ACM, 1982. (Cited on page 136.)

Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of ASE*, pages 590–600. IEEE, 2017. (Cited on page 109.)

Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing Java programs using alloy specifications. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *Proceedings of ASE*, pages 608–611. IEEE, 2011. (Cited on page 138.)

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. (Cited on pages 8 and 98.)

Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Proceedings of TGC*, volume 4661 of *LNCS*, pages 244–262. Springer, 2006. (Cited on page 47.)

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002. (Cited on page 137.)

Volker Klasen et al. *Verifying Dijkstra's Algorithm with KeY*. PhD thesis, Universität Koblenz-Landau, 2010. (Cited on page 43.)

Stephen Cole Kleene, NG de Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952. (Cited on page 136.)

Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006. (Cited on page 47.)

Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. (Cited on page 27.)

Robert-P Kurshan. *Computer-aided verification of coordinating processes: The Automata-theoretic Approach*. Princeton University Press, 1994. (Cited on pages 9, 91, and 103.)

Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, volume 15, page 35, 2011. (Cited on page 42.)

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. (Cited on pages 6, 20, 41, 42, 47, and 52.)

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In Shmuel Sagiv, editor, *Proceedings of ETAPS*, volume 3444 of *LNCS*, pages 124–140. Springer, 2005. (Cited on page 137.)

Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In Jens Palsberg, editor, *Proceedings of SAS*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000. (Cited on pages 22 and 137.)

Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of ISSTA*, pages 26–38. ACM, 2000. (Cited on page 137.)

Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science*, 5(2), 2009. (Cited on page 42.)

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of POPL*, pages 607–618. ACM, 2014. (Cited on pages 52 and 129.)

Jørn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In Nicolas Halbwachs and Doron A. Peled, editors, *Proceedings of CAV*, volume 1633 of *LNCS*, pages 316–327. Springer, 1999. (Cited on pages 9, 91, and 103.)

Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded program verification using an SMT solver: A case study. In Giuliano Antoniol, Antonia

Bertolino, and Yvan Labiche, editors, *Proceedings of ICST*, pages 101–110. IEEE, 2012. (Cited on pages 10 and 11.)

Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In Eran Yahav, editor, *Proceedings of HVC*, volume 8855 of *LNCS*, pages 284–299. Springer, 2014. (Cited on pages 15 and 129.)

Tianhai Liu, Shmuel S. Tyszberowicz, Mihai Herda, Bernhard Beckert, Daniel Grahl, and Mana Taghdiri. Computing specification-sensitive abstractions for program verification. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Proceedings of SETTA*, volume 9984 of *LNCS*, pages 101–117, 2016. (Cited on pages 13 and 14.)

Tianhai Liu, Shmuel S. Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. Computing exact loop bounds for bounded program verification. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Proceedings of SETTA*, volume 10606 of *LNCS*, pages 147–163. Springer, 2017. (Cited on page 12.)

Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of CGO*, pages 136–146. IEEE, 2009. (Cited on page 63.)

Feifei Ma, Jun Yan, and Jian Zhang. Solving generalized optimization problems subject to SMT constraints. In Jack Snoeyink, Pinyan Lu, Kaile Su, and Lusheng Wang, editors, *Proceedings of FAW-AAIM*, volume 7285 of *LNCS*, pages 247–258. Springer, 2012. (Cited on page 52.)

Robin Mange and Jonathan Kuhn. Verifying dijkstra algorithm in jahob. student project, EPFL, 2007. (Cited on page 43.)

Kenneth L. McMillan. Lazy annotation for program testing and verification. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Proceedings of CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010. (Cited on page 109.)

Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Proceedings of VSTTE*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012. (Cited on page 88.)

Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. ISBN 0-13-629155-4. (Cited on page 19.)

Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proceedings of RTCSA*, pages 161–166. IEEE, 2008. (Cited on pages 8, 51, and 63.)

Aleksandar Milicevic and Hillel Kugler. Model checking using SMT and theory of lists. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings of NFM*, volume 6617 of *LNCS*, pages 282–297. Springer, 2011. (Cited on pages 8, 51, and 63.)

Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of ICSE*, pages 771–774. IEEE, 2007. (Cited on page 138.)

Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of PLDI*, pages 221–231. ACM, 2001. (Cited on page 137.)

Michal Nagal. Bounded verification of an optimizedshortest path implementation. Master's thesis, KIT, 2013. (Cited on page 43.)

ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. Syminfer: inferring program invariants using symbolic states. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of ASE*, pages 804–814. IEEE, 2017. (Cited on page 109.)

Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992. (Cited on pages 8 and 136.)

Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: achieving higher statement coverage faster. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *Proceedings of FSE*, page 35. ACM, 2012. (Cited on page 130.)

Corina S. Pasareanu and Neha Rungta. Symbolic pathfinder: symbolic execution of Java bytecode. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *Proceedings of ASE*, pages 179–180. ACM, 2010. (Cited on pages 110 and 128.)

Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Proceedings of SPIN*, volume 2989 of *LNCS*, pages 164–181. Springer, 2004. (Cited on page 109.)

John Plevyak, Andrew A. Chien, and Vijay Karamcheti. Analysis of dynamic structures for efficient parallel execution. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Proceedings of LCPC*, volume 768 of *LNCS*, pages 37–56. Springer, 1993. (Cited on page 136.)

Jan Reineke. Shape analysis of sets. In *OASIcs-OpenAccess Series in Informatics*, volume 3. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006. (Cited on page 137.)

John C. Reynolds. An overview of separation logic. In Bertrand Meyer and Jim Woodcock, editors, *proceedings of VSTTE*, volume 4171 of *LNCS*, pages 460–469. Springer, 2005. (Cited on page 138.)

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In Jeanne Ferrante and P. Mager, editors, *Proceedings of POPL*, pages 12–27. ACM, 1988. (Cited on page 113.)

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. (Cited on page 136.)

Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with ${\mathcal LA}$ (ll) cost functions. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of IJCAR*, volume 7364 of *LNCS*, pages 484–498. Springer, 2012. (Cited on page 52.)

Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. In Daniel Kroening and Corina S. Pasareanu, editors, *Proceedings of CAV*, volume 9206 of *LNCS*, pages 447–454. Springer, 2015. (Cited on pages 6, 52, and 119.)

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of FSE*, pages 263–272. ACM, 2005. (Cited on page 128.)

Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of FM*, volume 5850 of *LNCS*, pages 757–772. Springer, 2009. (Cited on pages 42 and 135.)

Danhua Shao, Divya Gopinath, Sarfraz Khurshid, and Dewayne E. Perry. Optimizing incremental scope-bounded checking with data-flow analysis. In *Proceedings of ISSRE*, pages 408–417. IEEE, 2010. (Cited on pages 42 and 135.)

Olha Shkaravska, Rody Kersten, and Marko C. J. D. van Eekelen. Test-based inference of polynomial loop-bound functions. In Andreas Krall and Hanspeter Mössenböck, editors, *Proceedings of PPPJ*, pages 99–108. ACM, 2010. (Cited on page 63.)

Junaid Haroon Siddiqui and Sarfraz Khurshid. Parsym: Parallel symbolic execution. In *Proceedings of ICSTE*, volume 1, pages V1–405. IEEE, 2010. (Cited on pages 128 and 129.)

Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of SSV*. USENIX, 2010. (Cited on pages 22 and 138.)

Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of ISSTA*, pages 183–194. ACM, 2010. (Cited on page 129.)

Wilfried Steiner. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. In *Proceedings of RTSS*, pages 375–384. IEEE, 2010. (Cited on page 129.)

Jan Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Inf. Comput.*, 101(1):70–102, 1992. (Cited on page 136.)

Mana Taghdiri. *Automating modular program verification by refining specifications*. PhD thesis, Massachusetts Institute of Technology, 2008. (Cited on pages 7, 21, 27, 46, 49, and 135.)

Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007. (Cited on pages 9, 94, 103, and 104.)

Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: guided path exploration for efficient regression test generation. In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of ISSTA*, pages 1–11. ACM, 2011. (Cited on pages 128 and 129.)

Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, 2004. (Cited on pages 8, 51, and 63.)

Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Proceedings of TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. (Cited on pages 110 and 128.)

Emina Torlak, Mandana Vaziri, and Julian Dolby. Memsat: checking axiomatic specifications of memory models. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of PLDI*, pages 341–350. ACM, 2010. (Cited on pages 7, 21, 27, 46, 49, and 135.)

Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Proceedings of TACAS*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001. (Cited on page 47.)

Mandana Vaziri-Farahani. *Finding bugs in software with a constraint solver*. PhD thesis, Massachusetts Institute of Technology, 2004. (Cited on pages 7, 27, 33, 46, 49, and 135.)

Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In Bernhard Beckert and Reiner Hähnle, editors, *Proceedings of TAP*, volume 4966 of *LNCS*, pages 154–170. Springer, 2008. (Cited on page 64.)

Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2): 203–232, 2003. (Cited on page 65.)

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *Proceedings of FSE*, page 58. ACM, 2012. (Cited on pages 109, 110, and 128.)

Milena Vujosevic-Janicic and Viktor Kuncak. Development and evaluation of LAV: an smt-based error finding platform - system description. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Proceedings of VSTTE*, volume 7152 of *LNCS*, pages 98–113. Springer, 2012. (Cited on page 138.)

Dolores R Wallace and D Richard Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. *Journal of Reliability, Quality and Safety Engineering*, 8(04):351–371, 2001. (Cited on page 3.)

Benjamin Weiß. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011. (Cited on pages 21 and 24.)

Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of DAC*, pages 542–545. ACM, 2001. (Cited on page 129.)

Siert Wieringa. *Incremental Satisfiability Solving and its Applications*. PhD thesis, Aalto University, 2014. (Cited on page 129.)

Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin C. Rinard. Verifying complex properties using symbolic shape analysis. *The Computing Research Repository*, abs/cs/0609104, 2006. (Cited on page 137.)

Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013. (Cited on page 6.)

Guowei Yang, Sarfraz Khurshid, and Corina S. Pasareanu. Memoise: a tool for memoized symbolic execution. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *Proceedings of ICSE*, pages 1343–1346. IEEE, 2013. (Cited on page 109.)

Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology*, 24(1):3:1–3:42, 2014. (Cited on pages 128 and 129.)

Kuat T Yessenov. A lightweight specification language for bounded program verification. Master's thesis, Massachusetts Institute of Technology, 2009. (Cited on page 43.)

Publications

1. **Tianhai Liu**, Michael Nagel, and Mana Taghdiri. Bounded Program Verification Using an SMT Solver: A Case Study. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST), Canada, Proceedings.* pages 101-110. IEEE, 2012.
2. **Tianhai Liu**, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In Eran Yahav, editors, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference (HVC), Israel, Proceedings.* volume 8855 of *LNCS*, pages 284–299. Springer, 2014.
3. **Tianhai Liu**, Shmuel S. Tyszberowicz, Mihai Herda, Bernhard Beckert, Daniel Grahl, and Mana Taghdiri. Computing Specification-Sensitive Abstractions for Program Verification. In Martin Fränzle and Deepak Kapur and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium (SETTA), China, Proceedings.* volume 9984 of *LNCS*, pages 101–117. Springer, 2016.
4. **Tianhai Liu**, Shmuel S. Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. Computing Exact Loop Bounds for Bounded Program Verification. In Kim G. Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Third International Symposium (SETTA), China, Proceedings.* volume 10606 of *LNCS*, pages 147–163. Springer, 2017.
5. Daniel Grunwald, Christoph Gladisch, **Tianhai Liu**, Mana Taghdiri, and Shmuel S. Tyszberowicz. Generating JML Specifications from Alloy Expressions. In Eran Yahav, editors, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference (HVC), Israel, Proceedings.* volume 8855 of *LNCS*, pages 99–115. Springer, 2014.

Erklärung

Ich versichere wahrheitsgemäß, die Dissertation bis auf die angegebenen Hilfen selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und als kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.