ThanhVu Huy Nguyen

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Stephanie Forrest, Chairperson

Deepak Kapur, Co-Chairperson

Darko Stefanovic

Westley Weimer

Automating Program Verification and Repair Using Invariant Analysis and Test Input Generation

by

ThanhVu Huy Nguyen

B.S., Computer Science, Penn State University, 2003

M.S., Computer Science, Penn State University, 2006

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2014

Dedication

To my parents, Thanh and Thu, for their love, support, and encouragement.

Acknowledgments

The research that has gone into this dissertation has been thoroughly enjoyable. This enjoyment is largely a result of the interaction that I have had with my colleagues and also of the support by my family and friends. Their supports and encouragements are the major source of the joy of my years in New Mexico.

First and foremost, I thank my advisers Stephanie Forrest and Deepak Kapur. I thank Stephanie for taking me as her research assistant since my arrival at UNM and especially for the patience and time she invested to make me become a more mature researcher. I thank Deepak for introducing me to the research field of program analysis, for the countless discussions about mathematics and logic, and importantly, for always requiring me to think more.

On the academic side, I thank Westley Weimer for providing many technical resources for my research as well as valuable guidance during my job searching process. I thank Darko Stefanovic for joining my dissertation committee and providing helpful comments when reviewing this dissertation. I thank Matthias Horbach and Henjun Zhao for helping me in obtaining a fresh view on some of my approaches and in improving both my results and my writings. Thanks to colleagues and faculties from the CS department, who were always ready to discuss and solve technical problems: George Bezerra, Ben Edwards, Roya Ensafi, Michael Groat, Josh Karlin, Drew Levin, Lu Qi, Eric Shulte, Oleg Semenov, George Stelle, Jed Crandall, and Shuang Luan. I particularly enjoy many enlightening discussions and competitions with my office-mate George Bezerra. In addition, I am grateful for my friends in Albuquerque: Kun Huang, Thwan Goei, Yasushi Onigasawa, Dung Tang, and Han Tran. These guys and their families made my life as a graduate student much brighter with dinners, get-togethers, and sport activities.

This dissertation could not be completed without the endless love and continuous support from my family, my parents Thanh and Thu Nguyen. A heartfelt thanks goes out to my girlfriend Ha Nguyen for her patience when I was thinking only about strange formulas and while I was writing this dissertation.

Finally, I thank the (yet to come) readers of this dissertation for their interest in my work. I sincerely hope that you can benefit from it.

Automating Program Verification and Repair Using Invariant Analysis and Test Input Generation

by

ThanhVu Huy Nguyen

B.S., Computer Science, Penn State University, 2003

M.S., Computer Science, Penn State University, 2006

PhD., Computer Science, University of New Mexico, 2014

Abstract

Software bugs are a persistent feature of daily life—crashing web browsers, allowing cyberattacks, and distorting the results of scientific computations. One approach to improving software uses program invariants—mathematical descriptions of program behaviors—to verify code and detect bugs. Current invariant generation techniques lack support for complex yet important forms of invariants, such as general polynomial relations and properties of arrays. As a result, we lack the ability to conduct precise analysis of programs that use this common data structure. This dissertation presents DIG, a static and dynamic analysis framework for discovering several useful classes of program invariants, including (i) nonlinear polynomial relations, which are fundamental to many scientific applications; disjunctive invariants, (ii) which express branching behaviors in programs; and (iii) properties about multidimensional arrays, which appear in many practical applications. We describe theoretical and empirical results showing that DIG can efficiently and accurately find many important invari-

ants in real-world uses, e.g., polynomial properties in numerical algorithms and array relations in a full AES encryption implementation.

Automatic program verification and synthesis are long-standing problems in computer science. However, there has been a lot of work on program verification and less so on program synthesis. Consequently, important synthesis tasks, e.g., generating program repairs, remain difficult and time-consuming. This dissertation proves that certain formulations of verification and synthesis are equivalent, allowing for direct applications of techniques and tools between these two research areas. Based on these ideas, we develop CETI, a tool that leverages existing verification techniques and tools for automatic program repair. Experimental results show that CETI can have higher success rates than many other standard program repair methods.

Contents

Contents

Contents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"If I have a thousand ideas and only one turns out to be good, I am satisfied." – Alfred Bernhard Nobel[1]

Real software is buggy. Automated program analysis techniques and tools can improve it. In this dissertation, we develop efficient techniques and practical tools to capture precise program invariants to understand and verify programs. We also establish formal theories linking different areas of verification and synthesis, enabling application of tools from mature fields such as test input generation to the field of automatic program repair.

1.1 Motivation

Since the invention of computers, writing correct programs has been considered a great challenge. Generally, it takes much more time, effort, and money to debug,

---

[1]Swedish chemist, engineer and inventor of dynamite, who used his enormous fortune to institute the Nobel Prizes (1833 – 1896). In context, this dissertation has two ideas: generating powerful invariants to verify programs and applying verification techniques to synthesize program repairs.

i.e., find and eliminate errors, than to actually write programs. Some reports state software maintenance, of which debugging is a major component, accounts for as much as 90% of the total software production cost [Seacord et al., 2003]. A recent 2013 Cambridge University study estimates the global cost of software debugging at US $312 billion annually and finds that software developers spend over half of their programming time "fixing bugs" or "making code work" [Britton et al., 2013].

As software becomes omnipresent, the consequences of bugs become more significant, causing great financial and even human losses. Software defects range from simple misbehaviors to costly errors, such as the Ariane-5 crash [Dowson, 1997], and lethal catastrophes, such as the Therac-25 radiation therapy machines [Leveson, 1993]. Moreover, software today is incredibly complex, making it harder to debug. McConnel estimates in [McConnell, 2004] that delivered industry programs contain from 15 to 50 bugs for every 1000 lines of code. Major software projects are often forced to ship with both known and unknown bugs because they lack the development resources to deal with every defect [Liblit et al., 2003].

Software quality as a concern has motivated research and development on program analysis tools to help developers create more reliable software. A 2002 NIST report [Tassey, 2002] found that more than a third of debugging cost could be eliminated by enabling "earlier and more effective identification and removal of software defects." The aforementioned 2013 Cambridge study claims that analysis tools can decrease debugging time by an average of 26%, which translates to saving 13% of total programming time and $41 billion dollars annually.

1.2   Contributions

This dissertation focuses on automatic program verification and synthesis to improve software reliability. Program verification checks whether a program satisfies its spec-

ification while synthesis generates a program that meets a given specification. Thus, verification can reveal errors before the program is deployed, and synthesis can relieve developers from tedious programming details.

A popular approach to automatic verification is to generate program invariants, i.e., properties that hold of a given program, and use them to prove required specifications. Invariants can be computed using static analysis that examines the program code directly and dynamic analysis that learns about the program by running it. Both purely static and purely dynamic methods have drawbacks; static analysis produces sound results, but is usually expensive while dynamic analysis is more efficient, but might give incorrect invariants. Moreover, current approaches lack support for challenging yet important forms of invariants, including nonlinear polynomial relations, disjunctive invariants, and properties of data structures such as arrays.

Due to the difficulty of generating complete programs from scratch, practical synthesis methods typically create code under specific templates from partially complete programs. Such a template-based synthesis method can be applied to repair programs, i.e., modifying a buggy program to pass its specification. Automated program repair is a valuable approach for reducing software cost, and synthesis holds the promise to generate correct-by-construction repairs automatically. In general, program synthesis and repair are becoming more popular even though these research areas are not as mature as program verification, which has witnessed significant development in the last three decades.

The thesis of this dissertation is that *we can build expressive and efficient techniques to automatically discover program invariants and synthesize program repairs by encoding these tasks as solutions to existing problem instances in the mathematical and verification domains.* We reduce programs invariants to a set of equations and constraints, which can be solved by mechanical and efficient constraint solving techniques. We also show that certain formulations of verification and synthesis

are equivalent, facilitating the exchange of ideas and optimizations between different fields. However, invariant generation and program synthesis cannot be encoded directly as constraint solving and program verification problems. Thus, we develop theoretical work to formally connect these problems, so that solutions to constraint solving and verification problems directly map to discovered invariants and synthesized programs, respectively.

1.2.1   Invariant Generation

We present and evaluate DIG (Dynamic Invariant Generator), a hybrid tool that dynamically infers invariants from program execution traces and statically verifies candidate results against program code. DIG supports both conjunctive and disjunctive forms of nonlinear polynomial invariants. Polynomials are fundamental to many scientific and engineering applications, e.g., nonlinear polynomials are useful for the analysis of hybrid systems [Roozbehani et al., 2005, Sankaranarayanan et al., 2005]. Disjunctive invariants express the semantics of conditional statements and, thus, capture path-sensitive reasoning, such as those found in most sorting and searching tasks. At its heart, DIG interprets nonlinear polynomial formulas as convex geometric objects in high-dimensional space, such as hyperplanes and polyhedra. This representation allows the tool to employ mathematical techniques, e.g., equation solving and convex hull constructions, to dynamically generate conjunctive polynomial invariants. To identify a class of disjunctive polynomial invariants, DIG represents these relations as non-convex geometric objects using the non-standard "max-plus" and "min-plus" algebras. For efficiency, DIG also supports simpler forms of invariants expressible using more restricted geometric shapes such as octagons. Most dynamic methods have no guaranteed results; however, geometric reasoning ensures that DIG does not overapproximate the true program invariants if they are expressible using the supported forms. Finally, by checking candidate results with a custom k-inductive SMT theorem prover, DIG removes spurious results and produces

only true invariants.

DIG also discovers complex array properties, such as nested relations among multidimensional array variables, that appear in many practical applications. For example, over one half of the required invariants in a real-world AES (Advanced Encryption Standard) implementation involve relations among multidimensional arrays. We first formalize the problem of finding nested array relations and show its relationship to the problem of decomposing functions. We then prove that both problems can be solved in polynomial time in the number of array elements, but are NP-complete in the number of arrays or functions involved. Such theoretical results establish the run-time complexity of the array nesting and function composition problems and suggest directions to develop algorithms for solving them. To implement algorithms for finding array relations, DIG employs equation solving, performs reachability analysis, and then encodes the problem as a satisfiability query that can be handled using an SMT solver. The integration of equation and SMT solvers allows efficient analysis of complex array properties, such as those in AES, that have not been previously considered by either static or dynamic methods.

1.2.2 Program Repair

Program reachability, which decides if a program location is reachable, is a formulation of program verification that checks for the absence of program errors. Template-based program synthesis, which constructs programs under pre-specified forms to meet a required specification, is a practical approach to synthesizing programs. We present a constructive proof that program reachability and template-based program synthesis are equivalent. We encode a synthesis problem into a program consisting of a special location, reachable only when code could be generated for the synthesis problem. Conversely, we show that a reachability query can be reduced to a synthesis task such that a successful synthesis indicates the reachability of the targeted

location. These results demonstrate a link between the two subfields and, thus, allow exchange and combination of knowledge between them.

Based on these ideas, we develop CETI (Correcting Errors using Test Inputs), a tool for automated synthesis using test input generation techniques that solve reachability problems. CETI transforms a buggy program and its required specification into a specific program containing a location reachable only when the original program can be repaired. The transformed program is then used as input to an off-the-shelf test input generation tool to find test values that can reach the desired location. Those test values correspond exactly to repairs for the original program. Experimental case studies suggest that CETI has higher success rates than many other standard approaches.

1.3 Organization

The rest of this dissertation is structured as follows. Chapter 2 provides an overview of program verification and synthesis, focusing on the state of the art in automatic invariant generation, program repair, and test input generation. Chapters 3, 4, and 5 comprise the main research contributions of the dissertation. Chapter 3 presents the invariant analysis tool DIG and describes its geometric approach to finding polynomial invariants and its use of k-induction theorem proving to verify candidate invariants. Chapter 4 establishes the theoretical framework of function composition and array nesting problems, and then evaluates implementation techniques in DIG to find array invariants. Chapter 5 connects the program verification and synthesis problems and leverages this connection to develop the automatic repair tool CETI. Chapter 6 suggests future directions and offers concluding remarks.

Parts of the research on polynomial invariants in Chapters 3 and 4 have been published as conference papers [Nguyen et al., 2012, 2014c], which were presented

at the International Conference on Software Engineering in 2012 and 2014, respectively. A journal version [Nguyen et al., 2014a] of Chapters 3 and 4 is in press in the Transactions on Software Engineering and Methodology. The work on program synthesis and repair, described in Chapter 5, corresponds to a conference paper [Nguyen et al., 2014b], which was recently submitted to the Symposium on the Foundations of Software Engineering. These papers were co-written with Deepak Kapur, Stephanie Forrest, and Westley Weimer. The uses of "we", "our", and "us" in this dissertation refer to all four authors.

Chapter 2

Background and Related Work

"The history of mankind is the history of ideas." – Luigi Pirandello[1]

This chapter contains the background information of program verification and synthesis. We first provide an overview of the research done in these areas, focusing on invariant generation as an approach to verifying programs and program repair as an application of program synthesis. We then review the state of the art in the subfields of automatic invariant inference, program repair, and test input generation.

2.1 Program Verification

Program verification aims to automatically check that a computer program satisfies a given *specification* or formal property. Typically, formal properties are expressed using logical formulas encoding *correctness* or safety requirements, such as "the implementation of the cosine function returns values between -1 and $+1$" or "the program does not produce buffer-overflow errors." The two main approaches to verifying pro-

---

[1]Italian short-story writer, dramatist, and novelist, who was awarded the Nobel Prize in Literature in 1934 for his "bold and brilliant renovation of the drama and the stage," (1867 – 1936).

grams are static, which examines the program using its static representation, e.g., the program code, and dynamic analysis, which learns about the program by running it on sample inputs. There are also many fruitful combinations of static and dynamic analyses, i.e., hybrid approaches.

2.1.1 Static and Dynamic Program Analysis

Programs can be studied using *static analysis*, which inspects the program code without executing the program. Some static methods associate mathematical meanings to programs to analyze them formally and, thus, achieve *sound* results on all possible program behaviors. These *formal methods* for program verification have been used to validate safety-critical software. For example, the Astrée [Blanchet et al., 2003, Cousot et al., 2005] static analyzer has been applied to verify the absence of run-time errors in both the Airbus A340/A380 avionic systems and the docking software used in vehicles transporting payloads to the International Space Station. Moreover, industrial hardware and software manufacturers such as IBM, Intel and Microsoft have been developing and employing formal verification tools to improve the quality of their products [Ball et al., 2004, Kaivola et al., 2009, Schubert et al., 2011, Woodcock et al., 2009].

Formal static analysis can produce sound results, but it is computationally expensive and often does not scale up to complex programs. In contrast, *dynamic analysis* learns about programs from traces gathered from program executions over sample inputs. The accuracy of these results depends on the quality and completeness of the test inputs. However, by focusing on finite program traces, dynamic analysis is generally efficient and scales well to large programs. For these reasons, dynamic methods have received considerable attention in practice. An oft-quoted rule of thumb is that at least fifty percent of a commercial software project's budget is devoted to dynamic methods for testing software.

The major methods for formal program verification are model checking and theorem proving. The primary method for dynamic program checking is software testing.

- *Model Checking.* Given a model of a program, a model checker exhaustively checks if all reachable states of the model satisfy a property. If the property is not satisfied, the checker returns useful information for debugging, e.g., input values (*counterexamples*) that cause the violation. The approach, introduced by Clark and Emmerson [Clarke et al., 1986, 1999], has been applied successfully to many medium-sized *finite-state* programs, e.g., hardware designs or communication protocols [Bryant, 1986, Carbonell, 2006]. However, the method faces the *state explosion* problem, an exponential growth in state space when dealing with larger programs such as typical imperative programs with infinite states. To cope with this problem, model checking often employs approximations such as bounding the number of loop iterations [Biere et al., 1999] or representing large programs using smaller finite-state models [Clarke et al., 1994].

- *Theorem Proving.* A way to overcome the aforementioned state-explosion problem is to encode a program and a required property into a verification condition, i.e., a logical formula whose validity implies the correctness of the program with respect to the property. For imperative programs, formal rules from Floyd-Hoare logic [Floyd, 1967, Hoare, 1969] are often used to generate verification condition formulas. These formulas are then validated using a constraint solver such as a SAT (satisfiability) or SMT (Satisfiability Modulo Theories) solver[2]. Recent advances in constraint solving (also referred to as automatic theorem proving) allow for the efficient and automatic verification of complex formu-

---

[2]*SAT* solvers [Eén and Sörensson, 2004, Moskewicz et al., 2001] determine the satisfiability of formulas with boolean values while *SMT* solvers [De Moura and Bjørner, 2008, Dutertre and de Moura, 2006] operate on expressive formulas involving numerical variables or data structures such as lists and arrays.

las [Jovanović and De Moura, 2012]. However, this approach is not entirely automatic because the Floyd-Hoare method requires the user to annotate the program with asserted properties at various program locations, e.g., properties of loops and pre-/post-conditions of functions in the program.

- *Dynamic Software Testing.* In contrast to formal methods that verify a program using a formal specification, this dynamic approach tests a program using a *test-suite* or input-output specification, i.e., a set of finite pairs of inputs and expected outputs of the program. The process involves running the program on the test inputs and comparing the results to expected outputs defined by the programmer or by a reference program that is known to be correct. The results can also be examined to show presence of certain types of fatal errors such as division by zero and null-pointer dereferencing. Testing is the traditional and most popular approach to finding program defects before the software is deployed. However, as noted by Dijkstra that *"testing can be used to show the presence of bugs, but never to show their absence"* [Dahl et al., 1972], testing easily leaves bugs because it can observe only a limited number of program behaviors from finite tests.

  The demand for good test inputs leads to research interests in *test input generation*, an active software testing field that aims to create high-coverage test inputs to find deep errors in complex software. Test inputs can be generated using both static and dynamic methods, e.g., *concolic testing* [Burnim and Sen, 2008, Cadar et al., 2008a, Sen and Agha, 2006] combines static and dynamic analyses to create program path constraints that can be solved for inputs executing those program paths. Essentially, these test input generation techniques can be viewed as heuristics for solving the *program reachability* problem [Abdulla and Potapov, 2013] that checks if a particular program state containing an error is reachable. We review the state of the art in test input generation in Section 2.3.2.

2.1.2   Invariant Generation

A property met at a program location on every program execution is called an *invariant property* of that program location. For example, a loop invariant is a property that must hold when entering a loop. *Program invariant generation* aims to automatically discover invariant properties at certain program locations. Generated invariants are typically used to automate and guide formal program verification processes. For example, theorem proving techniques based on Floyd-Hoare logic can use discovered properties such as loop invariants to prove correctness of programs. Model checkers can employ invariants to prune the search space and reduce the state explosion problem. Program correctness can also be verified with invariants, by generating sufficiently strong invariants to imply the required property. In addition to program verification, invariants are useful in other phases of programming, including documentation, design, coding, testing, debugging, optimization, and maintenance [Ernst, 2000, Kataoka et al., 2001, Perkins et al., 2009]. In short, discovering invariants is critical for both program verification and general software development [Ernst et al., 2007, Jones et al., 1993, Karr, 1976], and it has been an active research area since the 1970s [Dershowitz and Manna, 1978, German and Wegbreit, 1975, Karr, 1976, Katz and Manna, 1976, Suzuki and Ishihata, 1977, Wegbreit, 1974].

The main approaches to statically and dynamically generating invariants are abstract interpretation and dynamic invariant inference, respectively.

- *Abstract Interpretation.* The abstract interpretation framework, introduced by P. Cousot and R. Cousot [Cousot and Cousot, 1976, 1977, Cousot and Halbwachs, 1978], automatically computes an invariant property that abstracts or overapproximates the set of (potentially infinite) reachable program states. The method starts from a weak invariant representing an initial approximation and gradually improves the invariant based on the structure of the program until no

more improvements can be made (a fixed point). The resulting invariant is then used to prove the absence of errors: if the overapproximation of the reachable states does not intersect the set of bad states, then the program is guaranteed to never reach a bad state, and thus, is safe. Abstract interpretation, with the capability of handling infinite-state programs automatically, has been employed to verify mission-critical systems such as the Airbus avionic systems [Cousot et al., 2005]. However, the abstraction process can lead to loss of information and produce false positives, i.e., the analysis may detect an error that does not actually exist. Thus, a major research direction in this area is to find *abstract domains* that can be implemented efficiently and are sufficiently expressive to retain key information from the original program. Techniques implementing abstracting interpretation also need to design an appropriate *widening* heuristic operator for fast convergence and termination [Cortesi, 2008, Cortesi and Zanioli, 2011, Cousot and Cousot, 1992].

- *Dynamic Invariant Inference.* A dynamic invariant detector [Ernst, 2000, Ernst et al., 2007, Perkins and Ernst, 2004] is typically initialized with a pre-defined collection of invariant templates postulated to be useful and likely to occur in programs. The detector filters out invalid templates based on observed program traces and returns the remainders as candidate invariants. Depending on the completeness of given traces, dynamic invariant analysis can produce *spurious invariants* that match some observations, but are not sound with respect to general program behaviors. Moreover, the approach has limited support for invariants that are inexpressible using the pre-defined templates. Examples of such invariants include general polynomial relations over numerical variables and properties of the array data structure. However, dynamic invariant techniques are generally efficient and scale well to large programs. Recently, for example, dynamically generated invariants have been used to prevent security attacks in Mozilla Firefox [Perkins et al., 2009].

The above classification of static and dynamic program analyses does not imply that these approaches are mutually exclusive. In fact, there has been fruitful cross-fertilization leading to hybrid methods such as the aforementioned "concolic" testing method to create program inputs. In this dissertation, we combine static and dynamic analyses to create sound and efficient techniques to generate expressive program invariants.

2.2   Program Synthesis

Program synthesis, which aims to automatically generate a program to meet a given specification [Manna and Waldinger, 1980, Srivastava et al., 2013], has been a "dream" of programming research since the 1960s [Manna and Waldinger, 1979, Solar-Lezama, 2008] and was considered by Pnueli in 1989 as *"one of the most central problems in the theory of programming"* [Pnueli and Rosner, 1989]. By construction, automated synthesis creates programs that are provably correct with respect to given specifications, relieving the tedium and error associated with low-level programming details and pushing the problem of correctness to the specification level. Moreover, synthesis could discover new non-trivial programs that are difficult for programmers to build [Srivastava et al., 2010]. In general, the goal of synthesis is that the user should be able to tell the computer *what* to do and let the synthesizer discover *how* to do it correctly and efficiently.

Despite the promise of significantly easing programming and verification, less research effort has been directed toward synthesis in comparison to program verification. Fully automatic synthesis is notoriously difficult, as Manna and Waldinger pointed out in 1979 that *"programming is one the most demanding of human activities, and is among the last tasks that computers will do well"* [Manna and Waldinger, 1979]. Researchers in the field realized that a synthesizer is unlikely to have the "intuition" to discover algorithms and implementation techniques whose original discovery

had challenged the ingenuity of the brightest minds in the fields. Consequently, practical synthesis approaches incorporate human insights to guide the synthesis process. Indeed, the success of synthesis depends on a proper synergy between the user and the synthesizer.

2.2.1 Deductive and Template-based Synthesis

Two well-known program synthesis approaches are deductive synthesis, which generates a program from a constructive proof of a given specification, and template-based synthesis, which automatically synthesizes partial programs from given templates.

- *Deductive Synthesis.* This method, pioneered by Manna and Waldinger [Manna and Waldinger, 1980, 1971], extracts a program from the satisfiability proof of a formula encoding the required specification, e.g., $\forall x.\ \exists y.\ pre(x) \Rightarrow post(x, y)$, where x and y are the input and expected output from a given test-suite specification. The synthesizer iteratively applies deductive rules to prove the formula and generates concrete program constructs corresponding to the proof rules. For example, a case split rule in the proof leads to a conditional branch and an induction rule leads to program loops. Deductive synthesis is not automatic because it involves the user's assistance in the proof process. However, with proper human-assistance, the method can be powerful and practical. For example, it has been used to synthesize an adaptive network protocol from formal specifications [Bickford et al., 2001] and an airlift scheduler for the Air Force [Emerson and Burstein, 1999].

- *Template-based Synthesis.* Template-based synthesis [Solar-Lezama, 2008, Srivastava, 2010, Srivastava et al., 2013] is a popular approach to generating codes automatically for partially complete programs. To reduce the search space, the approach creates code from specific *templates* instead of attempting to generate

arbitrary code. A synthesis template expresses specific forms of program constructs, but it also includes template parameters representing low-level details to be filled in by the synthesis process. The synthesizer first maps the synthesis templates and the required specification into a logical formula, and it then applies a constraint solver to find values for the parameters that satisfy the formula. Instantiating the synthesis templates with those values yields a complete program meeting the required specification. Template-based synthesis has been applied to create various sorting and geometric algorithms [Solar-Lezama, 2008, Srivastava et al., 2013]. Recently, it has been employed to synthesize patches to fix program errors [Könighofer and Bloem, 2013, Nguyen et al., 2013].

2.2.2   Program Repair

Program repair is a form of program synthesis that aims to automatically modify a program failing a given specification so that it passes that specification. Automatic program repair has tremendous value because debugging continues to be a mostly manual, time-consuming, and, thus, expensive task in software development and maintenance. For example, developers take 28 days on average to address security-critical defects and new general defects are reported faster than developers can handle them [Turner et al., 2006]. This has driven automatic repair tools to leverage cheap and abundant computer cycles to reduce costs and the burden on developers. Since automated program repair was demonstrated on real-world problems in 2009, interest in the field has grown steadily with multiple novel repair techniques proposed.

Two main approaches to program repair are generate-and-validate, which creates multiple candidate repairs and checks them against given specifications, and constraint-based repair, which generates correct-by-construction repairs.

- *Generate-and-validate.* Given a program that has an error, which violates a

test-suite specification consisting of inputs and expected outputs, this method first localizes the error to a small code region, then generates multiple repair candidates (e.g., using stochastic search) for the suspicious code region, and finally validates these candidates against the specification. Genetic algorithms are a well-known search method that has been successfully employed to find repairs for complex applications. A genetic algorithm-based repair technique [Weimer et al., 2009] searches for repairs using genetic operators including deleting existing program statements and inserting or swapping statements from other parts of the program. To reduce the search space, the technique constructs repairs only from extant code and, thus, lacks the ability to introduce new code, which might be necessary to repair programs.

Instead of using genetic algorithm, other techniques apply random mutations on repair templates to create repairs of specific forms [Debroy and Wong, 2010, Kim et al., 2013]. Some techniques employ dynamically inferred invariants to help guide the repair process, e.g., deriving repairs that minimize differences between expected and unexpected invariants mined from passing and failing program runs [Dallmeier et al., 2009, Perkins et al., 2009, Wei et al., 2010].

- *Constraint-based Repair.* This approach applies template-based program synthesis to automatic program repair [Bloem et al., 2013, Gopinath et al., 2011, Jin et al., 2011, Nguyen et al., 2013]. Once the defect has been localized to a specific code region, the synthesizer replaces statements in that region using synthesis templates. Next, the synthesizer applies symbolic or concolic execution techniques to encode the program with synthesis templates and its required specification into a satisfiability formula. Finally, the synthesizes employs a constraint solver to find values that satisfy the formula. These values correspond to repairs to the original buggy program, allowing it to satisfy the given specification. Several studies have shown that constraint-based repair has higher success-rate than genetic algorithm-based techniques and produces

program repairs faster [Bloem et al., 2013, Nguyen et al., 2013].

There has been more research on program verification than on program synthesis, even though both are long-standing problems in computer science research. In this dissertation, we prove that certain formulations of the verification and synthesis problems are equivalent, and this connection creates opportunities for collaboration between researchers in these fields. To demonstrate the connection, we develop a technique for automatic program repair using an off-the-shelf test input generation tool.

2.3   Related Work

This research focuses on using invariants to verify programs and using test input generation techniques to synthesize program repairs. In this section, we review related work in automatic invariant inference, program repair, and test input generation. We also highlight the main differences among these techniques and those developed in this dissertation.

2.3.1   Invariant Generation

As mentioned in Section 2.1.2, the main static and dynamic approaches to invariant generation are abstract interpretation and dynamic invariant inference, respectively. Some work, including our invariant detection tool DIG introduced in Section 1.2, combine both static and dynamic analyses to efficiently discover sound and expressive program invariants.

Static Invariant Inference

Rodríguez-Carbonell *et al.* [Carbonell, 2006, Rodríguez-Carbonell and Kapur, 2007] provide an abstract interpretation framework to generate conjunctions of nonlinear

polynomial equalities. They first observe that a set of polynomial invariants forms the algebraic structure of an ideal, then compute the polynomial invariants using Gröbner basis and operations over the ideals, based on the structure of the program until reaching a fixed point. The technique can analyze precisely only programs with assignments and loop guards that are expressible as polynomial equalities. To ensure termination when analyzing programs with nested loops, the method uses a widening operator that pre-specifies an a priori bound on polynomial degrees. Related work [Carbonell and Kapur, 2007] from the same authors removes the requirement for upper bounds on polynomial degrees but is restricted to programs with non-nested loops. These techniques do not support the conjunctive inequalities, disjunctive polynomial relations, or array invariants considered in this dissertation.

Allamigeon *et al.* [Allamigeon, 2009, Allamigeon et al., 2008] use abstract interpretation to approximate disjunctive program properties under the max and min-plus domains. The method first computes a formula representing an initial approximation of the program state space and gradually improves that approximation based on the program structure until a fixed point is reached. In addition, the method uses an ad hoc widening operator to ensure termination similar to other abstract interpretation approaches for inferring disjunctive invariants such as [Popeea and Chin, 2007, Sankaranarayanan et al., 2006].

Recent static analysis work by Kapur *et al.* [Kapur et al., 2013] uses *quantifier elimination*, rather than abstract interpretation, to produce sound loop invariants using the octagonal [Miné, 2006] and max-plus forms Allamigeon [2009]. The technique uses table look-ups to modify geometric objects representing invariant relations based on the program structure, e.g., to determine how an inequality is changed after an assignment $a = a + 10$. The tabular approach has a lower theoretical time complexity than traditional abstract interpretation for certain forms of invariants, e.g., octagonal inequalities. Currently, this work focuses on specific program constructs

for efficiency. For example, the analysis supports assignments and guards that are restricted to linear expressions.

A high-level difference between DIG and these static techniques is that DIG focuses on inferring invariants dynamically from program traces. However, our work on polynomial invariants is inspired by abstract interpretation, and thus, DIG supports and extends many forms of polynomial relations that are considered by static methods, e.g., octagonal inequalities, nonlinear inequalities, and disjunctive relations. We also introduce additional forms of polynomial invariants, e.g., the weak max-plus relations, that allow static techniques to be practically applied to more general classes of programs. To achieve sound results, we augment dynamic analysis with theorem proving, which verifies candidate invariants statically against the program code.

Dynamic Invariant Inference

Daikon, the canonical example of dynamic invariant analysis developed by Ernst *et al.* [Ernst, 2000, Ernst et al., 2000a, 2001, 2007, Perkins and Ernst, 2004], infers candidate invariants from traces and templates. By default, Daikon reports invariants at the entry and exit points of functions, although it is possible to extract invariants from other locations by manual instrumentation. Daikon provides a large list of assorted invariant templates that are considered to be useful to programmers, but it also supports user-supplied invariants. For polynomial relations, the tool can find linear relations over at most three variables, e.g., $x + 2y - 3z + 4 = 0$, and has a small number of fixed nonlinear templates such as $x = y^2$. Daikon can find simple disjunctive information using "splitting" conditions [Ernst et al., 2000b]. Given a predicate c, Daikon first obtains the invariants a and b when c and $\neg c$ are true, respectively, and it then combines these results to yields the disjunctive invariant `if` c `then` a `else` b. Relations among arrays have limited support in Daikon, e.g., the relations $A[i] = B[C[i]]$, $A[i] = 2B[i] + C[5] + 7$ are not considered.

There is related work on dynamic methods for finding invariants for debugging, e.g., the detected properties are used to find certain type of errors. The Diduce tool analyzes what happens when an error occurs by examining the differences between previous and current values of variables [Hangal and Lam, 2002]. Statistical debugging [Liblit et al., 2005], a fault localization technique, looks for relations, e.g., $\{<, =, >\}$, between two variables or a variable and a constant. The Spin model checker can also find relations over two variables [Vaziri and Holzmann, 1998]. In general, these approaches find invariants that are relatively simple compared to those provided by Daikon.

DIG considers more general forms of polynomial and array invariants than those supported by Daikon and the other dynamic approaches. Instead of using traces to filter out pre-defined templates, DIG applies geometric techniques to compute polynomial relations directly from program traces, and thus, it can capture more precise invariants. Moreover, the nonconvex geometric algorithm presented in Chapter 3 does not depend on manually-provided splitting conditions and generates powerful disjunctive invariants directly from traces. DIG also supports complex array properties, such as the nested form of array relation introduced in Chapter 4, that are not considered by the other static or dynamic invariant approaches.

Hybrid of Static and Dynamic Analyses

Nimmer and Ernst [Nimmer and Ernst, 2001] integrate Daikon with the ESC/Java static checker framework [Flanagan et al., 2002], allowing them to validate candidate invariants using Floyd-Hoare logic. This work is very similar in motivation and architecture to DIG. Key differences include: DIG detects richer forms of invariants, e.g., disjunctive invariants; DIG verifies invariants with respect to full program correctness ("rather than proving complete program correctness, ESC detects only certain types of errors" [Nimmer and Ernst, 2001, Sec. 2]), and DIG's empirical evaluation

(Section 3.6) shows that it proves over four times as many non-redundant invariants valid and considers over four times as many benchmark kernels as Daikon and ESC/Java).

Recently, Sharma *et al.* [Sharma et al., 2013a] combine dynamic and static analyses to detect sound equality invariants in program loops. They first use the algorithm developed in Section 3.2.1 and published in [Nguyen et al., 2012] to compute polynomial equalities from traces, and they then apply SMT solving to verify the candidate invariants. Counterexamples to candidate invariants are iteratively used to produce new traces to generate better candidate invariants. They prove that the candidate equalities are sound and that the approach terminates after a finite number of iterations. This technique does not support inequalities or disjunctive polynomial relations. An interesting area of future work would be extending this technique to include inequality and disjunctive invariants generated by DIG, which also can be checked for satisfiability using SMT solvers.

Sharma *et al.* [Sharma et al., 2013b] propose an approach based on machine learning to finding disjunctive polynomial invariants. The method operates on traces representing good and bad program states: good traces are obtained by running the program on random inputs and bad traces correspond to runs on which an assertion or post-condition is violated. They use a *probably approximately correct* machine learning model [Valiant, 1985] to find candidate invariants expressed as a predicate, which separates the good and bad traces. For efficiency, they restrict attention to the octagon domain and search only for predicates that are arbitrary boolean combinations of octagonal inequalities. Finally, they use induction to check the candidate invariants with an SMT solver. While we share their focus on disjunctive invariants, a key difference between their work and ours is their results depend on existing annotated program assertions. By contrast, we do not make such assumptions about the input program, and DIG generates these assertions automatically.

Uses of k-induction

The above methods from Nimmer *et al.* and Sharma *et al.*, as well most work using Floyd-Hoare logic, employ the standard technique of mathematical induction to verify detected invariants against program code. Chapter 3 presents a custom theorem prover called KIP (k-Inductive Prover), which based on k-induction, a stronger form of induction. We use KIP to prove dynamically inferred invariants (Section 3.5). The application of k-induction is becoming popular for proving invariants represented by logical formulas that may not admit standard induction. For example, Sheeran *et al.* apply k-induction to verify hardware designs using SAT solvers [Sheeran et al., 2000]; the PKIND model checker of Kahsai and Tinelli [Kahsai and Tinelli, 2011, Kahsai et al., 2011] uses k-induction and SAT/SMT solvers to verify synchronous programs in the Lustre language; and recently, Donaldson *et al.* [Donaldson et al., 2011] apply k-induction to imperative programs with multiple loops.

A main distinction between the design of KIP and these approaches is that KIP offers four of the properties (SMT, lemma learning, redundancy elimination, and parallelism), which we found were critical for efficiently verifying large numbers of candidate invariants over programs with complex properties, such as nonlinear arithmetic. Moreover, we note that the programs and candidate invariants evaluated in Chapter 3 could serve as a benchmark suite for the evaluation of such theorem provers because they include hundreds of valid and invalid formulas involving nonlinear arithmetic, many of which are k-inductive.

2.3.2   Program Repair and Test Input Generation

A key contribution of this dissertation is the equivalence theorem presented in Chapter 5 between the reachability formulation of program verification and template-based synthesis. This result allows automatic program repair and synthesis approaches to

take advantage of powerful, off-the-shelf test input generation tools that find inputs to reach certain program locations. For example, the repair tool CETI described in Chapter 5 applies the symbolic execution tool KLEE [Cadar et al., 2008a] to generate test inputs that map directly to program repairs. In this section, we review the state of the art in automatic program repair and test input generation.

Automatic Program Repair

Due to the pressing demand for reliable software, automatic program repair has steadily gained research interests and produced many novel repair techniques. *Synthesis* repair methods generate constraints and solve them to produce patches that are correct by construction, i.e., guaranteed to adhere to a specification or pass a test suite. For example, AFix generates correct fixes specifically for single variable atomicity violations [Jin et al., 2011]. Gopinath *et al.* [Gopinath et al., 2011] encode a buggy program and its specification into a constraint, which is solvable using a SAT solver. SemFix uses symbolic execution to create repair constraints with templates and solve them to produce program repairs [Nguyen et al., 2013]. The FoREn-SiC project employs several template-based repair techniques including concolic execution [Könighofer and Bloem, 2013], equivalence-based checking [Könighofer and Bloem, 2013], and counterexample guided refinement [Könighofer and Bloem, 2011] [Bloem et al., 2013]. Jobstmann *et al.* [Jobstmann et al., 2005] model the task of repairing a program with LTL specifications as a game and uses a model checker to find a winning strategy corresponding to a success repair.

In contrast, *generate-and-validate* repair approach generates multiple repair candidates using stochastic search, and verifies them against given specifications. For example, GenProg [Forrest et al., 2009, Le Goues et al., 2012, Le Goues et al., 2012, Nguyen et al., 2009, Weimer et al., 2009, 2010] employs genetic programming to modify suspicious code regions. Debroy and Wong [Debroy and Wong, 2010] produce new

program variants by applying mutation operators to suspicious program expressions. PAR [Kim et al., 2013] leverages human expertise to repair programs by applying common repair patterns learned from human-created patches. Several techniques capture program behaviors (invariants) at run time to guide the repair process. For example, Pachika derives repairs by analyzing differences between expected and unexpected behaviors mined from passing and failing program runs [Dallmeier et al., 2009]; Clearview detects invariants from program runs to identify errors and creates binary repairs satisfying desired program invariants [Perkins et al., 2009], and AutoFix-E extracts invariants from program runs and exploits Eiffel program contracts to create candidate repairs [Wei et al., 2010].

Test Input Generation

The subfield of test input generation has produced many practical techniques to generate high coverage test data for complex software. Fuzz testing techniques [Forrester and Miller, 2000, Miller et al., 1990] create test values by randomly mutating well-formed inputs of a program. Concolic execution approaches combine dynamic and static analyses to generate program constraints that can be solved with a SAT or SMT constraint solver. The DART [Godefroid et al., 2005], CUTE/jCute [Sen and Agha, 2006], and CREST [Burnim and Sen, 2008] techniques combine random testing and symbolic execution to generate test inputs for C and Java programs. CUTE and jCute in particular can find test inputs and thread schedules for multithreaded programs. EXE [Cadar et al., 2008b] and KLEE [Cadar et al., 2008a] perform concrete and dynamic execution, model memory, and employ a variety of constraint solving optimizations to achieve high code coverage. These two tools are designed for testing complex systems software, such as network servers, file systems, device drivers and library code. Microsoft developed SAGE [Godefroid et al., 2008] to discover bugs in x86 binaries and PEX [Tillmann and de Halleux, 2008] as an add-in tool for the Visual Studio .NET framework. Other organizations, such as NASA [Anand et al.,

2007], IBM [Artzi et al., 2008], and Fujitsu [Li et al., 2011] have also developed test input generation tools. Software model checkers such as BLAST [Beyer et al., 2007] and SLAM [Ball and Rajamani, 2002] are also applicable to generating test inputs by representing them as counterexamples that violate program correctness properties.

Chapter 3

Polynomial Invariants

"Where there is matter, there is geometry." – Johannes Kepler[1]

The following two chapters present the research results of this dissertation. This chapter and the next are devoted to the topic of invariant generation. This chapter covers the generation and verification of polynomial invariants using geometric concepts, while Chapter 4 focuses on the generation of array invariants. Parts of this chapter have been published previously in [Nguyen et al., 2012, 2014a,c].

3.1   Introduction

A *polynomial* is an algebraic expression of the form $c_0t_1 + \cdots + c_nt_n$, where each coefficient c_i is either real-valued and each monomial t_i is a single variable with a non-negative exponent or a product of variables, each with non-negative exponents. Polynomial relations, such as equalities and inequalities among polynomials, are important throughout mathematics and science. Two famous polynomial equations

---

[1]German mathematician, astronomer, and astrology, who is best known for the laws of planetary motion (1571 – 1630).

include the Pythagorean theorem for right triangles $x^2 = y^2 + z^2$ and Einstein's famous mass-energy relationship $e = mc^2$.

In computer science, polynomial relations among numerical program variables appear in many computer algorithms and applications. For example, the gcd of x, y is $nx+my$, and the location of the chosen pivot in a binary search is $l+u \geq 2p \geq l+u-1$. Polynomial relations can also be used to model pointer/array arithmetic and other memory related properties in programming languages like C [Cousot et al., 2005]. Thus, many algorithms for checking buffer overflow errors and memory leaks in C programs require reasoning over polynomial relations [Allamigeon, 2009, Miné, 2004]. For these reasons, polynomial relations are often considered by both static and dynamic invariant techniques. In fact, the main research focus of the abstract interpretation framework introduced in Chapter 2 is on finding different forms of polynomial invariants.

The form of a polynomial invariant, e.g., the type of relations among the polynomials, determines the information the invariant captures and suggests its computational complexity. Important forms of polynomial invariants include the type of polynomials, the relation among polynomials, and the boolean connection among formulas representing polynomial relations.

- *Type of Polynomial.* The *degree* of a monomial in a polynomial is the sum of the exponents of the variables appearing in that monomial. The degree of a polynomial is the highest degree of its monomials, e.g., the polynomial $2x^3y^4 + y^5 + 9$ has degree 7. A polynomial is *linear* if its degree is 1 and is *nonlinear* if the degree is ≥ 2. Linear relations are used in several classical data flow analysis techniques, including constant propagation, copy propagation, and common subexpression elimination [Gulwani, 2005]. Nonlinear relations are more complex, but appear in many scientific, engineering, and safety-critical applications. For example, the commercial static analyzer Astrée mentioned

in Chapter 2 implements the ellipsoid abstract domain to represent and analyze a class of quadratic inequality invariants.[2] Nonlinear invariants have also been found useful in the analysis of hybrid systems [Roozbehani et al., 2005, Sankaranarayanan et al., 2005].

- *Relational Operator. Equality* $(=)$ and *inequality* $(\geq)$ are two widely used forms of polynomial relations. Most constraint solving problems focus on finding values for variables that satisfy a given set of polynomial equalities and/or inequalities. Linear equations can be solved efficiently in polynomial time using a standard Gaussian elimination technique [Farebrother, 1988]. The problem of linear optimization (programming), which finds the best values satisfying a set of linear inequalities, is often handled using exponential-time techniques such as Simplex [Cormen et al., 2001, Nelder and Mead, 1965] and Fourier-Motzkin [Dantzig, 1998, Dantzig and Curtis Eaves, 1973].

- *Boolean Connection.* Polynomial invariants are commonly represented using logical formulas using *conjunctive* or *disjunctive* forms. A formula in conjunctive form is a conjunction or a set of polynomial relations, e.g., $a = b \land x = y$. Most abstract interpretation techniques rely on convex geometry to generate conjunctive polynomial relations. In contrast, a disjunctive formula is a disjunction of polynomial relations, e.g., $a = b \lor x = y$. Disjunctive invariants, which represent the semantics of program branching, are crucial to many programs such as sorting and searching algorithms. However, these invariants are more difficult to analyze because general disjunctive forms of polynomial relations are not expressible using classical convex shapes.

Existing invariant approaches usually do not achieve soundness, efficiency, and expressive power simultaneously. Sound and efficient static approaches target rel-

---

[2]The ellipsoid domain [Feret, 2004] for this case is expressed by the quadratic form $x^2 + axy + y^2 \geq k$ where $1 > b > 0$ and $4b > a^2$.

atively simple invariants, while efficient dynamic approaches find invariants under restricted templates but are not guaranteed to be sound. For example, the afore-mentioned Astrée analyzer and Interproc [Jeannet, 2014], a popular static analyzer that employs different abstract domains, consider only conjunctive polynomial in-variants, and, thus, lack expressive power. The dynamic tool Daikon reviewed in Chapter 2 detects only conjunctive linear relations over, at most, three variables and has limited support for nonlinear polynomials or disjunctive invariants.

In this chapter, we present and evaluate DIG (Dynamic Invariant Generator), a hybrid invariant analysis tool that efficiently and correctly generates expressive polynomial invariants. DIG combines dynamic geometric inference for complex poly-nomial invariants with static analysis for validating invariants by formal proof. DIG supports both conjunctive and disjunctive forms of nonlinear polynomials by repre-senting these relations as geometric objects in high-dimensional space. DIG inter-prets conjunctive polynomial relations as convex geometric shapes such as hyper-planes and polyhedra. DIG represents certain forms of disjunctive relations, which are not expressible using classical convex polyhedra, as convex polyhedra in the non-standard *max*- and *min*-plus algebras. Finally, by verifying candidate invariants with a custom theorem prover against the program code, DIG removes spurious invariants and returns only true invariants. In short, DIG achieves soundness, efficiency, and expressive power simultaneously by leveraging the observation that it is easier to infer complex candidate invariants dynamically and verify them statically.

In the remainder of this section, we provide a motivating example, present an overview of DIG, and list our contributions to the generation of polynomial invariants.

3.1.1   Motivating Example

We use an example program to highlight the important insights underlying DIG and to motivate key design decisions. The `cohen` program in Figure 3.1 implements the

```
def cohen(x, y):
  q = 0; r = x
  while r >= y:
    a = 1; b = y
    while r >= 2*b:
      [L] a = 2*a; b = 2*b
    r = r-b
    q = q+a
  return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
			$\vdots$		
4	1	1	1	0	4
4	1	2	2	0	4
			$\vdots$		

Figure 3.1: Cohen integer division algorithm and its traces at location L on inputs $(x = 15, y = 2)$ and $(x = 4, y = 1)$. From such traces, DIG generates three nonlinear invariants $b = ya, x = qy + r$, and $r \geq 2ya$.

integer division algorithm by Cohen [Cohen, 1990], which takes as input two integers x, y and returns the integer q as the quotient of x and y. We consider invariants at location L, the head of the inner while loop. The table in Figure 3.1 consists of several sets of trace values from the six variables $\{a, b, q, r, x, y\}$ in scope at L for inputs $(x = 15, y = 2)$ and $(x = 4, y = 1)$.

From such traces, DIG identifies three nonlinear relations $b = ya, x = qy + r, r \geq 2ya$. These relations are program invariants that describe precisely the semantics of the inner while loop in Cohen's algorithm. In particular, the nonlinear equality $x = qy + r$ asserts that the dividend x equals the divisor y times the quotient q plus the remainder r.

To obtain these nonlinear invariants, DIG first generates *terms* t_i to represent monomials up to a certain degree over the variables $\{a, b, q, r, x, y\}$. For equality relations, an equation template of the form $c_1 t_1 + \cdots + c_n t_n = 0$ is created from the terms t_i. DIG then instantiates the template with the traces from Figure 3.1 to obtain a set of equations, which the tool then solves for the unknowns c_i using a standard equation solver. This allows DIG to identify the two equations $b = ya, x = qy + r$ at location L from the execution traces of `cohen`. For inequality relations, DIG creates
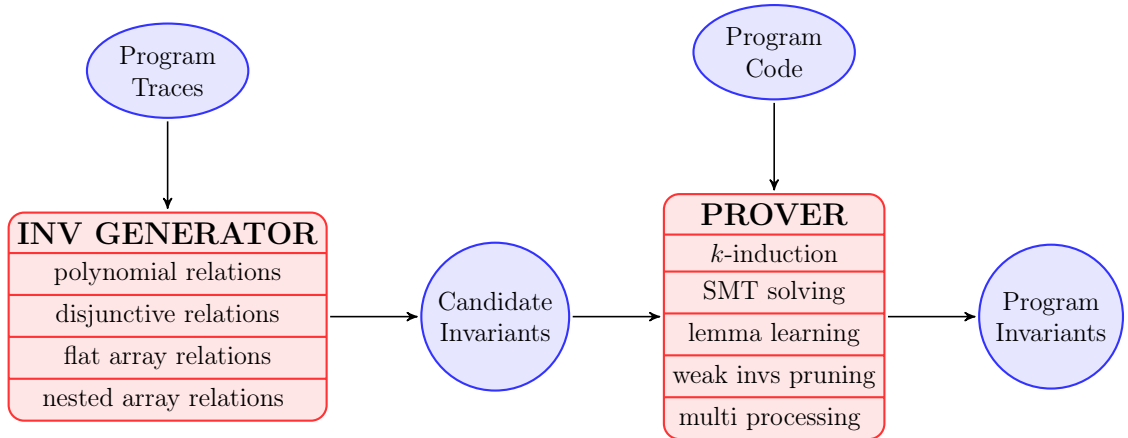
Figure 3.2: An overview of DIG. The generator finds different types of candidate invariants from input traces. The prover distinguishes between true and spurious invariants using the program code.

points from terms using the execution traces, builds a convex hull enclosing the points, and finally extracts the facets of the hull. These facets represent inequalities of the form $c_1 t_1 + \cdots + c_n t_n \geq 0$ and, thus, allow DIG to obtain inequalities such as $r \geq 2ya$. These nonlinear invariants cannot be discovered by current dynamic analysis tools such as Daikon and are also challenging for methods based on static analysis.

3.1.2 Overview of DIG

Figure 3.2 gives an overview of DIG (Dynamic Invariant Generator), an invariant analysis tool that generates invariants from input traces consisting of values from numerical or array variables (covered in Chapter 4). First, terms are created to represent variables whose values are captured in the traces. Depending on the type of the variables, DIG next generates polynomial relations and/or array relations over the terms. Finally, DIG uses a theorem prover to remove redundant and incorrect candidate invariants.

Nonlinear Terms

DIG uses terms to represent nonlinear properties over program variables and other information of interest. From a set V of variables and a degree d, a set T of terms is created to represent monomials up to degree d from V. For instance, the set T of ten terms $\{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ contains all monomials up to degree 2 over the variables $\{r, y, a\}$. Nonlinear relations over program variables can now be specified as linear relations over terms, which allows us to generate nonlinear invariants from existing techniques for linear constraint solving.

In addition to monomials, the user can manually define terms to capture other desirable properties, e.g., $t_1 = \frac{r}{y}, t_2 = y^a, t_3 = \text{mod}(r, 256)$. Users can also query DIG for relations among a specific set of terms, e.g., only inequalities among $\{r, y, a^2\}$. These customizations allow DIG to identify specific relations among potentially interesting terms and reduce the overall complexity of the process.

Post-processing

DIG uses three techniques, pruning, filtering, and static checking, to help remove redundant and spurious invariants:

- *Pruning.* To reduce the number of candidate invariants, DIG removes any invariants that are logical implications of other invariants. For instance, if both relations $x = y$ and $x^2 = y^2$ are found, then DIG suppresses the latter because it is implied by the former. These redundant invariants arise because DIG treats each term as an independent variable for the purpose of finding nonlinear polynomials. For example, if $t_1 = x, t_2 = y, t_3 = x^2, t_4 = y^2$ then $x = y$ implies $x^2 = y^2$; however, their corresponding term relations, $t_1 = t_2$ and $t_3 = t_4$, have no direct relation. To check an implication, DIG uses an SMT solver to show the negation of that implication is unsatisfiable.

- *Filtering.* DIG uses a subset of the random input traces for invariant generation and the remaining traces to check the resulting invariants. A program invariant holds for any set of traces, thus, it is likely that DIG can find that same invariant using a smaller subset of the available traces. The candidate invariant, which is obtained using a subset of traces and might not be true for all observed traces, is then verified against the remaining traces and removed if it fails for any of them. This strategy improves the run time of DIG because it is more expensive to generate a complex relation than to verify that relation holds over input traces.

- *Static Checking.* DIG uses a custom theorem prover based on k-induction and constraint solving to validate candidate invariants when the program code is available. The design and implementation of the prover are discussed in Section 3.5.

We note that existing strategies from other approaches could also be integrated with techniques used in DIG. For example, if DIG's algorithms were incorporated into Daikon, then most of Daikon's optimization techniques [Perkins and Ernst, 2004] could be applied directly to DIG's generated invariants. As an example, the static analysis work reported in [Sharma et al., 2013a] has integrated ideas from DIG to generate sound equality invariants.

3.1.3   Contributions

We make the following contributions to the generation of polynomial invariants in this chapter:

- *Geometric Invariant Inferring (Section 3.2).* We develop DIG, a dynamic tool that leverages geometric concepts for polynomial invariant analysis. We formulate the problem of generating conjunctions of equalities and inequalities to

the tasks of solving linear questions and constructing polyhedra, respectively. This formulation treats relations over numerical program variables as geometric shapes in multidimensional space, i.e., trace data as points, equations as hyperplanes, and inequalities as convex polyhedra. We also consider simpler geometric shapes, such as octagons, which are more tractable because they encode less expressive polynomial relations.

- *Disjunctive Invariants (Section 3.3).* We present a technique to infer certain disjunctive polynomial invariants by constructing convex max and min-plus polyhedra over trace points. We also introduce a novel restricted class of max and min-plus invariants, called "weak" invariants, that strike a balance between expressive power and computational complexity. Weak invariants express useful max and min-plus relations and can be computed efficiently.

- *Algorithmic Analysis (Section 3.4).* We formally analyze the complexity of all presented algorithms. Using geometric reasoning, we prove an underapproximation property of polynomial invariants that is guaranteed in DIG, but not in other dynamic invariant analyses.

- *Automatic Theorem Proving (Section 3.5).* We develop KIP, a theorem prover based on iterative k-induction proving and constraint solving, to verify polynomial invariants against program code. When parallelized, KIP efficiently and correctly processes many complex and potentially spurious invariants.

- *Experimental Evaluation (Section 3.6).* We evaluate DIG and KIP on difficult kernels involving nonlinear arithmetic and abstract arrays. Experimental results show that these tools are efficient, both at learning complex polynomial invariants and at proving them correct.

3.2 Inferring Conjunctive Invariants Dynamically

At a high level, DIG treats numerical trace data as points in Euclidean space and computes geometric shapes enclosing these points. For example, the trace values of the two variables v_1, v_2 are points in the (v_1, v_2)-plane. DIG then determines if these points lie on a line, represented by a linear equation of the form $c_0 + c_1 v_1 + c_2 v_2 = 0$. If such a line does not exist, DIG builds a bounded convex polygon from these points. The edges of the polygon are represented by linear inequalities of the form $c_0 + c_1 v_1 + c_2 v_2 \geq 0$. The technique generalizes to equations and inequalities among three or more variables by constructing hyperplanes and polyhedra in a high-dimensional space.

DIG takes as input the set V of numerical variables that are in scope at location L, the associated traces X, and a maximum degree d, and returns the set (a conjunction) of possible polynomial relations among the variables in V whose degree is at most d. The post-processing techniques described in Section 3.1.2 are then applied to the obtained relations to suppress redundant relations and to remove spurious invariants.

3.2.1 Equality Invariants

DIG treats polynomial equalities as unbounded geometric shapes, e.g., lines and planes, to obtain a conjunction of equality invariants of the form

$$c_1 t_1 + \cdots + c_n t_n = 0, \tag{3.1}$$

where c_i are real-valued and t_i are terms.

Figure 3.3 outlines DIG's algorithm for finding equality invariants from the inputs: set V of variables, set X of traces, and max degree d. First, we create T terms representing monomials over the input variables V up to degree d (see Section 3.1.2). The equation template F given in Equation (3.1) is formed using these terms. Next,

procedure FINDEQS(V, X, d)
    $T \leftarrow$ GENTERMS(V, d)
    $F \leftarrow$ GENTEMPLATE(T)
    $E \leftarrow$ GENEQTS(F, X)
    $S \leftarrow$ SOLVE(E)
    return S ▷ return equalities of the form given in Equation (3.1)

Figure 3.3: Algorithm for finding polynomial equations from the inputs: set V of variables, set X of traces, and max degree d. The algorithm consists of four steps: creating terms over the input variables(GENTERMS), using terms to form an equation template (GENTEM-PLATE), instantiating the template with input traces to obtain a set of linear equations (GENEQTS), and solving equations for the unknown coefficients, which map to concrete equality invariants (SOLVE).

we obtain a set of linear equations $E = \{e_1, \ldots, e_{|X|}\}$ by instantiating F with the traces in X. Finally, we solve E for the unknown coefficients c_i. The nontrivial solutions for c_i, if any, suggest relations among the terms in T.

The nontrivial solutions of E for the unknown coefficients c_i have the form $c_i = v_i$. The values v_i are free variables that range over the reals. The terms in the template F that have zero-valued coefficients are not related because the only way to satisfy equations in E is by setting the coefficients of these terms to zero. In contrast, terms that have coefficients sharing some free variable v are related. To find relations among the terms sharing the variable v, we fix v to a concrete value, e.g., $v = 1$ and other v' to 0, and instantiate F with $v = 1$ and $v' = 0$. This step is repeated for each shared variable v to obtain relations among terms sharing v.

Example 3.2.1. We demonstrate how DIG discovers the nonlinear equalities $b = ya$ and $x = qy + r$ for the `cohen` program from Figure 3.1. For illustration purposes, we focus on the case where $d = 2$, in which DIG generates quadratic equations.

For the six variables $\{a, b, q, r, x, y\}$, together with degree $d = 2$, DIG creates the set $T = \{1, a, \ldots, y^2\}$ of monomials of degree ≤ 2 containing 28 terms. T is then used to form the template $F : c_1 + c_2 a + \cdots + c_{28} y^2 = 0$ with 28 unknown coefficients

c_i to be solved for. Next, F is instantiated with the traces in X to form the set E of equations. For example, instantiating F with the values $(a = 1, \ldots, y = 2)$ from the first trace in Figure 3.1 gives the equation $c_1 + c_2 + \cdots + 4c_{28} = 0$. Solving E for the unknowns c_i results in the solution:

$$c_3 = v_1, \quad c_5 = v_3, \quad c_6 = -v_3, \quad c_{11} = -v_2,$$
$$c_{12} = v_2, \quad c_{13} = -v_1, \quad c_{15} = -v_2 \quad c_{22} = v_3,$$

and all other $c_i = 0$.

To find the relation between the terms t_3 and t_{13}, whose coefficients c_3 and c_{13} share the value v_1, DIG sets $v_1 = 1$ and $v_2 = v_3 = 0$ (since the terms t_3, t_{13} are not related by the values v_2, v_3). The template F, when being instantiated with $(v_1 = 1, v_2 = 0, v_3 = 0)$, gives the relation $t_3 - t_{13} = 0$, which is $b = ay$ because $t_3 = b$ and $t_{13} = ay$. After repeating this process for all shared variables, DIG achieves the equations:

$$t_3 = b, \quad t_{13} = ya, \qquad \rightarrow \quad b = ay,$$
$$t_5 = r, \quad t_6 = x, \quad t_{22} = qy \quad \rightarrow \quad x = qy + r,$$
$$t_{11} = ra, \quad t_{12} = xa, \quad t_{15} = bq \quad \rightarrow \quad xa = ra + bq.$$

These generated equations are true invariants; however, the relation $xa = ra + bq$ is redundant because it can be obtained from the other two equations $b = ay$ and $x = r + qy$ by substitution. The post-processing step described in Section 3.1.2 suppresses these redundant invariants using theorem proving. The resulting set of equations for the `cohen` program after post-processing is $\{b = ya, x = r + qy\}$.

3.2.2   Inequality Invariants

DIG interprets inequalities among terms as geometric shapes over points created from program traces. Figure 3.4 illustrates several shapes supported by DIG in two-dimensional space. Figure 3.4a shows a set of points created from input traces.
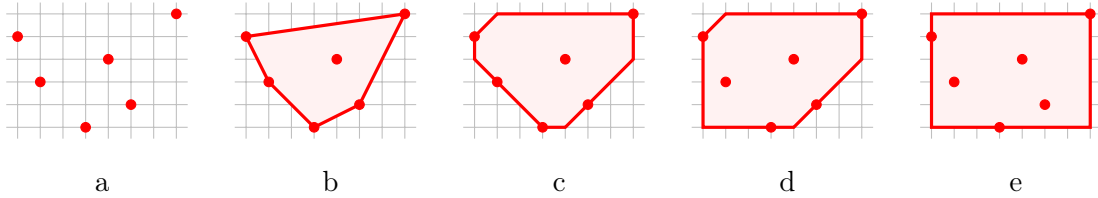
Figure 3.4: (a) A set of points in 2D and its approximation using the (b) polyhedral, (c) octagonal, (d) zonal, and (e) interval regions.

Figures 3.4b, 3.4c, 3.4d, and 3.4e approximate the area enclosing these points using the polygonal, octagonal, zonal, and interval shapes that are represented by the conjunctions of inequalities of the forms $c_1 v_1 + c_2 v_2 \geq c$, $\pm v_1 \pm v_2 \geq c$, $v_1 - v_2 \geq c$, and $\pm v \geq c$, respectively. In principle, these forms of relations are sorted in decreasing order of expressive power and computational cost. For instance, interval inequalities are less expressive than zonal inequalities, and the cost of computing an interval, i.e., the upper and lower bound of a variable, is lower than the computation of the convex hull of an zone. The number of generated invariants representing the facets also varies for different shapes; a polygon can have an unbounded number of facets (edges) whereas an octagon, a zone, and an interval region over two variables have at most eight, six, and four edges, respectively.

General (Polyhedral) Inequalities

DIG finds inequality invariants of the form

$$c_1 t_1 + \cdots + c_n t_n \geq 0, \tag{3.2}$$

where c_i are real-valued and t_i are terms. These general inequalities can also express octagonal inequalities (two terms with $\{-1, 0, 1\}$ integral coefficients) and interval inequalities (single terms with unit coefficients).

Figure 3.5 outlines two techniques for finding general inequalities. Both techniques yield sound relations with respect to input traces; however the deduction

procedure FINDIEQS(V, X, d, *ieqs* (optional))
    if *ieqs* $= \emptyset$ **then**
        $T \leftarrow$ GENTERMS(V, d)
        $P \leftarrow$ GENPOINTS(T, X)
        $H \leftarrow$ CREATEPOLYHEDRON(P)   ▷ construct a polyhedron over points
        $S \leftarrow$ EXTRACTFACETS(H)
    else
        eqts $\leftarrow$ FINDEQS(V, X, d)   ▷ find equalities using the algorithm in Figure 3.3
        $S \leftarrow$ DEDUCEIEQS(*eqts*, *ieqs*)
    return S ▷ return inequalities of the form given in Equation (3.2)

Figure 3.5: Algorithm for finding polynomial inequalities. The convex hull technique consists of four steps: creating terms to represent products of program variables (GENTERMS), instantiating points from terms using input traces (GENPOINTS), building a convex polyhedron enclosing the points (CREATEPOLYHEDRON), and extracting its facets to represent inequalities among terms (EXTRACTFACETS). When additional information is available, the deduction technique combines the discovered equations (FINDEQS) with the given information to deduce new inequalities (DEDUCEIEQS).

method, with the help of additional information, runs much faster. We now illustrate how both techniques produce the nonlinear inequality $r \geq 2ay$ in the `cohen` program given in Figure 3.1.

Using Polyhedra

After obtaining the set T of terms, we use the traces in X to create points in $|T|$-dimensional space and compute the convex hull of these points to represent a polyhedron H. The bounded convex polyhedron H can be described by a set of linear inequalities of the form given in Equation (3.2). This is called the *half-space* representation of a polyhedron. The facets of H, corresponding to the solutions of the set of linear equalities, represent the inequalities among the terms in T. Figure 3.4b depicts a 2D polyhedron (polygon) that has five facets.

Building a convex polyhedron in high-dimensional space is expensive as discussed in Section 3.4.1. Moreover, program invariants often involve only a small subset of

all possible program variables, e.g., the relation $b - ay = 0$ involves only $\{a, b, y\}$, even though all six variables in scope were considered. Based on this observation, we consider several heuristics, such as iteratively searching for invariants involving all possible combinations of a small, fixed number of variables.

Example 3.2.2. For the `cohen` program, DIG first generates possible inequalities that contain at most three out of the six variables $\{a, b, q, r, x, y\}$. There are $\binom{6}{3} = 20$ combinations that contain three variables, one of which is $\{r, y, a\}$. To find nonlinear inequalities, terms of degree d are built on the variables under consideration. With $d = 2$, DIG generates the set $T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$ of terms.

The terms in T are instantiated with the traces in X to form a set P of points. For instance, the first trace in Figure 3.1 gives the point $[1, 15, 2, 1, 30, 15, 2, 225, 4, 1]$ in 10-dimensional space, corresponding to the terms in T. The convex polyhedron H is then constructed to enclose the points in P. One of the facets of H corresponds to the inequality $r \geq 2ya$. The inequalities represented by other facets are also valid with respect to the input traces, although they might be spurious invariants. Section 3.4.2 provides additional discussion on spurious invariants. The static theorem proving technique in Section 3.5 distinguishes between true and false invariants.

Deduction From Loop Conditions

The convex hull technique for general inequalities can be computationally expensive depending on the numbers of terms and trace points. Consequently, we develop an alternative technique using *deduction* to find inequalities of the form given in Equation (3.2) if additional information is available. More specifically, if some inequalities are asserted at location L, then we can use them, together with the discovered equalities from Section 3.2.1, to deduce new nontrivial inequalities. For instance, if the location L is the head of a loop, then L can be reached if and only if the loop conditions are met. Such loop conditions are an example of additional information, which

can be given as input from the user (or automatically mined from the source code as in the `cohen` program) to facilitate the process of generating additional invariants. Deduction is related to the strategy of adding known facts or proved results as lemmas in interactive theorem provers such as PVS [Owre et al., 1992].

Example 3.2.3. For the running `cohen` example, DIG generates the set of equations $\{b = ay, x = qy + r\}$ representing possible invariants at location L as described in Section 3.2.1. The head of the inner loop at location L is reached only when the condition of that loop $r \geq 2b$ is met, thus, $r \geq 2b$ is also an invariant at L. New and nontrivial inequalities can be deduced from this additional information using deduction, term rewriting, and substitution. In the current implementation, DIG pairs inequalities from the loop conditions with the obtained equations to deduce new inequalities. For the running example, $r \geq 2ay$ is deduced from the pair $(r \geq 2b, b = ay)$, and $x \geq qy + 2b$ is deduced from $(r \geq 2b, x = qy + r)$. Hence, deduction finds the inequalities $r \geq 2ya$ and $x \geq qy + 2b$ among the variables $\{a, b, q, r, x, y\}$, both of which are program invariants at location L in the `cohen` program.

Deduction could theoretically produce many results by combining discovered equalities and loop conditions. However, the technique is efficient in our experiments because the numbers of loop conditions and generated equality invariants are small (one or two guards at most loops and fewer than four equalities at a particular program location). Our experience shows that deduction allows for effective inequality invariant discovery that otherwise would require the more expensive convex hull method or would not be possible in the case of incomplete traces.

3.2.3   Weak Inequality Invariants

To avoid building complex polyhedra in high dimensions, DIG supports simpler (weaker) inequalities representing simpler geometric shapes such as octagons.[3] By balancing expressive power with computational cost, octagonal relations are especially useful in practice for detecting bugs in flight-control software, performing array bound and memory leak checks [Cousot et al., 2005, Miné, 2004].

DIG builds an octagon, a polygon with eight edges in 2D, depicted in Figure 3.4c, over trace points to obtain a conjunction of eight inequalities of the form

$$c_1 t_1 + c_2 t_2 \geq k, \tag{3.3}$$

where t_1, t_2 are terms, $c_1, c_2 \in \{-1, 0, 1\}$ are coefficients, and k is real-valued.

Given the points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, we compute the half-space representation of an octagon enclosing these points, i.e., the set of eight linear relations $\{u_1 \geq x \geq l_1, u_2 \geq y \geq l_2, u_3 \geq x - y \geq l_3, u_4 \geq x + y \geq l_4\}$, as follows:

$$
\begin{aligned}
u_1 &= \max(x_i), & l_1 &= \min(x_i), \\
u_2 &= \max(y_i), & l_2 &= \min(y_i), \\
u_3 &= \max(x_i - y_i), & l_3 &= \min(x_i - y_i), \\
u_4 &= \max(x_i + y_i), & l_4 &= \min(x_i + y_i).
\end{aligned}
$$

The algorithm to find octagonal invariants from inputs X, V, d is similar to one listed in Figure 3.5, where the CREATEPOLYHEDRON function computes octagonal invariants for each pair of terms in T. The post-processing techniques from Section 3.1.2 also apply to the obtained invariants.

Like general inequalities, octagonal inequalities can also represent interval inequalities, e.g., $u_1 \geq x \geq l_1, u_2 \geq y \geq l_2$ as illustrated in Figure 3.4a. However,

---

[3]This approach is inspired by the abstract interpretation framework in static analysis introduced in Section 2.1.1, which finds simpler forms of inequalities, such as those in Figure 3.4, to avoid the cost of computing general polyhedra [Miné, 2004].

octagonal relations are less expressive than general relations due to the restriction to two terms with specific integral coefficients. For instance, octagonal relations cannot represent the inequality $t_1 \leq 2t_2$, where $t_1 = r, t_2 = ya$, in the `cohen` program due to the coefficient 2. However, the inequality $r \geq -2ay$ can be obtained through octagonal relations by using a term to representing $2ya$.

Example 3.2.4. Consider the below C code fragment `flatten` that puts the contents of a 2-dimensional array $A[M][N]$ into a 1-dimensional array $B[MN]$.

```
for (i = 0; i < M; ++i){
  for (j = 0; j < N; ++j){
    k   = i*n+j;
    [L]
    B[k] = A[i][j];
  }
}
```

The nonlinear relation $0 \leq k \leq MN - 1$ at location L is essential for the safety of `flatten` and is identified by DIG using octagonal constraints with terms representing quadratic polynomials over variables. The array relation $A[i][j] = B[iN + j]$, which asserts the correctness of `flatten`, is also generated by DIG using the technique described in Chapter 4.

3.3   Inferring Disjunctive Invariants Dynamically

Convex geometric shapes can represent conjunctions, but not disjunctions, of polynomial relations. Disjunctive invariants are more difficult to analyze, but are also crucial to many programs. For example, after `if (p) a = 1; else a = 2;` neither $a = 1$ nor $a = 2$ is an invariant, but $(p \wedge a = 1) \vee (\neg p \wedge a = 2)$ is an invariant. Thus, disjunctive invariants capture path-sensitive reasoning, such as those in any non-trivial program.

```
def ex1(x):
    y = 5
    if x > y:
        x = y
    while[L] x <= 10:
        if x >= 5:
            y = y+1
        x = x+1

    assert y == 11
```

x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11
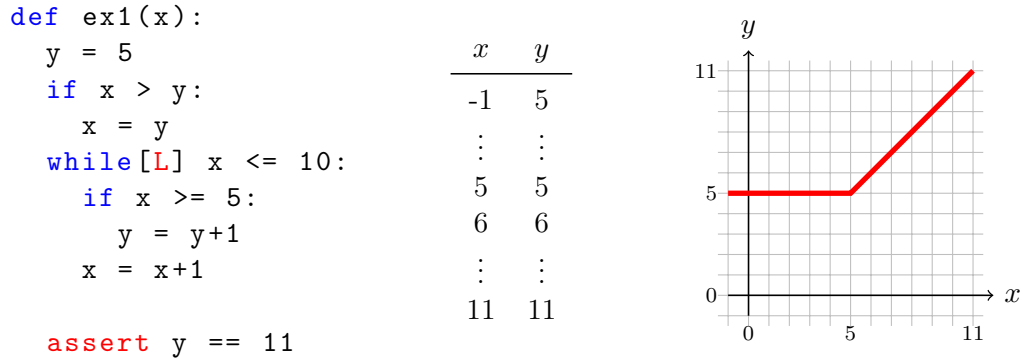
Figure 3.6: Program `ex1`, the observed traces on input $x = -1$, and the geometric representation of its invariant $(x < 5 \land y = 5) \lor (5 \leq x \leq 11 \land x = y)$ at location L.

To capture disjunctive information, we build convex hulls for a special type of nonconvex polyhedra in the *max-plus* algebra. Max-plus algebra [Allamigeon et al., 2008, Kapur et al., 2013] is analogous to standard algebra, but operates over the reals and $-\infty$ with max and $+$ as the additive and multiplicative operators, respectively. A max-plus polyhedron is a set of relations of the form $\max(c_0, c_1 + v_1, \ldots, c_n + v_n) \geq \max(d_0, d_1 + v_1, \ldots, d_n + v_n)$ over program variables v_i with coefficients $c_i, d_i \in \mathbb{R} \cup \{-\infty\}$. For instance, the max-plus polyhedron $\max(x, y) \geq \max(-\infty, z)$ encodes the disjunctive information $(x < y \land y \geq z) \lor (x \geq y \land x \geq z)$ or simply $y \geq z \lor x \geq z$. Dually, we also consider min-plus polyhedra and combine max and min-plus polyhedra to capture if-and-only-if information.

Motivating Example

We illustrate the approach with a simple example program containing a disjunctive invariant. Figure 3.6 shows program `ex1`, adapted from Gulwani and Jojic [Gulwani, 2007]. Program `ex1` first initializes y to 5 and ensures $x \leq y$, then enters a loop that increments y conditionally on the value of x. Figure 3.6 also shows the trace values for x, y at location L on input $x = -1$, and it depicts the nonconvex region (a bent

line) covering these trace points. Validating the post-condition `y == 11` requires analyzing the semantics of the loop by identifying the invariants at location L.

From the given trace data, existing tools such as Daikon can generate only conjunctive invariants such as:

$$
\begin{aligned}
y &\geq x, \\
11 &\geq x, \\
11 &\geq y \geq 5.
\end{aligned}
$$

These relations are not expressive enough to capture the disjunctive dependency between x and y, and they fail to prove the desired post-condition.

By building a max-plus polyhedra over the trace points in Figure 3.6, DIG obtains relations that simplify to:

$$
\begin{aligned}
11 &\geq x \geq -1, \\
11 &\geq y \geq 5, \\
0 &\geq x - y \geq -6,
\end{aligned}
$$

$$
(x < 5 \ \wedge \ 5 \geq y) \ \vee \ (x \geq 5 \ \wedge \ x \geq y),
$$

where the last relation is disjunctive. Next, DIG uses a custom k-inductive theorem prover to statically verify these candidate invariants against the program code given in Figure 3.6 using and removes the spurious relations $x \geq -1$ and $x - y \geq -6$. The rest are true invariants at L.

We note that the invariant $y \geq x$ is not directly k-inductive. However, by using the previously proven results $y \geq 5$ and $(x < 5 \wedge 5 \geq y) \ \vee \ (x \geq 5 \wedge x \geq y)$ as lemmas, the prover also verifies this relation $y \geq x$. Further, the prover shows that $11 \geq x$ is redundant (i.e., implied by other proved results) and can be removed. The remaining invariants are:

$$
\begin{aligned}
11 &\geq y \geq 5, \\
0 &\geq x - y,
\end{aligned}
$$

$$(x < 5 \ \wedge \ 5 \geq y) \ \vee \ (x \geq 5 \ \wedge \ x \geq y).$$

Intuitively, the code in Figure 3.6 has two phases: either $x < 5$ (at which point the `if` inside the `while` loop is not true and y remains 5), or x is between 5 and 11 (at which point the `if` inside the `while` loop is true, and $y = x$ because they are both incremented). The inferred invariants are logically equivalent to the encoding of that intuitive explanation:

$$(x < 5 \ \wedge \ y = 5) \ \ \vee \ \ (5 \leq x \leq 11 \ \wedge \ y = x).$$

They are also the precise invariants of the loop and can prove the post-condition `y == 11`. This example requires that both the dynamic analysis and the static prover be expressive and efficient enough to infer disjunctive invariants and prove them correct. We describe these methods in detail in the remainder of the section.

3.3.1   Max-plus Algebra

As discussed earlier, programs containing loops or conditional branches are not adequately modeled by purely conjunctive invariants. Figure 3.6 depicts the nonconvex region defined by the loop invariant $(x < 5 \wedge y = 5) \vee (5 \leq x \leq 11 \wedge x = y)$ in the `ex1` program. Such disjunctive information cannot be expressed as a conjunction of polynomial relations, including octagonal or even general polyhedral forms. Although some disjunctive invariants can be simulated using polynomials of higher order, e.g., $(a = 0) \vee (b = 0)$ is equivalent to $a \times b = 0$, this approach generates terms with impractically high degrees and computational cost, especially when there are more than two disjunctions. Thus, the representation of disjunctive information requires a fundamentally different approach.

To model disjunctive invariants, we use relations representing max-plus polyhedra [Allamigeon and Katz, 2013, Allamigeon et al., 2008], i.e., nonconvex hulls that are convex in a max-plus algebra [Daniel-Cavalcante et al., 2006, Heidergott and

van der Woude, 2006]. Max-plus formulas allow disjunctions of zonal relations over two variables [Cousot et al., 2005, Miné, 2004], i.e., inequalities of the forms $\pm v \geq c$ and $v_1 - v_2 \geq c$. Formally, max-plus relations have the form

$$\max(c_0, c_1 + v_1, \ldots, c_n + v_n) \geq \max(d_0, d_1 + v_1, \ldots, d_n + v_n), \qquad (3.4)$$

where v_i are program variables, c_i, d_i are real numbers or $-\infty$, and $\max(t_0, \ldots, t_m)$ returns the largest term t_i, e.g., $\max(x, y) \equiv \text{if } x > y \text{ then } x \text{ else } y$. The max operator allows max-plus formulas to encode certain disjunctions. For example, the max-plus relation $\max(0, x-5, y-\infty) = \max(-\infty, x-\infty, y-5)$, i.e., $\max(0, x-5) = \max(y-5)$, encodes the disjunction $(5 > x \ \wedge \ y = 5) \vee (5 \leq \ x \ \wedge x = y)$, or $y = 5 \ \vee \ x = y$.[4]

Table 3.1 compares linear algebra with max-plus algebra. Max-plus relations are analogous to linear relations, but use $(\max, +)$ instead of the $(+, \times)$ of standard arithmetic. These operators allow max-plus relations to form geometric shapes that are nonconvex in the classical sense. For example, the max-plus relation $(x = y) \vee (y = 5)$ represents a nonconvex region consisting of two lines $x = y$ and $y = 5$. Moreover, the structure of max-plus relations produces a relatively unusual geometric shapes. Table 3.1 depicts the three possible shapes of a max-plus line in 2D. In general dimensions, two points are always connected by lines that run parallel, perpendicular, or at a 45 degree angle to all the coordinate axes. A bounded convex max-plus polyhedron consists of these connections and the area surrounded by them. Table 3.1 illustrates an example of a max-plus convex hull, i.e., a bounded convex polyhedron, consisting of five lines connecting the five marked points. Although a max-plus convex hull is not convex in the classical sense, it is convex in the max-plus sense because it contains max-plus lines between any pair of its points.

---

[4]Because $\max(v_0, v_1 - \infty, v_2, \ldots, v_n) = \max(v_0, v_2, \ldots, v_n)$, we often drop $-\infty$ max-arguments for presentation purpose. We also abbreviate max-plus notations, e.g., $\max(x, y) \geq z$ for $\max(x, y, z - \infty, -\infty) \geq \max(x - \infty, y - \infty, z, -\infty)$ and $x \geq 9$ for $\max(9, x - \infty, y - \infty) \geq \max(-\infty, x, y - \infty)$. An equality is also used to express the conjunction of two inequalities, e.g. $\max(x, y) = z$ for $\max(x, y) \geq z \wedge z \geq \max(x, y)$.
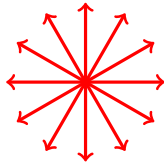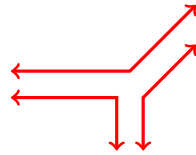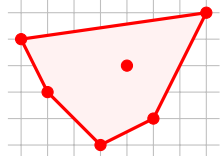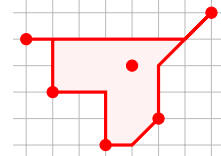
	Linear	**Max-plus**
Domain	$\mathbb{R}$	$\mathbb{R} \cup \{-\infty\}$
Addition	$+$	max
Multiplication	$\times$	$+$
Zero element	0	$-\infty$
Unit element	1	0
Relation form	$c_0 + c_1 t_1 + \cdots + c_n t_n \geq 0$	$\max(c_0, c_1 + v_1, \ldots, c_n + v_n) \geq$ $\max(d_0, d_1 + v_1, \ldots, d_n + v_n)$
Line shapes		
Convex hull examples		

Table 3.1: Comparison between linear algebra and max-plus algebra. Max-plus lines have three possible shapes: $\max(x + a, b) \geq y$ (top), $\max(y + a, b) \geq x$ (right), and $\max(x + a, y + b) \geq 0$ (left). All max-plus convex hulls are built using these lines.

In general, a bounded max-plus polyhedron can have finitely many facets represented by max-plus relations. For example, even a 2D complex polygon may contain multiple edges. Thus, a disjunctive formula representing a max-plus polyhedron has no fixed bounds on the number of disjuncts. However, constructing a max-plus polyhedron in high dimensions is computationally expensive as shown in Section 3.4.1. In Section 3.3.3, we introduce a simpler form of max-plus relations that strikes a reasonable compromise between efficiency and expressiveness.

3.3.2 Max-plus Invariants

DIG infers max-plus invariants dynamically using an algorithm similar to that used for general inequality invariants described in Section 3.2.2. Figure 3.7 outlines the main steps of the algorithm for generating max-plus inequalities.

procedure FINDMAXPLUS(V, X, d)
    $T \leftarrow$ GENTERMS(V, d)
    $P \leftarrow$ GENPOINTS(T, X)
    $H \leftarrow$ CREATEMAXPLUSPOLY(P)   ▷ construct a max-plus convex hull over points
    $S \leftarrow$ EXTRACTFACETS(H)
    return S ▷ return polynomial relations of the form given in Equation (3.4)

Figure 3.7: Algorithm for finding (max-plus) disjunctive inequalities from the inputs: set V of variables, set X of traces, and max degree d. The main steps of the algorithm are: using terms to represent products of program variables (GENTERMS), instantiating points from terms using input traces (GENPOINTS), creating a max-plus convex hull enclosing the points (CREATEMAXPLUSPOLY), and extracting its facets, which are represented by max-plus relations among terms (EXTRACTFACETS).

Similar to the process of generating general invariants, DIG employs heuristics to search iteratively for max-plus invariants containing all possible combinations of a small, fixed number of variables. The tool considers max-plus relations over triples of program variables that represent max-plus polyhedra in three-dimensional space. DIG also supports nonlinear max-plus relations by using terms to represent nonlinear polynomials over variables. However, the number of possible terms is exponential in the number of degrees as shown in Section 3.4.1 and, thus, DIG targets linear max-plus relations by default for efficiency.

Example 3.3.1. We illustrate how DIG derives the invariant $(x < 5 \wedge y = 5) \vee (5 \leq x \leq 10 \ \wedge \ x = y)$ at location L in program `ex1` in Figure 3.6. The trace values for x, y in Figure 3.6 form a set of eleven points, e.g., the first is $(-1, 5)$. DIG then computes a max-plus polyhedron over these points. The half-space representation of that polyhedron consists of the max-plus relations:

$$
\begin{aligned}
11 &\geq x \geq -1 \\
11 &\geq y \geq 5 \\
0 &\geq x - y \geq -6 \\
\max(0, x - 5) &\geq y - 5
\end{aligned}
$$

The conjunction $0 \geq x - y \ \wedge \ 11 \geq x \ \wedge \ \max(x - 5, 0) \geq y - 5$, which forms the

nonconvex region in Figure 3.6, is logically equivalent to the invariant $(x < 5 \wedge 5 = y) \vee (5 \leq x \leq 11 \wedge x = y)$.

Because x has no lower bound, $x \geq -1$ and $x - y \geq -6$ are spurious relations. The post-processing step in Section 3.1.2 removes these spurious invariants if given additional traces, such as running `ex1` on $x = -5$. More generally, the static technique in Section 3.5 formally verifies candidate invariants and removes spurious results.

3.3.3 Weak Max-plus Invariants

We introduce and define a weaker form of max-plus relations that retains much expressive power, but avoids the high computational cost of computing a general max-plus polyhedron. To the best of our knowledge, this is the first attempt to consider a simpler form of max-plus inequalities for invariant generation and program analysis.

We define a *weak* max-plus relation to be of the form:

$$
\begin{aligned}
\max(c_0, c_1 + v_1, \ldots, c_k + v_k) &\geq v_j + d, \\
v_j + d &\geq \max(c_0, c_1 + v_1, \ldots, c_k + v_k),
\end{aligned}
\tag{3.5}
$$

where v_i are program variables, $v_j \in \{v_1, \ldots, v_n\}$, $c_i \in \{0, -\infty\}$, d is a real numbers or $-\infty$, and k is constant, e.g., $k = 2$. Unlike general max-plus relations of the form given in Equation (3.4), weak max-plus relations have some restrictions:

- They restrict the values of the coefficients c_i to $\{0, -\infty\}$. The general form allows $c_i \in \mathbb{R} \cup \{-\infty\}$.

- They fix the number of variables k to a small constant. The general form allows n variables.

- They allow only one unknown parameter d. The general form allows $d_0 \ldots d_n$.

General Max-plus   Weak Max-plus
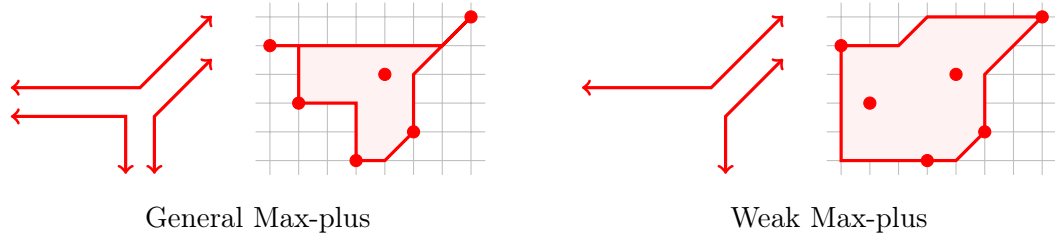
Figure 3.8: General max-plus and weak max-plus shapes

Weak max-plus relations are thus a strict subset of general max-plus relations. For example, the weak max-plus form cannot general max-plus relations like $\max(x + 7, y) \geq z$ or $\max(x, y) \geq \max(z, w)$, but it does support zonal relations like $x - y \geq 10, x = y$ and disjunctive relations like $\max(x, y) \geq z$ and $\max(x, 0) \geq y + 7$.

Geometrically, weak max-plus relations represent a restricted class of general max-plus polyhedra. Figure 3.8 compares the shapes of general max-plus relations with those of weak max-plus relations. While general max-plus lines have the possible three shapes, weak max-plus lines have only two shapes represented by the formulas $\max(x, b) \geq y$ and $\max(y, b) \geq x$. That is, weak max-plus shapes include only lines that run in parallel or at a 45 degree angle. Lines with a perpendicular shape cannot occur because their formula, $\max(x, y) \geq 0$, is inexpressible using the weak max-plus form.

Algorithm for Computing Weak Max-plus Convex Hulls

The advantage of the above restrictions is that they admit an efficient algorithm to compute the weak max-plus convex hull over a set of finite points in a fixed k dimensions, e.g., $k = 2$. The algorithm first enumerates all possible weak relations over k variables and then finds the unknown parameter d in each relation from the given points. The resulting set of relations is the half-space representation of the weak max-plus polyhedron enclosing the points. In general, the number of weak max-plus relations enumerated over k variables is $O(k2^{k+2})$, and the number of facets of a weak

max-polyhedron thus has a fixed upper bound for each k. For example, $k = 2$ has at most 32 facets. This fixed number of facets is more manageable than the unbounded number of facets of a general max-plus polyhedron.

Note that this algorithm does not apply to the general max-plus form because the coefficients c_i are not enumerable over the reals, and the problem becomes more complex when more than one unknown is involved. For instance, it is nontrivial to compute the unknowns c, d in the max-plus relation $\max(c, x) \geq y + d$ because the values of c and d depend on each other.

Example 3.3.2. The following illustrates how DIG finds the weak max-plus polyhedron enclosing the points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ in the two-dimensional plane. First, DIG enumerates relations of the weak max-plus form by instantiating the coefficients c_i over $\{0, -\infty\}$. For the form $\max(c_0, c_1 + x, c_2 + y) \geq x + d$, the tool obtains eight max-plus relations (two choices each for three coefficients):

$$\max(0, x, y) \geq x + d, \qquad \ldots$$
$$\max(0, x) \geq x + d, \qquad -\infty \geq x + d$$

The eight additional max-plus relations for each of the other three forms $\max(c_0, c_1 + x, c_2 + y) \geq y + d$, $x + d \geq \max(c_0, c_1 + x, c_2 + y)$, $y + d \geq \max(c_0, c_1 + x, c_2 + y)$ are obtained similarly. Redundant relations can be removed (e.g., $\max(y, 0) \geq x$ implies $\max(x, y, 0) \geq x$).

Next, DIG computes the parameter d in each of the 32 obtained relations using the given points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. For instance, $\max(y, 0) \geq x + d$ has $d = \min(\max(y_i, 0) - x_i)$ and $x + d \geq \max(y, 0)$ has $d = \max(\max(y_i, 0) - x_i)$. The resulting relations form an intersecting region that represents a bounded weak max-plus polygon over the given points.

Figure 3.9: (a) Three possible shapes of a min-plus line segment and (b) a min-plus polyhedron built over four points.

3.3.4 General and Weak Min-plus Invariants

DIG finds *min-plus* inequalities of the form

$$\min(c_0, c_1 + v_1, \ldots, c_n + v_n) \geq \min(d_0, d_1 + v_1, \ldots, d_n + v_n), \qquad (3.6)$$

where v_i are program variables and $c_i, d_i \in \mathbb{R} \cup \{\infty\}$. Similar to its max-plus dual, a min-plus polyhedron is a formed by the intersection of finite min-plus lines. However, min and max-plus relations describe different forms of disjunction information and have different geometric shapes. For example, the relation $\min(x, y) = z$ encodes the disjunction $(x < y \Rightarrow x = z) \wedge (x \geq y \Rightarrow y = z)$ that is not expressible using a max-plus relation. Figure 3.9 depicts the min-plus version of the shapes shown in Table 3.1.

A conjunction of max and min-plus invariants can describe information that is inexpressible using either max or min-plus relations alone. Consider the `ex2` program in Figure 3.10, which has the invariant $y \leq 10 \Leftrightarrow b = 0$ at location L. By building max and min-plus polyhedra over the traces given in Figure 3.10, DIG obtains the relations $1 \geq b \geq 0$, $\max(y - 10, 0) \geq b$, and $b + 10 \geq \min(y, 11)$. Given $1 \geq b \geq 0$, the max-plus relation implies $b = 0 \Rightarrow y \leq 10$, and the min-plus relation implies $b \neq 0 \Rightarrow y > 10$. These disjunctions are logically equivalent to the if-and-only-if condition $y \leq 10 \Leftrightarrow b = 0$.

```
def ex2(x):
    if x >= 0:
        y = x+1
    else:
        y = x-1
    b = y > 10
    [L]
    return b
```

x	y	b
-50	-51	0
-33	-34	0
9	10	0
10	11	1
12	13	1
40	41	1

Figure 3.10: Program `ex2` and its trace data at location L for several input values.

As a dual to the weak max-plus relations introduced in Section 3.3.3, we define weak min-plus relations to be of the form:

$$\min(c_0, c_1 + v_1, \ldots, c_k + v_k) \geq v_j + d_i,$$
$$v_j + d_i \geq \min(c_0, c_1 + v_1, \ldots, c_k + v_k), \tag{3.7}$$

where v_i are program variables, $v_j \in \{v_1, \ldots, v_n\}$, $c_i \in \{0, -\infty\}$, $d_i \in \mathbb{R} \cup \{-\infty\}$, and k is constant. The algorithm for building weak min-plus polyhedra over finite points is similar to the one for weak max-plus polyhedra as given in Section 3.3.3.

3.4 Algorithmic Analysis

In this section, we first give the computational complexity of DIG's algorithms for generating different forms of invariants. We then show that the convex hull method generates precise inequality invariants, but it also generate spurious results if the program invariants do not appear in the traces.

3.4.1 Computational Complexity

Figure 3.2 summarizes the time complexity of DIG's algorithms for generating different forms of polynomial invariants as a function of the number of traces $|X|$ and terms $|T|$, where terms are used to represent polynomials over variables as described

Invariant Type	Form	Complexity
Equality	(3.1)	$O(|T|^3)$
Polyhedral (general) inequality	(3.2)	$O(|X|^{\frac{|T|}{2}})$
Octagonal inequality	(3.3)	$O(|X||T|^2)$
Max/Min-plus inequality	(3.4)	$O(|X||T|^2(|X|+|T|)^{|T|})$
Weak Max/Min-plus inequality	(3.5)	$O(|X|2^{|T|})$

Table 3.2: Time complexity of polynomial invariant generation algorithms. T represents the set of terms and X the set of traces.

in Section 3.1.2. Given a set V of numerical variables and a positive degree d, the set T of terms representing monomials over V up to degree d has size $\binom{|V|+d}{d}$. The number of terms thus increases exponentially in the number of variables and degrees.

- *Equalities.* To find equality invariants of the form given in Equation (3.1), DIG applies a standard equation solver using Gaussian elimination over the $|T|$ independent equations instantiated from the traces in X as shown in Section 3.2.1. The complexity of Gaussian elimination to solve $|T|$ linear equations for $|T|$ unknowns is $O(|T|^3)$ [Farebrother, 1988]. Hence, generating invariants representing equations among $|T|$ terms takes $O(|T|^3)$.

- *General Inequalities.* As described in Section 3.2.2, DIG builds polyhedra to obtain polyhedral (general) invariants of the form given in Equation (3.2). Constructing a convex polyhedron over $|X|$ points in $|T|$ dimensions has a theoretical exponential upper bound $\Theta(|X|^{\lfloor\frac{|T|}{2}\rfloor})$ [de Berg et al., 1997]. Thus, the cost of generating general inequalities is $O(|X|^{\frac{|T|}{2}})$, exponential in the number of terms (because each new term essentially defines a new variable representing a new dimension).

  In the case of inequalities among a *fixed* number k of terms over program variables, the heuristic described in Section 3.2.2 builds polyhedra for all term combinations of size k. The complexity of such a heuristic is $O(\binom{|T|}{c}|X|^k)$,

which is polynomial in $|X|$ and $|T|$ because k is fixed.

- *Octagonal Inequalities.* To avoid the high cost of building general polyhedra in high-dimensional space, DIG finds octagonal invariants of the form given in Equation (3.3) representing inequalities between two terms. The algorithm given in Section 3.2.3 first instantiates each pair of terms with the traces in X to obtain the set of $|X|$ points in two dimensions, and it then applies the $\min, \max$ operations on these points. These two operations run in linear time in $|X|$, thus identifying the octagonal inequalities for each pair of terms takes $O(|X|)$. There are $O(|T|^2)$ such pairs from the set of terms T; hence generating octagonal relations for all pairs of terms takes $O(|X||T|^2)$.

- *Max/Min Inequalities.* DIG constructs max-plus polyhedra as shown in Section 3.3.2 to obtain max-plus inequalities of the form given in Equation(3.4). DIG uses the algorithm in [Allamigeon, 2009, Allamigeon et al., 2008], which takes $O(|X||T|^2(|X|+|T|)^{|T|})$, to build a max-plus polyhedron over $|X|$ points in $|T|$ dimensions. Thus, the cost of finding max-plus inequalities is exponential in the number of terms, similar to general inequality computations.

  The technique in Section 3.3.4 builds min polyhedra for finding min inequalities of the form given in Equation (3.6) and has the equivalent complexity.

- *Weak Max/Min Inequalities.* For weak max-plus inequalities of the form given in Equation (3.5), DIG enumerates weak relations over terms and computes unknown parameters in these relation using trace points as shown in Section 3.3.3. The number of enumerated relations over k terms is $O(k2^{k+2})$ and the time to find the single parameter d in each relation is linear in the number of points. Thus, constructing a weak max-plus polyhedron over $|X|$ points in $|T|$ dimensions takes $O(|X|2^{|T|})$, polynomial in the number of points when $|T|$ is constant and is exponential in the number of dimensions when $|T|$ is not fixed. Note

that even this worst case is still smaller than $O(|X||T|^2(|X| + |T|)^{|T|})$, the complexity of building a general max-plus polyhedron as given above.

The computation of weak min inequalities of the form given in Equation (3.7) has the same complexity.

After generating polynomial invariants, DIG filters these candidate results against additional traces. The filtering technique in Section 3.1.2 takes $O(|X||T|)$ to instantiate and check a polynomial relation among $|T|$ terms over $|X|$ traces.

3.4.2 Underapproximation Property and Spurious Invariants

The convex hull method described in Sections 3.2 and 3.3 merit additional discussion because it generates candidate invariants that underapproximate the *true* program invariants that are expressible using the considered inequality forms. However, if the program invariants do not fall under the considered forms, then the convex hull method can create a complex polyhedron whose facets represent many spurious invariants.

Underapproximation

A dynamically inferred invariant can either be equivalent to, underapproximate (i.e., be a spurious invariant that is too strong and does not always hold), or overapproximate (i.e., be too weak and possibly not useful) the program invariant. For instance, when the template $x \leq y$ is used to infer the program invariant $x \leq y - 10$, then this template, an overapproximation of the program invariant, is returned as the candidate invariant. We show that this overapproximation situation cannot happen in DIG, i.e., the tool only generates candidate inequalities that are equivalent to or underapproximate the program invariant. This property is useful because it can be used to detect program errors. The violation of this property, i.e., the inferred

invariant strictly overapproximates the program invariant, indicates that the program invariant fails for some observed traces and, thus, the program has a bug. For example, consider the below `flatten` code in Section 3.2.3 with an off-by-one error.

```
for (i = 0; i < M; ++i){
    for (j = 0; j <= N; ++j){ //bug, should be j < N
        k   = i*n+j;
        [L]
        B[k] = A[i][j];
    }
}
```

Depending on the given traces, DIG may generate at L the octagonal relation $0 \leq k \leq MN + 5$, which is an overapproximation of the program invariant $0 \leq k \leq MN - 1$. This indicates an error because DIG would never generate such a relation unless the value $k = MN + 5$ is in the traces, i.e., a counterexample that violates the program invariant.

Theorem 3.4.1 (Underapproximation Theorem). *If a program invariant belongs to an inequality form supported by DIG, then a candidate inequality generated from DIG using convex hulls is guaranteed to either be equivalent to or underapproximate the program invariant.*

Proof. The above theorem states that if F is the program invariant (i.e., a conjunction of inequalities of the form given in Equation (3.2), (3.3), (3.4), (3.5), (3.6), or (3.7) representing a bounded convex object in multidimensional space), then the candidate invariant F' of that form is equivalent to or underapproximates F, i.e., $F' \Rightarrow F$.

This underapproximation property is established using on the facts that the observed traces are a *subset* of all possible traces and that a convex hull of a set of

points is the *smallest* convex set containing those points. First, the geometric object represented by F encloses all trace points because F is the true program invariant that holds for all program traces. Next, the candidate invariant F' has the same form as F (i.e., F has the same geometric shape as F), but encloses only a subset of the program trace points. Finally, because F' is computed as the convex hull of that subset of trace points, the geometric object represented by F' is enclosed in the object represented by F. Thus, $F' \Rightarrow F$. □

The underapproximation property $F' \Rightarrow F$ also holds if the form of F' is more expressive than the form of F. However, this property is not guaranteed if the form of F' is less expressive than the form of F. For example, in Fig. 3.4, a program invariant representing an octagon (Fig. 3.4c) overapproximates a candidate invariant representing a polygon (Fig. 3.4b) and underapproximates a candidate invariant representing an interval (Fig. 3.4e).

Observe that *equivalence*, i.e., $F = F'$, is achieved when the observed traces consist of the *extreme* points describing the form of the program invariant. For instance, DIG can find the exact inequalities representing an octagon from any set of traces consisting of the eight extreme points of that octagon. Similarly, DIG also generates the correct equalities when given sufficient traces describing the program invariant, e.g., three distinct points for a plane. As mentioned in Chapter 2, the generation of test inputs to obtain such desirable traces is an popular research area that have many active projects. In particular, we can take advantage of an entire body of work on generating test inputs specifically for dynamic invariant detection [Gupta and Heidepriem, 2003, Harder et al., 2003, Xie and Notkin, 2003].

We note that the underapproximation property also holds for equalities of the form given in Equation (3.1) generated by DIG, as proved in [Sharma et al., 2013a].

Spurious Invariants

The convex hull method has high computational complexity because complex polyhedra with multiple facets in high dimensions might be generated depending on the given trace points. Importantly, if the traces do not precisely capture the program invariant, then the polyhedron will consist of many facets representing spurious inequalities. For instance, if x, y can take any value over the reals, then an n-facet polygon computed over any set of traces for x, y produces n spurious invariants because no bounded polygons can capture the unbounded ranges of x, y.

Although filtering (Section 3.1.2) reduces spurious invariants by removing facets of the polyhedron (i.e., widening it), the modified polyhedron may still retain facets representing spurious relations. Thus, DIG does not automatically invoke the convex hull method for general inequalities. The convex hull method described in Section 3.2.2 is effective when the user of DIG has certain expectations about the program invariants. The user can change the parameter d in Algorithm 3.7 to generate higher degree relations (e.g., $d = 2$ for quadratic relations) and can also manually define terms to capture other desirable properties. For example, a user with knowledge about the form of the desired invariants might hypothesize a *spherical* form $c_1 x^2 + c_2 y^2 + c_3 z^2$. With that as input, DIG searches for that exact form (i.e., computes the coefficients c_i, d_i) from the polyhedron built over the trace points of the terms representing the nonlinear polynomials x^2, y^2, z^2. In addition to reducing computational cost, the use of weaker polyhedral form in DIG helps mitigate the number of spurious invariants, e.g., the octagonal form of invariants in Section 3.2.3 admits exactly eight candidate invariants. In Section 3.5, we present a more general technique based on theorem proving, which distinguishes between true and spurious invariants using the program code.

In contrast to the convex hull construction, methods using equation solving give

few spurious equalities because equalities are stricter constraints than inequalities. For example, we can always compute a convex polygon representing many inequalities over any set of finite points in 2D, but we can have at most a line representing an equality over these points. Moreover, assuming traces are obtained from random program inputs, it is unlikely that a large set of traces would exhibit random false equalities.

3.5 Proving Invariants Statically

As discussed in Chapter 2, dynamic invariant generation is efficient but not sound. To address this limitation, we next show how to augment dynamic invariant generation with static theorem proving and produce sound program invariants. Specifically, we describe an automatic theorem prover that DIG uses to verify candidate invariants. The theorem prover, called KIP (k-Inductive Prover), is based on iterative k-induction and uses constraint solving to verify candidate invariants. In this approach, $k+1$ base cases are specified, and all $k+1$ previous instances are available for proving the inductive step (e.g., [Donaldson et al., 2011]). This additional power allows KIP to prove many invariants that are not provable with standard 0-induction. KIP leverages recent advances in SMT solving [De Moura and Bjørner, 2008, Jovanović and De Moura, 2012, Nuzzo et al., 2010] and can efficiently analyze formulas encoding complex programs and properties such as nonlinear arithmetic. The architecture of KIP supports parallel checking of invariants, dramatically improving efficiency.

Example 3.5.1. Consider the `sqrt` program in Figure 3.11, which computes the square root of an integer using only addition. From observed traces at location L, DIG generates candidate loop invariants such as $t = 2a+1, 4s = t^2 + 2t + 1, s = (a+1)^2, s \geq t$ and $9989 \geq x$ using the geometric techniques described in Section 3.2. KIP successfully distinguishes true and false invariants from these results. Specifically,

```
def sqrt(x):
    assert x >= 0
    a = 0; s = 1; t = 1;
    while[L] s <= x :
        a = a + 1
        t = t + 2
        s = s + t
    return a
```

Figure 3.11: A program computing square root using only addition.

the prover identifies $t = 2a + 1$ and $4s = t^2 + 2t + 1$ as inductive invariants and $s = (a + 1)^2$ as a 1-inductive invariant (i.e., would not be proved using standard 0-induction). By using proved results as lemmas, KIP proves the invariant $s \geq t$, which is not k-inductive for $k \leq \mathsf{maxK}$, where $\mathsf{maxK} = 5$ is a parameter in the prover and the default setting of KIP. The prover also rejects spurious relations such as $9989 \geq x$ by producing counterexamples that invalidate those relations in `sqrt`. The parallel implementation allows KIP to check these candidate results simultaneously.

3.5.1 Analyzing Programs using k-Induction

A program execution can be modeled as a state transition $M = (I, T)$ with I representing the initial state of M, and T specifying the transition relation of M from a state $n - 1$ to a state n. To prove that p is a *state invariant*, which holds at every state of M, k-induction requires that p hold for the first $k + 1$ states (the base case) and that p hold for the state $n + k + 1$ if it holds for the $k + 1$ previous states (the induction step). Formally, k-induction proves the state invariant p of $M = (I, T)$ by checking the base case and induction step formulas:

$$I \ \wedge \ T_1 \ \wedge \ \dots \ \wedge \ T_k \ \Rightarrow \ p_0 \ \wedge \ \dots \ \wedge \ p_k \qquad (3.8)$$

$$p_n \ \wedge \ T_{n+1} \ \wedge \ \dots \ \wedge \ p_{n+k} \ \wedge \ T_{n+k+1} \ \Rightarrow \ p_{n+k+1} \qquad (3.9)$$

procedure KPROVE(I, T, p)
    for $k \leftarrow 0 \ldots \mathsf{maxK}$ **do**
        ▷ base case
        if $k = 0$ **then**
            $\mathsf{S_b}.assert(I)$
        else
            $\mathsf{S_b}.assert(T_k)$
        if $\neg \mathsf{S_b}.entail(p_k)$ **then**
            return *disproved*, $\mathsf{S_b}.cex$ ▷ p is not an invariant

        ▷ induction step
        $\mathsf{S_s}.assert(p_k, T_{k+1})$
        if $\neg \mathsf{S_s}.entail(p_{k+1})$ **then**
            return *proved*   ▷ p is a k-inductive invariant

    return *unproved*   ▷ p is not k-inductive

Figure 3.12: Algorithm for incremental k-induction using SMT solvers $\mathsf{S_b}$ and $\mathsf{S_s}$.

If both formulas can be proved then p is a k-inductive invariant. If the base case (3.8) fails, then p is disproved and is not an invariant of M (assuming that M correctly models the program). However, if the base case holds but the induction step (3.9) fails, then p is not a k-inductive invariant but it could still be a program invariant. Thus, k-induction is a sound but incomplete proof technique.

By considering multiple consecutive transitions, k-induction can prove invariants that cannot be proved by standard induction (0-induction in this formulation). For example, the invariant $x \neq y$ of the transition $M(I : (x = 0 \land y = 1 \land z = 2)_0,$ $T_n : x_n = y_{n-1} \land y_n = z_{n-1} \land z_n = x_{n-1})$ that rotates the values $0, 1, 2$ through the variables x, y, z is not provable by standard induction, but is k-inductive with $k \geq 3$. The notation $(P)_i$ denotes the formula P with all free variables subscripted by i, e.g., $(x + y = 1)_0$ is $x_0 + y_0 = 1$.

3.5.2 k-Induction and SMT Solving

Figure 3.12 outlines the algorithm for verifying a property p using iterative k-induction with SMT solving. The algorithm consists of a loop that performs incremental k-induction, starting from $k = 0$. The loop terminates when either the base case fails (P is not an invariant), both the base case and the induction step are proved (P is an invariant), or maxK is reached. In the last case, we say that P is not a maxK-inductive invariant.

We use two independent SMT solvers S_b and S_s to check the two formulas corresponding to the base case (3.8) and induction step (3.9).[5] For a solver S and a formula f, we append f to S through *assertions* and check if the assertions $a_1, \ldots, a_n$ in S imply f using *entailment* [De Moura and Bjørner, 2008]. If S does not entail f, then the solver returns a counterexample (cex) satisfying $a_1 \wedge \cdots \wedge a_n$ but not f.

3.5.3 The Architecture of KIP

At a high level, proving a candidate invariant against a program requires two steps: (i) computing a formula that encodes the program's semantics, and (ii) deciding whether the candidate invariant is consistent with that formula or not. To increase expressive power in practice, the prover also (iii) incorporates knowledge of all invariants learned thus far.

Figure 3.13 outlines the architecture of KIP to verify a set P of candidate obtained at location L for program S. We first generate from the program S and the location L the formulas I, T to represent the state transition $M = (I, T)$ described above. Essentially, the formulas I, T are *verification conditions* (vcs) based on weakest pre-conditions (wps) from program analysis using Floyd-Hoare logic [Floyd, 1967,

---

[5]The two SMT solvers can share the same implementation: "independent" merely indicates that they may hold different assumptions at runtime.

```
procedure KIP(S, L, P)
    I, T ← vcgen(S, L)              ▷ verification conditions from S to check properties at L
    Pp, Pd, Pu ← ∅

    repeat
        Newp, Newu ← ∅
        for p ∈ P do
            r ← KPROVE(I, T, p)
            if r = proved then
                Pp.add(p); Newp.add(p)
            else if r = unproved then
                Newu.add(p)
            else
                Pd.add(p)
    until Newp = ∅ ∨ Newu = ∅

    Pu ← P
    Pi, Pr = CHECKREDUNDANCY(Pp)
    return Pi, Pr, Pd, Pu
```

Figure 3.13: Algorithm to verify candidate invariants. P_i and P_r are proved results, but P_r is redundant because $P_i \Rightarrow P_r$. P_d is disproved, and P_u is unknown.

Hoare, 1969]. The backward analysis method [Dijkstra, 1975] provides the necessary rules to create I, T for imperative programming constructs such as assignments, conditional branches, and loops. This area is well-established; tools such as Microsoft Boogie [Leino, 2008] and ESC [Flanagan et al., 2002] implement various methods based on backward analysis to automatically generate vcs using wps.

KIP progresses by trying to prove the invariants in the context of the vcs. While unproved invariants remain, KIP re-attempts to prove them by adding newly proved results as lemmas to KIP. In many cases, this additional knowledge allows KIP to prove properties that could not be proved previously (see Section 3.6.1). A disproved invariant is likely spurious, a proved invariant is definitely correct, and an unproved invariant (e.g., one that is not maxK-inductive) can be conservatively rejected.

KIP's design admits a parallel implementation, checking candidate invariants (the

`for` loop in Figure 3.13) simultaneously using multiple threads. In a post-processing step, KIP uses implication to partition all proved invariants into two sets: those that are independent, i.e., strongest and those that are implications of others, i.e., weaker. Implied invariant are redundant and need not be presented to the developer. This partitioning uses the backend SMT solver to check if each invariant $p \in P_p$ can be inferred by the conjunction of the other proved invariants $P_p \setminus \{p\}$.

To summarize, KIP combines several established techniques and provides the five properties we desire for the efficient verification of complex invariants: (i) use of k-induction for expressive power; (ii) use of SMT solvers for reasoning about program-critical theories like nonlinear arithmetic; (iii) incorporate lemmas iteratively to prove otherwise non-inductive properties; (iv) explicit parallelism for performance; and (v) removing weaker implied results for human consumption.

3.6   Experiments

The invariant analysis prototype DIG is implemented in Python using the Sage mathematical environment [Stein et al., 2014]. DIG uses built-in Sage functions to solve equations and construct convex hulls for classical polyhedra, and it uses TPLib [Allamigeon and Katz, 2013] to manipulate max and min-plus polyhedra. The prover prototype KIP is also implemented in Python and uses the Z3 [De Moura and Bjørner, 2008] solver to check the satisfiability of SMT formulas. The website `https://bitbucket.org/nguyenthanhvuh/dig/` contains the source code of DIG and KIP, benchmark programs, and experimental results given in this chapter.

To evaluate the efficiency and expressive power of the hybrid approach of DIG and KIP, we consider the two research questions:

- Can DIG and KIP together effectively generate on *complex* correctness properties, such as those that are not classically inductive or involve nonlinear

arithmetic, and prove them correct?

- Can DIG and KIP together efficiently discover powerful *disjunctive* invariants and prove them correct?

To investigate the first question, we applied DIG to a benchmark suite of algorithms involving nonlinear arithmetic [Nguyen et al., 2012]. To investigate the second question, we used a benchmark suite of kernels consisting of abstractions of string and array processing and involving disjunctive information [Nguyen et al., 2014c].

Each program was run on 300 random inputs to provide traces for invariant generation and 100 random inputs for filtering. For small kernels, this yields traces that are sufficient to generate accurate invariants [Nimmer and Ernst, 2002, Sharma et al., 2013b]. These programs include annotated invariants at various locations such as loop heads and function exits. For evaluation purposes, we instrumented the values of variables at those locations and generated invariants among the resulting traces.

DIG first generates equality relations and then proceeds to generate inequalities, using the deduction method when additional information such as loop guards is available. By default, DIG automatically finds the octagonal relations given in Section 3.2.3, and it does not generate general inequalities using the polyhedral method unless the user specifies it. The tool generates only the weak linear max and min-plus relations given in Section 3.3.3 unless the number of variables is three or less, in which case it is also practical to use the general forms. The prototype KIP sets maxK = 5 by default and takes as input the verification conditions corresponding to $M = (I, T)$ (Section 3.5.3); a more efficient tool such as Microsoft Boogie could also be used to generate these verification conditions. The experiments reported here were performed on a 32-core 2.60GHz Intel Linux system with 128 GB of RAM; KIP used 64 threads of parallelism.

3.6.1   Nonlinear Invariants

We evaluate DIG on complex programs, such as those that are not classically inductive or use nonlinear arithmetic, by studying the NLA (nonlinear arithmetic) test suite [Nguyen et al., 2012]. The suite, shown in Table 3.3, consists of 27 programs from various sources collected previously by Rodríguez-Carbonell and Kapur [Carbonell, 2006, Carbonell and Kapur, 2007, Rodríguez-Carbonell and Kapur, 2007]. These programs are relatively small, on average two loops of 20 lines of code each. However, they implement nontrivial mathematical algorithms and are often used to benchmark static analysis methods [Carbonell, 2006]. The documented correctness assertions for these 27 programs require nonlinear invariants, mostly equalities among nonlinear polynomials.

For these programs, we generate and check loop invariants of two polynomial forms: nonlinear equations and linear max-plus inequalities among program variables. In this experiment, we define a single parameter $\alpha = 200$ to bound DIG's running time. DIG automatically adjusts the maximum degree so that the number of generated terms does not exceed α. For example, the tool will consider invariants up to degree 5 for a program with four variables and invariants up to degree 2 for a program with twelve variables.

Table 3.3 reports experimental results. The number of generated invariants shown in the **Gen** column speaks to the expressive power of the algorithm: higher is better, indicating that DIG can reason about more complex relationships over program variables. Time indicates the efficiency of DIG: lower is better. The generated invariants were disproved three times as often as they were proved redundant. The significant presence of invariants requiring k-induction or learned lemmas validates the KIP architecture design choice. KIP is able to formally validate 118 of the generated invariants, or 4.3 per program on average, proving them correct and non-

redundant. Some of the theorem prover queries issued caused the underlying Z3 SMT solver to return an *unknown* error or to stop responding. These errors are likely due to recent revision in Z3 to support nonlinear arithmetic, and we reported these errors to the developers. In the interim, however, such candidate invariants must be rejected.

Ultimately, the invariants generated and validated by DIG can be used to statically prove the correctness of 22 of these 27 programs using Floyd-Hoare logic. Of the remainder, `divbin` and `hard` require novel invariant forms, `egcd1` requires invariants that are not k-inductive, and `prodbin` and `dijkstra` are correct but beyond the capability of current SMT solvers. For the first type, `divbin` requires the invariant $\exists k.x = 2^k$, and DIG does not currently support exponential forms. The `hard` program also has exponential invariants. For the second type, DIG generates three nonlinear equalities that precisely capture `egcd1`'s semantics, and manual inspection verifies that they are not k-inductive for any k, and thus KIP does not prove them. For the third type, DIG generates invariants that precisely capture the semantics of `prodbin` and `dijkstra` and KIP can process them, but the backend SMT solver hangs instead of proving them (we manually verified that they are otherwise correct). Thus, KIP could prove two more programs with an improved SMT solver, two more programs with a better theorem prover architecture, but it could not prove the last without a new algorithm for invariant generation.

3.6.2 Disjunctive Invariants

We also evaluate DIG on several benchmark kernels for disjunctive invariant analysis [Allamigeon et al., 2008], listed in Table 3.4. These programs typically have many execution paths, e.g., `oddeven5` contains 12 serial conditional blocks and thus 2^{12} possible execution paths through the program. The documented correctness as-

sertions for these programs require reasoning about disjunctive invariants,[6] but do not involve higher-order logic. For example, the sorting algorithms are asserted to produce sorted output, but are not asserted to produce a permutation of the input.

Table 3.4 shows the experimental results, in a format similar to that of Table 3.3. The table shows that the DIG approach is efficient. DIG can infer about 3000 disjunctive relations per minute, on average, and KIP validates about 300 per minute using the 32-core Linux system mentioned earlier. DIG is also effective; it produced 264 non-redundant, proved-correct disjunctive invariants, and those invariants were sufficient to statically prove each program's contract. For all of these programs, the invariants generated and validated by DIG—an average of 18 per program—were sufficient for a static proof of program correctness.

For example, for the C string function `strncpy`, which copies the first n characters from a (null-terminated) source s to a (unconstrained) destination d, DIG inferred the relation:

$$(n \geq |s| \ \wedge \ |d| = |s|) \ \vee \ (n < |s| \ \wedge \ |d| \geq n)$$

This captures the desired semantics of the function: if $n \geq |s|$, then the copy stops at the null terminator of s, which is also copied to d, so d ends up with the same length as s. However, if $n < |s|$, then the terminator is not copied to d, so $|d| \geq n$.

As a second example, for $\mathtt{bubble}_N$ and $\mathtt{oddeven}_N$, which sort the input elements $x_0, \ldots, x_N$ and store the results in $y_0, \ldots, y_N$, DIG's generated invariants prove the outputs y_0 and y_N hold the smallest and largest elements of the input. However, DIG cannot show that y is a permutation of x because that is only expressible using higher-order logics, but the obtained invariants here are similar to those of purely

---

[6]Note that this suite is relatively small. Max-plus algebra is still relatively new, and although it has practical applications such as network traffic shaping [Daniel-Cavalcante et al., 2006, Heidergott and van der Woude, 2006] and biological sequence alignment [Comet, 2003], to our knowledge this is the first work on dynamic inference for max-plus invariants and, thus, few benchmarks are yet available.

static analyses [Allamigeon et al., 2008].

3.7 Summary

This chapter presented DIG, the first dynamic invariant generator that discovers conjunctions and disjunctions of polynomial relations over numerical variables. To find conjunctions of nonlinear equalities, DIG generates terms representing nonlinear polynomials among variables and uses an equation solver to find linear relations among the terms; this yields nonlinear relations among the original variables. DIG represents a conjunction of inequalities using geometric shapes and reduces the task for inferring general inequalities to generating convex polyhedra. To find the max-plus class of disjunctive polynomial invariants, we reformulate the problem of convex invariant detection in a non-standard max-plus algebra. DIG generates terms and then builds max-plus polyhedra consisting of nonconvex facets represented by the desired disjunctive invariants. DIG gains expressive power with dual min-plus constraints, capturing if-and-only-if behavior. By generating invariants directly based on input traces, DIG produces very accurate results with respect to given traces. To deal with spurious results, we presented KIP, a parallel k-inductive SMT theorem prover, and integrated it with DIG to formally check candidate invariants statically against program code.

We evaluate DIG using difficult benchmark kernels involving nonlinear arithmetic and abstract arrays. DIG is efficient and effective at finding and validating disjunctive, nonlinear and complex invariants. Ultimately, DIG finds and verifies invariants that are powerful enough to prove 36 of 41 programs from the given benchmark suites correct using Floyd-Hoare logic, taking two minutes per program, on average, and producing no spurious answers.

Program	Loc	Var	Gen	$T_{Gen}(s)$	Val	kI	$T_{Val}(s)$	Strength
cohendiv	2	6	152	26.2	7	14	8.2	✓
divbin	2	5	96	37.7	8	15	8.7	–
manna	1	5	49	19.2	3	2	5.6	✓
hard	2	6	107	14.2	11	4	9.2	–
sqrt1	1	4	27	25.3	3	1	4.3	✓
dijkstra	2	5	61	30.7	8	6	10.9	–
freire1	1	3	25	22.5	2	0	2.2	✓
freire2	1	4	35	26.0	3	1	5.1	✓
cohencb	1	5	31	23.6	4	1	4.2	✓
egcd1	1	8	108	43.1	1	8	12.8	–
egcd2	2	10	209	60.8	8	12	14.6	✓
egcd3	3	12	475	67.0	14	25	23.4	✓
lcm1	3	6	203	38.9	12	0	14.2	✓
lcm2	1	6	52	14.9	1	10	0.9	✓
prodbin	1	5	61	28.3	3	10	1.1	–
prod4br	1	6	42	9.6	4	7	8.6	✓
fermat1	3	5	217	75.7	6	1	6.2	✓
fermat2	1	5	70	25.8	2	0	5.2	✓
knuth	1	8	113	57.1	4	6	24.6	✓
geo1	1	4	25	16.7	2	4	1.5	✓
geo2	1	4	45	24.1	1	10	2.1	✓
geo3	1	5	65	22.1	1	12	2.7	✓
ps2	1	3	25	21.1	2	0	4.0	✓
ps3	1	3	25	21.9	2	0	4.2	✓
ps4	1	3	25	23.5	2	0	4.9	✓
ps5	1	3	24	24.9	2	0	7.4	✓
ps6	1	3	25	25.0	2	0	69.5	✓
total			2392	825.9	118	149	266.3	22/27

Table 3.3: Nonlinear arithmetic experimental results. The **Loc** column lists the number of locations where invariants were generated. The **Var** column reports the number of distinct variables involved in the invariants. The **Gen** column counts the number of unique candidate invariants generated by DIG. The T_{Gen} column reports the generation and filtering time, in seconds, averaged over five runs. The **Val** column reports the number of generated invariants that KIP proved correct and non-redundant with respect to the program. The k**I** column counts the number of invariants that require k-induction to be proved or disproved. The T_{Val} column counts the time, in seconds, to analyze all of the generated invariants. The **Strength** column indicates whether the validated invariants were sufficient to prove program correctness using Floyd-Hoare logic. These benchmark NLA programs and experimental results are available at `https://bitbucket.org/nguyenthanhvuh/dig/`.

Program	Loc	Var	Gen	$T_{Gen}(s)$	Val	$T_{Val}(s)$	Strength
ex1	1	2	15	0.2	4	1.5	✓
strncpy	1	3	69	1.1	4	7.7	✓
oddeven3	1	6	286	3.7	8	16.0	✓
oddeven4	1	8	867	12.7	22	46.0	✓
oddeven5	1	10	2334	56.8	52	1319.4	✓
bubble3	1	6	249	4.1	8	4.9	✓
bubble4	1	8	832	11.7	22	47.6	✓
bubble5	1	10	2198	53.9	52	938.2	✓
partd3	4	5	479	10.5	10	50.8	✓
partd4	5	6	1217	23.3	15	181.1	✓
partd5	6	7	2943	53.3	21	418.1	✓
parti3	4	5	464	10.3	10	45.5	✓
parti4	5	6	1148	22.4	15	165.1	✓
parti5	6	7	2954	53.6	21	405.6	✓
total			16055	317.6	264	3647.5	14/14

Table 3.4: Disjunctive invariant experimental results. **Loc** lists the number of locations where invariants were generated. **Var** reports the number of distinct variables involved in the invariants. **Gen** counts the number of unique candidate invariants generated by DIG. T_{Gen} reports the generation and filtering time, in seconds, averaged over five runs. **Val** reports the number of generated invariants that KIP proved correct and non-redundant with respect to the program. T_{Val} counts the time, in seconds, to analyze all of the generated invariants. **Strength** indicates whether the validated invariants were sufficient to prove program correctness using Floyd-Hoare logic. These benchmark programs and experimental results are available at `https://bitbucket.org/nguyenthanhvuh/dig/`.

Chapter 4

Array Invariants

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." – Linus Benedict Torvalds[1]

This chapter is about the dynamic generation of array invariants from traces. We formally introduce and analyze a form of nested relation among arrays. DIG implements algorithms developed in the analysis to generate array invariants that appear in many applications. Parts of this chapter have been published in [Nguyen et al., 2012, 2014a]. The complexity analysis of the array nesting problem is a collaboration with Matthias Horbach.

4.1 Introduction

Arrays are a widely-used data structure that is fundamental to many programs. For example, in Hoare's seminal 1971 paper on algorithm verification, *Proof of a program: FIND*, the overall goal is to prove an array invariant that lies at the heart of the correctness of quicksort [Hoare, 1971, p.40]. Many data structures, including lists,

---

[1]Finnish-American software engineer, who was the principal force behind the development of the Linux kernel (1969 – present).

strings, heaps, queues, stacks, and hash tables, are frequently implemented on top of arrays. Fixed-size arrays are also present in many system programs, and proper analysis is often critical for security, e.g., buffer overruns. Finally, the ubiquity of arrays in general software makes reasoning about arrays crucial for performance, e.g., for bounds check elimination [Bodík et al., 2000].

This chapter describes methods for analyzing complex relations among multidimensional array variables that appear in real-world applications. For example, these array relations are involved in over one half of the required invariants in a well-known AES (Advanced Encryption Standard) implementation [Yin et al., 2009]. First, we show that the generation of a certain form of nested relations among arrays is related to the mathematical problem of composing functions. Then, we prove that both problems are strongly NP-complete in the number of arrays or functions involved, but they can be solved in polynomial time in the number of array elements or function inputs. These results establish the run-time complexity for both problems and suggest directions to develop techniques for solving them.

Based on this theoretical work, we develop techniques in DIG to discover array invariants, including nested and flat relations among multidimensional array variables. To find these invariants, DIG employs equation solving, performs reachability analysis, and then encodes the problem as a satisfiability problem that can be handled by an SMT solver. The integration of equation and SMT solvers allows DIG to analyze efficiently complex array invariants that have not been previously considered by either static or dynamic methods. Experimental results provide evidence that DIG is effective at generating invariants for practical software like AES.

4.1.1 Contributions

We make the following contributions to the dynamic inference of array invariants in this chapter:

- *Complexity Analysis (Section 4.2).* We formally analyze the relation between the problem of finding of nested relations among arrays and the task of composing functions. We prove the complexity of both problems to be strongly NP-complete in the number of involved functions or arrays and show that they are polynomial in the sizes of the functions or arrays.

- *Nested Array Relations (Section 4.3).* We implement of the algorithms developed in Section 4.2 to infer nested array relations among array variables with multiple dimensions and functions of multiple arguments. In particular, we encode the problem of finding nested array relations as a satisfiability problem that can be efficiently solved by an SMT solver.

- *Flat Array Relations (Section 4.4).* We use equation solving to dynamically infer flat relations among multidimensional array variables from program execution traces. The technique also identifies flat relations over certain subsets of array elements, i.e., a form of conditional invariant.

- *Experimental Evaluation (Section 4.5).* We empirically evaluate DIG on a full implementation of the AES encryption algorithm that contains many array invariants. DIG successfully discovers all annotated invariants of the considered forms of array relations.

4.2 Function Composition and Array Nesting

In this section, we provide a theoretical framework for finding a form of nested relation among arrays. We first define the problem of function composition and analyze the complexity of this problem and its variants. We then show that finding nested array relations is a special case of composing functions and present efficient algorithms for finding these relations.

4.2.1   The Function Composition (FC) problem

Functions, which describe the relation between a set of inputs (a domain) and a set of outputs, are *"the central objects of investigation"* in most fields of modern mathematics [Spivak, 2006]. In mathematics and computer science, the *function composition* problem searches for applications of functions to produce a target function. For example, applying $f : Y \to Z$ to $g : X \to Y$ yields a function mapping $x \in X \mapsto (f \circ g)(x) \in Z$. The problem has many practical values, e.g., computer programs are typically written by composing smaller programs or functions.

This research focuses on the composition of functions with *finite domains*. Cardinal examples of finite functions include the array data structure, because an array of size n can be viewed as a function that takes inputs from the finite set of indices $\{0, \ldots, n-1\}$. In practice, infinite functions are often treated as partial functions that operate on a finite set of inputs. For example, the correctness of a program function, which might be defined over an infinite domain, is usually tested only over a finite set of test inputs. Many test input generation techniques aim to create small test suites with high code coverage [Gupta and Heidepriem, 2003, Harder et al., 2003, Xie and Notkin, 2003].

Let f be a unary function with a finite domain and G be a finite set of unary functions with finite domains. The size of a function is the cardinality of the function domain, e.g., $|f| = |\mathrm{dom}(f)|$. A function *composition* from G is an ordered tuple $(g_1, \ldots, g_l)$ from G, where the g_i's are distinct, i.e., $i \neq j \Rightarrow g_i \neq g_j$. Section 4.2.3 generalizes this definition to allow repeats in the compositions and to support functions with multiple input arguments.

Definition 4.2.1 (The Function Composition (FC) problem)**.** *Given a function f and a set G of functions as defined above, does there exist a composition from G that*

produces f? That is, a composition $(g_1, \ldots, g_l)$ from G such that

$$\forall x \in dom(f). \ f(x) = g_1(\cdots(g_l(x))\ldots)? \qquad (4.1)$$

Example 4.2.2. The composition (g_1, g_4) from $G = \{g_1 = \{j \mapsto y, v \mapsto z, k \mapsto v\}, g_2 = \{v \mapsto y, y \mapsto z\}, g_3 = \{a \mapsto k, b \mapsto j\}, g_4 = \{a \mapsto j, b \mapsto v\}\}$ produces $f = \{a \mapsto y, b \mapsto z\}$. Another composition from G that produces f is (g_2, g_1, g_3).

4.2.2 Complexity of FC

Because the FC problem takes as input a set G of functions and a function f, we analyze the complexity of FC with respect to $|G|$, the size of G and in $|f|$, the size of f. We first show that FC is strongly NP-complete in $|G|$, denoted as $\text{NPC}_{|G|}$, when $|G|$ is the dominant parameter. That is, when $|G|$ is asymptotically equivalent to or larger than f, e.g., $|f|$ is polynomial in G. We then show that FC can be solved in polynomial in $|f|$, denoted as $\text{P}_{|f|}$, when $|f|$ is dominant, e.g., $|G|$ is constant or less than polynomial in $|f|$.

Theorem 4.2.3. *If $|f|$ is polynomial in $|G|$, then FC is strongly $\text{NPC}_{|G|}$.*

Proof. The proof consists of two parts that show (i) FC is in $\text{NP}_{|G|}$ and (ii) is at least strongly $\text{NP}_{|G|}$-Hard. FC is in $\text{NP}_{|G|}$ because verifying that a composition of length l from G producing f, i.e., checking the relation given in Equation (4.1), takes $O(l|f|)$, which is polynomial in $|G|$ with $|f|$ is polynomial in $|G|$ and $l \leq |G|$.

FC is strongly $\text{NP}_{|G|}$-Hard by the reduction from Exact Covering (EC), a well-known strongly NPC problem introduced by Karp [Karp, 2010] and defined as follows.

Definition 4.2.4 (The Exact Covering (EC) problem)**.** *Given a collection $S = \{S_1, \ldots, S_q\}$ of subsets of a set $X = \{x_1, \ldots, x_p\}$, does there exist a subcollection (or an exact cover) $S' \subseteq S$ such that each $x_i \in X$ occurs in exactly one subset of S'?*

Example 4.2.5. Given the sets $S = \{\{5\}, \{7\}, \{0, 7\}, \{-2, 0\}, \{-2, 5, 7\}\}$ and $X = \{-2, 0, 5, 7\}$, the subcollection $\{\{5\}, \{7\}, \{-2, 0\}\}$ from the set S is the only exact cover of the set X. The subcollection $\{\{5\}, \{7\}, \{0, 7\}, \{-2, 0\}\}$ of S is not an exact cover of X because 0 appears in both $\{0, 7\}$ and $\{-2, 0\}$.

The reduction from an arbitrary instance of EC with $S = \{S_1, \ldots, S_q\}$ and $X = \{x_1, \ldots, x_p\}$ to a specific instance of FC consists of two steps. In the following, $\perp$ denotes the value $1 + 2p$, y' the value $y + p$, and $h = [y_0, \ldots, y_k]$ the function $h = \{0 \mapsto y_0, \ldots, k \mapsto y_k\}$, i.e., an array-like function that takes as inputs the finite set of non-negative integers $\{0, \ldots, k\}$.

- Create a function $f = [\perp, 1', 2', \ldots, p']$, i.e., a function of size $p + 1$.

- For each $S_i \in S$, create a function g_{S_i} of size $2p + 1$ using the rules

$$
g_{S_i}[y \in \{0, \ldots, 2p\}] \mapsto
\begin{cases}
y', & \text{if } x_y \in S_i \\
\perp, & \text{if } x_{y-p} \in S_i \\
y, & \text{otherwise}
\end{cases}
$$

This reduction is polynomial in $|S|$ and transforms an arbitrary EC instance to a specific FC instance, which consists of array-like functions.[2] The reduction guarantees that the input EC problem has a solution if and only if the resulting FC problem has a solution.[3] Function g_{S_i} maps the input y to the special value y' if $x_y \in S_i$. Essentially, y' specifies that y has been used as a valid input previously. In the input EC problem, this means x_y has been covered by some set S_i. If y' is later used as input to a function g_{S_j} that has $y \in S_j$, then g_{s_j} maps y' to $\perp$ to indicate an

---

[2]There are other (potentially) simpler reductions from EC to FC. However, the presented reduction, which constructs array-like functions, can easily apply to the Array Nesting problem in Section 4.2.4.

[3]Note that $|S| = O(2^{|X|})$ because S contains the subsets of X. Thus, this reduction from EC shows that the FC problem is strongly `NPC` $|G|$ even when N is $\log(|G|)$.

invalid composition. In the input EC problem, the sets S_i and S_j cannot be part of an exact cover because x_y is in both of sets. □

Example 4.2.6. The EC instance from Example 4.2.5 is reduced to the following FC instance:

$$f = [\bot, 1', 2', 3', 4']$$

and

$$
\begin{aligned}
G = \{ g_{\{5\}} &= [\bot, 1, 2, 3', 4, 1', 2', \bot, 4'], \\
g_{\{7\}} &= [\bot, 1, 2', 3, 4', 1', 2', 3', \bot], \\
g_{\{0,7\}} &= [\bot, 1, 2', 3, 4', 1, \bot, 3', \bot], \\
g_{\{-2,0\}} &= [\bot, 1', 2', 3, 4, \bot, \bot, 3, 4], \\
g_{\{-2,5,7\}} &= [\bot, 1', 2, 3', 4', \bot, 2, \bot, \bot] \}.
\end{aligned}
$$

This reduced FC instance has only one composition $(g_{\{7\}}, g_{\{5\}}, g_{\{-2,0\}})$ producing f, corresponding to the only exact cover $\{\{7\}, \{5\}, \{-2, 0\}\}$ in the input EC instance.

Because FC is strongly $\mathtt{NPC}_{|G|}$, the problem remains $\mathtt{NPC}$ even when all of its numerical parameters, e.g., the values of the elements in the functions and domains, are *small* in the size of G (bounded by a polynomial of $|G|$). Readers who are interested in knowing more about the two important classes of strongly and weakly $\mathtt{NPC}$ should refer to [Wisniewski, 2006].

In general, unless $\mathtt{P} = \mathtt{NP}$, FC is not likely to have a polynomial algorithm in $|G|$. However, in practice, the sizes of the functions typically exceed the number of functions, which is usually a fixed number. The following shows that FC can be solved in polynomial time in $|f|$ when $|G|$ is constant.

Theorem 4.2.7. *If $|G|$ is constant, then FC is in $\mathtt{P}_{|f|}$*

Proof. FC can be solved by enumerating all possible compositions $(g_1, \ldots, g_l)$ from G over different lengths $l = 1, \ldots, |G|$ and checking if any of these compositions

produces f. The set G contains $\frac{|G|!}{l!(|G|-l)!}$ subsets of size l, where each subset has $l!$ compositions. Thus, G has $\frac{|G|!}{(|G|-l)!}$ compositions of size l. Moreover, checking if a composition of length l produces f, i.e., verifying the relation given in Equation (4.1), takes $O(l|f|)$. In total, this algorithm has the complexity

$$\sum_{l=1}^{|G|} \frac{|G|!}{(|G|-l)!} \times O(l|f|), \tag{4.2}$$

which is in $\mathsf{P}_{|f|}$ because l is bounded by the constant $|G|$. □

4.2.3 Generalizations of FC

Functions often take multiple input arguments and can be used more than once in various composition situations, e.g., recursive calls. We now generalize the definition of FC to allow repeats in the composition and to support functions with multiple arguments. We then show that the complexity of these generalizations are similar to those of FC, i.e., NPC in $|G|$, but can be solved in polynomial time in $|f|$.

Definition 4.2.8 (FC with bounded repeats (r-FC))**.** *Given a function f, a set G of functions as in Definition 4.2.1, and an integer $d \geq 1$, does there exist a composition from G_d that produces f, where G_d is a multiset containing d copies of each function in G?*

We note that r-FC can be treated as a version of FC by labeling the functions in G_d, e.g., the d copies of g_i are labelled as $g_i^1, \ldots, g_i^d$. Because $|G_d|$ is only a constant factor d larger than $|G|$, the analysis of FC in Section 4.2.2 also applies to r-FC.

Corollary 4.2.9. *If $|f|$ is polynomial in $|G|$, then r-FC is strongly $\mathsf{NPC}_{|G|}$*

Proof. r-FC is in $\mathsf{NP}_{|G|}$ because checking if a composition from G_d produces f takes $O(l|f|)$, and $|f|$ and the length l of the composition are both polynomial in $|G|$.

Moreover, r-FC is at least strongly $\text{NP}_{|G|}$-Hard because FC is a specific instance of r-FC with $d = 1$. $\qquad\qquad\square$

Corollary 4.2.10. *If $|G|$ is constant, then r-FC is in $\text{P}_{|f|}$*

Proof. The polynomial time algorithm to enumerate compositions for FC in Theorem 4.2.7 also applies to r-FC. The resulting complexity, analogous to Equation (4.2) with G_d replacing G, is also in $\text{P}_{|f|}$ because $|G_d| = d|G|$ is constant with both d and $|G|$ are constants. $\qquad\qquad\square$

Definition 4.2.11 (FC with functions with multiple inputs (k-FC)). *Given a function f with k input arguments, i.e., a k-ary function, and a set G of k-ary functions, where the integer $k \geq 1$, does there exist a composition from G that produces f?*

Example 4.2.12. Basic binary functions, e.g., `AND, OR, XOR`, are often used to build complex digital circuits such as multiplexers, memory controllers, and microprocessors. For example, the `XOR` $\oplus$ gate can be composed from a set of `NAND` $\uparrow$ gates as $x \oplus y = \uparrow (x \uparrow (x \uparrow y)) \uparrow (y \uparrow (x \uparrow y))$.

Corollary 4.2.13. *If $|f|$ is polynomial in $|G|$, then k-FC is strongly $\text{NPC}_{|G|}$*

Proof. k-FC is in $\text{NP}_{|G|}$ because checking if a composition from G produces f takes $O(l|f|)$, where both $|f|$ and the size $l \leq |G|$ of the composition are polynomial in $|G|$. Moreover, k-FC is $\text{NP}_{|G|}$-Hard because FC is a specific instance of k-FC with $k = 1$. $\qquad\qquad\square$

Corollary 4.2.14. *If $|G|$ is constant, then k-FC is in $\text{P}_{|f|}$*

Proof. The polynomial time algorithm to enumerate compositions for FC in Theorem 4.2.7 also applies to k-FC. However, more compositions are generated from a set of l k-ary functions than from a set of l unary functions. In fact, the enumeration of function compositions for k-FC is equivalent to the counting of trees, a well-known combinatorial problem summarized below.

Definition 4.2.15. *The tree counting problem asks for C_l^k, the number of full k-ary trees[4] that can be formed using l unlabeled nodes. This number, also known as the Fuss-Catalan number, has the closed form $C_l^k = \frac{1}{(k-1)l+1}\binom{kl}{l}$. The number of trees increases to $l!C_l^k$ when the nodes are labelled, i.e., the position of each node in the tree matters.*

Example 4.2.16. Figure 4.1 depicts the formulation of two binary trees using two unlabeled nodes. Figure 4.2 depicts the formulation of four binary trees using two labelled nodes. In these figures, $\bullet$ represents an unlabeled node, $\bullet_i$ represents a node with label i, and $-$ represents a leaf.

Figure 4.1: The formulation of binary trees using two unlabelled nodes
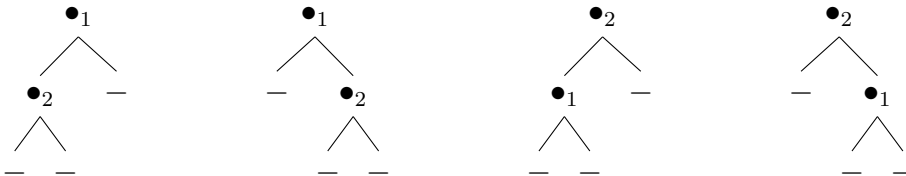
Figure 4.2: The formulation of binary trees using two labelled nodes

By modeling functions as labelled tree nodes, the enumeration of compositions from l k-ary functions is equivalent to the counting of k-trees with l labelled nodes. Thus, the enumeration algorithm given in Theorem 4.2.7 produces $l!C_l^k$ compositions of size l from G. In total, the complexity of solving the k-FC problem is

$$\sum_{l=1}^{|G|} \frac{|G|!C_l^k}{(|G|-l)!} \times O(l|f|), \tag{4.3}$$

---

[4]A full k-ary tree is a rooted tree in which each node has exact k children

which is also in $\mathsf{P}_{|f|}$ because l is bounded by the constant $|G|$. $\qquad\qquad\square$

4.2.4 The Array Nesting (AN) problem

As mentioned, arrays can be viewed as a specific class of functions that operates over a finite, non-negative integral domain $\{0, \ldots, n-1\}$ representing array indices. Due to this relation, the analysis for functions in Section 4.2.1 also applies to the following problem of finding nested relations among arrays, a form of relation that appears in many applications.

Let a be a 1-dimensional array and B be finite set of 1-dimensional arrays. The size of an array is the cardinality of its domain or its set of indices, e.g., $\mathrm{dom}(a) = \{0, \ldots, |a|\}$. An array *nesting* from B is an ordered tuple $(b_1, \ldots, b_l)$ of B, where the b_i's are distinct, i.e., $i \neq j \Rightarrow b_i \neq b_j$. Section 4.2.6 generalizes this definition to allow repeats in the nestings and to support multidimensional arrays.

Definition 4.2.17 (The Array Nesting (AN) problem). *Given an array a and a set B of arrays as defined above, does there exist a nesting from B that produces a? That is, a nesting $(b_1, \ldots, b_l)$ from B such that*

$$\forall i \in dom(a).\ a[i] = b_1[\ldots b_l[i] \ldots]\ ? \tag{4.4}$$

Definition 4.2.18 (The Array Nesting 2 (AN$_2$) problem). *Given an array a, a set B of arrays as defined above, and the reals z', z'', w', w'', does there exist a nesting $(b_1, \ldots, b_l)$ from B and two reals (z, w), where $z' \leq z \leq z''$ and $w' \leq w \leq w''$, such that*

$$\forall i \in dom(a).\ a[i] = b_1[\ldots b_l[z + wi] \ldots]\ ? \tag{4.5}$$

Definition 4.2.19 (The Array Nesting 3 (AN$_3$) problem). *Given an array a, a set B of arrays as defined above, and the reals $x', x'', y', y'', z', z'', w', w''$, does there exist a nesting $(b_1, \ldots, b_l)$ from B and the reals $x_1, y_1, \ldots, x_l, y_l, z, w$, where $x' \leq x_i \leq x''$,*

$y' \leq y_i \leq y''$, $z' \leq z \leq z''$ *and* $w' \leq w \leq w''$, *such that*

$$\forall i \in dom(a).\ a[i] = x_1 + y_1 b_1[\ldots [x_l + y_l b_l[z + wi]]\ldots]\ ? \tag{4.6}$$

The reals c'_i, c''_i in the definitions of AN_2 and AN_3 specify the ranges of the coefficients c_i. These explicit ranges allow for the reduction of one problem to another. Specificaly, AN is an instance of AN_2 with $z' = z'' = 0, w' = w'' = 1$ and AN_2 is an instance of AN_3 with $x' = x'' = 0, y' = y'' = 1$. Hence, in term of complexity, $\text{AN} \subseteq \text{AN}_2 \subseteq \text{AN}_3$.

Example 4.2.20. AN finds nested array relations such as $a[i] = b_1[b_2[i]]$, AN_2 finds relations such as $a[i] = b_1[b_2[2i + 3]]$, and AN_3 finds relations such as $a[i] = -2b_1[4 + \frac{1}{5}b_2[2i + 3]]$.

Note that the FC problem in Definition 4.2.1 generalizes AN because array is a special class of function. However, AN_2 and AN_3, the generalizations of AN, are not related to FC, i.e., neither AN_2 nor AN_3 reduce to FC, and vice versa.

4.2.5 Complexity of the AN's

Similar to the study of FC in Section 4.2.2, we analyze the complexity of the AN problems in $|B|$, the size of the set B of arrays and $|a|$, the size of array a. As will be shown, the AN problems are strongly $\text{NPC}_{|B|}$ when $|B|$ is the dominant factor and in $\text{P}_{|a|}$ when $|a|$ is the dominant parameter.

Theorem 4.2.21. *If $|a|$ is polynomial in $|B|$, then AN is strongly* $\text{NPC}_{|B|}$

Proof. This proof is similar to the proof of Theorem 4.2.3 showing that FC is NPC. First, AN is in $\text{NP}_{|B|}$ because it is an instance of FC. Next, AN is strongly NP-Hard by the same reduction from the Exact Cover problem, with arrays a and $b_i \in B$ represented by functions f and $g_i \in G$, respectively. $\qquad \square$

Corollary 4.2.22. *If $|a|$ is polynomial in $|B|$, then both AN_2 and AN_3 are strongly* $\mathtt{NPC}_{|B|}$

Proof. AN_2 is in $\mathtt{NP}_{|B|}$ because checking the relation given in Equation (4.2.18) from a nesting $\{b_1, \cdots, b_l\}$ and a pair of reals (z, w) is polynomial in $|B|$. Similarly, AN_3 is in $\mathtt{NP}_{|B|}$ because checking the relation given in Equation (4.2.19) from $\{b_1, \cdots, b_l\}$ and $(x_1, y_1, \ldots, x_l, y_l, z, w)$ is in $\mathtt{P}_{|B|}$. AN_2 and AN_3 are both at least $\mathtt{NP}$-hard because AN can be reduced to AN_2 with $(z' = z'' = 0, w' = w'' = 1)$ and to AN_3 with $(x' = x'' = 0, y' = y'' = 1, z' = z'' = 0, w' = w'' = 1)$. $\qquad\square$

In real-world applications, the number of array elements often exceeds the number of arrays involved, which is usually a fixed number. The following proves that AN_3 is in $\mathtt{P}_{|a|}$ when the number of array elements is the dominant parameter. This implies that AN_2 and AN, which are specific instances of AN_3, are also in $\mathtt{P}|a|$. The proof of AN_3 in $\mathtt{P}_{|a|}$ relies on the solutions to the following auxiliary problems.

Definition 4.2.23 (The Array Nesting 3a (AN_{3a}) problem). *Given an array nesting $(b_1, \ldots, b_l)$ from B and the inputs $a, B, x', x'', y', y'', z', z'', w', w''$ from AN_3, does there exist the reals $x_1, y_1, \ldots, x_l, y_l, z, w$, where $x' \le x_i \le x''$, $y' \le y_i \le y''$, $z' \le z \le z''$ and $w' \le w \le w''$, such that*

$$\forall i \in dom(a).\ a[i] = x_1 + y_1 b_1[\ldots[x_l + y_l b_l[z + wi]]\ldots] \ ? \qquad (4.7)$$

Definition 4.2.24 (The Array Nesting 3b (AN_{3b}) problem). *Given a positive integer l and the inputs $a, B, x', x'', y', y'', z', z'', w', w''$ from AN_3, does there exist a nesting $(b_1, \ldots, b_l)$ from B and the reals $x_1, y_1, \ldots, x_l, y_l, z, w$, where $x' \le x_i \le x''$, $y' \le y_i \le y''$, $z' \le z \le z''$ and $w' \le w \le w''$, such that*

$$\forall i \in dom(a).\ a[i] = x_1 + y_1 b_1[\ldots[x_l + y_l b_l[z + wi]]\ldots] \ ? \qquad (4.8)$$

AN_{3b} is more explicit than AN_3 because it requires that the size l of the nesting be provided as an input. AN_{3a} is even more explicit because its input includes a

specific nesting $(b_1, \ldots, b_l)$. As with AN_3, neither AN_{3a} nor AN_{3b} are related to the FC problem, i.e., they do not reduce to FC, and vice versa.

We now present an algorithm to solve AN_{3a} in polynomial time in the size of array a. This algorithm is subsequently used to show that AN_{3b} and AN_3 are in $\mathsf{P}|a|$.

Lemma 4.2.25. *If $|B|$ is constant and $|b_i|$ is polynomial in $|a|$, then AN_{3a} is in $\mathsf{P}_{|a|}$.*

Proof. The algorithm consists of two steps: (i) generating nesting relations of the form given in Equation (4.7) for an arbitrarily chosen pair of elements $(a[i], a[j])$ and then (ii) checking if any of these relations also holds for other elements of a.

- *Generating Nesting Relations.* We arbitrarily choose *two* distinct elements $a[i] \neq a[j]$ in array a and find all pairs of indices in b_1 whose elements can linearly express $(a[i], a[j])$. That is, for each pair (s_1, t_1) of indices in b_1, we construct a set of two equations $\{a[i] = x_1 + y_1 b_1[s_1], a[j] = x_1 + y_1 b_1[t_1]\}$ and then solve it for (x_1, y_1), subjecting to the constraints[5] about the ranges of x_1, y_1 from the input problem. The answer to the input AN_{3a} problem is no if no solution is obtained for (x_1, y_1). Otherwise, a solution $(x_1 = u_1, y_2 = v_1)$ means that $(a[i], a[j])$ can *reach* b_1 through the elements $(b_1[s_1], b_1[t_1])$ with the real-valued coefficients $(x_1 = u_1, y_2 = v_1)$. This *reachability analysis* gives the nesting relation $\{a[i] = u_1 + v_1 b_1[s_1], a[j] = u_1 + v_1 b_1[t_1]\}$.

  For each nesting relation $\{a[i] = u_1 + v_1 b_1[s_1], a[j] = u_1 + v b_1[t_1]\}$ obtained at b_1, we again apply reachability analysis to check whether (s_1, t_1) can reach b_2. If (s_1, t_1) reaches b_2 through $(b_2[s_2], b_2[t_2])$ with $(x_2 = u_2, y_2 = v_2)$, then $(a[i], a[j])$ reaches $(b_2[s_2], b_2[t_2])$ by the relations $\{a[i] = u_1 + v_1 b_1[u_2 + v_2 b_2[s_2]], a[j] = u_1 + v_1 b_1[u_2 + v_2 b_2[t_2]]\}$. Repeating the analysis for all b_i's results in a set

---

[5]Essentially, this is a linear programming problem and can be solved using methods such as Simplex or Fourier-Motzkin in constant time because the number of constraints and unknowns coefficients are both constants.

of relations of the form $\{a[i] = u_1 + v_1 b_1[\ldots[u_l + v_l b_l[s_l]]\ldots], a[j] = u_1 + v_1 b_1[\ldots[u_l + v_l b_l[t_l]]\ldots]\}$ at b_l. Finally, for each obtained relation at b_l, we construct a set of two equations $\{s_l = z + wi, t_l = z + wj\}$ and solves it for the coefficients (z, w).

This step has polynomial time complexity in $|a|$. At b_1, we create at most $|b_1|^2$ or $O(|b_1|^2)$ relations. Each relation obtained from b_1 produces $O(|b_2|^2)$ relations at b_2. Thus, the number of relations is $O(|b_1|^2|b_2|^2)$ at b_2, and more generally, is $O(|b_1|^2 \cdots |b_l|^2)$ at b_l. Because $|b_i|$ is polynomial in $|a|$, this step produces $O(|a|^{2l})$ relations, which is polynomial in $\mathsf{P}_{|a|}$ since l is bounded by the constant $|B|$.

- *Verifying Nesting Relations.* This step checks if any of the obtained relations, which are guaranteed to hold for the initially chosen pair of elements $(a[i], a[j])$, holds for *all* elements of a. Verifying a relation of the form given in Equation (4.7) takes $O(l|a|)$. Thus, the verification of $O(|a|^{2l})$ such relations is $O(|a|^{2l}l|a|)$, which is in $\mathsf{P}_{|a|}$ because l is bounded by the constant $|B|$.

When *all* elements of a are the same (or a has only one element), the algorithm only needs to find a nesting relation that holds for the first element $a[i = 0]$. Thus, the above analysis is performed to obtain all nesting relations of the form $a[0] = u_1 + v_1 b_1[\ldots[u_2 + v_l b_l[x]]\ldots]$. Note that w can be anything within its range constraint because $i = 0$. The verification step is also not necessary because the nesting relation that holds for $a[0]$ also holds for other elements $a[i]$ because all elements of a are the same. The complexity of this algorithm is $O(|a|^l)$, which is in $\mathsf{P}_{|a|}$ using similar analysis as above.

□

A solution of the form given in Equation (4.7) that holds for all elements of A must also hold for the initially chosen pair of elements $(A[i], A[j])$. Thus, we can find such

a solution, if it exists, by trying all possible sets of relations generated by reachability analysis on the two elements $A[i]$, $A[j]$. Moreover, it is sufficient to apply the analysis on *two* distinct, instead of all, elements of the 1-dimensional A because only two independent equations are needed to solve for the pair of unknown coefficients as shown above. In general, we apply reachability analysis on tuples of $k + 1$ elements of a k-dimensional array a (see the k-AN$_3$ problem in Section 4.2.6) because the nested relation now involves a linear expression consisting of $k + 1$ unknowns. The analysis also has a polynomial time complexity when k is a fixed number.

We now apply this reachability analysis to solve the AN$_{3b}$ and AN$_3$ problems. In Section 4.3, we implement these solutions to discover nested array relations dynamically from traces.

Lemma 4.2.26. *If $|B|$ is constant and $|b_i|$ is polynomial in $|a|$, then AN_{3b} is in $\mathsf{P}_{|a|}$*

Proof. The AN$_{3b}$ problem can be solved by applying the reachability algorithm for AN$_{3a}$ given in Lemma 4.2.25 on each nesting from the $\frac{|B|!}{(|B|-l)!}$ nestings of size l generated from the set $|B|$. This takes

$$O(\frac{|B|!}{(|B| - l)!}|a|^{2l}l|a|),$$

which is in $\mathsf{P}_{|a|}$ because l is bounded by the constant $|B|$. $\qquad\square$

Theorem 4.2.27. *If $|B|$ is constant and $|b_i|$ is polynomial in $|a|$, then AN_3 is in $\mathsf{P}_{|a|}$*

Proof. AN$_3$ can be solved by applying the algorithm for AN$_{3b}$ given in Lemma 4.2.26 over the lengths $l = 1, \ldots, |B|$. This takes

$$\sum_{l=1}^{|B|} O(\frac{|B|!}{(|B| - l)!}|a|^{2l}l|a|),$$

which is polynomial in $|a|$ because l is bounded by the constant $|B|$. $\qquad\square$

Corollary 4.2.28. *If $|B|$ is constant and $|b_i|$ is polynomial[6] in $|a|$, then AN and*
AN_2 are in $\mathtt{P}_{|a|}$

Proof. AN and AN_2 are specific versions of AN_3 and, thus, are also in $\mathtt{P}_{|a|}$ by Theorem 4.2.27. □

4.2.6   Generalizations of AN

Similar to FC, AN's can be extended to allow a bounded number of repeats in the nesting and to support multidimensional arrays. The following introduce two extensions of AN_3, which is the most general version of the AN problems.

Definition 4.2.29 (AN_3 with repeats (r-AN_3)). *Given an array a, a set B of arrays*
as in Definition 4.2.19, and an integer $d \geq 1$, does there exist an array nesting of a
from B_d, where B_d is a multiset containing d copies of each array in G?

Corollary 4.2.30. *r-AN_3 is strongly $\mathtt{NPC}_{|B|}$ and in $\mathtt{P}_{|a|}$*

Proof. r-AN_3 is a version of AN_3 with the set B increased by a constant factor d, i.e., by giving distinct labels to the d copies of each array g_i. Thus, the proof for AN_3 being strongly $\mathtt{NPC}_{|B|}$ is similar to the one given in Corollary 4.2.9 showing the strongly $\mathtt{NP}$-Completeness of r-FC. The polynomial time argument for r-AN_3 is also similar to the one given in Corollary 4.2.10 showing r-FC is in $\mathtt{P}$. □

Definition 4.2.31 (AN_3 with k-dimensional arrays (k-AN_3)). *Given a k-dimensional*
array a and a set B of k-dimensional arrays, where k is a constant ≥ 1, does there
exist a nesting from B that produces a?

---

[6]The requirement of b_i being polynomial in $|a|$ is not necessary to show AN is in $\mathtt{P}_{|a|}$ because it is an instance of FC. However, this constraint is used in the polynomial time algorithm for AN_3 in Lemma 4.2.25.

Similar to functions with multiple inputs in Section 4.2.14, a k-dimensional array can be modeled as a k-ary trees, where each array index is represented by a node in the tree. Moreover, because a is a k-dimensional array, the non-nested array indices in the nesting, i.e., the leaf nodes in the tree, are now a linear combination of the k indices of a, e.g., $a[i][j][k] = b[c[i - j + k + 2]][i + k]$.

Corollary 4.2.32. *If $|a|$ is polynomial in $|B|$, then k-AN_3 is strongly* $\mathtt{NPC}_{|B|}$.

Proof. k-AN_3 is at least as hard as AN_3, which is strongly $\mathtt{NPC}_{|B|}$. Moreover, k-AN_3 is in $\mathtt{NP}_{|B|}$ because checking a nesting of a from B is $O(l|a|)$, which is polynomial in $|B|$ for $l \leq |B|$. $\square$

Corollary 4.2.33. *If $|B|$ is constant and $|b_i|$ is polynomial in $|a|$, then k-AN_3 is in* $\mathtt{P}_{|a|}$

Proof. The enumeration of k-dimensional array nestings is equivalent to the counting of k-ary trees as given in Corollary 4.2.14. Hence, we enumerate $l!C_l^k$ nestings for each set of size $l = 1, \ldots, |B|$ using the reachability algorithm given in Theorem 4.2.27. As mentioned in the proof of Lemma 4.2.25, the algorithm now applies reachability analysis on a tuple of $k+1$ elements of the k-dimensional array. Finally, the algorithm checks if any of the $\sum_{l=1}^{|B|} \frac{|B|!}{(|B|-l)!} C_l^k$ enumerated nestings satisfies the array $|a|$. Lemma 4.2.25 shows the verification of a given nesting of size l is $O(|a|^{2l}l|a|)$, thus the complexity of this algorithm is

$$\sum_{l=1}^{|B|} O(\frac{|B|!}{(|B| - l)!} C_l^k |a|^{2l} l|a|),$$

which is in $\mathtt{P}_{|a|}$ because l is bounded by the constant $|B|$. $\square$

4.3 Inferring Nested Array Invariants Dynamically

DIG implements the algorithms developed in Section 4.2 to generate nested relations among multidimensional arrays dynamically from program traces. Specifically, we combine SMT solving with the reachability algorithm presented in Section 4.2.5 to solve the k-AN$_3$ problem in Definition 4.2.31. To achieve an efficient and practical implementation, we consider only unit coefficients in the nesting. For instance, if the inputs to k-AN$_3$ are a 2-dimensional array a and a set B of 1-dimensional arrays, then DIG finds a nesting from B that produces a, i.e.,

$$\forall i \in \text{dom}(a).\ a[i_1][i_2] = b_1[\dots [b_l[z + w_1 + w_2] \dots]. \tag{4.9}$$

This form covers simpler relations than those supported in k-AN$_3$. For instance, it does not consider the relation $A[i] = 8 - 9B[5 + 2C[2i - j]]$ because it has non-unit coefficients. However, this form covers the nested array relations that are in real-world applications such as AES, and it enables a more straightforward reachability algorithm as shown below.

DIG takes as input the set V of (possibly multidimensional) array variables that are in scope at location L and the associated traces X, and it returns a set of possible relations among the arrays in V. Figure 4.3 outlines the three steps that generate nested array relations. The first step (GENNESTINGS) enumerates nested array structures, such as $A = B[C[\dots]], B = A[C[\dots]], \dots$. The next step (REACHANALYSIS) applies reachability analysis to identify relations among individual array elements using each enumerated nesting, such as $A[0] = B[C[1]], A[1] = B[C[2]], A[2] = B[C[3]]$. The last step analyzes this information for potential nested array relations, e.g., $A[i] = B[C[i+1]]$, by encoding the problem as a satisfiability query that can be handled using an SMT solver (GENFORMULA and SMT). Essentially, this algorithm uses the ideas given in Theorem 4.2.27, but encodes the results of reachability analysis as a satisfiability problem.

procedure FINDNESTEDARRAYS(V, X)
 $S \leftarrow \emptyset$
 $nestings \leftarrow$ GENNESTINGS(V)
 for $nesting \in nestings$ **do**
  $R \leftarrow$ REACHANALYSIS($nesting$,X)
  if $R \neq \emptyset$ **then**
   $f \leftarrow$ GENFORMULA(R)
   $s \leftarrow$ SMT(f)
   if $s \neq \emptyset$ **then**
    $S \leftarrow S + \{s\}$
 return S $\triangleright$ return array relations of the form given in Equation (4.9)

Figure 4.3: Algorithm for finding nested array relations from inputs: the set V of array variables and the associated traces X. The algorithm consists of three steps: enumerating nested structures among the input array variables (GENNESTINGS), applying reachability analysis on each enumerated nesting to find relations among individual array elements (REACHANALYSIS), encoding the obtained relations as a satisfiability query that can be checked using an SMT solver (GENFORMULA and SMT).

For simplicity, we illustrate this method using three 1-dimensional arrays, i.e., $V = \{A, B, C\}$, although the implementation of DIG generalizes the method to multidimensional arrays.

- *Nestings.* We first enumerate the nested structures among the arrays in V. A nested array structure, or *nesting*, from a set V of arrays is a tuple (a, B) where a is an array in V and B is a non-empty and non-repeating sequence of arrays in V that does not contain the array P. For the input $V = \{A, B, C\}$, we generate the nestings $(A, [B]), (A, [C]), \dots, (C, [B, A])$.

  Intuitively, each nesting represents an input to the r-AN$_3$ problem. Moreover, the enumeration step resembles the counting of trees shown in Corollary 4.2.33, where a nesting $(a, [\dots])$ is a tree with root node a.

- *Reachability Analysis.* A nesting $(A, [B, C])$ implies the relation $A[i] = B[C[k]]$ where elements of the array A are related to elements of B using elements of C as indices into B. For such a relation to hold, the elements of A must be in

B. Moreover, the indices of B, where the elements of A appear in, must also be in C. We use the reachability analysis in Lemma 4.2.25 to determine how the elements of A are related to the elements of B using C as indices into B.

We arbitrarily choose two distinct elements $A[x] \neq A[y]$ from array A. For $A[x]$, we find the indices j_x in B where $B[j_x] = A[x]$. For each of the obtained indices j_x in B, we again find the indices k_x in C where $C[k_x] = j_x$. We then form a set of relations of the form $A[x] = B[C[k_x]]$ from these results, which indicate that the element $A[x]$ is related to elements of B using elements $C[k_x]$ as indices into B. Repeating this process for $A[y]$, we obtain a set of relations of the form $A[y] = B[C[k_y]]$. Each set R from the cross product of the two sets of relations consists of two equations of the form $\{A[x] = B[C[k_x]], A[y] = B[C[k_y]]\}$.

If any of the above checks fails, e.g., $A[x]$ is not in B or the obtained indices j_x of B are in C, then relation $A[i] = B[C[k]]$ is invalid and disregarded. We can further optimize this step by starting with the two distinct elements of A that occur least often in B. However, such a greedy approach does not guarantee the smallest number of relation sets generated at the end because the indices j_x of B can occur many times in C.

- *Relations Among Array Indices.* From a set $R = \{A[x] = B[C[k_x]], A[y] = B[C[k_y]]\}$ of relations obtained from reachability analysis, we determine the relation between the indices of A and C, which is represented by the parameterized linear expression $k = ip + q$. Instantiating $k = pi + q$ with the information from R, we get a set of two equations $\{k_x = xq + q, k_y = yp + q\}$. The solution for p, q of these equations gives a relation of the form $A[i] = B[C[pi + q]]$ for $i = \{x, y\}$, i.e., a conditional relation that is further discussed in Section 4.4. We now verify that this relation also holds for other indices i of A, instead of just x, y. If it is verified, we return it as the candidate invariant. Otherwise, we repeat this step on another set R of relations to find a different nested array
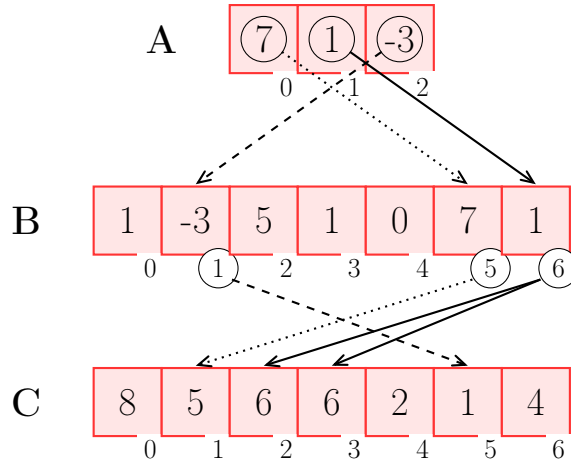
Figure 4.4: Reachability analysis showing $A[0] = B[C[1]]$ (dotted), $A[1] = B[C[2]] \vee B[C[3]]$ (solid), and $A[2] = B[C[5]]$ (dashed).

relation.

Example 4.3.1. We demonstrate how DIG finds the relation $A[i] = B[C[2i + 1]]$ from the trace $A = [7, 1, -3]$, $B = [1, -3, 5, 1, 0, 7, 1]$, and $C = [8, 5, 6, 6, 2, 1, 4]$. Figure 4.4 illustrates reachability analysis on the three elements of array A over the nesting $(A, [B, C])$.

Among the nestings generated from the input $V = \{A, B, C\}$, those representing relations such as $B[i] = C[\ldots]$ are ruled out immediately because the element -3 of B is not in C. Note that the use of traces is essential here as it allows us to quickly filter out invalid nestings. For the nesting $(A, [B, C])$, we apply reachability analysis on two arbitrarily chosen elements $A[1]$ and $A[2]$ of A. For $A[1]$, the analysis generates $\{A[1] = B[C[2]], A[1] = B[C[3]]\}$ because $A[1] = B[0], B[3], B[6]$ and $6 = C[2], C[3]$ (the index values $0, 3$ of B do not occur in C). For $A[2]$, we obtain the set $\{A[2] = B[C[5]]\}$ because $A[2] = B[1]$ and $1 = C[5]$. The cross product of these two sets yields the sets $R_1 = \{A[1] = B[C[2]], A[2] = B[C[5]]\}$ and $R_2 = \{A[1] = B[C[3]], A[2] = B[C[5]]\}$ of relations.

The information from either the set R_1 or R_2 suggests the possibility of a nested

relation $A[i] = B[C[k]]$ for $i = \{1, 2\}$ and k is the parameterized linear expression $k = pi + q$. Instantiating $k = pi + q$ with the information from R_1 gives two equations $\{2 = p + q, 5 = 2p + q\}$. The unique solution $\{p = 2, q = -1\}$ for these equations yields the relation $A[i] = B[C[3i - 1]]$ where $i = \{1, 2\}$. This relation does not hold for all indices of A, e.g., $A[0] \neq B[C[-1]]$, and is thus disregarded. Next, we instantiate $k = pi + q$ with the information from R_2 and obtain the equations $\{3 = p + q, 5 = 2p + q\}$. The unique solution $\{p = 2, q = 1\}$ for these yields the relation $A[i] = B[C[2i + 1]]$, for $i = \{1, 2\}$. This relation holds for all indices of A, and is returned as the candidate invariant.

As shown in Lemma 4.2.25, the algorithm finds a relation of the form $A[i] = B[C[k]]$, if it exists, by trying all possible sets R of relations generated by reachability analysis on the two elements $A[x], A[y]$. The technique is generalized to multidimensional arrays by applying reachability analysis on a tuple of $k + 1$ elements of a k-dimensional array A because the relation among the indices of A and C is represented by a linear expression consisting of $k + 1$ unknowns $z, w_1, \ldots, w_k$.

4.3.1 Satisfiability Problem Formulation

In practice, arrays often have large sizes with multiple duplicate elements, causing reachability analysis to generate many sets R of relations to be solved for. We can accommodate this issue by encoding the results of reachability analysis as a satisfiability formula in the theory of linear integer arithmetic, which can be solved efficiently with modern SMT technologies [Dutertre and De Moura, 2006].

Example 4.3.2. Returning to the running example, DIG creates a clause consisting of two atoms $(2 = p + q \ \lor \ 3 = p + q)$ to represent the result $\{A[1] = B[C[2]], A[1] = B[C[3]]\}$ from reachability analysis. Similarly, the atom $5 = 2p + q$ is created for $\{A[2] = B[C[5]]\}$. Since the relation should hold for the two chosen elements of A, i.e., $A[i] = B[C[pi + q]]$ for $i = \{1, 2\}$, DIG combines these formulae into the final

CNF formula $f = (2 = p + q \lor 3 = p + q) \land (5 = 2p + q)$. Next, DIG queries the SMT solver to return, if possible, an assignment of integers (since array indices are integers) to the variables p and q that satisfies f. In this example, the solver might assign $p = 3, q = -1$ for f, which implies the relation $A[i] = B[C[3i - 1]]$ for $i = \{1, 2\}$. This relation cannot be verified because it does not hold for all indices of A, e.g., $A[0] \neq B[C[-1]]$. DIG then adds the constraint $\neg(p = 3 \land q = -1)$ to f and queries the SMT solver for a new assignment for p, q. The solver now assigns $p = 2, q = 1$, implying the relation $A[i] = B[C[2i + 1]]$. This relation is verified to hold for all indices of A and thus is returned as the candidate invariant.

We avoid having to verify each relation by applying the analysis on all elements of A. Doing so for the running example results in the CNF $f = (1 = q) \land (2 = p + q \lor 3 = p + q) \land (5 = 2p + q)$, where the atom $1 = q$ represents the relation $A[0] = B[C[1]]$ as illustrated in Fig. 4.4. The solution $\{p = 2, q = 1\}$, returned by the SMT solver on the formula f, implies the similar relation $A[i] = B[C[2i + 1]]$ as above. This relation is valid for all elements of A because the analysis is applied on all of those elements. Thus, DIG invokes the solver only once, but over a more complex formula f (the number of clauses in f is the size of A).

The problem of finding nested array relations has polynomial time complexity by the analyses and algorithms given in Section 4.2.2. However, the implementation of DIG for nested array relations involves SMT technologies and, hence, does not guarantee polynomial run time.[7] The experimental results described in Section 4.5 were obtained by applying reachability analysis on all elements of A.

---

[7]This depends on the technique implemented in SMT solvers for satisfiability checking over CNFs of the discussed form.

4.3.2 Functions

Nested relations involving both arrays and functions, e.g., $A[i] = f(C[i], g(D[i]))$, require special treatment. We view a function f with n arguments as an n-dimensional array F, where the element $F[i_1] \ldots [i_n]$ contains the output of $f(i_1, \ldots, i_n)$. For example, if f is the `mult` function, then $F[4][7] = F[7][4] = 28$. For efficiency, F is represented in DIG as a *partial* array that stores only observed values. For example, if $A = [4, 7]$ and $B = [5]$ are considered, then F contains just the elements $F[4][4], F[4][5], F[4][7], \ldots, F[7][7]$. This abstraction of infinite functions to finite arrays extends to function composition such as $g(f(A[\ldots], B[\ldots]))$. For instance, if g is mod_2 which maps even and odd inputs to 0 and 1 respectively, then the corresponding array G has as its indices the elements of A, B, F, e.g., $G[4] = G[28] = 0, G[5] = G[7] = 1$.

DIG pre-defines a set of basic functions, e.g., `mult,add,xor,mod`, and automatically generates the corresponding partial arrays based on given traces as described above. By treating functions as partial arrays, DIG generates nested array invariants involving functions, such as the relation $R[i] = T(\text{mod255}(\text{add}(L(A[i]), L(B[i]))))$ in the `multWord` function in AES.

4.4 Inferring Flat Array Invariants Dynamically

Another popular form of array invariant involves non-nested, linear expressions ranging over arrays such as $A[i] = B[i] + C[i] + 5$. These *flat* array relations can be viewed as a special case of nested relations by using a partial array to represent addition as shown in Section 4.3.2. For example, the flat relation $A[i] = B[i] + C[i] + 5$ is the nested relation $A[i] = \text{add}[[\text{add}[B[i]][C[i]]]][5]$, where `add` is a 2-dimensional array. Thus, we can generalize the reachability analysis given in Section 4.3 to find flat array relations. However, the non-nested, linear expression form of these relations

allows for a more efficient algorithm than reachability analysis. DIG implements an algorithm that uses only standard equation solving to infer flat array relations dynamically.

DIG finds flat relations among array elements of the form

$$A = b_1 B_1 + \cdots + b_n B_n + c, \tag{4.10}$$

where A, B_i are distinct (possibly multidimensional) arrays whose elements are real-valued. The array A, called the *pivot array*, is privileged in our approach because the indices of arrays B_i and the coefficients b_i, c are hypothesized as linear expressions ranging over the indices of A. The invariant $A[i][j] = B[iN + j]$ (N is a constant) shown at the end of Section 3.2.3 is an example of flat array relation. DIG also supports more complex relations of this form, e.g., $A[i][j] = \frac{1}{2}jB[2i + j] - (j + 1)C[7i][3] + 5$.

The algorithm for finding flat array relations, outlined in Figure 4.5 consists of two main parts: (i) identifying groups of relations among individual array elements such as $\{A[1] = B[0] + 2, A[4] = 3B[7] - 4\}$ and $\{C[0] = D[1], C[1] = D[2], C[2] = D[3], \ldots\}$ and (ii) analyzing this information for potential flat array relations like $C[i] = D[i + 1]$ in the second group.

For simplicity, we demonstrate this method using two 1-dimensional arrays, i.e., $V = \{A, B\}$, although the implementation of DIG generalizes the method to multi-dimensional arrays.

- *Relations Among Array Elements.* We first generate a set V' of new variables representing elements of the arrays in V. Next, the technique from Section 3.2.1 is used to identify linear equalities of the form given in Equation (3.1) over the variables in V' from the input traces X. The obtained equations represent relations among array elements, e.g., $A_4 = 3B_7 - 4$ where the variables A_4, B_7,

procedure FINDFLATARRAYS(V, X)
    ▷ obtain linear relations among array elements
    $V' \leftarrow$ GENNEWVARS(V)
    $eqts \leftarrow$ FINDEQS(V',X,$d = 1$) ▷ find eqts using the algorithm given in Algorithm 3.3
    $Rs \leftarrow$ GROUP($eqts$)
    if $Rs \neq \emptyset$ **then**
        for $R \in Rs$ **do**
            $pivot \leftarrow$ GENPIVOT(R)
            $exps \leftarrow$ GENLINEXPS($pivot$)   ▷ generate linear exps over the indices of *pivot*
            $s \leftarrow$ SOLVE($exps, R$)
            $S \leftarrow S + \{s\}$
    return S ▷ return array relations of the form given in Equation (4.10)

Figure 4.5: Algorithm for finding flat array relations from the inputs: set V of numerical array variables and the associated traces X. The algorithm has of two main parts (i) identifying groups of relations among individual array elements and (ii) analyzing this information for potential flat array relations in the obtained groups. The first part consists of the steps: creating new variables to represent array elements (GENNEWVARS), finding equality relations among these array elements (FINDEQS), and grouping the obtained relations (GROUP). The second part consists of the steps: representing the relations among the indices of a selected pivot array and other arrays as a parameterized linear expression (GENPIVOT), instantiating this expression with information from the obtained group of equalities (GENLINEXP), and solving these equations (SOLVE).

represent the array elements $A[4], B[7]$, respectively. Currently, we do not find relations among similar arrays, e.g., $A[i] = A[2i]$, and thus keep only equations that express relations among array elements of different arrays. These relations are then grouped so that each group contains relations among elements from a same set of arrays. For example, $\{A_1 = B_0 + 2, A_4 = 3B_7 - 4\}$ and $\{C_0 = D_1, C_1 = D_2, C_2 = D_3\}$ are two different groups.

- *Relations Among Array Indices.* From each obtained group, we consider only the set R of relations of the form:

$$
\begin{aligned}
A_{i_0} &= b_0 B_{j_0} + c_0, \\
A_{i_1} &= b_1 B_{j_1} + c_1, \\
&\vdots
\end{aligned}
$$

where b_x, c_x are real-valued and A_{i_x}, B_{j_x} are the variables in V' representing $A[i_x], B[j_x]$, respectively.

In such a set R, we select A as the pivot array and hypothesize that the coefficients b_x, c_x and the indices j_x of array B are linear expressions ranging over the indices i_x of A. For instance, we represent the relation between j_x and i_x through the parameterized linear expression $j_x = p_1 i_x + q_1$, where p_1 and q_1 are unknowns to be solved for. This expression is then instantiated with the information from R to obtain a set of equations $\{j_0 = p_1 i_0 + q_1, j_1 = p_1 i_1 + q_1, \dots\}$. Any solution for p and q of these equations implies a relation of the form $A[i_x] = (p_0 i_x + q_0)B[p_1 i_x + q_1] + (p_2 i_x + q_2)$, where i_x are the indices of A obtained from R.

Similar to the algorithm given in Section 4.3 for nested arrays, this algorithm yields flat relations under the disjunctive or *conditional* form $i \in \{\dots\} \Rightarrow r$, i.e., the relation r holds only for specific indices i. Such invariants are useful and appear in many programs, e.g., in the following code fragment

```
for (i=0; i < M; ++i){
    if (i < 6)
        A[i] = [B[4*i],  B[4*i+1],
                B[4*i+2],B[4*i+3]];
}
[L]
```

For this code fragment, DIG generates the invariant $A[i][j] = B[4i + j]$ for $i = \{0, \dots, 5\}$ and $j = \{0, \dots, 3\}$, indicating a relation among certain elements of the arrays A and B at location L.

Example 4.4.1. The following illustrates how DIG finds the relation $A[i] = 7B[2i] + 3i$ between two arrays A, B, using traces X that exhibit that relation. An example trace in X contains the values $A = [-546, -641, 34]$ and $B = [-78, 3, -92, -34, 4]$.

Eight variables are created to represent the elements of A and B. Based on the given trace, the set $R = \{A_0 = 7B_0, A_1 = 7B_2 + 3, A_2 = 7B_4 + 6\}$ of linear equations is obtained using the technique in Section 3.2.1. From R, DIG chooses A as the pivot and extracts the information $i_x = \{0, 1, 2\}$. The relation between j_x and i_x is expressed as $j_x = p_1 i_x + q_1$. DIG instantiates $j_x = p_1 i_x + q_1$ with the information from R and obtain the set of equations $\{0 = 0p_1 + q_1, 2 = 1p_1 + q_1, 4 = 2p_1 + q_1\}$. The unique solution $\{q_1 = 0, p_1 = 2\}$ of these equations yields $j_x = 2i_x$, i.e., $A[i_x] = b_x B[2i_x] + c_x$. Similarly, DIG instantiates the analogous equations for b_x and c_x. After solving these, the array relation $i_x = \{0, 1, 2\} \Rightarrow A[i_x] = 7B[2i_x] + 3i_x$ is obtained.

Notice that all relations in R have 7 as the coefficient of B_i. DIG can divide these equations by 7 to obtain $R' = \{B_0 = \frac{1}{7}A_0, B_2 = \frac{1}{7}A_1 - \frac{3}{7}, B_4 = \frac{1}{7}A_2 - \frac{6}{7}\}$. From R', DIG selects B as the pivot array and extracts the information $i_x = \{0, 2, 4\}$. Applying the above process of creating and solving linear equations gives the relation $i_x = \{0, 2, 4\} \Rightarrow B[i_x] = \frac{1}{7}A[\frac{1}{2}i_x] - \frac{3}{14}i_x$. DIG recognizes such a scenario and, thus, generates both array relations.

When given as inputs the trace data $|X|$, array variables $|V|$, and array elements $|E|$ consisting of elements from all arrays in V, the complexity of the algorithm to find flat array relations of the form given in Equation (4.10) is dominated by solving equations. We create $|E|$ new variables to represent array elements and use the equation solving technique in Section 3.2.1 to find equalities among them. As analyzed in Section 3.4.1, generating equalities among these variables (terms) takes $O(|E|^3)$, the time of solving $|E|$ equations for $|E|$ unknowns using Gaussian elimination technique.

4.5   Experiments

We evaluate DIG's ability to generate array invariants using traces from an AES implementation described by Yin *et al.* [Yin et al., 2009]. This implementation exemplifies a practical security-critical application and contains nontrivial array invariants. To show that the implementation conforms to the formal AES specification, the authors of AES inspected and documented the invariants of each function in AES and then fully verified the result using SPARK Ada [Barnes, 2003] and PVS [Owre et al., 1992]. The annotated invariants represent the manual effort required to fully verify the functionality of an AES implementation using axiomatic semantics. AES contains 868 lines of Ada code organized into 25 functions containing 30 invariants: 8 flat array relations, 7 nested array relations, 2 linear equations, and 13 other relations.

Similarly to the evaluation of DIG on polynomial invariants in Section 3.6, we use the parameter $\alpha = 200$ to bound DIG's running times. For flat array relations, DIG automatically adjusts the sizes of the considered arrays in such a way that the total number of array elements does not exceed α. Analyzing over smaller array ranges helps improve the run time of DIG and does not affect the result quality; e.g., the relation $A[i] = B[i]$, which holds for indices $i = 0 \ldots 199$, would also hold for indices $i = 0 \ldots 99$. There is no parameter for nested array relations because reachability analysis enumerates all possible non-repeating nestings to consider array relations up to any nesting depth.

We currently do not statically check array relations against the program code. Although modern SMT solvers can handle nonlinear polynomial arithmetic, they have limited support for array operations, especially the complex form of nested array relations considered by DIG. Indeed, most static analysis techniques abstract operations over arrays because they cannot formally reason over this data structure.

For example, concolic execution techniques, e.g., [Burnim and Sen, 2008], replace symbolic array values directly with concrete values and operations over arrays as uninterpreted functions so that the resulting constraints can be simplified and solved by existing constraint solvers.

Table 4.1 reports experimental results on all 25 functions from AES. DIG discovered 30 candidate invariants for AES, none of which is spurious, i.e., we manually verify that all of the generated array invariants are valid relations. Comparing to the documented invariants, we found all 17 documented relations that are expressible using the considered forms. In many cases, DIG also discovered undocumented invariants. In the three relations $r[i] = \mathsf{xor}(a[i], b[i]), a[i] = \mathsf{xor}(r[i], b[i]), b[i] = \mathsf{xor}(r[i], a[i])$ obtained from `xor2Word`, the first one is a documented invariant but the other two are true properties of the *xor* operator. In addition to the documented invariant $r[i][j] = S[t[i][j]]$ in `subWord`, we found the relation $t[i][j] = Si[r[i][j]]$, which is valid because the array Si contains the reversed values of the array S. The results generated for the `keySetupEnc` functions are conditional invariants, e.g., in `keySetupEnc8`, DIG found $r[i][j] = k[4i + j]$ for $i = 0, \ldots, 7, j = 0, \ldots, 3$ and $r[i][j] = 0$ for other indices i, j. In several cases, the algorithms for both flat and nested array relations discover similar invariants such as $S[i][j] = R[4i + j]$ in `state2Block` because these flat array relations are also nested array relations with nesting depth 0.

On average, DIG took under 35 seconds to generate invariants for each AES function. However, this run time on the AES example could be significantly improved with additional information about the desired invariants. For instance, DIG found the relation $t[i][j] = c[4i + j]$ in `aesDecrypt` in under 10s using reachability analysis, but it took over 60s using the algorithm for flat array relations (for solving 200 equations). The last eleven functions in Table 4.1 has similar run times (77s on average) because the considered arrays in each function were automatically resized to contain $\alpha = 200$ elements. We note that a smaller parameter value is also sufficient

to obtain similar results for AES with much shorter run time, e.g., DIG took on average 14.3s for the last eleven functions when run with $\alpha = 50$.

The 13 documented invariants that were not discovered fall into categories that are not supported by DIG and are left for future work. These can be grouped into three categories: Others$_{1-3}$. Others$_1$ includes nested array relations such as $A[i] = 4B[6C[\ldots]]$. We do not currently handle nested invariants if the elements of A are not exactly nested in B. Others$_2$ includes nested array invariants such as $A[i] = B[C[\ldots]]$ and $A[j] = B[C[\ldots]]$ where $i \neq j$, i.e., a conditional form of nested array relations. We require that generated relations such as $A[i] = B[C[\ldots]]$ hold for all i. Others$_3$ includes array invariants involving functions whose inputs are arrays, such as $f([1, 2])$. We consider only functions with scalar inputs such as $g(7, 8)$. We note that existing dynamic analysis methods cannot find these array relations either.

The manual annotation of AES with sufficient invariants to admit machine-checked full formal verification was a significant undertaking involving hours of tool-assisted manual effort [Yin et al., 2008, 2009]. Annotating pre- and post-conditions and loop invariants has not been solved in general and is known to be a key bottleneck in approaches based on axiomatic semantics [Flanagan and Leino, 2001]. It is not surprising that our approach was unable to discover all relevant invariants; indeed, we view reducing the manual verification annotation burden by one-half as a strong result.

4.6   Summary

This chapter introduced and analyzed a form of relation among arrays that appears often in real-world programs. We show that the task of finding such array relations is related to the well-known mathematical problem of function composition. Our analysis proves that both problems are strongly NP-complete in the number of involved

arrays or functions, but can be solved in polynomial time in the number of array elements or function arguments.

DIG implements the above ideas to infer nested and flat array relations dynamically from program traces. For nested array relations, we build an SMT query using information obtained from a reachability analysis; the satisfying assignment provided by the SMT solver yields the desired invariant. For flat array relations, we look for relations among individual array elements and extract from those results the possible relations among the array indices. These flat array relations also express conditional information, capturing array relations that hold for specific indices. The integration of equation and SMT solvers enables efficient analysis of complex array properties that have not been considered by either static or dynamic methods.

Our evaluation demonstrates the feasibility and potential of DIG by successfully discovering 60% of the documented array relations necessary for full formal verification of an AES implementation under 15 minutes.

Function	Desc	Gen	V, D	$T_{Gen}(s)$	Vs Doc
multWord	mult	1 N_4	7, 2	11.0	1/1
xor2Word	xor	3 N_1	4, 2	0.8	1/1
xor3Word	xor	4 N_1	5, 3	2.0	1/1
subWord	subs	2 N_1	3, 1	1.3	1/1
rotWord	shift	1 F	2, 1	0.5	1/1
block2State	convert	1 F	2, 2	4.1	1/1
state2Block	convert	1 F	2, 2	4.2	1/1
subBytes	subs	2 N_1	3, 2	3.2	1/1
invSubByte	subs	2 N_1	3, 2	3.3	1/1
shiftRows	shift	1 F	2, 2	3.7	1/1
invShiftRow	shift	1 F	2, 2	3.6	1/1
addKey	add	2 N_1	4, 2	3.5	1/1
mixCol	mult	0	-	1.0	0/1 O_3
invMixCol	mult	0	-	1.0	0/1 O_3
keySetEnc4	driver	1 F	2, 2	76.4	1/2 O_2
keySetEnc6	driver	1 F	2, 2	78.8	1/2 O_2
keySetEnc8	driver	1 F	2, 2	79.3	1/2 O_2
keySetEnc	driver	1 F	2, 1	76.3	0/1 O_3
keySetDec	driver	0	-	73.0	0/1 O_3
keySched1	driver	0	-	77.9	0/1 O_1
keySched2	driver	1 F	2, 2	79.5	0/1 O_1
aesKeyEnc	driver	1 F, 1 eq	2, 1	76.2	1/2 O_3
aesKeyDec	driver	1 eq	2, 1	73.6	1/2 O_3
aesEncrypt	driver	1 F	2, 2	70.5	0/1 O_3
aesDecrypt	driver	1 F	2, 2	73.8	0/1 O_3
25 functions		**30**		**878.5**	**17/30**

Table 4.1: Experimental results on 25 functions from AES. The **Gen** column counts the number of unique candidate invariants generated by DIG. The **V, D** column reports the number of distinct array variables and the highest dimension of the arrays in the candidate invariants. The types of the generated invariants, reported in the **Invs** columns, include *N*ested, *F*lat, and linear equality invariants (N_l indicates that the depth of the generated nested array relation is l). The **Vs Doc** column reports the number of documented invariants matched by DIG's results. The column also indicates the types of documented invariant (called *O*thers) that DIG could not identify. The *driver* functions are composed from other functions in this table. The benchmark AES implementation and experimental results are available at `https://bitbucket.org/nguyenthanhvuh/dig/`.

Chapter 5

From Program Verification to Synthesis

"Get the habit of analysis– analysis will in time enable synthesis to become your habit of mind." – Frank Lloyd Wright[1]

This chapter establishes a direct link between the problem of reachability in program verification and the problem of template-based synthesis. Such a connection enables the direct application of ideas and tools from one field to another, e.g., leveraging techniques for test input generation (a reachability task) to repair programs (a synthesis task). Parts of this chapter have been submitted for publication in [Nguyen et al., 2014b].

---

[1]American architect and writer, who was considered the most abundantly creative genius of American architecture. His Prairie style became the basis of 20th century residential design in the United States (1867 – 1959).

5.1   Introduction

Automatic program verification and synthesis are two important problems in computer science that have tremendous value in program research. Verification is the task of validating program correctness with respect to a given specification [Srivastava et al., 2013]. Synthesis is the task of finding a program that meets a given specification [Srivastava et al., 2013]. Both problems are notoriously difficult and have been proved undecidable in general cases. However, there has been less work on automatic program synthesis, which was considered to be *"among the last tasks that computers will do well"* [Manna and Waldinger, 1979], compared with verification. The advent of powerful verification techniques using constraint solving, e.g., SAT and SMT solvers, in the past decade has changed the view of automatic verification from being intractable to being realizable [Srivastava, 2010]. These technologies may help revise the view of synthesis from being impossible to being plausible. For example, recent work in synthesis adopts many verification techniques to create programs, e.g., using symbolic execution to generate program repairs [Könighofer and Bloem, 2011, Nguyen et al., 2013].

In this chapter, we establish a formal connection between certain formulations of program verification and synthesis. We view program verification as a *reachability* problem, which checks if the program can reach an undesirable state, and we consider *template-based synthesis*, which generates missing code for a partially complete program. We then constructively prove that reachability and template-based synthesis are *equivalent*. We reduce a template-based synthesis problem, which consists of a program with parameterized templates to be synthesized and a test-suite specification, to a program consisting of a specific location that is reachable only when that template can be instantiated such that the program has the desired behavior. To reduce reachability to synthesis, we transform a reachability instance consisting of a program and a given location into a synthesis instance that can be solved only

when the location in the original problem is reachable. Thus, the task of synthesizing code is used as a procedure to decide if a program location can be reached, and conversely, the determination of reachability allows for synthesizing programs. This connection between the two areas enables ideas, optimizations and tools developed for one problem to be applied to the other. For example, existing tools from the well-established field of program verification can be leveraged to solve problems in the relative new field of program synthesis. Dually, advances in program synthesis can potentially contribute to research in program verification, e.g., finding test inputs to reach nontrivial program locations.

To demonstrate the potential impact of the above ideas, we use the construction from the reduction proof to develop a new automatic program repair technique using existing test input generation tools. We view program repair as a special case of program synthesis in which "patch" code is generated so that the program behaves correctly. We develop a prototype tool called CETI (Correcting Errors using Test Inputs) that automatically repairs C programs violating test-suite specifications. Given a test suite and a program failing at least one test in that suite, CETI first applies fault localization to obtain a list of ranked suspicious statements from the buggy program. For each suspicious statement, CETI transforms the buggy program and the information from its test suite into a program reachability instance. The reachability instance is a new program containing a special `if` branch, whose `then` branch is reachable only when the original program can be repaired by modifying the considered statement. By construction, any input value that allows the special location to be reached can map directly to a repair template instantiation that fixes the bug. To find a repair, CETI invokes an off-the-shelf automatic test input generation tool on the transformed code to find test values that can reach the special branch location. CETI stops when such test input values are found. These values correspond to changes that, when applied to the original program, cause it to pass the given test suite.

```
1   int is_upward(int in,
2        int up, int down){
3     int bias, r;
4     if (in)
5       bias = down;
6     else
7       bias = up;
8     if (bias > down)
9       r = 1;
10    else
11      r = 0;
12    return r;
13  }
```

| Test | Inputs | | | Output | Passed |
	in	up	down	expected	
1	1	0	100	0	✓
2	1	11	110	1	−
3	0	100	50	1	✓
4	1	-20	60	1	−
5	0	0	10	0	✓
6	0	0	-10	1	✓

Figure 5.1: Given this buggy program and its test suite, CETI suggests replacing line 5 with the statement `bias = up + 100;` to fix the bug.

In the remainder of this section, we provide a motivating example and list our contributions.

5.1.1 Motivating Example

We present a concrete instance of the reduction from program reachability to synthesis to motivate important design decisions. Consider the program in Figure 5.1, a code excerpt from a traffic collision avoidance system [Do et al., 2005]. The intended behavior of this program can be precisely described as $is\_upward(in, up, down) = in*100+up > down$. The table in Figure 5.1 lists a test suite describing the intended behavior. The test suite also demonstrates that the buggy program fails two tests.

To reduce the search space for possible patches, automated repair approaches typically restrict attention to certain simple types of edits (e.g., tree-structured operators [Weimer et al., 2009], mutation testing operators [Debroy and Wong, 2010], repair templates learned from human fixes [Kim et al., 2013], etc). Although these edits could be applied anywhere, practical optimization program repair approaches

Line	Statement	Score	Rank
5	`bias = down;`	0.75	1
11	`r = 0;`	0.60	2
4	`if (in)`	0.50	3
8	`if (bias > down)`	0.50	3
7	`bias = up;`	0.00	5
9	`r = 1;`	0.00	5
12	`return r;`	0.00	5

Table 5.1: Fault localization results for the program and its test suite in Figure 5.1

often begin by considering statements or locations implicated by off-the-shelf fault localization techniques [Qi et al., 2013]. In this example, we explain CETI in terms of these two design decisions: the shapes of possible edits and the use of fault localization.

CETI first applies fault localization to the program to bias modifications toward regions of the code that are likely to be implicated in the defect. Using Tarantula, a statistical technique that ranks program statements in descending order of their suspiciousness [Jones and Harrold, 2005], CETI obtains a ranked list of statements and their suspiciousness scores as shown in Table 5.1. The tool then considers each statement in the list in descending order until a repair is found.

We consider repairs formed by modifications to the program based on repair templates. In particular, we assume that the program can be repaired by synthesizing expressions involving program variables and unknown coefficients—a design decision used in most template-based program synthesis and program repair algorithm such SemFix [Nguyen et al., 2013] and Forensic [Könighofer and Bloem, 2013]. For example, the template $c_0 + c_1 v_1 + c_2 v_2$ is a linear combination of program variables v_i and unknown constants c_i. This template can be instantiated to yield concrete expressions such as $200 + 3v_1 + 4v_2$ via $c_0 = 200, c_1 = 3, c_2 = 4$. Using this template, CETI considers a repair by replacing the highest ranked statement in Table 5.1, `bias`

`= down;`, with

$$\texttt{bias = } c_0 \texttt{ + } c_1 \texttt{*bias + } c_2 \texttt{*in + } c_3 \texttt{*up + } c_4 \texttt{*down;}$$

where `bias`, `in`, `up`, and `down` are the variables in scope at line 6 and the value of each c_i must be found by our method.

Although program repair is currently phrased as a synthesis problem, we use test input generation to find values for each c_i in the repair template. Given the program and its test suite in Figure 5.1 with the template statement shown above, CETI creates a related program reachability problem instance consisting of a program and a special target location. The created program, shown in Figure 5.2, contains a function $\texttt{p}_{\texttt{is\_upward}}$ that is similar to the function `is_upward` in the original code but with line 6 replaced by the template statement. In addition, the program also contains a starting function $\texttt{p}_{\texttt{main}}$ that encodes the inputs and expected outputs from the given test suite as the guards to a conditional statement leading to the reachability target location L. Intuitively, this reachability problem instance asks if we can find values for each c_i that allow control flow to reach location L.

The reachability problem instance can be given as input to any off-the-self test input generation tool. In this example CETI employs KLEE [Cadar et al., 2008a] to find values for each c_i. KLEE determines that the values $c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0$ allow control flow to reach location L. Finally, we map this solution to a reachability problem back to a solution to the original program repair problem. Those test input values, when applied to the template

$$\texttt{bias = } c_0 \texttt{ + } c_1 \texttt{*bias + } c_2 \texttt{*in + } c_3 \texttt{*up + } c_4 \texttt{*down;}$$

yield the statement

$$\texttt{bias = 100 + 0*bias + 0*in + 1*up + 0*down;}$$

which simplifies to `bias = 100 + up;`. Replacing the statement `bias = down;` in

```
int pis_upward(int in, int up, int down,
        int c0, int c1, int c2, int c3, int c4){
  int bias, r;
  if (in)
    bias = c0 + c1*bias + c2*in + c3*up + c4*down;
  else
    bias = up;
  if (bias > down) r = 1;
  else r = 0;
  return r;
}

int pmain(){
    int c0,c1,c2,c3,c4;
    if(pis_upward(1,  0,100,c0,c1,c2,c3,c4) == 0 &&
       pis_upward(1, 11,110,c0,c1,c2,c3,c4) == 1 &&
       pis_upward(0,100, 50,c0,c1,c2,c3,c4) == 1 &&
       pis_upward(1,-20, 60,c0,c1,c2,c3,c4) == 1 &&
       pis_upward(0,  0, 10,c0,c1,c2,c3,c4) == 0 &&
       pis_upward(0,  0,-10,c0,c1,c2,c3,c4) == 1){
      [L]
    }
    return 0;
}
```

Figure 5.2: The reachability problem instance derived from the buggy program and test suite in Figure 5.1. Location L is reachable with values such as $c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0$. These values suggest using the statement `bias = 100 + up;` at line 4 in the buggy program.

the original program with the new statement `bias = 100 + up;` causes the original program to pass all of the test cases.

The use of fault localization to prioritize candidate repairs and restrict attention to certain shapes or templates of candidate repairs closely resemble design decisions used by other repair methods. Indeed, the novelty of many repair techniques relates to where and how they modify existing code or generate new code. For instance, GenProg reuses existing code [Weimer et al., 2009], Debroy and Wong use muta-

tion operators [Debroy and Wong, 2010], while SemFix [Nguyen et al., 2013] and FoREnSiC [Könighofer and Bloem, 2013] create and solve constraints. The particular fault localization and repair template schemes shown above are not novel; they were chosen as indicative examples. By contrast, the novelty in CETI lies in its use of the equivalence between program reachability and program synthesis and, in essense, framing program repair (synthesis) as a test input generation (reachability) problem. This equivalence is established by the constructive proof presented in the next section.

5.1.2 Contributions

In this chapter, we make the following contributions to the research areas of program verification/reachability and program synthesis/repair:

- *Equivalence Theorem.* We prove that that the problems of reachability in program verification and template-based program synthesis are equivalent. This result opens doors to a profitable cross-fertilization between the two research areas of program verification and synthesis.

- *Program Repair Technique.* We present a new automatic program repair technique that leverages the constructive nature of the equivalence proof. Once potential error-causing locations have been identified, we create a template-based synthesis program from the buggy program and the suspicious locations, and transform the synthesis problem and test-suite specification defining its expected behavior into a reachability program. We then apply an off-the-shelf reachability tool to the transformed code to find test inputs to reach the target location in the reachability program and map those test inputs to concrete program patches.

- *Experimental Evaluation.* We empirically evaluate CETI using 41 defects from

the `Tcas` program in the SIR benchmark [Do et al., 2005]. Although the primary contributions are theoretical, even this small case study demonstrates the effectiveness of the equivalence theorem in practice, as CETI achieves higher success rates than many other standard repair approaches.

5.2 Program Reachability is Equivalent to Template-based Synthesis

Program reachability and synthesis are both classical problems in computer science. They both have theoretical and practical impacts in software development. In this section, we first review these problems and then prove that certain formulations of them are equivalent.

5.2.1 Preliminaries

We consider standard imperative programs in a language like C. The language includes usual program constructs such as assignments, conditionals, loops, and functions. A function takes as input a (potentially empty) tuple of values and returns an output value. The correctness of a function is specified using a test suite consisting of a finite set of input/output pairs. A function may call other functions, including itself. A program P consists of finite set of functions including a special starting function p_{main}. For brevity, we write $P(x_i, \ldots, x_n) = y$ to denote that the result of evaluating the function $p_{\mathrm{main}} \in P$ on the input tuple $(x_i, \ldots, x_n)$ is the value y.

To simplify the presentation of the proofs in this section, we assume that the language also supports exceptions. That is, it admits non-local control flow by raising and catching exceptions as in modern programming languages such as C++, Java or Python. This assumption is not necessary for the proof, and we discuss how to remove it in Section 5.2.2.

Program Reachability

The program reachability problem asks if a particular program state or location is reachable. This problem is not decidable in the general case because it can encode the halting problem: a loop terminates if and only if the location immediately after that loop is reachable (cf. Rice's Theorem [Rice, 1953]). Nonetheless, reachability remains a popular and well-studied problem in practice. For example, it is used in model checking [Clarke et al., 1999] to discover if program states representing undesirable program behaviors could occur in practice. Reachability is also a major research interest in the area of test input generation [Cadar and Sen, 2013], which aims to produce test values to explore all reachable program locations.

Definition 5.2.1 (The Program Reachability problem)**.** *Given a finite program* $P(t_1, \ldots, t_n)$ *with a target location* L, *does there exist input values* t_i *such that the execution of* $P(t_1, \ldots, t_n)$ *reaches* L *in a finite number of steps?*

For example, the program in Figure 5.3 has a reachable location L using the solution $\{x = -20, y = -40\}$. Reachability is often formulated in terms of labeled transition systems (for model checking) or Turing machine configurations (for more theoretical analyses). For brevity, we do not reproduce any of that well-studied machinery here, and instead assume a standard formalization of what it means for an execution to visit a labeled location; the reader is referred to [Jhala and Majumdar, 2009] for a thorough treatment. The decision problem formulation of reachability asks merely if such input values exist; in this presentation we use the searching problem formulation and require that the input values be produced.

Program Synthesis

As introduced in Section 2.2, program synthesis is a subfield in artificial intelligence and software engineering that aims to generate automatically program code to meet

```python
def P(x, y):
 if 2*x == y:
   if x > y+10:
       [L]

   return 0
```

Figure 5.3: An instance of program reachability. Program P reaches location L using the solution $\{x = -20, y = -40\}$

a required specification. The problem of creating a complete program is undecidable in general cases [Srivastava, 2010], thus most synthesis techniques concentrate on generating code for partially complete programs. That is, practical synthesis approaches often fill in holes in otherwise-complete programs [Solar-Lezama et al., 2005, 2006]. Moreover, these techniques use specific forms or templates instead of producing arbitrary code [Srivastava et al., 2010]. A synthesis *template* expresses the shape of program constructs, but includes holes (sometimes called template parameters) rather than concretely specifying all low-level details. For example, the template $c_0 + c_1 v_1 + \cdots + c_n v_n$ is a linear combination involving program variables v_i and real-valued template parameters c_i. A synthesis template with unfilled template parameters is incomplete and only has meaning as a program when the synthesizer has filled in all parameters. Borrowing notation from contextual operational semantics, we write $P[c_0, \ldots, c_n]$ to denote the result of instantiating the template program P with the template parameter values $c_0 \ldots c_n$. To find values for the parameters in a program containing one or more synthesis templates, many modern synthesis techniques (e.g., [Solar-Lezama et al., 2007, Srivastava et al., 2010]) encode the program and its specification as a logical formula (e.g., using axiomatic semantics). A constraint solver is then used to find values for the parameters c_i that satisfy that formula. Instantiating the templates with those values yields a complete program that adheres to the required specification. This formulation of template-based pro-

```
def Q(i, u, d):
    if i:
        #stmt with template
        b = c₀+c₁*u+c₂*d
    else:
        b = u
    if (b > d): r = 1
    else: r = 0
    return r
```

Test	Inputs			Output
	in	up	down	expected
1	1	0	100	0
2	1	11	110	1
3	0	100	50	1
4	1	-20	60	1
5	0	0	10	0
6	0	0	-10	1

Figure 5.4: An instance of program synthesis. Program Q passes the test suite T using the solution $\{c_0 = 100, c_1 = 1, c_2 = 0\}$

gram synthesis has many practical applications. For example, it has recently been used for automatic program repair (e.g., [Könighofer and Bloem, 2013, Nguyen et al., 2013]): once the defect has been localized to a small segment of code, that code can be erased and replaced by a synthesis template. Solving the synthesis problem is then equivalent to rewriting that segment of the program to repair the bug.

Definition 5.2.2 (The Template-based Program Synthesis problem). *Given a template program Q with a finite set of template parameters $S = \{c_1 \ldots c_n\}$ and a finite test suite of input/output pairs $T = \{(i, o)\}$, does there exist parameter values c_i such that $\forall (i, o) \in T . (Q[c_1, \ldots, c_n])(i) = o$?*

For example, the program in Figure 5.4 passes the given test suite T using the solution $\{c_0 = 100, c_1 = 1, c_2 = 0\}$. The decision formulation of the problem asks merely if satisfying values $c_1 \ldots c_n$ exist; in this presentation we use the searching problem formulation and require that the concrete values of $c_1 \ldots c_n$ be produced.

5.2.2 Reducing Synthesis to Reachability

We reduce the problem of program synthesis to the problem of program reachability so that the solutions to the reachability problem can be used to synthesize programs.

Theorem 5.2.3 (Templated-based Synthesis is reducible to Program Reachability)**.**
The synthesis problem in Definition 5.2.2 of Section 5.2.1 is reducible to the reachability problem in Definition 5.2.1 of Section 5.2.1.

Proof. Let Q be a template program with a finite set of template parameters $S = \{c_1, \ldots, c_n\}$ and T a finite test suite. The reduction from a general instance Q, T, S of program synthesis to a specific instance P, L of reachability is as follows:

- For every function $q \in Q$, define a similar function $p \in P$ that also has additional formal parameters $c_1, \ldots, c_n$ (the parameters in S). Replace each function call to q with a corresponding call to p with additional actual arguments $c_1, \ldots, c_n$.

- Define a starting function p_{main} that has input arguments $c_1, \ldots, c_n$.

  - Encode the specification information from the test suite T as a conjunctive expression e:
  $$\bigwedge_{(x,y) \in T} p_q(x, c_1, \ldots, c_n) = y$$
  where p_Q is the starting function in Q.

  - p_{main} contains a conditional statement leading to a target location L only when e is true.

  - Thus, p_{main} has the form

```
def p_main(c_1,..,c_n):
    if e:
        [L]


    return 0
```

- The program P consists of the function p_{main} and other p_i.

```
def pQ(i, u, d,
       c0, c1, c2):
    if i:
        b = c0+c1*u+c2*d
    else:
        b = u
    if b > d:
        r = 1
    else:
        r = 0
    return r
```

```
def pmain(c0, c1, c2):
    e = pQ(1, 0,100, c0,c1,c2) == 0 and
        pQ(1,11,110, c0,c1,c2) == 1 and
        pQ(0,100,50, c0,c1,c2) == 1 and
        pQ(1,-20,60, c0,c1,c2) == 1 and
        pQ(0, 0, 10, c0,c1,c2) == 0 and
        pQ(0, 0,-10, c0,c1,c2) == 1

    if e:
        [L] #pass the given test suite

    return 0
```

Figure 5.5: Reducing the synthesis instance in Figure 5.4 to a reachability program.

This reduction transforms an arbitrary synthesis instance Q, S, T to a specific reachability instance P, L such that the location L is reachable if and only if Q can be synthesized successfully. Suppose that L can be reached using the solution c_i, then the predicate e is true (because L is guarded by exactly e) and hence Q, when instantiated with c_i, passes the test suite T (because e is constructed as a conjunction of all of input-output behaviors specified by T). Conversely, if L in P cannot be reached for any input values c_i, then no values of c_i can ever make e true (since e depends entirely on the values of c_i) and hence Q cannot be instantiated for any values c_i to satisfy T. This is also a polynomial time reduction from the input instance Q, S, T. The constructed program P consists of all functions in Q (with $|S|$ extra parameters) and a starting function p_{main} having $|S|$ parameters and an expression encoding the test suite T.

□

Example 5.2.4. Figure 5.5 illustrates the reduction using the synthesis example in Figure 5.4.

5.2.3 Reducing Reachability to Synthesis

Conversely, we reduce reachability to synthesis so that program synthesis can be used to solve reachability.

Theorem 5.2.5 (Program Reachability is reducible to Template-based Synthesis)**.** *The reachability problem in Definition 5.2.1 of Section 5.2.1 is reducible to the synthesis problem in Definition 5.2.2 of Section 5.2.1.*

Proof. Let P be a program containing a location L. The reduction of a general instance P, L of reachability to a specific instance Q, S, T of program synthesis is as follows:

- For every function $p \in P$, define a similar function $q \in Q$. Replace each function call to p with the corresponding call to q.

- Raise a unique exception REACHED at the location in Q corresponding to the location L in P. The exception REACHED will be caught if and only if the location in Q corresponding to $L \in P$ has been reached.

- Define a starting function q_{main} that has no inputs and returns an integer value.

  - Let q_{P} be the function in Q corresponding to the starting function in P. At the beginning of q_{main}, insert a finite set of template assignment of the form $x_i = c_i$, where x_i are the inputs to function q_{P} and c_i are parameters to be synthesized.

  - Next, insert a *try-catch* construct that calls q_p on inputs $x_1, \ldots, x_n$ and returns the value 1 if the exception REACHED is caught.

  - At the end of q_{main}, return the value 0.

  - Thus, q_{main} has the form

```
def q_main():
    x_1 = c_1
    ...
    x_n = c_n
    try:
        q_p(x_1,..,x_n)
    catch REACHED:
        return 1

    return 0
```

- The template program Q consists of the finite set of template parameters $S = \{c_1, \ldots, c_n\}$, functions q_{main}, and other q_i.

- The test suite T for Q consists of exactly one test case $Q() = 1$.

The reduction transforms an arbitrary reachability instance P, L into a specific synthesis instance Q, T, S that can be synthesized if and only if location L in P is reachable. Suppose that Q can be synthesized with values c_i such that $Q() = 1$ (the only test in T), then the location corresponding to L in Q is reachable (to return 1 the exception must have been caught, to be caught it must have been raised, it is raised only at that location), and hence $L \in P$ is reachable using the same input values c_i (because P and Q share control- and data-flow). Conversely, if Q cannot be synthesized with any values c_i such that $Q() = 1$, then the location corresponding to L in Q and hence $L \in P$ is not reachable for any input values of c_i. This is also a polynomial time reduction from the input instance P, L. The constructed program Q consists of all functions in P and a starting function q_{main} having n template assignments, where n is the number of inputs to P. $\quad\square$

```
                                def q_main():
                                 #synthesized stmts
 def q_P (x, y):                 x = c_0
  if 2*x == y:                   y = c_1
   if x > y+10:                  try:
     [L]                           q_P(x,y)
     raise REACHED              catch REACHED:
                                  return 1 #success
  return 0
                                return 0
```

Test suite $\mathbf{T}$: Q() = 1

Figure 5.6: Reducing the reachability instance in Figure 5.3 to a synthesis program.

Example 5.2.6. Figure 5.6 illustrates the reduction using the reachability example in Figure 5.3.

The exception REACHED represents a signal, unique from all other values returned by q_p, indicating to q_{main} that the location corresponding to L has been reached. We present the reduction in terms of exception handling, a familiar mechanism supported by many modern languages. However, exceptions are not necessary for the reduction to proceed. Other (potentially language-dependent) implementation techniques can be also employed. We could use tuples or `struct`s as a signal, returning (v, false) from a function that normally returns v if the location corresponding L has not been reached and $(1, \mathsf{true})$ as soon as it has. BLAST [Beyer et al., 2007], a model checker for C programs (which do not support exceptions), uses *goto* and labels to indicate when a desired location has been reached.

5.2.4 Synthesis ≡ Reachability

Together, the above two theorems establish the equivalence between program synthesis and reachability.

Theorem 5.2.7. *The synthesis problem in Definition 5.2.2 of Section 5.2.1 is equiv-*

alent to the reachability problem in Definition 5.2.1 of Section 5.2.1.

Proof. This follows from Theorem 5.2.3 stating that synthesis is reducible to reachability and Theorem 5.2.5 stating that reachability is reducible to synthesis. Thus, the two problems are reducible to each other, i.e., equivalent. □

This result connects the two fields and the constructive nature of the proof allows for insights and techniques from one problem to be used to the other. In the next section, we demonstrate just such a concrete application for this theoretical result.

5.3 CETI: Automatic Program Repair using Test Input Generation

In this section, we describe a new approach for solving automated program repair problems (a synthesis task) using techniques for test input generation (a reachability task). We define the problem of program repair in terms of template-based program synthesis:

Definition 5.3.1 (Single-Edit Program Repair problem)**.** *Given a program P that fails at least one test in a finite test suite T and a finite set of parameterized templates S, does there exist a location $L \in P$ and parameter values $c_1, \ldots, c_n$ for the templates in S such that L can be replaced with $S[c_1, \ldots, c_n]$ and the resulting program passes all tests in T?*

We present an automatic program repair tool called CETI (Correcting Errors using Test Inputs) to solve this repair problem. The tool implements the key ideas from Theorem 5.2.3 in Section 5.2.2 to transform the repair task into a reachability problem, which can then be solved by tools developed in verification subfields such as model checking and test input generation in software testing. Figure 5.7 gives an
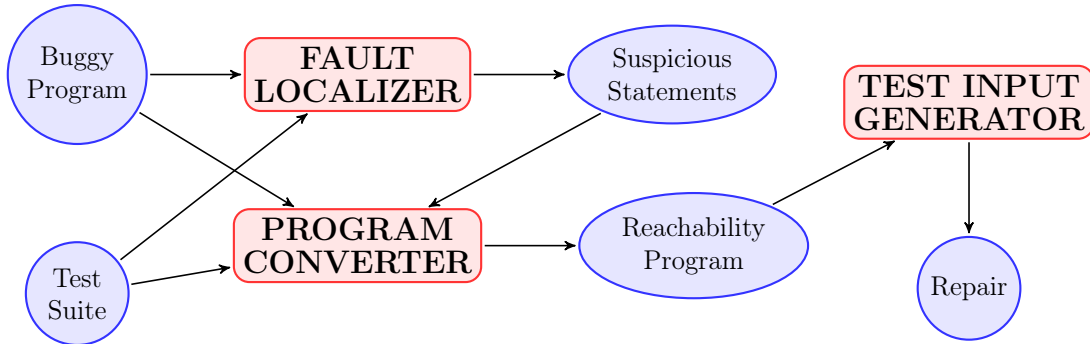
Figure 5.7: CETI: Automatic Program Repair using Test input Generation.

overview of CETI. Given a test suite and a buggy program that fails some test in the suite, CETI employs an existing statistical fault localization technique to rank suspicious statements. Next, for each suspicious statement and synthesis template, CETI transforms the buggy program, the test suite, the statement and the template into a new program containing a location reachable only when the original program can be repaired. The transformed program is then sent to an off-the-shelf test input generation tool, which produces test values that can reach the designated location. Such test input values, when combined with the synthesis template and the suspicious statement, correspond exactly to a patch that repairs the bug.

5.3.1 Repair Components

We review the main repair components of CETI. These include fault localization, repair templates, and test input generation.

Statistical Fault Localization

To transform a program repair instance into a synthesis task, we first identify particular code regions for synthesis. These correspond exactly to patch locations. Although some program repair approaches consider all possible locations [Weimer et al., 2013], a common optimization is to consider locations likely to be implicated in the defect

first. We use a statistical *fault localization* technique [Jones and Harrold, 2005, Qi et al., 2013] to identify program statements likely related to the defect. Statistical fault localization techniques typically compute a *suspiciousness score* for each program statement based on static and dynamic information, such as the frequency of its appearance in passing and failing program test case runs. We consider techniques that return a list of ranked statements in descending order of their suspiciousness with respect to a given defect.

CETI implements the popular Tarantula statistical algorithm [Jones and Harrold, 2005], which assigns a score for a program statement s as

$$\mathsf{score}(s) = \frac{\mathsf{fail}(s)/\mathsf{totalfail}}{\mathsf{fail}(s)/\mathsf{totalfail} + \mathsf{pass}(s)/\mathsf{totalpass}}$$

where $\mathsf{fail}(s)$ is the number of failing runs in the test suite that visit statement s and $\mathsf{totalfail}$ is the number of failing runs (regardless of whether s is reached or not). Similarly, $\mathsf{pass}(s)$ and $\mathsf{totalpass}$ are the corresponding numbers for passing runs. Table 5.1 lists the Tarantula scores of the program statements in Figure 5.1.

Repair Templates

Like many template-based program synthesis and repair methods [Nguyen et al., 2013, Solar-Lezama et al., 2005, Srivastava et al., 2013], CETI modifies program statements using predefined templates. We consider repairs that change only the right hand sides of *assignment* statements. We assume an intermediate program representation such as CIL [Necula et al., 2002] that simplifies expressions that control side-effects or control-flow through the introduction of well-typed temporaries. For example, `x = y = z;` is treated as `y = z; x = y;` and `if(e){..}` is treated as `temp = e; if(temp){..}`. This allows for many defects, including those occurring at non-assignment statements such as conditions and loops, to be addressed even when attention is restricted to statement-level [Weimer et al., 2009] or assignment-level [Nguyen et al., 2013] patches.

At a high level, we assume that the program is mostly correct; if the programmer is using a comparison operation at a suspicious statement, we consider replacing it with another comparison operation, instead of arbitrary code. Given an assignment `x = e;`, where e is an expression, CETI considers the repair templates given in Table 5.2, defined based on the structure of e and the variables it references. For example, if e is $x \leq y$, then CETI creates the $\mathrm{T_{op}}$ template $c_1 * (x \leq y) + c_2 * (x < y) + \cdots + c_5 * (x \geq y)$, where the parameter c_i have boolean values and exactly one c_i is true. The template instantiates different logical comparisons between x and y depending on the value of c_i. For example, $c_5 = \mathsf{true}$ corresponds to $x \geq y$. CETI automatically analyzes the recursive structure of e to apply appropriate operators. The tool also uses the template $\mathrm{T_{const}}$ to replace all constants in an expression with parameters, e.g., $2 * x + 3 * y + 4$ becomes $c_1 * x + c_2 * y + c_3$.

We also assume that defects can often be addressed in terms of in-scope variables. The k-linear template is a combination among k variables and $k + 1$ parameters. Unlike the operator-directed templates, this template is applicable to any assignment statement `x = e;` regardless of the structure of e. Applying this template on the assignment `x= ...;` results in a template assignment `x = `$\mathtt{c_0 + c_1 * v_1 + ... + c_k * v_k}$`;`, where v_i are k variables in scope at the assignment statement. Such a template can be instantiated to form statements such as `x = 7;`, `x = `$\mathtt{2 * v_1}$`;`, or `x = `$\mathtt{2 * v_1 + 5 * v_2 + 100}$`;`.

For single-edit repairs, CETI considers only one change at a time as a candidate patch. If e contains n operators then the tool creates and synthesizes n template expressions separately. Similarly, when there are more than k variables available, CETI creates all possible combinations of variables of size k then applies the k-linear template on each combination.

Template	Description
T_{op}	operators
	logics $\wedge, \vee$
	comparison $\leq, <, =, >, \geq$
	arithmetic $+, -$
T_{const}	const c_i
$T_{lincomb}$	k-linear comb $c_0 + c_1 v_1 + \cdots + c_k v_k$

Table 5.2: Repair templates used in CETI.

Test Input Generation

Test input generation is a subfield in program verification that aims to generate high-coverage test data and, thus, to find deep errors in complex software. Although black-box approaches that do not inspect the program are possible (e.g., fuzz testing [Miller et al., 1990, Tillmann and de Halleux, 2008]), many approaches force a program to visit a particular path by calculating the associated path predicate and using a constraint solver to find satisfying values. This is complicated by common program features that are difficult to reason about statically, such as aliasing.

Concolic execution [Cadar and Sen, 2013] is a modern test input generation technique that combines concrete and symbolic executions to create test inputs that explore as many different program locations as possible. To find inputs leading to a program location, the technique encodes the conditions leading to that location as a path constraint, which can be solved with a constraint solver for concrete input values. For the program in Figure 5.3, the technique can generate inputs such as $(x = 0, y = 1)$ from the constraint $2x \neq y$ leading directly to the `return` statement and the inputs $(x = -20, y = -40)$ from $2x = y \wedge x > y + 10$ leading to location L.

CETI employs KLEE [Cadar et al., 2008a], an automatic test input generation tool based on concolic execution, to find inputs reaching a desired location in a

C program. KLEE has been shown to generate high coverage inputs on to complex real-world programs, e.g., above 90% line coverage of tools in the GNU COREUTILS utility suite. Because KLEE repeatedly produces test inputs to make a high coverage test suite, we terminate the test input generation process as soon as the desired location is reached.

Although CETI uses KLEE for test input generation, other tools and techniques, such as CREST [Burnim and Sen, 2008], SAGE [Godefroid et al., 2008], PEX [Tillmann and de Halleux, 2008], BLAST [Beyer et al., 2007], or SLAM [Ball and Rajamani, 2002] could be used to find inputs reaching a program location. In fact, for reachability purposes, software model checkers such as BLAST and SLAM are potentially more efficient because they target the reachability of specified locations, rather than exploring all possible locations as in the case of test input generation tools. Finally, because CETI treats a test input generator as an untrusted black box, it runs multiple test input generation tools in parallel to take advantage of techniques that make different performance and correctness trade-offs. A key advantage of CETI is that it admits other test input generation or reachability tools, regardless of the technologies used in these tools.

5.3.2 Repair Algorithm

Figure 5.8 outlines the repair algorithm of CETI, which takes as inputs a test suite T, a program Q failing T, and returns a modified program Q' satisfying T. The algorithm synthesizes correct-by-construction repairs, i.e., the repair, if found, is guaranteed to pass the test suite. As discussed in Section 5.3.1, we use a pre-processing step to parse the program into an intermediate representation that exposes expressions for the benefit for fault localization and repair template locations. Next, we obtain a list of suspicious statements using an off-the-shelf fault localization algorithm.

The algorithm then applies each applicable predefined repair template given in

procedure Repair(Q, T)
    $Q \leftarrow$ parse(Q)   ▷ Parse and simplify complex program constructs
    susp_stmts $\leftarrow$ faultLoc(Q,T)   ▷ Apply statistical fault localization

    ▷ Create programs with template statements
    tpl_progs $\leftarrow \emptyset$
    for $ss \in$ susp_stmts, in ranked order **do**
        for $tpl \in$ predefined_templates **do**
            tpl_stmts $\leftarrow$ applyTemplate(tpl,ss)
            for $ts \in$ tpl_stmt **do**
                $q \leftarrow Q.replace(ss, ts)$
                tpl_progs.$add(q)$

    ▷ Convert to reachability programs and apply test input generation (parallel code)
    for $q \in$ tpl_progs **do**
        $p \leftarrow$ convert(q,T)
        test_vals $\leftarrow$ genTestInputs(p)
        if test_vals $\neq \emptyset$ **then**
            return $q.instantitate$(test_vals)
    return "no single-edit repair found"

Figure 5.8: Algorithm for single-edit program repair from the inputs: a test suite T and a program Q failing T. The main step of this algorithm are: pre-processing the input program to an intermediate representation (parse), using fault localization to obtain a ranked list of suspicious statements (faultLoc), creating template-based synthesis programs using predefined templates and suspicious statements (applyTemplate), converting synthesis programs to reachability problems using equivalence theorem (convert), and applying test input generation to find solutions for reachability problems, which correspond to program repairs (genTestInputs).

Table 5.2 on each suspicious statement, in order, to create template statements containing parameters to be synthesized. We focus on single-statement modifications by creating, for each template statement ts, a new program similar to the input program but with the suspicious statement replaced by ts. This step produces a set of template programs tpl_prog, each of which, when coupled with the test suite T, is an instance of the program synthesis problem (Definition 5.2.2 of Section 5.2.1).

Using the constructive reduction step shown in Section 5.2.2, we convert each

program $q \in$ tpl_prog to a reachability problem p containing a location L that corresponds exactly to passing the test suite T. Next, we send each transformed program p to an off-the-shelf test input generation tool. If such test values are found, they are instantiated into the corresponding parameters in template program q to obtain a repaired program that passes the test suite T.

As an optimization, the algorithm parallelizes the process of converting programs and running the test input generator. Although the program transformation step can be done quickly, finding test cases to reach a certain program location can be expensive depending on the given program and the test input generation tool. Moreover, the task is embarrassingly parallel because each test input problem can be considered independently. Parallelization allows us to quickly find a repair by running multiple test generation instances simultaneously and stopping when the first repair is found by any of the parallel tasks[2]. This optimization mirrors other parallelism approaches found in previous ad hoc techniques. For example, each candidate patch and each test case can be considered independently in GenProg.

5.4   Experiments

The prototype tool CETI takes as input a buggy C program and a test suite and makes use of an off-the-shelf fault localization tool (Tarantula) and an off-the-shelf reachability tool (KLEE) as well as a set of predefined repair templates (described in Table 5.2). The tool converts each candidate repair (i.e., the application of a given repair template at a given suspicious statement) into a test input generation problem; any satisfying test input values are mapped back into a repair. We use the CIL [Necula et al., 2002] front end to parse and modify program constructs

---

[2] We note that a test input generation tool *designed for* use in such a repair algorithm could re-use intermediate results and pre-processing steps on these similar queries and thus operate even more rapidly. However, we do not assume anything about the test input tool used.

followed by other scriptings to invoke the test input generator tool in parallel. The website `https://bitbucket.org/nguyenthanhvuh/ceti/` contains the source code of CETI, benchmark programs, and experimental results given in this chapter.

The behavior of CETI is controlled by customizable parameters. For fault localization, we implement both the Tarantula and Ochiai [Abreu et al., 2006] statistical score metrics. We consider the top $n = 80$ statements with a score $s > 0.2$ from the ranked list of suspicious statements and, then, apply the predefined templates to these statements. We use $k = 2$-linear templates, representing expressions of the form $c_0 + c_1 v_1 + c_2 v_2$. For efficiency, we restrict synthesis parameters to be within certain value ranges: constant coefficients c_0 are confined to the integral range $[-100000, 100000]$ while the variable coefficients c_1, c_2 are drawn from the set $\{-1, 0, 1\}$.

To evaluate CETI, we use the `Tcas` program from the SIR benchmark [Do et al., 2005]. The program, which implements an aircraft traffic collision avoidance system, has 180 lines of code and 12 integer inputs. The program comes with a test suite of about 1608 tests and 41 faulty functions, consisting of seeded defects such as changed operators, incorrect constant values, missing code, and incorrect control flow. This program has been used to benchmark modern bug repair techniques including SemFix [Nguyen et al., 2013], FoREnSiC [Könighofer and Bloem, 2013], and Debroy and Wong [Debroy and Wong, 2010].

We manually modify `Tcas`, which normally prints its result on the screen, to instead return its output to its caller, e.g., `printf("output is %d\n",v);` becomes `return v;`. Similar techniques were used in FoREnSiC and SemFix because they also need to encode the input and expected output specifications as constraints. We expand constant array values (e.g., `int arr[2];` becomes two variables, `arr0` and `arr1`) when appropriate to simplify integration with KLEE. We also observe that the test case generator KLEE works slowly on programs containing large numbers

of global variables. Therefore, we convert global variables to local variables whenever possible (following algorithms and tools to do this automatically, e.g., [Yang et al., 2009]). For efficiency, many repair techniques initially consider a smaller number of tests in the suite and then verify promising candidate repairs on the entire suite [Fast et al., 2010] (e.g., the author of SemFix minimize the given test suite to 50 tests [Nguyen et al., 2013]). In contrast, we use all available test cases at all times to guarantee that any repair found by CETI is correct with respect to the test suite. We find that modern test input generation tools such as KLEE can handle the complex conditionals that encode such information efficiently, finding the desired solutions within seconds. For brevity, we elide all tests not related to algorithmic correctness (e.g., sanity checks ensuring that the correct number of inputs are passed to the function).

Table 5.3 shows the experimental results with 41 buggy `Tcas` versions. These experiments were performed on a 32-core 2.60GHz Intel Linux system with 128 GB of RAM. We were able to correct 26 of 41 defects, including multiple defects of different types. On average, CETI takes 22 seconds for each successful repair. The tool found 100% of repairs for which the required changes are single edits according to one of our predefined templates (e.g., generating arbitrary integer constants or changing operators at one location). In several cases, defects could be repaired in several ways. For example, defect v_{28} can be repaired by swapping the results of both branches of a conditional statement or by inverting the conditional guard. CETI also obtained unexpected repairs. For example, the bug in v_{13} is a comparison against an incorrect constant; the buggy code reads `< 700` while the human-written patch reads `< 600`. Our generated repair of `< 596` also passes all tests. The human acceptability and maintainability of automatic patches is an area of ongoing research (e.g., [Fry et al., 2012, Kim et al., 2013]), but we note that the acceptability of our repairs to humans are comparable to those reported for other tools.

We were not able to repair 15 of 41 defects, each of which requires edits at multiple locations or the addition of code that is beyond the scope of the current set of templates. As expected, CETI takes longer for these programs because it tries all generated template programs before giving up. One common pattern among these programs is that the bug occurs in a macro definition, e.g., `#define C = 100` instead of `#define C = 200`. Since the CIL front end automatically expands such macros, CETI would need to individually fix each use of the macro in order to succeed. This is an artifact of CIL, rather than a weakness inherent in our algorithm.

CETI, which repairs 26 of 41 `Tcas` defects, performs well compared to other reported results from repair tools on this benchmark program. GenProg, which finds edits by recombining existing code, can repair 11 of these defects [Nguyen et al., 2013, Tab. 5]. The technique of Debroy and Wong, which uses random mutation, can repair 9 of these defects [Debroy and Wong, 2010, Tab. 2]. FoREnSiC, which uses the concolic execution in CREST, repairs 23 of these defects [Könighofer and Bloem, 2013, Tab. 1]. SemFix out-performs CETI, repairing 34 defects [Nguyen et al., 2013, Tab. 5], but also uses fifty manually selected test cases instead of the entire suite of thousands. Other repair techniques, including equivalence checking [Könighofer and Bloem, 2013] and using counterexample guided refinement [Könighofer and Bloem, 2013], repair 15 and 16 defects, respectively.

The closest related tool, SemFix, directly uses and customizes a concolic execution engine. In contrast, we use KLEE off-the-shelf to demonstrate that the constructive reduction works competitively in general, even without domain-specific optimizations. This is a trade-off: customizing a reachability solver to the task of program repair may increase the performance or the number of repairs found, but may also reduce the generality or ease-of-adoption of the overall technique. We note that our unoptimized tool CETI already outperforms published results for GenProg, Debroy and Wong, and FoREnSiC on this benchmark, and is competitive with SemFix.

The goal of this proof-of-concept evaluation was to demonstrate that the constructive reductions between program synthesis and program reachability allow tools and algorithms from one area to be used directly in the other.

5.5 Summary

The reachability problem in program verification and template-based program synthesis are two important problems in computer science with many theoretical and practical applications. This chapter proved that the two problems are equivalent. We reduce a general program synthesis instance to a specific reachability instance consisting of a special location that is reachable when code could be generated for the synthesis problem. Conversely, we reduce a general reachability instance to a specific synthesis instance such that a successful synthesis indicates the reachability of the target location in the original problem. This equivalence result connects the fields of synthesis and reachability and enables the application of ideas, optimizations, and tools developed for one problem to the other.

To show the potential impact of these theoretical results, we leverage the constructive nature of the reduction proof to develop a new algorithm for automatic program repair (a synthesis problem) using test input generation techniques (which solve reachability problems). We use existing statistical fault localization algorithms to rank suspicious program statements, consider template code transformations on the buggy program, apply an off-the-shelf test input generation tool to each transformed program to find test inputs, and map those inputs back into concrete patches. We implement these ideas in a prototype tool called CETI to automatically repair C programs, evaluating on a benchmark that has been targeted by multiple program repair algorithms. The preliminary results suggest the usefulness of the equivalence in practice because CETI has a higher success rate than many other standard repair approaches.

	Bug Type	R-Progs	Time (s)	Repair?	Template
v1	incorrect op	6143	21	✓	T_{op}
v2	missing code	6993	27	✓	$T_{lincomb}$
v3	incorrect op	8006	18	✓	T_{op}
v4	incorrect op	5900	27	✓	T_{const}
v5	missing code	8440	394	–	–
v6	incorrect op	5872	19	✓	T_{op}
v7	incorrect const	7302	18	✓	T_{const}
v8	incorrect const	6013	19	✓	T_{const}
v9	incorrect op	5938	24	✓	T_{op}
v10	incorrect op	7154	18	✓	T_{op}
v11	multiple	6308	123	–	–
v12	incorrect op	8442	25	✓	T_{op}
v13	incorrect const	7845	21	✓	T_{const}
v14	incorrect const	1252	22	✓	T_{const}
v15	multiple	7760	258	–	–
v16	incorrect const	5470	19	✓	T_{const}
v17	incorrect const	7302	12	✓	T_{const}
v18	incorrect const	7383	18	✓	T_{const}
v19	incorrect const	6920	19	✓	T_{const}
v20	incorrect op	5938	19	✓	T_{op}
v21	missing code	5939	31	✓	$T_{lincomb}$
v22	missing code	5553	175	–	–
v23	missing code	5824	164	–	–
v24	missing code	6050	231	–	–
v25	incorrect op	5983	19	✓	T_{op}
v26	missing code	8004	195	–	–
v27	missing code	8440	270	–	–
v28	incorrect op	9072	11	✓	T_{op}
v29	missing code	6914	195	–	–
v30	missing code	6533	170	–	–
v31	multiple	4302	16	✓	$T_{lincomb}$
v32	multiple	4493	17	✓	$T_{lincomb}$
v33	multiple	9070	224	–	–
v34	incorrect op	8442	75	✓	$T_{lincomb}$
v35	multiple	9070	184	–	–
v36	incorrect const	6334	10	✓	T_{const}
v37	missing code	7523	174	–	–
v38	missing code	7685	209	–	–
v39	incorrect op	5983	20	✓	T_{op}
v40	missing code	7364	136	–	–
v41	missing code	5899	29	✓	$T_{lincomb}$

Table 5.3: Repair results for 41 `Tcas` defects. Column **Bug Type** describes the type of defect. *Incorrect Const* denotes a defect involving the use of the wrong constant, e.g., 700 instead of 600. *Incorrect Op* denotes a defect that uses the wrong operator for arithmetic, comparison, or logical calculations, e.g., $\geq$ instead of $>$. *Missing code* denotes defects that entirely lack an expression or statement, *e.g.*, $a\&\&b$ instead of $a\&\&b||c$ or `return a;` instead of `return a+b;`. *Multiple* denotes defects caused by several actions such as missing code at a location and using an incorrect operator at another location. Column **Seconds** shows the time taken. Column **R-Prog** lists the number of reachability problem instances that were generated and processed by KLEE. Column **Repair?** indicates whether a repair was found and column **Template** names the template used to find that repair (see Table 5.2). These benchmark programs and experimental results are available at `https://bitbucket.org/nguyenthanhvuh/ceti/`.

Chapter 6

Conclusions

"Beware of bugs in the above code; I have only proved it correct, not tried it." – Donald Knuth[1]

As stated in Chapter 1, the thesis of this dissertation is to build effective techniques to automatically generate invariants and synthesize programs by encoding these tasks as solutions to existing problems in the mathematical and verification domains. In the following, we review the key contributions of this dissertation to support the thesis, suggest future directions, and offer final remarks.

6.1 Summary of Findings

In Section 1.2, we hypothesize that complex program invariants can be interpreted as linear equations and constraints, which can be efficiently solved using existing mechanical and powerful constraint solvers. This thesis is supported by the following contributions described in Chapters 3 and 4:

---

[1]American computer scientist and mathematician, who created the TeX computer typesetting system (1938 – present).

- *Geometric Invariant Inferring.* We present and evaluate a geometric approach for discovering general polynomial relations among numerical program traces. We reduce the inference of conjunctive polynomial equalities and inequalities to the task of building hyperplanes and convex polyhedra in high-dimensional space. We also construct nonconvex polyhedra in the non-standard max and min-plus algebras to handle disjunctive polynomial relations. For efficiency, we define and detect weaker forms of polynomial invariants that balance between expressive power and efficiency. Finally, by using terms to represent nonlinear polynomials among program variables, the geometric inference technique yields nonlinear relations among the original variables.

- *Static Invariant Proving.* We develop a custom, automatic theorem prover for verifying invariants based on iterative, parallel k-inductive SMT solving. Many program invariants are not classically inductive, and k-induction allows us to prove them. Similarly, the re-use of learned invariants as lemmas allows us to prove non-inductive invariants in practice. The explicit parallel structure of the prover is critical for performance. By construction, our geometric approach does not overapproximates true program invariants that are expressible using supported forms. Moreover, validating each candidate against the program code means that we do not underapproximate the true invariants; this approach helps address the issue of spurious or incorrect invariants.

- *Theory of Nested Array Relations.* We conduct a formal analysis that connects the tasks of composing functions and finding nested relations among arrays. The analysis establishes the complexity of the two problems; they are `NP`-complete in the number of involved arrays and functions, but have polynomial complexity in the number of array elements or function arguments. Based on this analysis, we develop a dynamic technique that employs equation and SMT solving to discover complex array invariants from program traces.

- *Implementation and Evaluation.* We implement DIG, an invariant analysis
  tool, to dynamically detect invariants, and KIP, a theorem prover based on
  k-induction, to verify candidate invariants. These tools are evaluated using
  difficult programs involving nonlinear arithmetic and a full implementation of
  an AES encryption algorithm. Experimental results show that the tools are
  efficient, both at learning complex invariants and proving them correct.

We also hypothesize in Section 1.2 that certain formulations of verification and
synthesis are equivalent, and this equivalence allows for the exchange of ideas and
techniques between different research areas. This thesis is supported by the following
contributions described in Chapter 5:

- *Equivalence Theorem.* We formally show that program reachability, a formula-
  tion of verification, and template-based synthesis, an approach to synthesis, are
  interreducible. We encode a reachability problem as a synthesis task, where a
  successful synthesis indicates the reachability of the target location in the orig-
  inal problem. Dually, we transform a synthesis task into a program containing
  a target location, reachable only when code could be generated for the original
  task.

- *Automatic Program Repair Technique.* To demonstrate the potential impact
  of the equivalence theorem, we present a new approach to automatic program
  repair (a synthesis task) using existing techniques for test input generation
  (which solves reachability problems). We rank suspicious code statements using
  an existing fault localization algorithm, perform a code transformation on the
  buggy program, apply a test input generation tool on the transformed code to
  find inputs, and map those test inputs back into concrete repairs.

- *Implementation and Evaluation.* We implement CETI, a prototype tool to
  repair C programs using off-the-shelf test input generation tools. Experimental

results from common bug benchmarks show that CETI can have higher success rates than many other standard repair approaches.

6.2 Future Work

The research presented in this thesis could be continued in many directions. The invariant detection tool DIG can be extended to new classes of polynomial invariants, such as the congruence relations of the form $c_i x_i + \cdots + c_n x_n \equiv c[m]$ used in abstract interpretation to analyze pointer alignment properties and bit vectorizations [Granger, 1989, 1991]. The modular design of DIG allows for easy extensions to other geometric shapes for other forms of relations. Currently, DIG can identify 60% of the required array relations necessary for full formal verification of the AES implementation. Our goal is to cover the remaining array invariants in AES, e.g., analyzing disjunctive nested array relations by exploiting the NP-completeness proof for arrays in Chapter 4. We are also interested in analyzing properties of other data structures related to arrays such as strings and trees. The prover KIP can be extended to support arrays and other data structures by using constraint solvers designed specifically for these data structures instead of general SMT solvers, e.g., Boolector for arrays and bit-vectors [Brummayer and Biere, 2009] or HAMPI for string constraints [Kiezun et al., 2009]. In addition, we are extending KIP to verify invariants for recursive programs and functions.

The tool CETI synthesizes program repairs that are correct with respect to a given test suite, not a formal specification. Consequently, these repairs may not be generalized to test inputs that are not given in the test suite. We propose to use theorem proving, e.g., KIP, to verify the repair against a formal specification if available. If the repair violates the formal specification, we can generate counterexamples and refine the repair iteratively. In addition, we intend to integrate CETI, which synthesizes code using repair templates, with evolutionary-based tech-

niques such as GenProg, which can evolve arbitrary extant code. This dissertation concretely demonstrates the applicability of program reachability (test input generation) to program synthesis (bug repair) but not the reverse direction of using program synthesis to solve reachability. We intend to apply advances in automatic program synthesis and repair to find test inputs to reach nontrivial program locations using the constructive proof given in Chapter 5.

6.3 Final Remarks

We believe that this research was a successful step toward making software more reliable. The development of invariant generation offers programmers the ability to understand and verify programs containing complex properties such as nonlinear polynomial, disjunctive, and array relations. The equivalent theorem opens doors to a profitable cross-fertilization between the two research areas of program verification and synthesis, e.g., leveraging verification techniques to synthesize program repairs. We hope that many of the ideas, techniques, and tools in this research will be adopted in the future, and that they will ultimately contribute toward more productive developers and more reliable software.

References

P. A. Abdulla and I. Potapov, editors. *International Workshop on Reachability Problems*, volume 8169 of *Lecture Notes in Computer Science*, 2013. Springer.

R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *International Symposium on Dependable Computing*, pages 39–46. IEEE, 2006.

X. Allamigeon. *Static analysis of memory manipulations by abstract interpretation: Algorithmics of tropical polyhedra, and application to abstract interpretation*. PhD thesis, École Polytechnique, France, 2009.

X. Allamigeon and R. D. Katz. Minimal external representations of tropical polyhedra. *Journal of Combinatorial Theory, Series A*, 120(4):907–940, 2013.

X. Allamigeon, S. Gaubert, and É. Goubault. Inferring min and max invariants using max-plus polyhedra. In *Static Analysis Symposium*, pages 189–204. Springer, 2008.

S. Anand, C. S. Păsăreanu, and W. Visser. JPF–SE: A symbolic execution extension to Java Pathfinder. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis*, pages 261–272. ACM, 2008.

REFERENCES

T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Notices*, 37(1):1–3, 2002.

T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

J. Barnes. *High integrity software: the SPARK approach to safety and security.* Addison-Wesley, 2003.

D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5-6): 505–525, 2007.

M. Bickford, C. Kreitz, R. Van Renesse, and R. Constable. An experiment in formal design using meta-properties. In *DARPA Information Survivability Conference*, volume 2, pages 100–107. IEEE, 2001.

A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without BDDs.* Springer, 1999.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Languages Design and Implementation*, pages 196–207, 2003.

R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC–an automatic debugging environment for C programs. In *Hardware and Software: Verification and Testing*, pages 260–265. Springer, 2013.

R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Programming Language Design and Implementation*, pages 321–333. ACM, 2000.

REFERENCES

T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Transactions on Computers*, 100(8):677–691, 1986.

J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.

C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, volume 8, pages 209–224. USENIX Association, 2008a.

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *Transactions on Information and System Security*, 12(2):10, 2008b.

E. R. Carbonell. *Automatic generation of polynomial invariants for system verification*. PhD thesis, Technical University of Catalonia, Barcelona, Spain, 2006.

E. R. Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

REFERENCES

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

E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 1999.

E. Cohen. *Programming in the 1990s: An introduction to the calculation of programs.* Springer, 1990.

J.-P. Comet. Application of max-plus algebra to biological sequence comparisons. *Theoretical computer science*, 293(1):189–217, 2003.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT Press, 2001.

A. Cortesi. Widening operators for abstract interpretation. In *International Conference on Software Engineering and Formal Methods*, pages 31–40. IEEE, 2008.

A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures*, 37(1):24–42, 2011.

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, pages 106–130, 1976.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM, 1977.

P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

REFERENCES

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, pages 84–96. ACM, 1978.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming.* Academic Press, 1972.

V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering*, pages 550–554. IEEE, 2009.

M. Daniel-Cavalcante, M. F. Magalhaes, and R. Santos-Mendes. The max-plus algebra and the network calculus. In *International Workshop on Discrete Event Systems*, pages 433–438. IEEE, 2006.

G. Dantzig. *Linear programming and extensions.* Princeton Press, 1998.

G. B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: algorithms and applications.* Springer, 1997.

L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.

REFERENCES

N. Dershowitz and Z. Manna. Inference rules for program annotation. In *International Conference on Software Engineering*, pages 158–167. ACM, 1978.

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

H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Static Analysis Symposium*, pages 351–368. Springer, 2011.

M. Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, Mar. 1997.

B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification*, pages 81–94. Springer, 2006.

B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

T. Emerson and M. H. Burstein. Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In *International Conference on Automated Software Engineering*, volume 99, page 267. IEEE, 1999.

M. D. Ernst. *Dynamically detecting likely program invariants*. PhD thesis, University of Washington, 2000.

REFERENCES

M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458. ACM, 2000a.

M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering program invariants involving collections. Technical report, University of Washington, 2000b.

M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering*, 27(2):99–123, 2001.

M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, pages 35–45, 2007.

R. Farebrother. *Linear least squares computations*. Marcel Dekker, Inc., 1988.

E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Conference on Genetic and Evolutionary Computation*, pages 965–972. ACM, 2010.

J. Feret. Static analysis of digital filters. In *Programming Languages and Systems*, pages 33–48. Springer, 2004.

C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Notices*, 37(5):234–245, 2002.

R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

REFERENCES

S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Conference on Genetic and Evolutionary Computation*, pages 947–954. ACM, 2009.

J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows System Symposium*, pages 59–68, 2000.

Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *Transactions on Software Engineering*, 1(1):68–75, 1975.

P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.

P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, pages 151–166, 2008.

D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188. Springer, 2011.

P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, pages 169–192. Springer, 1991.

S. Gulwani. *Program analysis using random interpretation*. PhD thesis, University of California, Berkeley, 2005.

REFERENCES

S. Gulwani. Program verification as probabilistic inference. In *Principles of Programming Languages*, pages 277–289. ACM, 2007.

N. Gupta and Z. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *International Conference on Automated Software Engineering*, pages 49–58. IEEE, 2003.

S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301. ACM, 2002.

M. Harder, J. Mellen, and M. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering*, pages 60–71. ACM, 2003.

B. Heidergott and J. W. van der Woude. *Max Plus at work: modeling and analysis of synchronized systems: a course on Max-Plus algebra and its applications*. Princeton Press, 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14:39–45, 1971.

B. Jeannet. Interproc analyzer for recursive programs with numerical variables, 2014. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.

R. Jhala and R. Majumdar. Software model checking. *Computing Surveys*, 41(4):21, 2009.

G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, pages 389–400. ACM, 2011.

REFERENCES

B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification*, pages 226–238. Springer, 2005.

J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated Software Engineering*, pages 273–282. IEEE, 2005.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

D. Jovanović and L. De Moura. Solving non-linear arithmetic. In *Automated Reasoning*, pages 339–354. Springer, 2012.

T. Kahsai and C. Tinelli. PKIND: a parallel k-induction based model checker. In *International Workshop on Parallel and Distributed Methods in Verification*, pages 55–62, 2011.

T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NASA Formal Methods*, pages 192–206. Springer, 2011.

R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, et al. Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In *Computer Aided Verification*, pages 414–429. Springer, 2009.

D. Kapur, Z. Zhang, M. Horbach, H. Zhao, Q. Lu, and T. Nguyen. Geometric quantifier elimination heuristics for automatically generating octagonal and max-plus invariants. In *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788, pages 189–228. Springer, 2013.

R. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010.

REFERENCES

M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6: 133–151, 1976.

Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance*, pages 736–743. IEEE, 2001.

S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *International Symposium on Software Testing and Analysis*, pages 105–116. ACM, 2009.

D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811. ACM, 2013.

R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer Aided Design*. IEEE, 2011.

R. Könighofer and R. Bloem. Repair with on-the-fly program analysis. In *Hardware and Software: Verification and Testing*, pages 56–71. Springer, 2013.

C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13. ACM, 2012.

C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.

REFERENCES

N. G. Leveson. An investigation of the Therac-25 accidents. *Computer*, 26:18–41, 1993.

G. Li, I. Ghosh, and S. P. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Computer Aided Verification*, pages 609–615. Springer, 2011.

B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, pages 141–154. ACM, 2003.

B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26. ACM, 2005.

Z. Manna and R. Waldinger. Synthesis: Dreams -> programs. *Transactions on Software Engineering*, 5(4):294–328, 1979.

Z. Manna and R. Waldinger. A deductive approach to program synthesis. *Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

A. Miné. *Weakly relational numerical abstract domains*. PhD thesis, École Polytechnique, France, 2004.

REFERENCES

A. Miné. The octagon abstract domain. *Higher Order Symbolic Computation*, 19(1):
31–100, 2006.

M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535.
ACM, 2001.

G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*,
pages 213–228. Springer, 2002.

J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer journal*, 7(4):308–313, 1965.

H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Software Engineering*, pages 772–781. ACM, 2013.

T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest. Using execution paths to evolve software patches. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 152–153. IEEE, 2009.

T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering*, pages 683–693. IEEE, 2012.

T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *Transactions on Software Engineering Methodology, to appear*, 2014a.

T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. From program synthesis to program reachability: Automatic program repair using test input generation. In *Symposium on Foundation of Software Engineering, submitted*. IEEE, 2014b.

REFERENCES

T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering*, pages 608–619. IEEE, 2014c.

J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276, 2001.

J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis*, pages 232–242. ACM, Jul 2002.

P. Nuzzo, A. Puggelli, S. A. Seshia, and A. Sangiovanni-Vincentelli. CalCS: SMT solving for non-linear convex constraints. In *Formal Methods in Computer-Aided Design*, pages 71–80. IEEE, 2010.

S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, volume 607, pages 748–752. Springer, 1992.

J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. *SIGSOFT Software Engineering Notes*, 29(6):23–32, 2004.

J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, pages 87–102. ACM, 2009.

A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Automata, Languages and Programming*, pages 652–671. Springer, 1989.

C. Popeea and W. Chin. Inferring disjunctive postconditions. In *Advances in Computer Science*, pages 331–345. Springer, 2007.

REFERENCES

Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 191–201. ACM, 2013.

H. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, Jan. 2007.

M. Roozbehani, E. Feron, and A. Megrestki. Modeling, optimization and computation for software verification. In *Hybrid Systems: Computation and Control*, pages 606–622. ACM, 2005.

S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking and Abstract Interpretation*, pages 25–41. Springer, 2005.

S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*, pages 3–17. Springer, 2006.

K.-D. Schubert, W. Roesner, J. M. Ludden, J. Jackson, J. Buchert, V. Paruthi, M. Behm, A. Ziv, J. Schumann, C. Meissner, et al. Functional verification of the IBM POWER7 microprocessor and POWER7 multiprocessor systems. *IBM Journal of Research and Development*, 55(3):10–1, 2011.

R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path

model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems*, pages 574–592. Springer, 2013a.

R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis Symposium*, pages 388–411. Springer, 2013b.

M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 127–144. IEEE, 2000.

A. Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.

A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. *SIGPLAN Notices*, 40:281–294, 2005.

A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGARCH Computer Architecture News*, 34:404–415, 2006.

A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *Programming Language Design and Implementation*, pages 167–178. ACM, 2007.

M. Spivak. *Calculus*. Cambridge Press, 2006.

S. Srivastava. *Satisfiability-based program reasoning and program synthesis*. PhD thesis, University of Maryland, 2010.

REFERENCES

S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Principles of Programming Languages*, pages 313–326. ACM, 2010.

S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013.

W. A. Stein et al. Sage Mathematics Software, 2014. `http://www.sagemath.org`.

N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Principles of Programming Languages*, pages 132–143. ACM, 1977.

G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

N. Tillmann and J. de Halleux. PEX, a white box test generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153. Springer, 2008.

D. Turner, S. Entwisle, O. Friedrichs, D. Hanson, M. Fossi, D. Ahmad, S. Gordon, P. Szor, and E. Chien. Symantec Internet security threat report. *Trends for January*, 1, 2006. URL `http://www.symantec.com/specprog/threatreport/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`.

L. G. Valiant. Learning disjunction of conjunctions. In *International Joint Conferences on Artificial Intelligence*, pages 560–566. AAAI, 1985.

M. Vaziri and G. Holzmann. Automatic detection of invariants in SPIN. In *SPIN Model Checking and Software Verification*. Springer, 1998.

B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2): 102–113, 1974.

REFERENCES

Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72. ACM, 2010.

W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367. IEEE, 2009.

W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering*, pages 356–366. IEEE, 2013.

D. Wisniewski. P versus NP question, 2006.

J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *Computing Surveys*, 41(4):19, 2009.

T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *International Conference on Automated Software Engineering*, pages 40–48. IEEE, 2003.

X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save RAM. *SIGPLAN Notices*, 44(7):60–69, 2009.

X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer. Formal verification by reverse synthesis. In *Conference on Computer Safety, Reliability, and Security*, pages 305–319. Springer, 2008.

X. Yin, J. C. Knight, and W. Weimer. Exploiting refactoring in formal verification. In *International Conference on Dependable Systems and Networks*, pages 53–62. IEEE, 2009.