5-31-2023

Mapping programs to equations

Hessamaldin Mohammadi
New Jersey Institute of Technology, hm385@njit.edu

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

MAPPING PROGRAMS TO EQUATIONS

by
Hessamaldin Mohammadi

Extracting the function of a program from a static analysis of its source code is a valuable capability in software engineering; at a time when there is increasing talk of using AI (Artificial Intelligence) to generate software from natural language specifications, it becomes increasingly important to determine the exact function of software as written, to figure out what AI has understood the natural language specification to mean. For all its criticality, the ability to derive the domain-to-range function of a program has proved to be an elusive goal, due primarily to the difficulty of deriving the function of iterative statements. Several automated tools obviate this difficulty by unrolling the loops; but this is clearly an imperfect solution, especially in light of the fact that loops capture most of the computing power of a program, are the locus of most of its complexity, and the source of most of its faults. This dissertation investigates a three-step process to map a program written in a C-like language into a function from inputs to outputs, or from initial states to final states. The semantics of iterative statements are captured (while loops, repeat loops, for loops), including nested iterative statements, by means of the concept of invariant relation; an invariant relation is a reflexive transitive relation that links program states separated by an arbitrary number of iterations.

But the function derived for large and complex programs may be too unwieldy to be useful, not unlike drinking from a fire hose. In order to enable the user to query the program at scale, four functions are proposed. We propose four functions: *Assume()*, which enables the user to make assumptions about program states or program parts; *Capture()*, which enables the user to capture the state of the program at some label of the function of some program part; *Verify()*, which enables the user to verify a unary assertion about the state

of the program at some label, or a binary assertion about a program part; and *Establish()*, which is envisioned to use program repair techniques to modify the program so as to make a Verify() query return true.

MAPPING PROGRAMS TO EQUATIONS

by
Hessamaldin Mohammadi

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2023

APPROVAL PAGE

MAPPING PROGRAMS TO EQUATIONS

Hessamaldin Mohammadi

Dr. Ali Mili, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. Iulian Neamtiu, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Ioannis Koutis, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Hai Phan, Committee Member — Date
Associate Professor of Data Science, NJIT

Dr. Tiantian Wang, Committee Member — Date
Professor of Computer Science and Technology, Harbin Institute of Technology, Harbin, China

---

Dr. Ye Yang, Committee Member — Date
software Project Manager, Amazon , New York, US

BIOGRAPHICAL SKETCH

Author: Hessamaldin Mohammadi

Degree: Doctor of Philosophy

Date: May 2023

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey institute of Technology, Newark, NJ, 2023
- Master of Science in Software Engineering,
  AmirKabir University of Tehran, Tehran, Iran, 2014
- Bachelor of Science in Software Engineering,
  Shahid Bahonar University of Kerman, Kerman, Iran, 2012

Major: Computer Science

Presentations and Publications:

H. Mohammadi, W. Ghardallou, R. Linger, A. Mili, "Computing Program Functions", *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, 2022.

H. Mohammadi, W. Ghardallou, A. Mili, "Assume, Capture, Verify, Establish: Ingredients for Scalable Software Analysis", *IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2021.

H. Mohammadi, W. Ghardallou, R. Linger, A. Mili, "Function Extraction: Mapping Programs Into Mathematica Equations", *Wolfram Technology Conference*, 2022.

W. Ghardallou, H. Mohammadi, R. Linger, A. Mili, "Invarint Relations for Affine Loops", *Science of Computer Programming*, under revision

To My Beloved Parents

ACKNOWLEDGMENT

Throughout the writing of this dissertation, I have received a great deal of support and assistance. First, I would like to thank my supervisor, Dr. Ali Mili, for his guidance and invaluable advice in formulating research questions and methodologies.

I would like to express my appreciation to Dr. Iulian Neamtiu, Dr. Ioannis Koutis, Dr. Hai Phan, Dr. Tiantian Wang, and Dr. Ye Yang for taking the time to serve as committee members and for their insightful feedback.

My thanks to Dr. Ziavras and the Department of Computer Science for providing me with financial support.

In addition, I appreciate the kind assistance and academic advice provided by Dr. Reza Curtmola, Dr. Baruch Schieber, Ms. Angel Butler, and Ms. Kathy Thompson in the Computer Science department.

I am also thankful for all the support I received from my fellow researchers, Dr. Wided Ghardalou and Hadi Ghahremannezhad.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter Page

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

LITERATURE REVIEW

1.1 Background

Ensuring that a program is correct with respect to a specification, involves the ability to write a valid/ vetted specification, and to match it against a candidate program through a meticulous analysis of the source code. This, in turn, requires a detailed definition of the semantics of the programming language, and the application of precise criteria of refinement. One feasible solution for this problem is to extract the function of a program through a static analysis. In the first chapter, we review the research around programs verification, loop invariant and symbolic execution solvers.

1.2 Verification and Validation

Despite several decades of research and development, and several attempts at technology transfer, the routine correctness verification of software artifacts remains largely an elusive goal. Several reasons can be cited for this failure [2, 90]:

1. **The complexity of the task**: To prove the correctness of a program requires that we have a complete, valid, detailed specification of what the program is supposed to do; this alone is a significant obstacle in practice, as generating specifications involves a multitude of stakeholders, requirements, and constraints. In addition, one must capture the semantics of the program (a challenging task in light of the size and complexity of typical programs), and must use a definition of refinement to match the program against the specification. [54, 82]
2. **The lack of automated tools**: The only way to control the complexity of the correctness verification tasks is to deploy automated tools. But some aspects of correctness verification are not automatable, and require creative human intervention. This is the case of loop invariant generation, which has mobilized a vast amount of research effort, with limited success. [4, 36, 42, 64, 83]
3. **The Lack of Qualified Personnel**: Learning about program correctness and correctness verification several years after learning to program is better than nothing,

but is not very good; by the time students are exposed to ideas of program correctness, they have been writing programs for several years, and have acquired programming habits and reflexes that do not integrate correctness concerns; they may perceive correctness concerns as a distraction that interferes with their operations and affects their productivity.

Before we get started, it is worthy to have a quick look at the definition of verification which has been a matter of debate since the early days of software invention [1]. In the software terminology, there are two terms tied to software quality which plays a key role to achieve the high quality and desired product, the verification and validation of the software. The term verification refers to satisfying the formal specification [53]. In a simple explanation, verification investigate if the software developer is building the product correctly and right, means to check whether we are on right track to develop the software or not. It generally involves formal checking, and reviewing the codes and designs without practical testing. The verified software must meet the requirements which has been set out at the start of production phase. On the other hand, the validation is about what the client is supposed to receive. Hailpern in [53] well defines the validation as: ”validation is the process of evaluating software, at the end of the development process, to ensure compliance with requirements.” In, validation phase, the team usually apply different methods to test the software, such as functionality, usability and performance test.

1.3 Abstract Interpretation

One of the general techniques for programs verification is abstract interpretation. This a framework proposed by Cousot in the 1970s [24]. Based on his definition: ”The abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems, for example, computations of programs. In particular, program analysis algorithms can be constructively derived from these abstract semantics.” This provides a practical and generic approach for static analysis, and recently has become a favourite methodology for defining and formalizing approximation computations across many various fields of computer science, like for instance in model checking [21], verification of distributed

memory systems [51], security [87], type inference [81] and theorem proving [92]. However, checking the undecidability of a program is still an interesting property in programs. The halting problem is a simple example of undecidability of a program [10]. Most program properties can be reduced to the halting problem. Therefore, we need to rely on the approximations. The approximations could be sound (correct), meaning the system gives a definite answer which is *true*, or the system responds as *maybe* which is not complete. To be able to prove the correctness of the analyses, these approximations must be formalized. In other words, Abstract Interpretation is a theory of approximation. The most popular type of abstract interpretation imitates forward program execution in such a way that the set of potential run-time states for all potential input values are finally over-approximated with matching abstract values. Thus, abstract interpretation offers a technique for creating invariants; it does so by generating a valid invariant in a finite amount of time for each program point, including the beginning and end of loop bodies [3]. There is practical tool to build static analyzer based on abstract interpretation from facebook, called SPARTA, which tries to simplify the engineering process of abstract interpretation [38].

1.4 Symbolic Execution

Symbolic execution was introduced in order to concisely describe the inputs that cause each component of a program to be executed or ”covered” [3]. Its early versions were developed in same period of time as abstract interpretation [11]. Unlike abstract Interpretation that each element of the abstract domain is an abstract value and approximates a group of “concrete” values, i.e., values that a variable can take during the program execution, Symbolic Execution uses symbolic expressions without concrete values to discover the potential program paths and reasoning about the conditions that goes to this or that branch [3]. Each path reflects a boolean relation and describes the conditions that are satisfied along with that path. There is also a symbolic memory store that maps the space of programs into symbolic expressions [5]. To verify the paths, a model checker is usually used to check if there is any violation along

the path or not. These model checkers, such as Z3, are basically a satisfiability modulo theories (SMT) solver [6,26]. Numeric challenges related to symbolic execution is identified in [5]. Due to huge cost for testing all of the paths, in traditional approaches, the execution engine solves one path at time. This needs to prioritize the paths to execute the most promising path first with some techniques like DFS or BFS. Other heuristic approaches are presented in [13, 14, 17]. Since symbolic execution is typically used as a complement to testing, or to support testing (e.g. test data selection), it has eluded researchers for decades. It can help us to determine whether a certain properties is violated or not. The aspect of interest could be checking of division by zero, NULL pointer dereferencing, any backdoor existence, and so on. [5, 11, 57, 63]

1.5 Concolic Testing

Symbolic execution generally has been advocated as a method of developing small sets that produce high coverage, or compact test input suites. But, a variety of problems have prevented it from being widely used.

1. Among these, we mention: The source program may call library functions or make system calls.
2. The underlying constraint solver may not be efficient or expressive enough to solve a given path condition.
3. Even simple programs tend to generate huge numbers of paths.

Concolic testing (or dynamic symbolic execution) was introduced by [49] to solve the first problem. Concolic, a combination of the words ”concrete” and ”symbolic,” refers to testing that maintains both concrete and symbolic states while running a program. Consequently, symbolic variables must be seeded with actual values. Concolic testing keeps track of alternate path limitations while it runs. This may result in new execution paths. To determine which are practical and to offer fresh and tangible inputs for the upcoming exploration path, constraint solvers are used [3].

Formally, we can define a concolic trace as a quadruple $(\rho, \pi, \sigma, \phi)$ where $\rho : Var \rightarrow Val$ is a concrete state and (π, σ, ϕ) is a symbolic trace. The concrete state ρ defines assignments of concrete values to symbolic variables. Concolic testing and symbolic execution share a lot of similarities in terms of semantics. Because symbolic variables now accept tangible values, the primary distinction is that we actually run the program. Concolic execution is therefore wholly deterministic. The conditions at each branch point are evaluated at runtime as the program is being run, hence the constraint solver is not used to do so. As opposed to that, it is employed to produce the actual values that will be applied to the relevant symbolic variables in the subsequent concolic iteration.

Some of the problems with symbolic execution are addressed by Concolic testing. In fact, by directly calling every function, we may execute concolic testing without modeling library functions, system calls, or external applications. Due to the simplification of constraints caused by the substitution of concrete values for symbolic variables, constraint solving is more lightweight. We can just ignore or approximate unsupported restrictions, which is significant. Completeness is compromised in this method, but symbolic execution can go more easily as a result. This is completely fine in many practical applications as long as "good enough" coverage is attained in a timely manner.

1.6 Model Checking

As software systems get more complicated, the conventional methods for programs verification could not be a applicable. Model checking as a new approach was introduced to remove the manual proofs and provide a fully trusted technique [60]. The model checking originally proposed by Clarke and Emerson, and separately by Sifakis [20], for verifying temporal properties of finite state systems. This described in [20] as: "This verification technology provides an algorithmic means of determining whether an abstract model—representing, for example, a hardware or software design—satisfies a formal specification expressed as a temporal logic (TL) formula. Moreover, if the property does

not hold, the method identifies a counterexample execution that shows the source of the problem."

A model checker's inputs consist of a description of the system to be analyzed(often in finite states) as well as a number of expected attributes, which are frequently stated as formulae of temporal logic. The model checker indicates any violations or validates that the properties are true. In the former situation, it offers a run that transgresses the property as a counter-example. Such a run might offer insightful feedback and highlight design flaws [16].

Several branches of computing have benefited from model checking theories and tools.. Wing in [108] presented algorithms to show how model checking can reveal some possible attacks in computer networks. In this research, not only one counter example is discovered, but they try to capture all the counterexamples, called a scenario graph. Each path in this graph is considered as an attack. Similar works has been done related to security of systems such as White [107] which compares two security models, and [80] which proposes a secure method for DoS detection in wireless sensor networks. For database systems and operations, model checking may also be employed. Chomicki in [19] tries to connect both temporal databases and model checking verification. The input would be all possible executions of the system and a query. In order to systematically examine all program pathways, including those whose execution depends on data supplied by database queries, Emmi and Majumdar [35] proposes concolic execution. Their method generates both the program's input data and relevant database records. To be able to test database applications, Marcozzi in [73] define symbolic execution of SQL statements coupled with other limitations in a program for generating test inputs.

The ability to evaluate large-scale systems using model checking necessitates the use of effective data structures to represent items like sets of system states and transition systems. These systems can be encoded as a set of binary variables like $\{b_1, ..., b_n\}$. To achieve this, a propositional formula over $\{b_1, ..., b_n\}$ could represents the system states, and for the actions that shows the relevant actions for a pair of states such as (s, t) could

be described as $\{b_1, ..., b_n, b'_1, ..., b'_n\}$. The unprimed variables shows the pre-state s and the primed variables represent the post-state t [74]. The size of the representing formula rely on the structure of the represented set but not on its size: for instance, the empty set and the set of all states are represented by false and true, both of size 1. For this reason, these representations are usually called symbolic, and model checking methods that work on symbolic representations are called symbolic model checking [12]. This feature makes symbolic model checking appropriate for hardware designs and other systems with a high branching degree [45].

Out of all these advantages, there are some limitations that makes model checking an idealistic but not realistic technique. We may have limited resources to analyze part of the system not the whole. Even if the specification verified, there is a chance for some hidden bugs in the system due to some simplification of the features. On the other hand, a counter example might be reported because of wrong specification and not related to the actual system [74].

1.7 Invariant Generation

One of the main challenges of symbolic execution solvers is loop summarizing, there could be an endless number of created branches if the loop condition involves one or more symbolic values. The explanation behind this problem, also known as path explosion, is because each iteration of the loop can be thought of as an if-goto statement, which results in a conditional branch in the execution tree [5]. To address this issue, partial summarizations is presented in [50]. By invoking the dependencies between the conditions of the loop and the symbolic variables, a loop summary uses pre- and post-conditions that are dynamically produced throughout the symbolic execution. The symbolic engine can eliminate repeated loop executions in the same program state by caching loop summaries, but it also makes it possible to generalize the summary to encompass various loop executions under various conditions. Early work were unable to handle multi-path loops or nested loops, which

are loops with branches within their bodies. In [112] a general framework, called Proteus, proposed for summarizing multi-path loops. It categorizes loops based on the ways in which the values of the path conditions change over time and how the paths are interleaved within the loop. The compaction technique described in [101] has a distinct flavor, since it produces templates that declaratively represent the program states produced by a chunk of code as a compact symbolic execution tree by analyzing cyclic paths in the control flowgraph. The symbolic execution engine can explore a substantially smaller set of program states by utilizing templates. The use of templates, which add quantifiers into the path constraints, has the disadvantage of potentially placing a heavy weight on the constraint solver. One of the remedies proposed to solve the loop complexities is loop invariants, a logical equation that serves as an abstract description of a loop. The term invariant simply means an assertion that holds before and after each iteration of the loop. let's take a look at the Hoare [55] definition: "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion". We simply call P and R as pre-condition and post-condition respectively. have a simple loop as an example:

```
while (i <= 100)
{
x++;
y++;
}
```

one invariant relation for this loop is $I : y - x == y' - x'$; here (x,y) denotes the pre-condition and (x', y') the post-condition. This relation is true for the loop above, despite how man times we iterate over the loop.

Inferring loop invariants has been approached in a variety of ways throughout the literature [18, 23–25, 62, 65, 93] . The benefits of program verification technologies are, however, severely constrained by the intractable nature of the issue and the difficulty of even solving practical cases. A decision trees strategy which is proposed in [46] learn loop invariants with simple linear features in the form of $x + y \leq c$ where c is a constant, so

they are unable to solve for three variables like $x + y \leq z$. In LOOPINVGEN [89], these features are systematically enumerated and generalized. A stochastic approach is carried out over a number of constraint templates in [97]. Although certain features or templates work effectively in some domains for them, they might not be able to adapt to other domains. Code2Inv [98] developed a rich tool to capture the loop invariants based on reinforcement learning. However, it has two drawbacks: 1- it works based on the trial and error and decision making, hence it fails to solve wide range of problems. 2- most of the problems that CODE2INV fails to solve can be represented in a compact disjunctive normal form (DNF). However, CODE2INV is designed to produce loop invariants in the conjunctive normal form (CNF). The reduction of loop invariants from DNF to CNF could incur an exponential blowup in size. In [36] a system called Daikon presented which tries to generate the loop invariant dynamically. Leveraging some machine learning techniques, Daikon executes candidate programs and monitors their behaviour at some user-selected points before reporting assertions that were true during the observed executions. Due to the nature of experimental observations, the system produce probabilistic relations as the invariant generations. For the goal of certifying the code, Denney and Fischer in [27] examine created code against safety properties. In order to achieve this, they begin by comparing the generated code to well-known code generator idioms, which they parametrize with pertinent safety features. Invariants, such as loop invariants, are used to construct safety properties and are deduced by propagation through the code. Colón et al. in [23] take into account loop invariants of numerical programs as linear expressions and derive the coefficients of the expressions by solving a set of linear equations. They expand their approach to non-linear expressions in [94]. Through solving recurrence relations, Kovacs and Jebelean create loop invariants in [65]. They set the loop invariants as solutions to recurrence relations and derive closed forms of the solution using a theorem prover (Theorema) to aid in the process. In [93] with strong theorem proving support, Rodriguez Carbonnell et al. derive loop invariants by forward propagation and fixed point computing; they describe loop bodies

as conditional concurrent assignments. In Mili [78], the difference between conventional loop invariants and the reflexive transitive loop invariants are discussed. Aligator [58] is another tool, previously implemented as a Mathematica package [69] and redesigned in Julia programming language, intends to discover the polynomial invariants of so-called extended P-solvable loops. Loop guards and test conditions are ignored in such loops, resulting in non-deterministic loops with sequencing and conditionals [59]. The fact is Aligator is unable to find the invariants if it is not a polynomial expression. Loopfrog [105] is another approach which works based on termination analysis. This tool is a light-weight static analyzer, which extends a loop summarization algorithm [66] in accordance with abstract interpretation [24]. The key distinction between the prior strategy and loopfrog is applying the transition invariants during summarization as opposed to state invariants. Similarly LOOPUS [100] developed a tool which automatically computes loop bounds for C programs. LOOPUS uses the LLVM compiler framework [67]. The irrelevant parts of the program the termination of the analyzed loop is removed and then a SMT solver used to run the queries about the program.

1.8 Functional Extraction

Discovery of the intention of a program, or as it is called the Program Function, has been a challenging and interesting branch in software engineering [9, 15]. This, which rooted from symbolic execution, can propose interesting opportunities to improve the state of the art in software verification. The function extraction approach tries to transform a source code into some symbolic equations. For instance, for a program P and a goal G, researchers try to find an equivalent program P′ which has same output for G. This has been introduced as program specialisation in [44] which intends to get the paths' characteristic of the program and produce an abstract interpretation of that path. Similarly, [103] describes a method called lazy evaluation which implements the program exception as an abstract data type.

More specifically, evaluating the loop behaviour has been the focus of much interest by computer science researchers [75]. In [32] a heuristic approach is given to analyse and find the function of a loop. As it works based on some heuristic hypothesis, not all the evaluations could be correct. Some researches like Obdr˘z´alek, leverage the symbolic execution to find some feasible path, from a start to a end label in a code, through a loop in between [86]. Another method in [76], maps loop body statements to a concurrent structure and then derives a lower bound for the loop to finally capture the function of the loop. To find an approximation of a loop, [99] deploys a rational vector addition system with resets (Q-VASR) that simulates the behaviour of an input loop, and then uses the reachability relation of that Q-VASR.

We adopt the following definition: Given a program P on space S, we let the function of P be the set of pairs (s, s') such that if P starts execution in state s then it terminates normally (i.e. after a finite number of steps, without raising any exceptional run-time condition such as division by zero, array reference out of bounds, referencing a nil pointer, overflow, underflow, etc) in state s'; by abuse of notation, we denote programs and their functions by the same symbol.

Although transforming a program into a function(sequence of logical relation) has been done in previous works [70, 84], there are some key differences and improvements in our research by the following factors:

1. We do not focus only on loops, but we try to get the function for the whole program
2. To produce the intermediate relations, we use the Abstract Syntax Tree(AST) which comes from running the input source code through a parser. On the other hand, the previous works was done through the Concurrent Conditional Assignments.
3. To find the proper invariant relation, instead of a syntactic matching, we adopt a semantic matching approach. Hence, we don't have the difficulties of having a complex source code. We can analyze any program with any number of nested conditions and nested loops.
4. The combination of AST and semantic matching enabled us to analyze nested loops, while the previous works were limited to analyze only simple loops.

CHAPTER 2

MATHEMATICS FOR PROGRAMMING

The importance of mathematics for programming as an essential foundation is evident. Hoare believes [56] "the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic. Such methods of program construction promise benefits in specifications, systems software, safety-critical programs, silicon design, and standards." He provides four principle to for his claim:

1. Computers are mathematical machines that are perfectly calculable in every facet of their conduct.
2. Computer programs are mathematical expressions.
3. A programming language is a mathematical theory that contains concepts, notations, definitions, axioms, and theorems.
4. Programming is a mathematical activity, it requires traditional methods of mathematical understanding, calculation, and proof.

While it is not very usual to include specification in software testing and verification, there are some reasons that motivate us to discuss them:

1. Test Oracles are Built on specification:A crucial step in software testing and verification is the construction of a test oracle, which entails choosing and implementing a specification against which to test the program. This procedure is crucial in figuring out how effective the test was.
2. Testing and Relative Correctness: We cannot discuss testing without discussing faults (testing entails exposing, identifying, and/or removing faults); we cannot discuss faults without discussing relative correctness (a program from which a fault has been removed is more correct, in some sense, than the original faulty program); and we cannot discuss relative correctness without discussing correctness (as correctness is

the ultimate form of relative correctness); and we cannot discuss correctness without discussing testing (correctness is relative to a specification).

3. A Connection Between Verification and Testing: It is common to argue that static verification and dynamic testing are complementary methods for ensuring the accuracy or dependability of software products. However the results of these two methodologies must be able to be articulated inside the same general framework for complementarity to be meaningful; specifications make this feasible.

4. A Foundation for Hybrid Validation: An key point is sometimes missed in the ongoing discussion about the relative benefits of static analysis versus dynamic testing: the fact that a method is unsuccessful not because of any inherent flaw in the method, but rather because it is being applied to the wrong specification. Using testing for some aspects of the specification and static analysis for others may be a cost-effective strategy for ensuring software quality. Only when both methods are created to work with the same specification framework is this strategy feasible.

2.1 Relational Specifications

A software product's specification is a representation of the attributes that the software product must possess to serve its function. The specification is often created by identifying all pertinent parties involved in the (current or future) software product, obtaining the requirements that they need the product to satisfy, creating, merging, and putting these requirements into a single document [79]. Typically specifications pertain to operational requirements and functional requirements, the former essentially means any information about how to run the system, monitoring, Logging, resource consumption, startup/shutdown controls, while the latter is related to the actions and processes that the system is supposed to do.

As a product, a specification must meet two conditions, which are as follows:

1. Formality: The specification must be described in such a way as to show precisely what functional behavior is required.

2. Abstraction: The specification must describe what requirements the software product must satisfy, not how to satisfy them. In other words, it must focus on what candidate programs must do rather than how they must do it, the latter being the prerogative of the designer.

As a process (the process of identifying stakeholders, eliciting requirements, compiling them, etc.), a specification must meet two conditions, which are as follows:

1. Completeness: The specification must capture all the relevant requirements of the product.
2. Minimality: The specification must capture nothing but the relevant requirements of the product.

2.2 Relational Mathematics

Here we want to to explain the specification principles. Although there are many different specification languages used in research, some of which are rather commonly used in industry, we opt to utilize mathematical notation and concentrate on models rather than languages.

Relational mathematics' role for operations research and informatics has been recognized as what numerical mathematics does for engineering. It is meant to aid with computing, modeling, and reasoning. Therefore, its applications are varied, spanning from machine learning and spatial reasoning to psychology, linguistics, decision support, and ranking [96]. In line with these topics, several attempts have been made to formulate the problems that have so far been tackled using relational methods. In [95] a programming language designed to transform the problems into relations terms. [71] works on formalization of matrices and intends to build a method that connects matrices to relational algebra and computer programming. Likewise, [7] presented an algorithm for general Boolean matrix factorization.

2.2.1 Sets and relations

We provide variable names and related data types to express sets in a manner same to programming (sets of values). Using the variable declarations to represent set S, for instance:

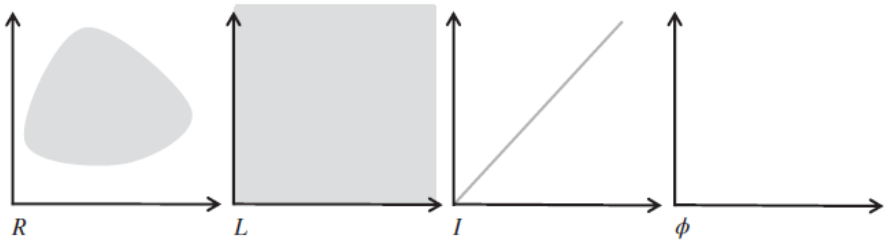


Figure 2.1 Special relations.

$$x : X; y : Y; z : Z,$$

then S is the Cartesian product X × Y × Z. Elements of S are triplets of elements of X, Y, and Z and are indicated by the lower case letter s. We denote the X, Y, and Z components of a given element s of S as x(s), y(s), and z, respectively (s). A relation on S is a subset of the Cartesian product $S \times S$; given a pair (s, s') in R, we say that s' is an image of s by R. Special relations on S include the universal relation $L = S \times S$, the identity relation I= $\{(s, s')|s' = s\}$, and the empty relation $\phi = \{\}$. We utilize the Cartesian plane to describe relationships graphically, with set S shown on the ordinates (for s) and abscissas (for s′). We represent an arbitrary relation on S, L, I, in Figure 2.1

2.2.2 Operations on relations

As a relation is a set, we are able to apply relations all the operations that are applicable to sets, such as union $(\cup)$, intersection $(\cap)$ and difference $(/)$. Moreover, the following operations are defined:

- The converse of relation R is the relation denoted by R and defined by $\hat{R} = \{(s, s')|(s, s') \in R\}$.
- The $domain$ of relation R is the subset of S denoted by $dom(R)$ and defined by $dom(R) = \{s|\exists s' : (s, s') \in R\}$.
- The range of relation R is the subset of S denoted by $rng(R)$ and defined as the domain of $\hat{R}$.

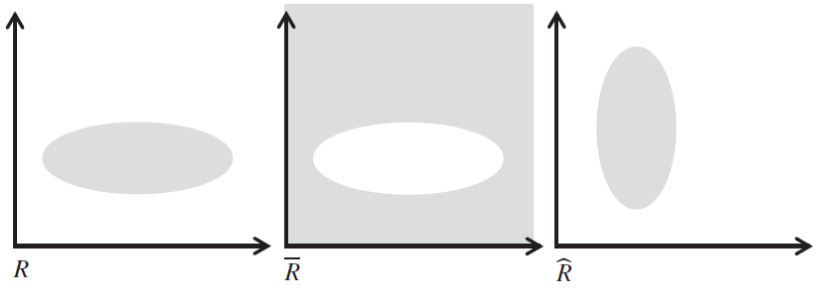


Figure 2.2 Complement and inverse.

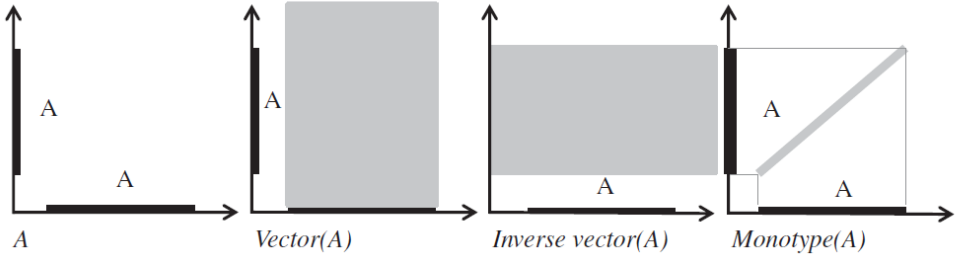


Figure 2.3 Relational representation of sets.

- The $(pre)restriction$ of R to (sub)set A is the relation denoted by ${}_{A\backslash}$R and defined by ${}_{A\backslash}R = \{(s, s')|s \in A \wedge (s, s') \in R\}$.
- The $postrestriction$ of R to (sub)set A is the relation denoted by $R_{A/}$ and defined by $R_{A/} = \{(s, s')|(s, s') \in R \ \wedge s' \in A\}$.

Figure 2.2 shows a graphic description of a relation, its complement, and its converse. Given a set A (subset of S), we define three relations of interest, which are as follows:

- The $vector$ defined by A is the relation A × S.
- The $inverse$ vector defined by A is the relation S × A.
- The $monotype$ defined by A is the relation denoted by I(A) and defined by $I(A) = \{(s, s')|s \in A \wedge s' = s\}$.

Figure 2.3 represents, for set A (a subset of S), the vector, inverse vector, and monotype defined by A.

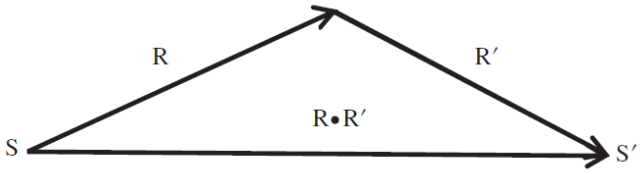


Figure 2.4 Relational product.

Given two relations R and R′, we let the product of R by R′ be denoted by $R \bullet R'$ (or RR′ , if no ambiguity arises) and defined by $R \bullet R' = \{(s, s') | \exists s" : (s, s") \in R \wedge (s", s') \in R\}$. The definition of relational product illustrates in Figure 2.4

If we denote the vector and the inverse vector defined by A by, respectively, $\omega(A)$ and $\mu(A)$, then the following identities hold, by virtue of the relevant definition:

- $\omega(A) = I(A) \bullet L$
- $\mu(A) = L \bullet I(A)$
- $\hat{\omega(A)} = \mu(A)$
- $I(A) = \omega(A) \cap I = \mu(A) \cap I$

When we want everything to be a relation, vectors are an easy (relational) approach to describe sets. As we can see in Figure 2.5, the range of relation R can be represented by the inverse vector LR and the domain of relation R can be represented by the vector RL, for instance.

2.2.3 Properties of relations

We list the following among the properties of relations:

- A relation R is said to be total if and only if $RL = L$.
- A relation R is said to be surjective if and only if $LR = L$.
- A relation R is said to be deterministic if and only if $\hat{R}R \subseteq I$.

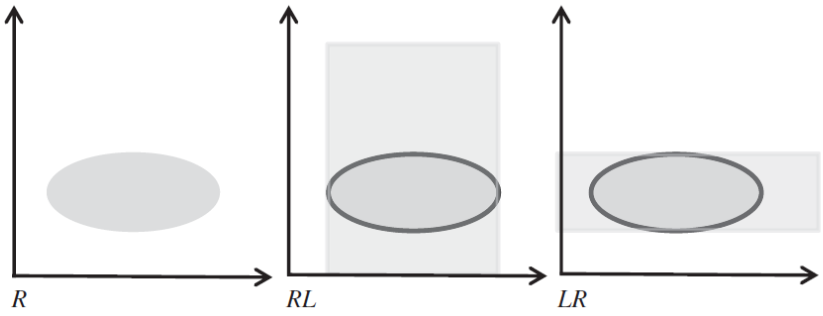


Figure 2.5 Multiplying with universal relation.

- A relation R is said to be reflexive if and only if $I \subseteq R$.
- A relation R is said to be symmetric if and only if $R \subseteq \hat{R}$.
- A relation R is said to be transitive if and only if $RR \subseteq R$.
- A relation R is said to be antisymmetric if and only if $R \cap \hat{R} \subseteq I$.
- A relation R is said to be asymmetric if and only if $R \cap \hat{R} \subseteq \phi$.
- A relation R is said to be connected if and only if $R \cup \hat{R} = L$
- A relation R is said to be an equivalence relation if and only if it is reflexive, symmetric, and transitive.
- A relation R is said to be a partial ordering if and only if it is reflexive, antisymmetric, and transitive.
- A relation R is said to be a total ordering if and only if it is a partial ordering and is connected. some of these properties are illustrated in the Figure 2.6 which are taken from [79].

2.3 Symbolic Equation Solving

The ability of modern computers to perform symbolic calculations in addition to the purely numerical calculations for which they were initially created opens up fascinating possibilities in the various engineering and scientific fields [91]. Systems that use symbolic manipulation are essentially ”expert systems” that incorporate mathematical knowledge [8]. in addition to capacity to numerical calculations, they have the ability to manipulate abstract

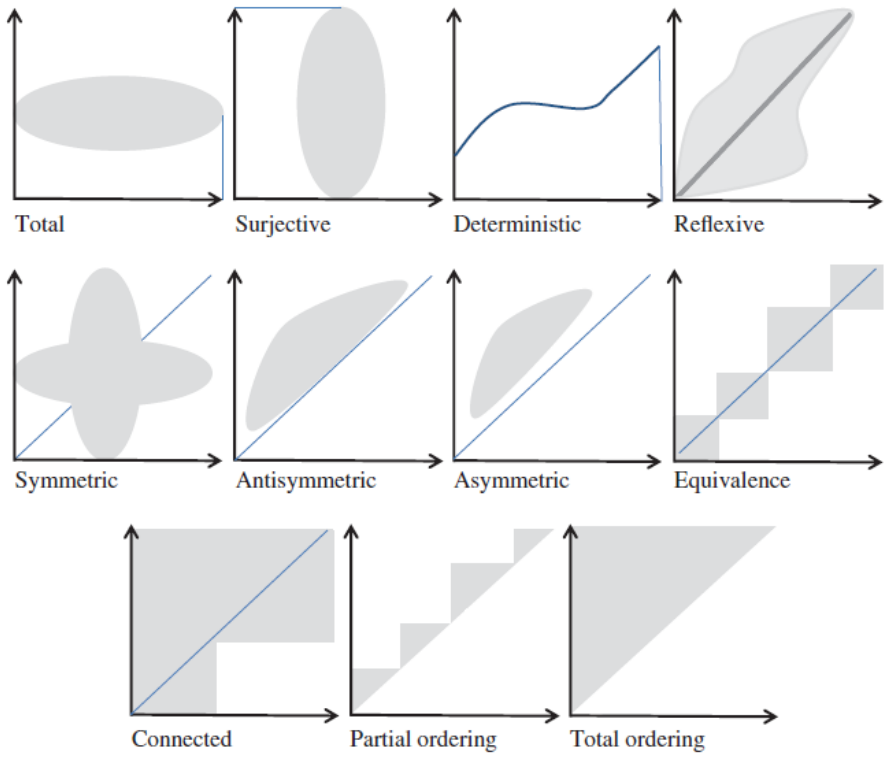


Figure 2.6 Properties of relations.

symbols that represent numerical values. Hence, they are more adaptable than conventional programming languages that simply execute numerical computations, such FORTRAN and BASIC. Recent years have seen a gradual increase in symbolic equation solver systems, in part due to the accessibility of relatively affordable yet powerful personal computers [8, 41, 85]. Three powerful systems, the older ALTRAN (ALgebra TRANslator) [22] and the more contemporary MATHEMATICA [69] and MatLab [37], can be used to show the aforementioned. Matlab and Mathematica provide Java programming API so we can call their functions from the Java applications. However, based on the multiple analysis and evaluations Mathematica is more effective than Matlab [33, 104] for our purposes. There are some reasons for this claim:

1. In terms of symbolic guiding, Mathematica is superior to Matlab and is simpler

2. Mathematica is used in Procedural, modular, object-oriented, and functional, but Matlab is used as a procedural language in maximum time

3. Mathematica excels at being a scientific calculator while Matlab is incapable of doing so

4. MatLab is optimized for use by engineer, whereas mathematica is better adapted to the needs of logicians.

2.3.1 Mathematica: modern technical computing

Mathematica is a symbolic mathematical calculation software that is also referred to as a computer algebra program. This computational engine has been used in several scientific, engineering, mathematical, and computing sectors. Stephen Wolfram invented and manage the Wolfram Research of Champaign in Illinois. The programming language used in Mathematica is The Wolfram Language which is so close to a scripting language.

To get more familiar with Mathematica, let's review some of the major capabilities and features of it [110]:

1. A Complex, Integrated System: there are more than 6,000 built-in functions in Mathematica that cover every aspect of technical computing. These functions are included in the fully integrated Mathematica system and all of them meticulously integrated so that they function flawlessly together.

2. A tool for all majors: Mathematica is applicable in multiple technical computing field, building on three decades of work, such as neural networks, machine learning, image processing, geometry, data science, visualizations and much more.

3. Algorithm Power: several of the algorithms included in Mathematica were developed with advanced development techniques and the special capabilities of the Wolfram Language.

4. Strength for Industry: Mathematica is designed to offer industrial-strength features, including parallelism, GPU processing, large-scale problems management, and strong, effective methods in all fields.

5. Powerful Ease of Use: a system that is particularly simple to use, with predictive predictions, natural language input, and more, has been made possible by Mathematica by using its computational strength as well as the thoughtful design of the Wolfram Language.

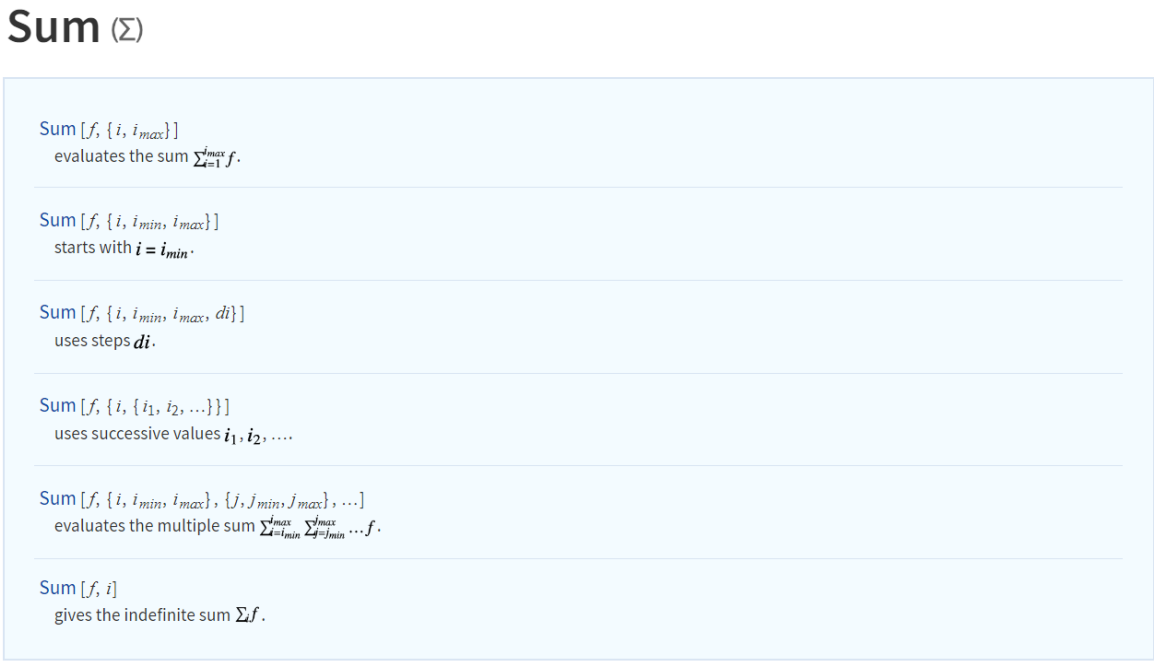


Figure 2.7 Sum overview.

6. User-friendly environment: With Mathematica, you can arrange everything you do in rich documents that include text, runnable code, dynamic images, user interfaces, and more. This is possible thanks to the Wolfram Notebook Interface.
7. Solid Cloud Integration: With a distinctive and potent hybrid cloud/desktop architecture, Mathematica is now fully connected with the cloud, enabling sharing, cloud computing, and more.

2.3.2 Arithmetic operations

In this section we review some major arithmetic functions included in Mathematicas' framework [109]:

Sum[f,i,imin,imax] : evaluates the sum $\Sigma_{i=imin}^{imax} f$ (Figure 2.7 , 2.8)

Input: **Sum [**$i^2, i, 1, n$**]**

Output: $\frac{1}{6}n(n+1)(2n+1)$

Product[f,i,imin,imax] : evaluates the product of $\Pi_{i=imin}^{imax} f$ (Figure 2.9 , 2.10)

Input: **Product [**$i^2, i, 1, n$**]**

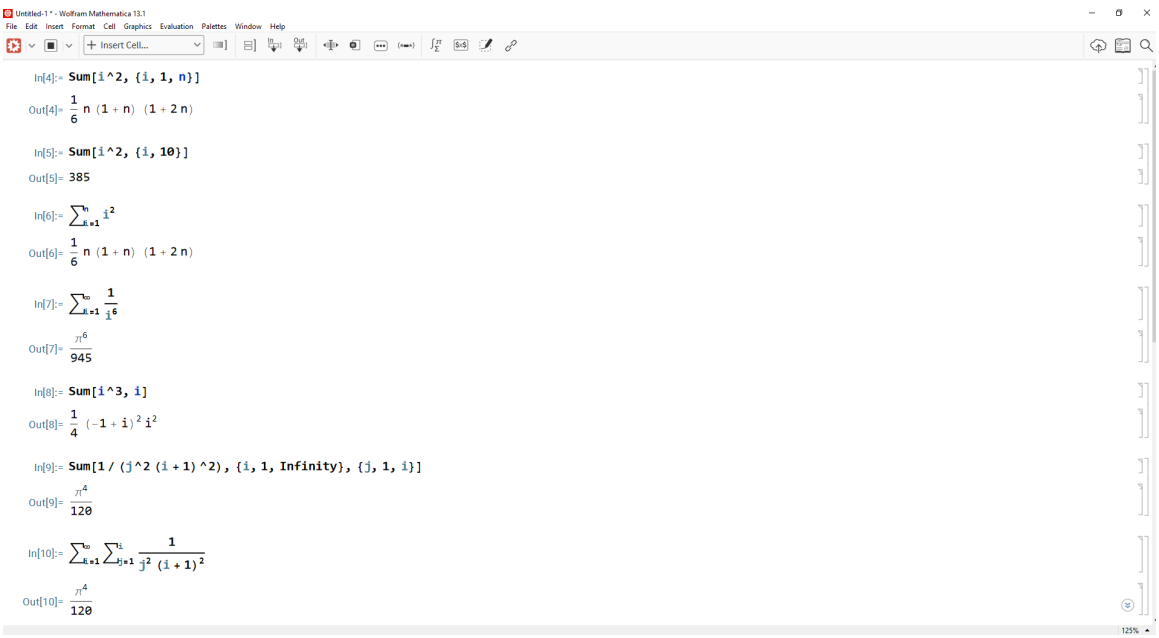


Figure 2.8 Sum examples.

Output: $(n!)^2$

Series[expr,{x,x0,n}] : Use Series to make a power series out of a function. The first argument is the function(expr). The second argument has the form $\{var, pt, order\}$, where var(x) is the variable, pt(x0) is the point around which to expand, and order is the order(n). In mathematical terms, Series can be viewed as a way of constructing Taylor series for functions [111]. (Figures 2.11 and 2.12)

Input : Series[$\mathbf{E}^x, \{x, 0, 10\}$], E is the exponential constant ϵ (base of natural logarithms), with numerical value $\simeq 2.71828$.

Output: $1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + \frac{x^{10}}{3628800} + O\left[x^{11}\right]$

Limit[f,$x \to x*$] : gives the limit $\lim_{x \to x*}$. (Figure 2.11, 2.12)

Input : $\lim_{x \to \infty} \frac{1}{x}$

Output: 0

Product (∏)

Product [*f*, {*i*, i_{max}}]
evaluates the product $\prod_{i=1}^{i_{max}} f$.

Product [*f*, {*i*, i_{min}, i_{max}}]
starts with $i = i_{min}$.

Product [*f*, {*i*, i_{min}, i_{max}, *di*}]
uses steps *di*.

Product [*f*, {*i*, {i_1, i_2, ...}}]
uses successive values i_1, i_2, ...

Product [*f*, {*i*, i_{min}, i_{max}}, {*j*, j_{min}, j_{max}}, ...]
evaluates the multiple product $\prod_{i=i_{min}}^{i_{max}} \prod_{j=j_{min}}^{j_{max}} \cdots f$.

Product [*f*, *i*]
gives the indefinite product $\prod_i f$.

Figure 2.9 Product overview.

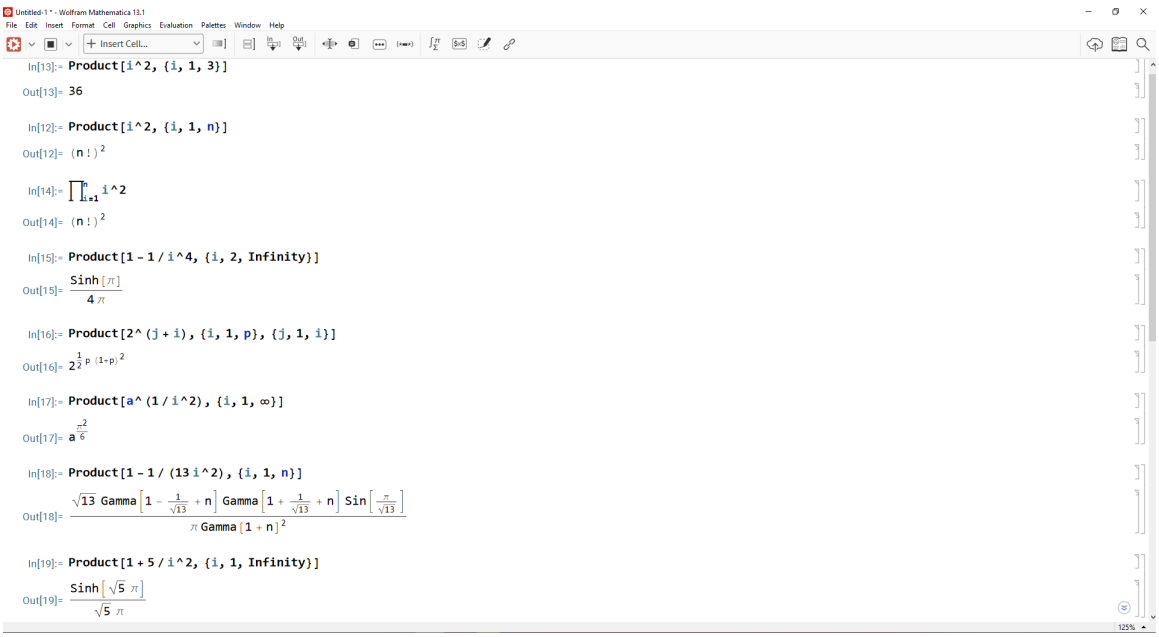


Figure 2.10 Product examples.

D[f,x] : gives the partial derivative $\partial f / \partial x$ (Figure 2.15, 2.16)

Input : D$[x^n, x]$

Series

Series [*f*, {*x*, x_0, *n*}]
generates a power series expansion for *f* about the point *x*=x_0 to order $(x\text{-}x_0)^n$, where *n* is an explicit integer.

Series [*f*, *x* → x_0]
generates the leading term of a power series expansion for *f* about the point *x*=x_0.

Series [*f*, {*x*, x_0, n_x}, {*y*, y_0, n_y}, ...]
successively finds series expansions with respect to *x*, then *y*, etc.

Figure 2.11 Series overview.

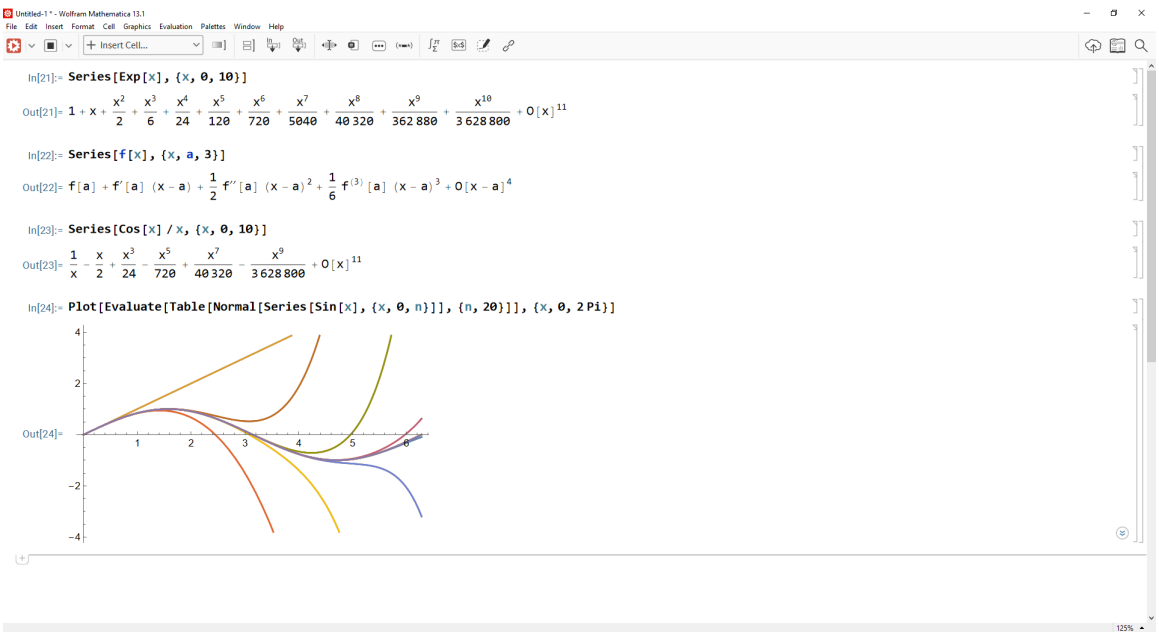


Figure 2.12 Series examples.

Output: $nx^{(-1+n)}$

2.3.3 Logical operations

Solve[expr,vars] : attempts to solve the system expr of equations or inequalities for the variables vars.(Figure 2.17 , 2.18)

Limit

Limit [f, $x \to x^*$]
gives the limit $\lim_{x \to x^*} f(x)$.

Limit [f, $\{x_1 \to x_1^*, \ldots, x_n \to x_n^*\}$]
gives the nested limit $\lim_{x_1 \to x_1^*} \cdots \lim_{x_n \to x_n^*} f(x_1, \ldots, x_n)$.

Limit [f, $\{x_1, \ldots, x_n\} \to \{x_1^*, \ldots, x_n^*\}$]
gives the multivariate limit $\lim_{\{x_1,\ldots,x_n\} \to \{x_1^*,\ldots,x_n^*\}} f(x_1, \ldots, x_n)$.

Figure 2.13 Limit overview.

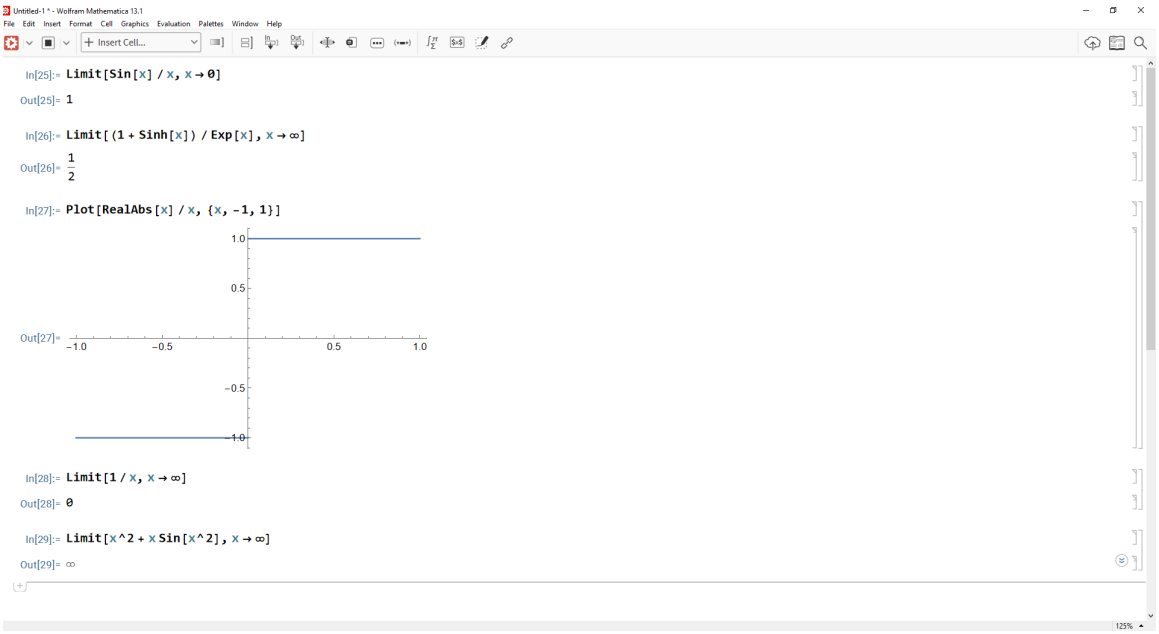


Figure 2.14 Limit examples.

Input: **Solve[**$x^2 + ax + 1 == 0, x$**]**

Output: $\left\{\left\{x \to \frac{1}{2}\left(-\sqrt{a^2-4} - a\right)\right\}, \left\{x \to \frac{1}{2}\left(\sqrt{a^2-4} - a\right)\right\}\right\}$

Reduce [expr,vars,dom] : does the reduction over the domain dom. Common choices of dom are Reals, Integers, and Complexes (2.19 , 2.20)

Input: **Reduce**[$x^2 + y^2 < 1, x, y$]

D (∂)

D [*f*, *x*]
gives the partial derivative $\partial f / \partial x$.

D [*f*, {*x*, *n*}]
gives the multiple derivative $\partial^n f / \partial x^n$.

D [*f*, *x*, *y*, ...]
gives the partial derivative $\cdots (\partial / \partial y) (\partial / \partial x) f$.

D [*f*, {*x*, *n*}, {*y*, *m*}, ...]
gives the multiple partial derivative $\cdots (\partial^m / \partial y^m) (\partial^n / \partial x^n) f$.

D [*f*, {{x_1, x_2, ...}}]
for a scalar *f* gives the vector derivative $(\partial f / \partial x_1, \partial f / \partial x_2, \ldots)$.

D [*f*, {*array*}]
gives an array derivative.

Figure 2.15 Derivation overview.

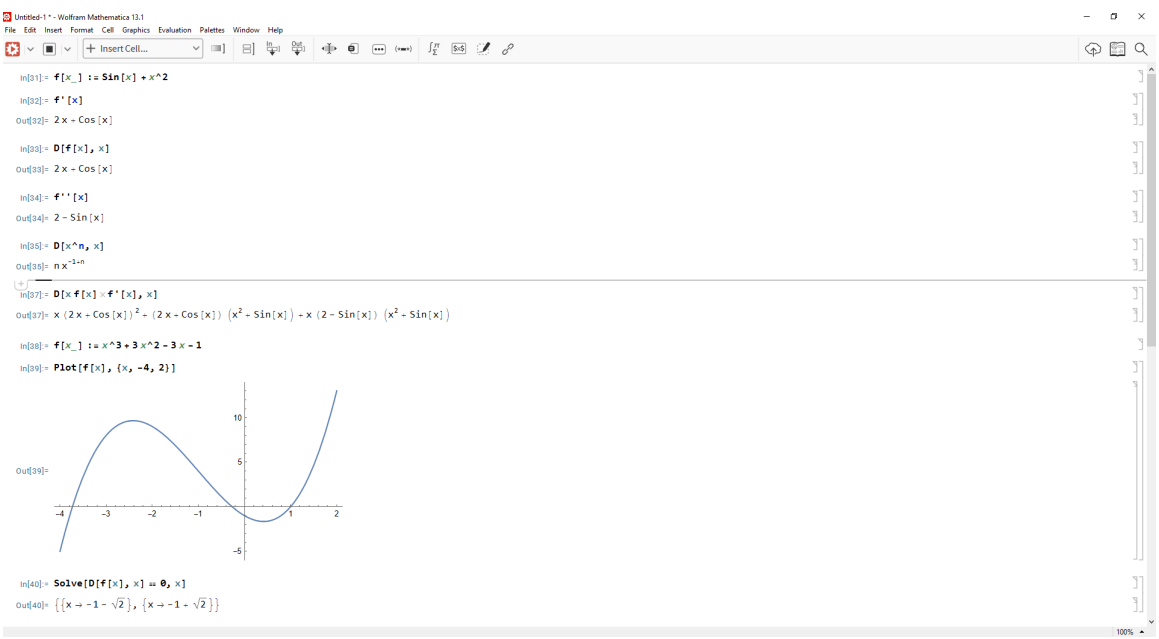


Figure 2.16 Derivation examples.

Output: $-1 < x < 1 \wedge -\sqrt{1-x^2} < y < \sqrt{1-x^2}$

Solve

Solve [*expr*, *vars*]
attempts to solve the system *expr* of equations or inequalities for the variables *vars*.

Solve [*expr*, *vars*, *dom*]
solves over the domain *dom*. Common choices of *dom* are Reals, Integers, and Complexes.

Figure 2.17 Solve overview.

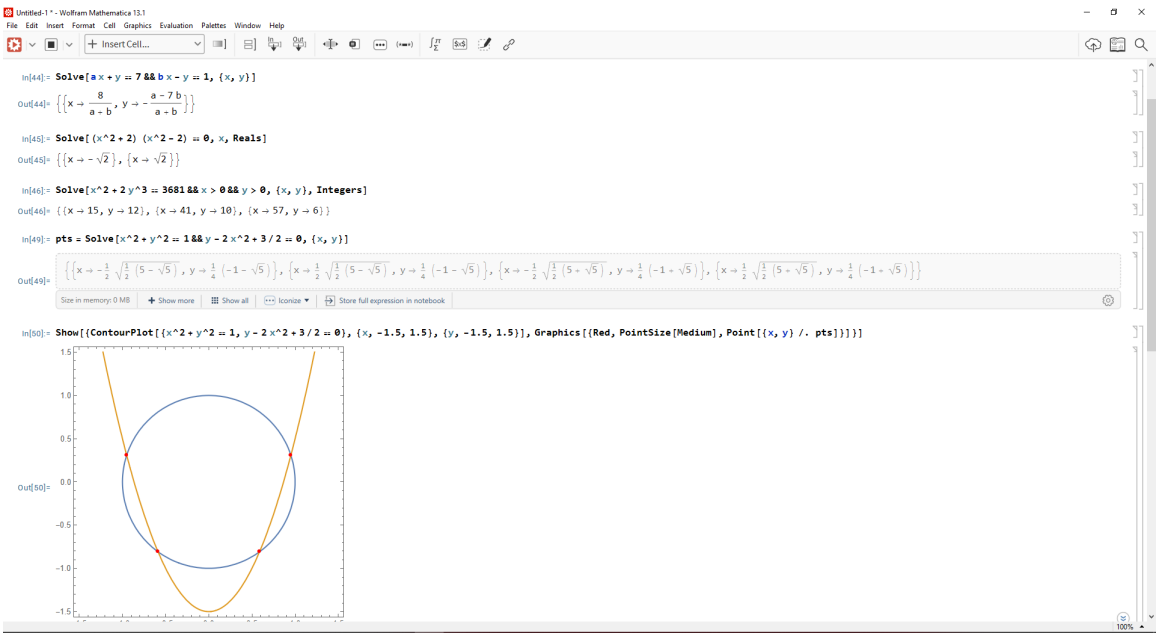


Figure 2.18 Solve examples.

Reduce

Reduce [*expr*, *vars*]
reduces the statement *expr* by solving equations or inequalities for *vars* and eliminating quantifiers.

Reduce [*expr*, *vars*, *dom*]
does the reduction over the domain *dom*. Common choices of *dom* are Reals, Integers, and Complexes.

Figure 2.19 Reduce overview.

Exists [x, cond, expr] : states that there exists an x satisfying the condition cond

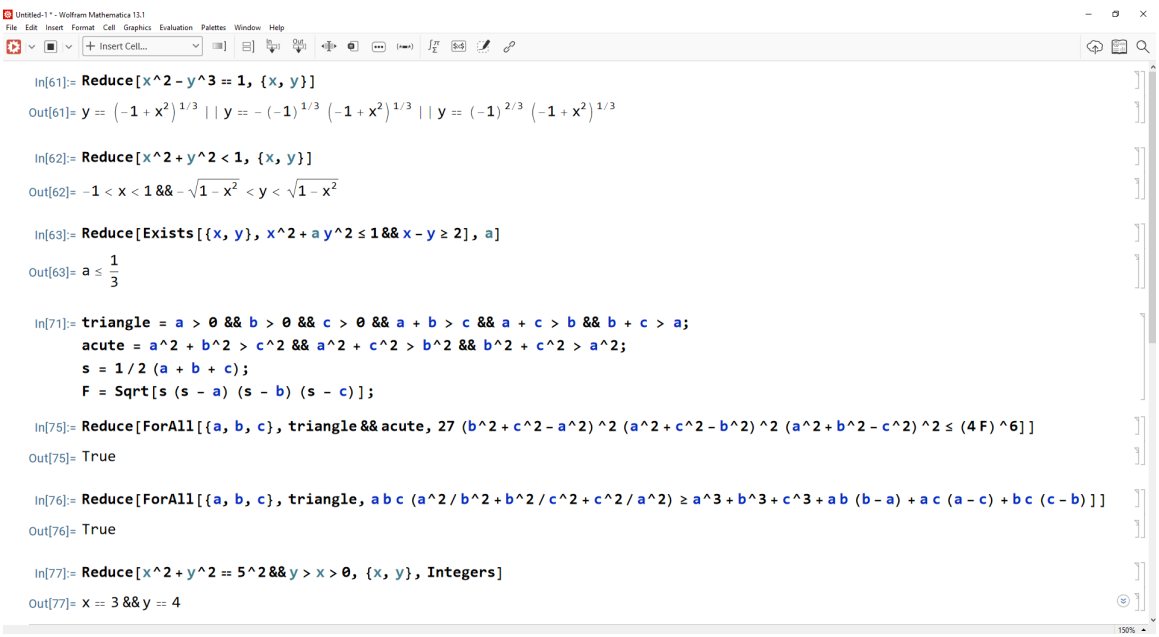


Figure 2.20 Reduce examples.

Exists (∃)

Exists [*x*, *expr*]
represents the statement that there exists a value of *x* for which *expr* is True.

Exists [*x*, *cond*, *expr*]
states that there exists an *x* satisfying the condition *cond* for which *expr* is True.

Exists [{x_1, x_2, ...}, *expr*]
states that there exist values for all the x_i for which *expr* is True.

Figure 2.21 Exists overview.

for which expr is True(Figure 2.21, 2.22)

Input: **Exists** $[x, a * x^2 + b * x + c == 0 \;\&\&\; x > 0]$

Output: $\exists_x \left(ax^2 + bx + c = 0 \wedge x > 0\right)$

LogicalExpand [expr] : expands out logical combinations of equations, inequalities, and other functions.(Figure 2.23, 2.24)

Input: **LogicalExpand**$[p \wedge \neg(q \vee r)]$

Output: $p \wedge \neg q \wedge \neg r$

Figure 2.22 Exists examples.

LogicalExpand

LogicalExpand [*expr*]
expands out logical combinations of equations, inequalities, and other functions.

Figure 2.23 Logical expand overview.

Implies [p,q] : represents the logical implication $p \implies q$.(Figure 2.25, 2.26)

Input: **Implies**$[p, q]$

Output: $p \implies q$

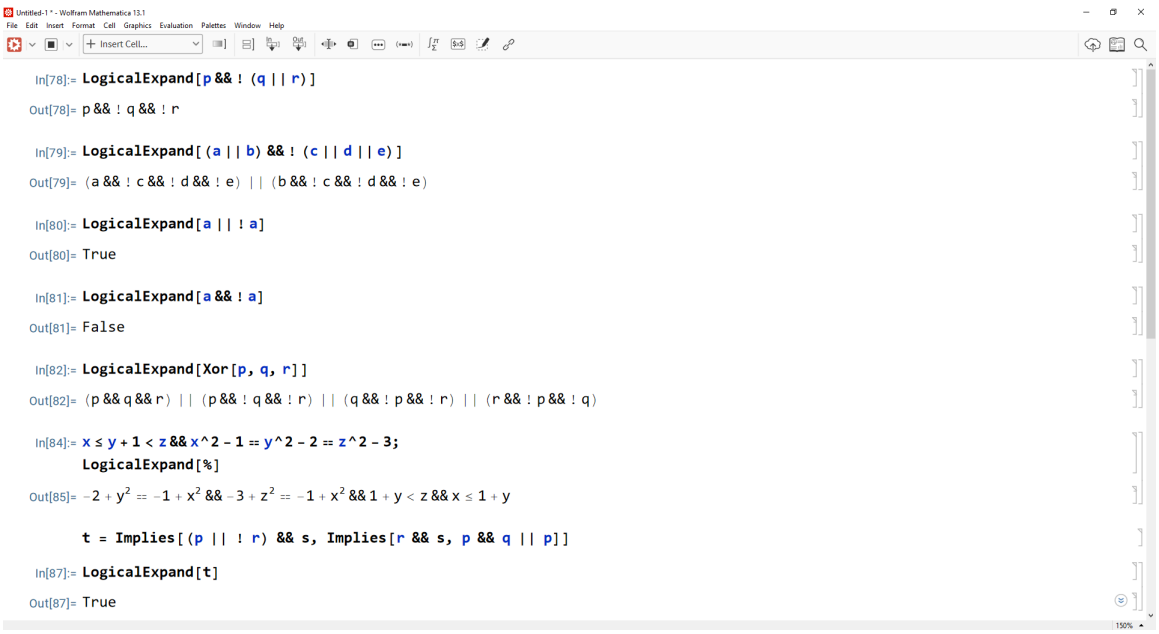


Figure 2.24 Logical expand examples.

Implies (⇒)

Implies [*p*, *q*]
represents the logical implication $p \Rightarrow q$.

Figure 2.25 Implies overview.

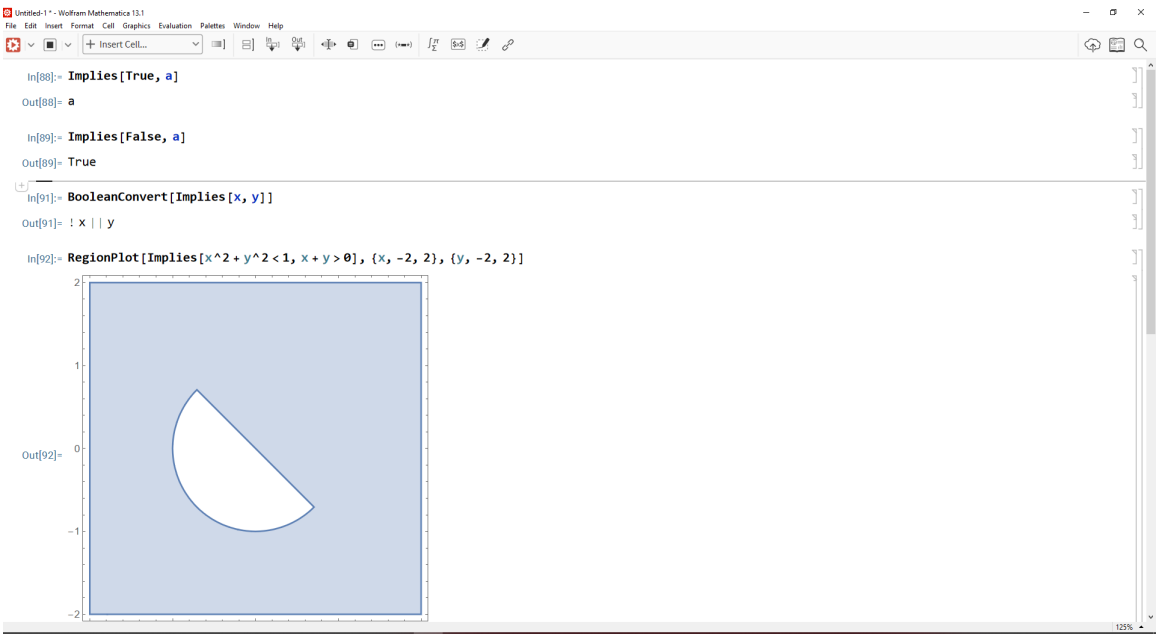


Figure 2.26 Implies examples.

CHAPTER 3

PROGRAM CORRECTNESS

The Program Correctness has been a serious concern since the early days of the invention of programming languages. Too many software products are shipped every day with known failures but undiagnosed / untreated faults. To prevent and solve these issues, significant efforts have been put into developing strategies either for proving the correctness of computer programs [31, 34, 72], or managing the process of producing the correct program [30]. Researchers have been interested to work toward programs' correctness because this can help us to prove that the program meet all the specifications, or they might find some bugs [68]. In the past times, verifying the programs correctness was being done through human works and manual proving but recently automatic tools has been offered which works based on satisfiability-modulo-theories (SMT) solvers [68], or even some data mining approaches to anticipate the program correctness [48].

3.1 Program Specification

We talked about the relational mathematics and its role in computer programs. In this part, we want to explain how the relation can be used to model specifications. As we know, specification is the crucial for the design and implementation of oracle. In general, They serve to describe key concepts in software testing, such as program correctness, faults, fault elimination, and relative correctness [28]. Assume we need a program to get a real number and produce its square root. Beside the simplicity of the task, there could be various conditions that affect on the code and we must consider. These conditions must be included in the specification. Let's check some of the varieties of this program:

1. Only nonnegative arguments will be submitted; the output is a (positive or nonpositive) square root of the input value:

$R1 = \{(s, s') | s \geq 0 \wedge s'^2 = s\}$

2. Only non-negative arguments will be submitted; the output is the nonnegative square root of the input value:

   $R2 = \{(s, s') | s \geq 0 \wedge s'^2 = s \wedge s' \geq 0\}$

3. Only nonnegative arguments will be submitted; the output is an approximation (within a precision e) of a (positive or non-positive) square root of the input value:

   $R3 = \{(s, s') | s \geq 0 \wedge |s'^2 = s| < \epsilon\}$

4. Only nonnegative arguments will be submitted; the output is an approximation (within a precision e) of the non-negative square root of the input value:

   $R4 = \{(s, s') | s \geq 0 \wedge |s'^2 = s| < \epsilon \wedge s' \geq 0\}$

5. Negative arguments may also be submitted; for negative arguments, the output is -1; for nonnegative arguments, the output is a (positive or nonpositive) square root of the input value:

   $R5 = \{(s, s') | s \geq 0 \wedge |s'^2 = s|\} \cup \{(s, s') | s < 0 \wedge s' = -1\}$

6. Negative arguments may also be submitted; for negative arguments, the output is -1; for nonnegative arguments, the output is the nonnegative square root of the input value:

   $R6 = \{(s, s') | s \geq 0 \wedge s'^2 = s \wedge s' \geq 0\} \cup \{(s, s') | s < 0 \wedge s' = -1\}$

7. Negative arguments may also be submitted; for negative arguments, the output is arbitrary; for nonnegative arguments, the output is an approximation (within a precision e) of a (positive or nonpositive) square root of the input value:

   $R7 = \{(s, s') | s \geq 0 \wedge |s'^2 = s| < \epsilon\} \cup \{(s, s') | s < 0\}$

8. Negative arguments may also be submitted; for negative arguments, the output is arbitrary; for nonnegative arguments, the output is an approximation (within a precision e) of the non-negative square root of the input value:

   $R8 = \{(s, s') | s \geq 0 \wedge |s'^2 = s| < \epsilon \wedge s' \geq 0\} \cup \{(s, s') | s < 0\}$

9. Only nonnegative arguments will be submitted; the output must be within e of the exact square root of the input (comparison with specification R4: Precision e applies to the square root scale rather than the square scale):

   $R9 = \{(s, s') | s \geq 0 \wedge |s' - \sqrt{s}| < \epsilon\}$

We can continue and have more samples out of that initial program. But we learned two important points from this example: The first is the significance of accuracy in defining program requirements, and the second is the assumption that relationships allow us to obtain the necessary precision.

1. Variable x is known to be in a; place in k an index where x occurs in a.

   $F1 = \{(s, s') | (\exists h : 1 \leq h \leq N : a[h] = x) \wedge (a[k'] = x)\}$

2. Variable x is known to be in a; place in k the first (smallest) index where x occurs in a.

   $F2 = F1 \cap \{\forall h : 1 \leq h \leq k' : a[h] \neq x\}$

3. Variable x is known to be in a; place in k an index where x occurs in a, while preserving a and x.

   $F3 = F1 \cap \{(s, s') | a' = a \wedge x' = x\}$

4. Variable x is known to be in a; place in k the first (smallest) index where x occurs in a, while preserving a and x.

   $F4 = F2 \cap \{(s, s') | a' = a \wedge x' = x\}$

5. Variable x is not known to be in a; if it is not, place 0 in k; else place in k an index where x occurs in a.

   $F5 = F1 \cup \{(s, s') | (\forall h : 1 \leq h \leq N : a[h] \neq x) \wedge k = 0\}$

6. Variable x is not known to be in a; if it is not, place 0 in k; else place in k the first (smallest) index where x occurs in a.

   $F6 = F2 \cup \{(s, s') | (\forall h : 1 \leq h \leq N : a[h] \neq x) \wedge k = 0\}$

7. Variable x is not known to be in a; if it is not, place 0 in k; else place in k an index where x occurs in a, while preserving a and x.

   $F7 = F3 \cup \{(s, s') | (\forall h : 1 \leq h \leq N : a[h] \neq x) \wedge k = 0\}$

8. Variable x is not known to be in a; if it is not, place 0 in k; else place in k the first (smallest) index where x occurs in a, while preserving a and x.

   $F8 = F4 \cup \{(s, s') | (\forall h : 1 \leq h \leq N : a[h] \neq x) \wedge k = 0\}$

If, instead of F1, we had written the specification as follows:

$F1' = \{(s, s') | a[k'] = x'\}$

then it would be possible to satisfy this specification by the following simple program:

```
{k=1; x=a[1];}
```

If, instead of F1, we had written the specification as follows:

$F1'' = \{(s, s') | a'[k'] = x\}$

then it would be possible to satisfy this specification by the following simple program:

```
{k=1; a[1] =x;}
```

If, instead of F1, we had written the specification as follows:

$F1''' = \{(s, s') | a'[k'] = x'\}$

then it would be possible to satisfy this specification by the following simple program:

```
{k=1;x =0; a[1] =0;}
```

Neither of these three programs is performing a search of variable x in array a.

3.2 Correctness Definitions

To have an insight about the correctness we refer to the explanations in [79].

We let space S be the set of natural numbers and let R be the following specification on S:

R = {(0,0) , (0,1) , (0,2) , (1,1) , (1,2) , (1,3) , (2,2) , (2,3) , (2,4) , (3,3) , (3,4) , (3,5)};

We consider the following candidate programs (represented by their functions on S), and we ask the question: which of these programs is correct with respect to R?

P1 = {(0,1) , (1,2) , (2,3) , (3,4)};

P2 = {(0,1) , (1,2) , (2,3) , (3,4) , (4,5) , (5,6) , (6,7)};

P3 = {(0,0) , (1,2) , (2,4)};

P4 = {(0,0) , (1,2) , (2,4) , (4,8) , (5,10) , (6,12)};

P5 = {(0,0) , (1,2) , (2,4) , (3,6)};

P6 = {(0,0) , (1,2) , (2,4) , (3,6) , (4,8) , (5,10) , (6,12)};

We submit:

- Only programs P1 and P2 are correct with respect to specification R since they return correct values for all the inputs of interest for R (which are inputs 0, 1, 2, 3).
- We say that programs P3 and P4 are partially correct with respect to R: they are not defined for all relevant inputs (which are 0, 1, 2, 3) since they are not defined for 3; but whenever they are defined for a relevant input (which is the case for 0, 1, 2), they return a correct value.
- We say that programs P5 and P6 are defined with respect to R (or that they terminate normally with respect to R): they produce an output for all relevant inputs (which are 0, 1, 2, 3), though for 3 they produce an incorrect output (6, rather than 3, 4, or 5).

As a second example, we let space S be defined by two integer variables x and y, and we let R be the following relation (specification) on S:

R = $\{(s, s')|x(s') = x(s) + y(s)\}$;

We consider the following candidate programs (written in C-like notation), and we ask the question: which of these programs is correct with respect to R?

p1: {`while (y ≠ 0) {x=x+1; y=y-1;}`}

p2: {`while (y > 0) {x=x+1; y=y-1;}`}

p3: {`if (y > 0) {while (y>0) {x=x+1; y=y-1;}`}

else {`while (y < 0) {x=x-1; y=y+1;}}`}

Before we make judgments on the correctness of these programs, we compute their respective functions:

p1 = $\{(s, s') | y \geq 0 \wedge x' = x + y \wedge y' = 0\}$,

p2 = $\{(s, s') | y \geq 0 \wedge x' = x + y \wedge y' = 0\} \cup \{(s, s') | y < 0 \wedge x' = x \wedge y' = y\}$,

p3 = $\{(s, s') | x' = x + y \wedge y' = 0\}$.

We submit:

- That p1 is partially correct with respect to R; it is not (totally) correct because it is not defined for negative values of y; but it is partially correct with respect to R because whenever it is defined (for non-negative values of y), it satisfies specification R (computing the sum of x and y into x).
- That p2 is defined with respect to R; it is not totally correct because for negative y it fails to compute the sum of x and y into x; but it is defined because it produces a final state for all relevant initial states (which, in the case of R, are all states in S).
- That p3 is totally correct with respect to R because it is defined for all relevant initial states and satisfies specification R for all relevant states, by computing the sum of x and y into x.

As a third example, we consider the following program on space S defined by integer variables x and y:

p: {`while (y ≠ 0) {x=x+1; y=y-1;};`}

and we consider the following specifications:

R1 = $\{(s, s') | x' = x + y\}$

R2 = $\{(s, s') | x' = x + y \wedge y' \geq 0\}$

R3 = $\{(s, s') | x' = x + y \wedge y' > 0\}$

R4 = $\{(s, s') | x' = x + y \wedge y' > 10\}$

R5 = $\{(s, s') | y \geq 0 \wedge x' = x + 1 \wedge y' = y - 1\}$

R6 = $\{(s, s') | y = 1 \wedge x' = x + 1 \wedge y' = y - 1\}$

R7 = $\{(s, s') | x' = x + 1 \wedge y' = y - 1\}$

As a reminder, consider that the function of program p is:

P = $\{(s, s') | y' \geq 0 \wedge x' = x + y \wedge y = 0\}$

Whence we submit:

- Program p is not correct with respect to R1 because it does not terminate for all relevant initial states (which, according to specification R1 are all initial states).
- Program p is correct with respect to specifications R2, R3, R4, and R6. For all these specifications, the program terminates normally for all relevant initial states and delivers a correct final state.
- Program p is defined with respect to specification R5; it is defined for all relevant inputs (which are states such that $y \geq 0$), but it is not partially correct, since it delivers a different output from what specification R5 demands.
- Finally, program p is neither correct, nor partially correct, nor defined with respect to specification R7.

We are now ready to cast the intuition gained through these examples into formal definitions.

Definition : **Correctness** Let R be a specification (relation) on space S and let p be a program on space S whose function we denote by P. We say that program p is correct (or totally correct) with respect to R if and only if:

$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in R$

Definition: Partial Correctness Let R be a specification (relation) on space S and let p be a program on space S whose function we denote by P. We say that program p is partially correct with respect to R if and only if:

$\forall s : s \in dom(R) \wedge s \in dom(P) \Rightarrow (s, P(s)) \in R$

Note that partial correctness provides for the correct behavior of the program only whenever the program terminates; so that a program that never terminates is, by default, partially correct with respect to any specification. Despite this gaping weakness, partial correctness is a useful property that is often considered valuable in practice.

Definition: Termination Let R be a specification (relation) on space S and let p be a

program on space S whose function we denote by P. We say that program p is defined (or terminates) with respect to R if and only if:

$\forall s : s \in dom(R) \Rightarrow s \in dom(P)$

We conclude this section with a simple proposition, which stems readily from the definitions.

Proposition: Correctness Properties Let R be a specification (relation) on space S and let p be a program on space S whose function we denote by P. Program p is totally correct with respect to specification R if and only if it is partially correct and defined with respect to R.

3.3 Correctness Verification

In this section, we provide ideas that express correctness in terms of formulas derived from set theory. They are more manageable in reality than either the definitions or the characterizations based on refining.

Proposition 1. ***Proposition: Correctness, Set Theoretic Formula:*** *Let R be a specification (relation) on space S and let p be a program on space S whose function we denote by P. Program p is correct with respect to specification R if and only if*

$$dom(R \cap P) = dom(R) \cap dom(P)$$

Clarification of this formula: The set dom (R ∩ P) is the set of initial states for which program p behaves as R mandates (see Figure 3.1). The set dom (R) ∩ dom (P) is the set of states for which program p terminates and specification R has a requirement. Program p is partially correct according to specification R if and only if it behaves with respect to R whenever it terminates.

Proposition 2. ***Proposition: Termination, Set Theoretic Formula:*** *Let R be a specification (relation) on space S and let p be a program on space S whose function we denote by P. Program p is defined with respect to R if and only if dom (R) ∩ dom (P) = dom (R).*
This condition simply means that dom(R) is a subset of dom(P).

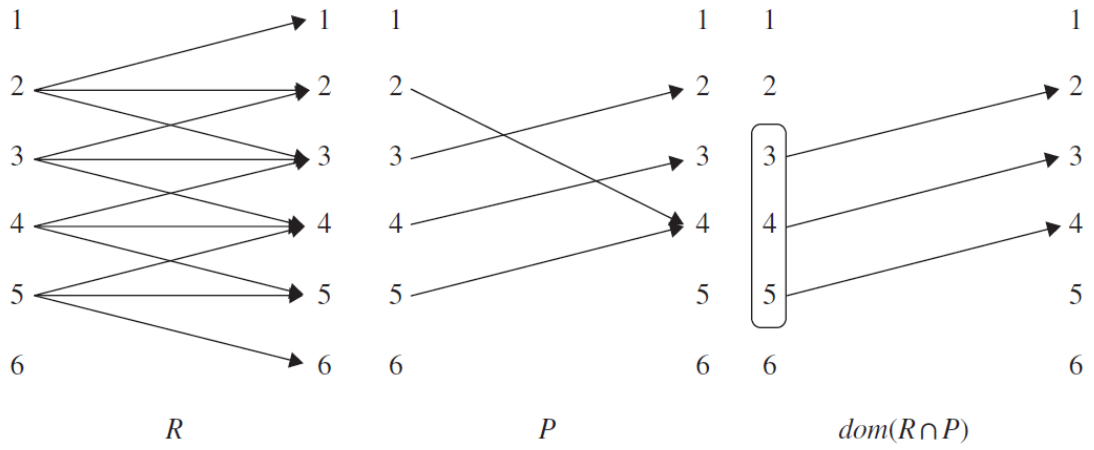


Figure 3.1 Interpretation of dom(R ∩ P).

Illustration We review the examples from previous section to serve as illustrations of the assertions stated, and we verify the formulae of these statements to make sure we arrive to the same conclusions. We review the specification and the candidate programs of the first example:

$R = (0,0),(0,1),(0,2),(1,1),(1,2),(1,3),(2,2),(2,3),(2,4),(3,3),(3,4),(3,5)$

$P1 = (0,1),(1,2),(2,3),(3,4),$

$P2 = (0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),$

$P3 = (0,0),(1,2),(2,4),$

$P4 = (0,0),(1,2),(2,4),(4,8),(5,10),(6,12),$

$P5 = (0,0),(1,2),(2,4),(3,6),$

$P6 = (0,0),(1,2),(2,4),(3,6),(4,8),(5,10),(6,12),$

The Figure 3.2 shows, for each candidate program P, the values of dom R P , dom(R), dom(P), and the correctness property of P: total correctness (TC), partial correctness (PC), termination (T), or none (N). Indeed, these are the conclusions we had previously drawn from the first scenario.

For the second example, we list the specification and programs, then we draw the same

Candidate	$dom(R \cap P)$	$dom(R)$	$dom(P)$	TC	PC	T	N
P_1	{0,1,2,3}	{0,1,2,3}	{0,1,2,3}				
P_2	{0,1,2,3}	{0,1,2,3}	{0,1,2,3,4,5,6}				
P_3	{0,1,2}	{0,1,2,3}	{0,1,2}				
P_4	{0,1,2}	{0,1,2,3}	{0,1,2,4,5,6}				
P_5	{0,1,2}	{0,1,2,3}	{0,1,2,3}				
P_6	{0,1,2}	{0,1,2,3}	{0,1,2,3,4,5,6}				

Figure 3.2 Correctness properties for example 1.

Candidate	$dom(R \cap P)$	$dom(R)$	$dom(P)$,	TC	PC	T	N
P_1	$\{(s,s')\|y \geq 0\}$	S	$\{(s,s')\|y \geq 0\}$				
P_2	$\{(s,s')\|y \geq 0\}$	S	S				
P_3	S	S	S				

Figure 3.3 Correctness properties for example 2.

table.

R = $\{(s, s')|x(s') = x(s) + y(s)\}$;

P1 = $\{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\}$;

P2 = $\{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\} \cup \{(s, s')|y < 0 \wedge x' = x \wedge y' = y\}$;

P3 = $\{(s, s')|x' = x + y \wedge y' = 0\}$;

In fact, these are the conclusions we had already drawn for the second scenario.

We had a single program and several requirements for the third example, which we describe below:

Specification	$dom(R \cap P)$	$dom(R)$	$dom(P)$,	TC	PC	T	N
R_1	$\{(s,s')\|y \geq 0\}$	S	$\{(s,s')\|y \geq 0\}$				
R_2	$\{(s,s')\|y \geq 0\}$	$\{(s,s')\|y \geq 0\}$	$\{(s,s')\|y \geq 0\}$				
R_3	$\{(s,s')\|y > 0\}$	$\{(s,s')\|y > 0\}$	$\{(s,s')\|y \geq 0\}$				
R_4	$\{(s,s')\|y > 10\}$	$\{(s,s')\|y > 10\}$	$\{(s,s')\|y \geq 0\}$				
R_5	$\{(s,s')\|y = 1\}$	$\{(s,s')\|y \geq 0\}$	$\{(s,s')\|y \geq 0\}$				
R_6	$\{(s,s')\|y = 1\}$	$\{(s,s')\|y = 1\}$	$\{(s,s')\|y \geq 0\}$				
R_7	$\{(s,s')\|y = 1\}$	S	$\{(s,s')\|y \geq 0\}$				

Figure 3.4 Correctness properties for example 3.

```
{p: {while (y != 0) {x=x+1; y=y-1;};}
```

R1 = $\{(s, s')|x' = x + y\}$;

R2 = $\{(s, s')|y \geq 0 \wedge x' = x + y\}$;

R3 = $\{(s, s')|y > 0 \wedge x' = x + y\}$;

R4 = $\{(s, s')|y > 10 \wedge x' = x + y\}$;

R5 = $\{(s, s')|y \geq 0 \wedge x' = x + 1 \wedge y' = y - 1\}$;

R6 = $\{(s, s')|y = 1 \wedge x' = x + 1 \wedge y' = y - 1\}$;

R6 = $\{(s, s')|x' = x + 1 \wedge y' = y - 1\}$;

We already know that the function of program p is P = $\{(s, s')|y \geq 0x' = x + y \wedge y' = 0\ \}$, and the domain of P is $\{s|y \geq 0;\}$ for each specification R we write, in the Figure 3.4, the values of dom (R $\cap$ P) , dom(R), and dom(P), then make a conclusion about the correctness properties of P according to the specification in question.

CHAPTER 4

FROM C-LIKE PROGRAMS TO PROGRAM FUNCTIONS

4.1 A Three_Step Transformation

In this part we discuss the process by that we generate the program function in three steps. Our tool automatically read and produce the summarized program in a series of relations which we call it program function. These steps are :

1. **J2A**: Mapping Java Code onto an Abstract Syntax Tree
2. **A2M**: Mapping each node of the Abstract Syntax Tree onto an equation between program states
3. **M2F**: Deriving the Program Function from the Aggregate Equations

4.2 J2A: Mapping Java Code onto an Abstract Syntax Tree

Working on Abstract Syntax Tree (AST) is not a new research topic. There has been numerous applications for AST such as finding the bugs in the source code [39] or transformation a C code to another language [88]. Overbey in [88], tried to make a fully re-writable AST that can be generated automatically from an annotated grammar, and then using the AST to transform the source code into target language. [40] designed a system to take a general, language independent AST and convert it into a more specific language dependent model. Similarly [106] generates a Java code from a transformation of UML to AST and AST to Java. Although all these research has been contributed very well to the use the AST effectively, none of them could reduce the complexity of the program. It means, even the output program still could be hard to read and complicated. We tried to transform the input program into a few lines of formula, that is by far easier to read, comparing to the whole source code.

In the first step, a java source code is given to a standard Java parser [102]. In its most basic form, the JavaParser library enables the interaction with Java source code in a Java context as a Java object representation. We refer to this object representation more formally as an Abstract Syntax Tree (AST). Additionally, it offers what it is called as "Visitor Support," a convenient way to browse the tree. The ability to change the source code's fundamental structure is the library's final key feature. Developers now have the ability to create their own code generation tools by writing this to a file. We will talk about Java Parser in detail in Chapter 9.

To elaborate more, we provided a simple but complete program with equivalent java Parser AST. In the next section we will see how to traverse this tree to map each node to relations.

```
public class Main
{
public static void main(String argv[])
{
            int x,y,z,w;
            y=z+x;
            x=x+y;
            if (x>y+z)
            {
                y=y+z;
                if (w>y)
                {w=x+4;}
                else
                {
                    if (x<5)
                    {
                        z=y+88;
                    }
                }
            }
            else
            {z=y*3;}
}
}
```

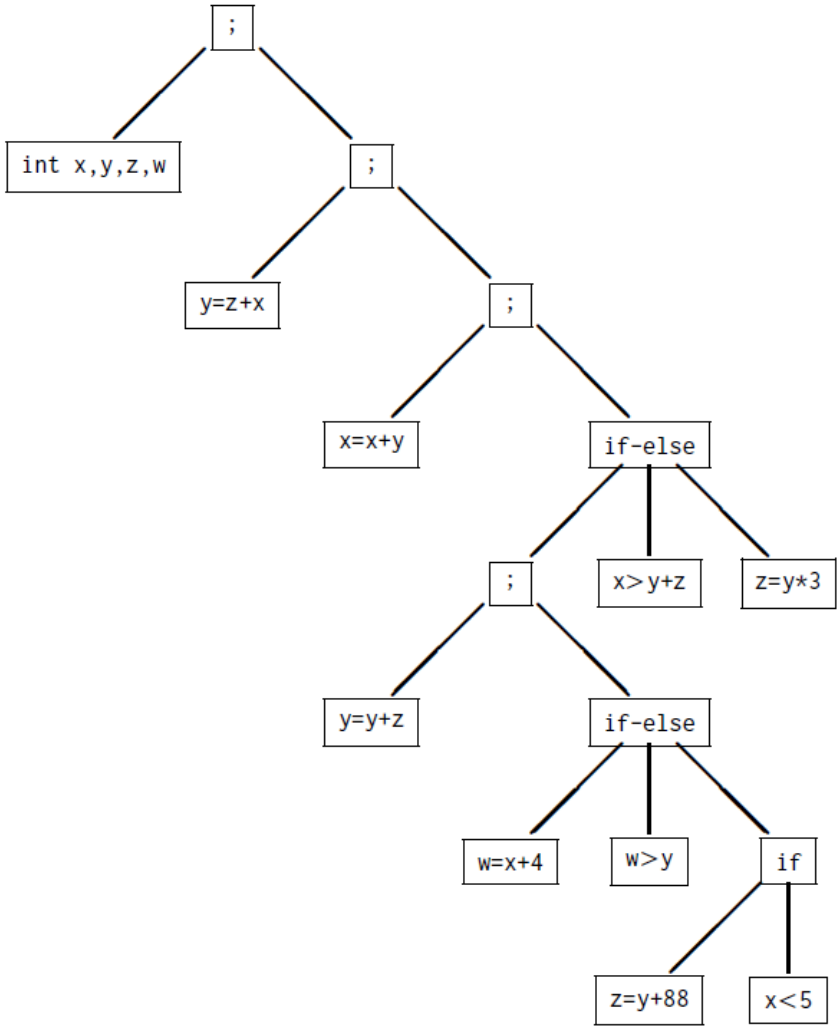


Figure 4.1 Abstract syntax tree for sample program.

4.3 A2M: Mapping Each Node to an Equation

As explained previously, we used the java parser for the first step in our tool. After reading a source code as an input of our program, we need to traverse the whole AST in order to evaluate each node and transform them into an equation of pre-post state. Next is an example of pre-post state equations which means the initial and final state.

assume we have two assignment expressions:

$y = z + x;$

$x = x + y;$

we set the final values of the variables as xP and yP. After running this two lines of code (in order) we will have:

$yP = z + x;$

$xP = 2 * x + z;$

While we are traversing the AST, each node must be properly converted into an equation. The AST is represented as a data structure in object oriented format. Each item of a program like variables, statements, functions etc, are inherited from a base Node class. There are multiple attributes for each type of node. Each node has a block or scope and the items like expressions or statements are stored in a list of children nodes. Then, at each node we can iterate over the children list, find the type of the node and map them to a series of pre-post state relations in the Wolfram language syntax. In the next chapters we will talk about how this mapping is implemented.

```
Simplify[Reduce[Reduce[Exists[{xPP,yPP,zPP,wPP},
Exists[{xP,yP,zP,wP}, xP==xPP&&yP==yPP&&zP==zPP&&wP==wPP&&
Reduce[Exists[{xPP,yPP,zPP,wPP},
Exists[{xP,yP,zP,wP},xP==xPP&&yP==yPP&&zP==zPP&&wP==wPP&&
yP==x+x&&xP==x&&zP==z&&wP==w]&&
Exists[{x,y,z,w},x==xPP&&y==yPP&&z==zPP&&w==wPP&&
xP==x+y&&yP==y&&zP==z&&wP==w],{xP,yP,zP,wP},
Backsubstitution -> True]]
Exists[{x,y,z,w},x==xPP&&y==yPP&&z==zPP&&w==wPP&&
((x>y+z&&Reduce[Exists[{xPP,yPP,zPP,wPP},
Exists[{xP,yP,zP,wP},xP==xPP&&yP==yPP&&zP==zPP&&wP==wPP&&
yP==y+z&&xP==x&&zP==z&&wP==wPP] &&
Exists[{x,y,z,w},x==xPP&&y==yPP&&z==zPP&&w==wPP&&
((w>y&&wP==x+1&&xP==x&&yP==y&&zP==z) ||
(!w>y&&((x<5&&zP==y+88&&xP==x&&yP==y&&wP==w)||
(!x<5&&xP==x&&yP==y&&zP==z&&wP==w))))]],
{xP,yP,zP,wP},Backsubstitution -> True]) ||
(!x>y+z&&zP==z+3&&xP==x&&yP==y&&wP==w))]],{xP,yP,zP,wP},
Backsubstitution -> True, {xP,yP,zP,wP}]
```

```
Elements[{x,y,z,w,xP,yP,zP,wP}Integers]]/.ConditionalExpression->And
```

The above relation is generated in our tool. It simply shows a production of two consecutive expression. The xP of first expression is applied and used for the second expression as x.

4.4 M2F: Equations to Program Function

The third steps is to find the program function through resolving the equations from previous step. By resolving means, finding the the final states as a function of the initial states, for each variable. To solve these equations we used a Mathematica equation solver [69]. After running the above relations on Mathematica notebook, (SHIFT+ENTER to execute the command on Mathematica notebook). It solves these equation; and we call this output as the function of the program. Therefore, if you give any input to (x,y,z,w) and run the program, the final values for x and y will be same as (xP,yP,zP,wP), means we can find the final state of variables without running the program. Our tool generates a formula which has same output as if we run the actual program. In Figure 4.2, we can see the generated output on Mathematica software and Figures 4.3 and 4.4 shows simplified and Tabular formats for the same output.

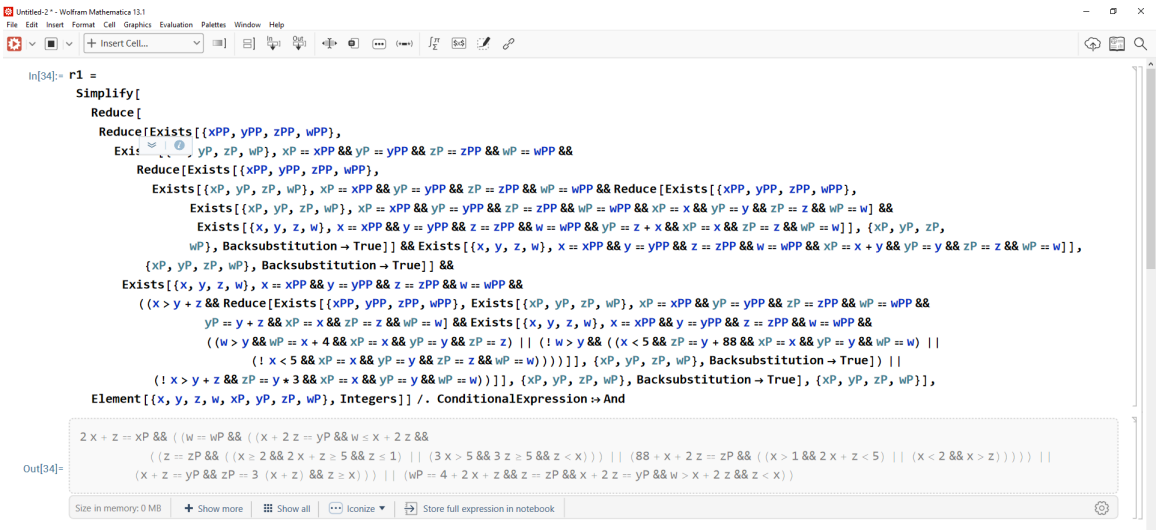


Figure 4.2 Mathematica output.

```
In[ ]:= le = LogicalExpand[te];

        TraditionalForm[
         (Simplify /@ le) //.
          Or → Composition[(Column[#, Right, Background → {{White, LightGray}}, Frame → All] &),
            List]]
```

Out[ ]//TraditionalForm=

$$
\begin{array}{|r|}
\hline
w = \mathrm{wP} \wedge x + z = \mathrm{yP} \wedge 2x + z = \mathrm{xP} \wedge \mathrm{zP} = 3(x+z) \wedge z \geq x \\
\hline
2x + z = \mathrm{xP} \wedge \mathrm{wP} = 2x + z + 4 \wedge x + 2z = \mathrm{yP} \wedge z = \mathrm{zP} \wedge w > x + 2z \wedge z < x \\
\hline
w = \mathrm{wP} \wedge 2x + z = \mathrm{xP} \wedge x + 2z = \mathrm{yP} \wedge x + 2z + 88 = \mathrm{zP} \wedge x > 1 \wedge 2x + z < 5 \wedge w \leq x + 2z \\
\hline
w = \mathrm{wP} \wedge 2x + z = \mathrm{xP} \wedge x + 2z = \mathrm{yP} \wedge x + 2z + 88 = \mathrm{zP} \wedge x > z \wedge x < 2 \wedge w \leq x + 2z \\
\hline
w = \mathrm{wP} \wedge 2x + z = \mathrm{xP} \wedge x + 2z = \mathrm{yP} \wedge z = \mathrm{zP} \wedge 3x > 5 \wedge 3z \geq 5 \wedge z < x \wedge w \leq x + 2z \\
\hline
w = \mathrm{wP} \wedge 2x + z = \mathrm{xP} \wedge x + 2z = \mathrm{yP} \wedge z = \mathrm{zP} \wedge x \geq 2 \wedge 2x + z \geq 5 \wedge w \leq x + 2z \wedge z \leq 1 \\
\hline
\end{array}
$$

Figure 4.3 Tabular output-type 1.

```
In[ ]:= TraditionalForm[
          te //.
           Or → Composition[(Column[#, Right, Background → {{White, LightGray}}, Frame → All] &),
             List]]
```

Out[ ]//TraditionalForm=

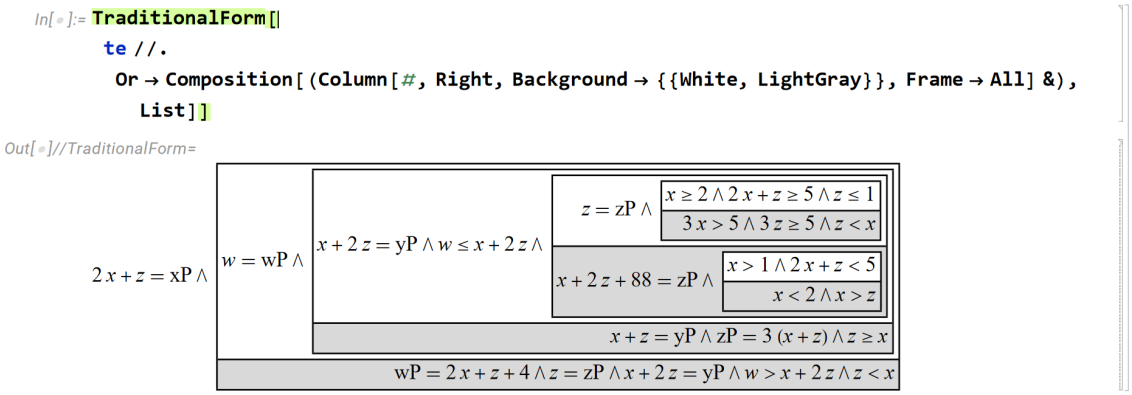


Figure 4.4 Tabular output-type 2.

CHAPTER 5

INVARIANT RELATIONS

The sample we analyzed in previous chapter was relatively easy. If we are evaluating a program with a loop, or even a nested loop, it is not as easy as a few expression production. We need to run some extra processes to find the function of the loop first. But let's analyze how we can get the loop function step by step.

5.1 Loop Function

We achieved the program function in last chapter, as a relation of pre-post state. But how about a loop? we can get a similar relation for a loop as the function of the loop. It means what would be the final state for the loop variables. The function of a loop can be achieved from Theorem 1.

Theorem 1. *Given a while loop* `w:` $\{$`while (t) {b}`$\}$ *on space* S*, the function of* `w` *is the function on* S *defined as:*

$$W = (T \cap B)^* \cap \widehat{\overline{T}},$$

where B *is the function of* `b` *and* T *is the vector defined by predicate* t*, i.e.* $T = \{(s, s')|t(s)\}$.

Without noticing the details, we can check few loops and their equivalent functions.

Program 1:

```
public class Main
{
    public static void main(String argv[])
    {
        int k, n, f;
        while (k != n)
        {
            f = f * k;
```

```
            k = k + 1;
        }
    }
}
```

Function:

W = $\{(s, sP)|k <= n \wedge nP == n \wedge kP == n + 1 \wedge fP == f * (n!/(k-1)!)\}$

Program 2:

```
public class Main
{
public static void main(String argv[])
{
        int x ,y, i,n;
        while (i != n)
        {
            x = x + y;
            y = x - y;
            i = i + 1;
        }
}
}
```

Function:

W = $\{ (s, sP)|i <= n \wedge xP == y * Fibonacci[-i + n] + x * Fibonacci[1 - i + n] \wedge yP ==$

$y * Fibonacci[-1 - i + n] + x * Fibonacci[-i + n] \wedge iP == n \wedge nP == n\}$

Program 3:

```
public class Main
{
    public static void main(String argv[])
    {
        int i, j, n, x,y;
        int a[];
        int b[];
        while (i != n)
        {
            x = x + a[i];
            i = i + 1;
            y = y + b[j];
```

```
            j = j - 1;
        }
    }
}
```

Function:

W = $\{ (s, sP) | i <= n \wedge iP == n \wedge nP == n \wedge xP == x \wedge jP == i + j - n \wedge xP == x + \sum_{k=i}^{n} a[k] \wedge yP == y + \sum_{k=j}^{i+j-n} b[k] \}$

5.2 Invariant Relations

Since it is typically quite challenging in reality to compute the reflexive transitive closure of a relation, this definition could be too difficult and not applicable to compute the function of a loop. We use invariant relations as an alternative way.

Definition 1. *Given a while loop* `w`: `{while (t) {b}}` *on space S, a relation R on S is said to be an* invariant relation *of* `w` *if and only if R is a reflexive transitive superset of* $(T \cap B)$.

There are some properties of invariant relations that we can use to find the loop function [76, 77]:

Proposition 3. $(T \cap B)^*$ *is an invariant relation*

Proposition 4. $(T \cap B)^*$ *is the smallest invariant relation*

Proposition 5. *The intersection of two invariant relation is an invariant relation*

Proposition 6. $E = I \cup T(T \cap B)$ *is an invariant relation*

As we stated in Chapter 1, an invariant relation of a while loop is a binary relation that contains pairs of states that are separated by any number of loop iterations. This means, after any number of the iteration and without considering the values of the variables, the invariant relation returns a true

statement.

Now for the same examples of previous section we can see and verify the invariant relations:

Program 1:

```
public class Main
{
public static void main(String argv[])
{
        int k,n,f;
        while (k != n)
        {
        f = f * k;
        k = k + 1;
        }
}
}
```

Invariant Relations

$R0 = \{(s, sP)|n = nP\}$

$R1 = \{(s, sP)|k \leq kP\}$

$R2 = \{(s, sP)|f/(k-1)! = fP/(kP-1)!\}$

$R = \{(s, sP)|x = xP \wedge k \leq kP \wedge f/(k-1)! = fP/(kP-1)!\}$

If we run the loop, these relations holds between any states s and s′ that are separated by an arbitrary number of iterations. expressions.

Program 2:

```
public class Main
{
public static void main(String argv[])
{
        int x ,y, i,n;
        while (i != n)
        {
            x = x + y;
            y = x - y;
            i = i + 1;
        }
```

```
}
}
```

Invariant Relations

$R0 = \{(s, sP) | xP = x*Fibonacci[iP+1-i]+y*Fibonacci[iP-i] \wedge yP = x*Fibonacci[iP-i] + y*Fibonacci[iP-i-1]\}$

$R1 = \{(s, sP) | i \leq iP\}$

Program 3:

```
public class Main
{
    public static void main(String argv[])
    {
        int i, j, n, x,y;
        int a[];
        int b[];
        while (i != n)
        {
            x = x + a[i];
            i = i + 1;
            y = y + b[j];
            j = j - 1;
        }
    }
}
```

Invariant Relations

$R0 = \{(s, sP) | y + \Sigma_{k=0}^{j} b[k] = yP + \Sigma_{k=0}^{jP} bP[k]\}$

$R1 = \{(s, sP) | x + \Sigma_{k=i}^{n} a[k] = xP + \Sigma_{k=iP}^{n} aP[k]\}$

$R2 = \{(s, sP) | a = aP\}$

$R3 = \{(s, sP) | b = bP\}$

5.3 Invariant Relation and Loop Function

Having invariant relations for the loop, we are able to extract an approximation of the loop function. With enough approximation we derive the function for the loop in full.

Theorem 2. *Given a while loop* `w:` $\{$`while (t) {b}`$\}$ *on space* S*, let* R *be an invariant relation of* `w` *and let* W *be the function of the loop, then the following condition holds:* $W \subseteq R \cap \widehat{\widehat{T}}$.

Based on this theorem, we provided a testing strategy works based on checking huge number or test cases. So far, we claimed that our tool is able to transform a source code and find an equivalent function for that program. Now, we intend to test it. The general structure for this test consists of an oracle method, a driver method and the actual program. The method that runs the actual source code, here it is **P1**, executes the actual program. The driver method runs too many possible test cases. For each test case, the actual program runs, and after that initial and final values of variables are sent to oracle method to test the function. In the end, we check the correctness of our claimed function(included in oracle method). We did this approach for each of three programs and results was True for all.

Program 1:

```
public class Main
{

public static int factorial(int x)
{
    if (x == 0) return 1;
    int r = x;
    while (r > 0)
    {
        x *= (x - 1);
        r--;
    }
        return x;
}
public static bool oracle(int k,int n,int f,int kP,int nP,int fP)
{
```

```
    return k0 <= n0 && nP == n0 && kP == n0 + 1 &&
    fP == f * (factorial(n) / factorial(k - 1));
}

public static void P1(int& k, int& n, int& f)
{
    while (k != n)
    {
        f = f * k;
        k = k + 1;
    }
}

public bool driver()
{
  bool testResult = true;
  for (int k0 = 1; k0 < 100; k0++)
    for(int n0 = 1; n0 < 100; n0++)
        for (int f0 = 1; f0 < 100; f0++)
        {
            int kt = k0, nt = n0, ft = f0;
            P1(kt, nt, ft);
            testResult = testResult && oracle(k0,n0,f0,kt,nt,ft);
        }
    return testResult;
}



 public static void main(String argv[])
 {
      driver();
 }
 }
```

Function 1

$W = R0 \cap R1 \cap R2 \cap \{(s, sP) | \neg(k \neq n)\}$

= $\{(s, sP) | \wedge k \leq n \wedge nP == n \wedge kP == n + 1 \wedge fP == f * (n!/(k-1)!)\}$

Program 2:

```
public class Main
{

public static int Fibonacci(int x)
```

```
{
    if (x <= 0) return 0;
    if (x <= 2) return 1;
    int f1= 1, f2= 1;
    int r = x;
    while (r > 2)
    {
        f2 = f2 + f1;
        f1 = f2 - f1;
        r--;
    }
        return f2;
}
public static bool oracle(int k,int n,int f,int kP,int nP,int fP)
{
    return
    i <= n && xP == y * Fibonacci(-i+n) + x * Fibonacci(1-i+n)
    && yP == y * Fibonacci(-1-i+n) + x * Fibonacci(-i+n)
    && iP == n && nP == n;
}

public static void P2(int& i, int& n, int& x, int& y)
{
  while (i != n)
  {
    x = x + y;
    y = x - y;
    i = i + 1;
  }
}

public bool driver()
{
    bool testResult = true;
    for (int i0 = 1; i0 < 100; i0++)
        for(int n0 = i; n0 < 100; n0++)
            for (int x0 = 1; x0 < 100;x0++)
                for (int y0 = x0;y0 < 100;y0++)
                {
                    int it = i0, nt = n0, xt = x0 , yt = y0;
                    P2(it, nt, xt, yt);
                    testResult =testResult &&
                    oracle(i0, n0, x0, y0, it, nt, xt, yt);
                }
    return testResult;
}
```

```
 public static void main(String argv[])
 {
      driver();
 }

}
```

Function 2:

$W = R0 \cap R1 \cap \{(s, sP) | \neg(i \neq n)\}$

= $i \leq n \wedge xP == y * Fibonacci[-i+n] + x * Fibonacci[1-i+n] \wedge yP == y * Fibonacci[-1-$

$i+n] + x * Fibonacci[-i+n] \wedge iP == n \wedge nP == n$

Program 3:

```
public class Main
{

public static int sum(int a[], int start, int end)
{
    int result=0;
    for (int i=start, i<=end,i++)
        result += a[i];
    return result;
}
public static bool oracle(int i,int n,int j,int x,int y,int iP,
                          int nP,int jP,int xP,int yP)
{
    return
    i <= n && iP == n && nP == n && xP == x && jP == i + j - n
    && xP == x + sum(a,i,n) && yP == y + sum(b,j, i + j - n)
}

public static void P3(int& i, int& n,int& j, int& x, int& y)
{
    int i, j, n, x,y;
    int a[100];
    int b[100];
    while (i != n)
    {
        x = x + a[i];
        i = i + 1;
        y = y + b[j];
        j = j - 1;
    }
```

```
}

public bool driver()
{
    bool testResult = true;
    for (int i0 = 1; i0 < 100; i0++)
        for(int j0 = 0;j < 100; j0++)
            for(int n0 = i; n0 < 100; n0++)
                for (int x0 = 1; x0 < 100;x0++)
                    for (int y0 = 1;y0 < 100;y0++)
                    {
                      int it = i0,nt = n0,xt = x0 ,yt = y0,jt=j0;
                      P3(it, nt, jt, xt, yt);
                      testResult =testResult &&
                      oracle(i0,n0,j0,x0,y0,it, nt, jt, xt, yt);
                    }
    return testResult;
}

 public static void main(String argv[])
 {
      driver();
 }

}
```

Function 3:

$W = R0 \cap R1 \cap R2 \cap R3 \cap \{(s, sP) | \neg(i \neq n)\}$

= $\{(s, sP) | i \leq n \wedge iP == n \wedge nP == n \wedge xP == x \wedge jP == i + j - n \wedge xP ==$

$x + \sum_{k=i}^{n} a[k] \wedge yP == y + \sum_{k=j}^{i+j-n} b[k]\}$

CHAPTER 6

INVARIANT RELATION GENERATION

6.1 Elementary Invariant Relation

There is only one invariant relation that can be generated constructively, using the parameters T and B of the loop, we call it as elementary invariant relation. This invariant can be obtained by $I \cup T(T \cap B)$. We refer to it as the *elementary invariant relation* of. Hence, for each three loop we analyzed above, we can present the elementary invariant relation.

elementary invariant relation for program 1

$E = I \cup \{(s, sP) | (k \neq n) \wedge kP == k + 1\}$

$E = \{(k == kP \wedge n == nP \wedge f == fP) || ((k \neq n) \wedge kP == np)\}$

elementary invariant relation for program 2

$E = I \cup \{(s, sP) | (i \neq n) \wedge xP == x + y \wedge yP == x \wedge iP == i + 1\}$

$E = (x == xP \wedge y == yP \wedge i == iP \wedge n == nP) || (i \neq n \wedge iP == nP)$

elementary invariant relation for program 3

$E = I \cup \{(s, sP) | (i \neq n) \wedge iP == nP\}$

$E = (x == xP \wedge y == yP \wedge i == iP \wedge n == nP) || (i \neq n \wedge iP == nP)$

6.2 Recognizers

Beside *elementary invariant relations,* we provided a database of invariant relations, shown in Table 6.1 which we call it recognizer table. Of course the more recognizer we provide, the more loops we may evaluate. The recognizers have been proved and tested manually. Here we are going to describe how we prove the correctness of the recognizers. We want to show the relations R as the invariant relation is reflexive transitive superset of the B which is the function of the loop's body.

1R1

Reflexive: $x = x' \Rightarrow x \pmod c = x' \pmod c$

Transitive: $\{(x, x') \in R \wedge (x', x'') \in R \Rightarrow (x, x'') \in R\}$

$x \pmod c = x' \pmod c \wedge x' \pmod c = x'' \pmod c \Rightarrow x \pmod c = x'' \pmod c$ After proving the reflexivity and transitivity of the relation, now we are checking whether $B \subseteq R$ is true or not. The B is the loop body function and R is the invariant relation. In the other words we want to investigate if the below conclusion is right or not:

$x' = x + c \Rightarrow x \pmod c = x' \pmod c$ Simply we replace the x' in right side of relation.

2R1

Reflexive: $x = x' \wedge y = y' \Rightarrow ay - bx = ay' - bx'$

Transitive: $ay - bx = ay' - bx' \wedge ay' - bx' = ay'' - bx'' \Rightarrow ay - bx = ay'' - bx''$

$x' = x + a \wedge y' = y + b \Rightarrow ay - bx = a(y + b) - b(x + a)$

$\Rightarrow ay - bx = ay + ab - bx - ba$

$\Rightarrow ay - bx = ay - bx$

3R1

Reflexive: $i \leq i'$ is reflexive, since $i \leq i$ is a tautology. The other two terms are of the form: f(s)=f(s′). by construction, these define a reflexive transitive relation. To prove that $B \subseteq R$, we take an element (s,s′) in B, and we prove that it is in R.

$(s, s') \in B$

By definition:

$= (i' = i + 1 \wedge x' = x + a[i] \wedge a' = a)$

by substitution

$i <= i' \wedge x' + \Sigma_{k=i'}^{N} a'[k] = x + a[i] + \Sigma_{k=i+1}^{N} a[k] \wedge a' = a$

simplification

Table 6.1 Sample Recognizers

ID	State Space	Condition	Code Pattern	Invariant Relation, R=
1R1	x: int; const c: int> 0;	true	x'=x+c	$\{(s,s')\|x \mod c = x' \mod c\}$
1R2	x: int; const c: int;	true	x'=x+c	$\{(s,s')\|x \times c \leq x' \times c\}$
1R3	x: int	true	x'=x-1	$\{(s,s')\|x \geq x'\}$
2R1:	x, y: int; const a, b: int	true	x' = x+a, y' = y+b	$\{(s,s')\|ay - bx = ay' - bx'\}$
2R2:	x, y: int; const a: int	true	x' = x*a, y' = y+x	$\{(s,s')\|y(1-a) + x = y'(1-a) + x\}$
2R4:	x, y: int;	true	x'=x+1; y'=y-y/(x+1)	$\{(s,s')\|xy = x'y'\}$
2R5:	x,y: int;const a,b: int;	true	x' = x+a;y'=y*b;	$\{(s,s')\|\frac{y}{b^{x/a}} = \frac{y'}{b^{x'/a}}\}$
2R6:	x,y: listType	true	y'=y.First(x),x'=Rest(x)	$\{(s,s')\|y.x = y'.x'\}$
2R11	x, y: int	$x\%2 = 0$	x'=x/2, y'=y+1	$\{(s,s')\|y + \log_2(x) = y' + \log_2(x')\}$
2R13	x, y: int	$x\%4 = 0$	x'=x/4, y'=y+2	$\{(s,s')\|y + \log_2(x) = y' + \log_2(x')\}$
2R14	x, y: int	$x\%c = 0$	x'=x/c, y'=c*y	$\{(s,s')\|xy = x'y'\}$
3R1	x, i: int a[N]: int	true	i'=i+1, a'=a, x' = x+a[i]	$\{(s,s')\|a' = a \wedge i \leq i'$ $\wedge x - \sum_{k=i}^{N} a[k] = x' - \sum_{k=i'}^{N} a'[k]\}$
3R2	x, i: int a[N]: int	true	i'=i-1,a'=a, x' = x+a[i]	$\{(s,s')\|a' = a \wedge i \geq i'$ $\wedge x + \sum_{k=1}^{i} a[k] = x' + \sum_{k=1}^{i'} a'[k]\}$
3R3	i: int; x,y: sometype	true	i'=i-1, x'=f(x), y'=y+x	$\{(s,s')\|y + \Sigma_{k=1}^{i} f^k(x) = y' + \Sigma_{k=1}^{i'} f^k(x')\}$

$i <= i' \wedge x' + \Sigma_{k=i'}^{N} a'[k] = x + \Sigma_{k=i}^{N} a[k] \wedge a' = a$

6.3 Recognizer Generation

To prove and generate the invariant relations, we use recurrence technique. We write the recurrence equations for each variable. Then, to solve these equations, the recurrence variable will be eliminated and by simplifying the remaining equation, the invariant will be achieved. To illustrate the procedure, two recognizer generation are provided below.

Referring to table 6.1, we try to solve the 2R1 record. To do this, we write the equations multiple times. In each equation, the x′ is replaced by the equivalent expression which achieved from previous

relation.

$x' = x + a, y' = y + b \implies x' = (x + a) + a, y' = (y + b) + b \implies x' = (x + a + a) + a, y' = (y + b + b) + b$

Then after n iteration we will have:

$x' = x + n * a, y' = y + n * b$

We can get the constant n from the first equation

$n = (x' - x)/a$

and place it in second to get the final equation.

$y' = y + ((x' - x)/a) * b \implies a * y' - a * y = (x' - x) * b \implies a * y' - a * y = (b * x' - b * x) \implies a * y - b * x = a * y' - b * x'$

Another instance is 3R1. In this record, the code pattern which reflects the function of the loop's body, is

$i' = i + 1, a' = a, x' = x + a[i]$

This relation is achieved after one iteration of the loop. After n iteration, we will have:

$x' = x + \sum_{j=i}^{i+n-1} a[j], i' = i + n \implies x' = x + \sum_{j=i}^{i'-1} a[j]$

to conform the left side with the right side, we add the $\sum_{j=0}^{i-1} a[j]$ to both sides. Then, we will get:

$x' + \sum_{j=0}^{i-1} a[j] = x + \sum_{j=0}^{i'-1} a \implies x' - \sum_{j=0}^{i'-1} a[j] = x - \sum_{j=0}^{i'-1} a[j]$

As we see, we could prove and find an equation based on the code pattern that we have.

6.4 Pattern Matching

To evaluate a loop node, we need some extra process to find the function of the loop. The function of the loop has the same concept as we explained in previous chapters, the final state for each variable of the loop. We showed how we can use invariant relations to get the loop function. But, how

can we derive the appropriate invariant relations for our loop? Unlike previous works that finds the invariants with syntactic matching [76, 77], we adopt a new semantic matching approach. The semantic matching is done by some codes we developed, encapsulated in Mathematica package. Before getting into detail, we can check this semantic matching in a few steps:

1. finding the function of the loop body.
2. running the imply for the loop body function and each recognizer $Implies[F, rec_i]$
3. return and collect every recognizer that makes the imply True

The first step is done similar to a regular program without a loop, assuming we do not have a loop here. Let's review our previous example here with different variables.

```
while (k != n)
{
      a = a + b;
      b = a - b;
      k = k + 1;
}
```

The function for the body of this loop is

```
aP == a + b && bP == a && kP == k + 1
```

Now in second step we intend to find the relevant invariant relation for this loop. As we see in figure 6.1 each recognizer has three parts: first segment is the code pattern which could be possibly part of the function of the loop's body, second are the variables involved in this relation, and third is the relevant invariant relation. In our work, we have three type of recognizers separated by number of logical clauses: one clause, two clauses and three clauses. In variable segment, we have the initial and final state for each variable along with its data type.

```
{xP == x + y && yP == x && iP == i + 1, {{x, xP, "int"},
{y, yP, "int"}, {i, iP, "int"}},
xP == x*Fibonacci[iP + 1 - i] + y*Fibonacci[iP - i] &&
yP == x*Fibonacci[iP - i] + y*Fibonacci[iP - i - 1]}
```

Figure 6.1 Sample recognizer in our database

Pattern Matching Procedure:

The pattern matching approach is different from previous works. Finding the relevant invariant is not done through some syntactic matching, i.e. string matching. Instead, we implemented a comprehensive package, implemented with Mathematica language. The procedure starts from calling the package function from our Java tool. the signature of the function provided below:

```
ComputeLFONE[Vars_,stateS_, T1_, F1_, OneRecList_,
TwoRecList_, ThreeRecList_, File1_]
```

the function takes few parameters and we explain each one in below:

- Vars: the variables that involved in the given loop. As we are traversing the Abstract Syntax Tree, we store the declared variables in a symbol table map data structure. Hence, for each scope we have the variables that are accessible.
- State (or space): the state of variables in the same format as Figure 6.1, like {x,xP,"int"}
- T : the condition of the given loop
- F : the function of the loop's body. We recursively investigate the nested scopes and find the function of the body. This is explained in next chapters.
- RecList: three recognizers list are sent for further pattern matching. These lists are stored in a separate file, as we call it Recognizer Database.
- File: output file to store and write the found invariant relation into

The first step in pattern matching procedure is to conform the formal variables of recognizers with the actual variables coming from real code. For example, in the previous sample we have a and b as actual integer variables, and there are x and y ,formal variables in the recognizer's record. In our code, we test and store all of the permutations of mapping formal variables to actual variables.

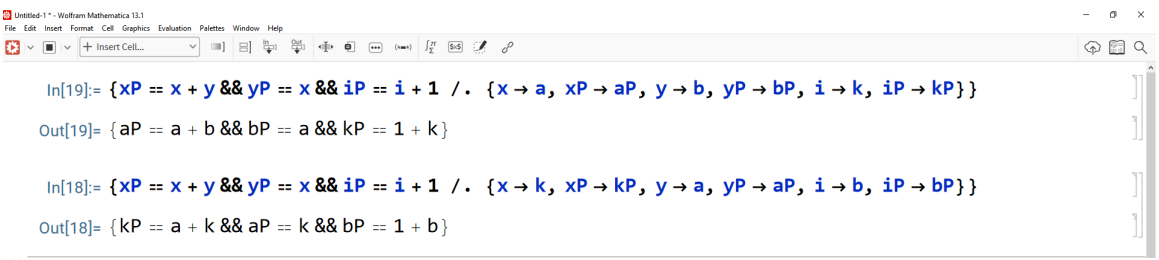


Figure 6.2 Mapping from formal to actual variables.

For instance, one permutation is to change x to a and y to b, and another would be x to b and y to a. In the end, we created a list of new relations from recognizers with this difference that the formal variables are replaced with actual. This should happen at run-time and stored as a temporary list. In Figure 6.2, a sample provided which shows how we transform the first part of recognizer to a new one with actual variables with Mathematica code.

In the second step, we need to find which one of the changed recognizers, now with actual variables, conform with the function of the loop's body. To do this, we take advantage of another Mathematica capability. $Imply$ function, explained in chapter 2, can take two statements and check whether P(first statement) does imply Q(second statement) or not. Any of the items that satisfies this implication, will be an invariant relation candidate. But we need to conform the variables of this invariant relation, the formal variables, with the actual variables in our loop. Then, as final step, the do the mapping of formal variables to actual ones and store it as one of the found invariant relations.

As we see in Figure 6.3, the first check returned True and it means this relation one of the invariant relations.

The last step is nothing but return the third part of the recognizer record as the invariant relation. But this time again, we need to return the invariant relation with actual variables. Hence, the same mapping procedure to replace formal variables with actual variables performs here too. In the end, we find and return:

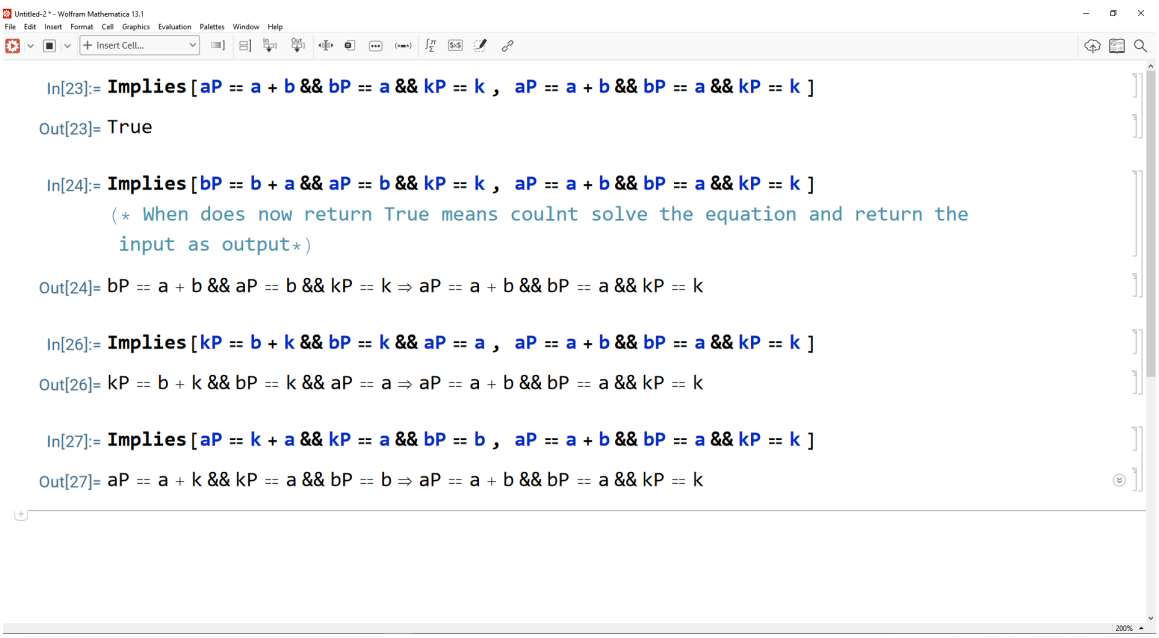


Figure 6.3 Sample Imply for pattern matching.

```
{aP == b Fibonacci[-k + kP] + a Fibonacci[1 - k + kP] &&
  bP == b Fibonacci[-1 - k + kP] + a Fibonacci[-k + kP]}
```

CHAPTER 7

PROGRAM SEMANTICS

In this chapter, we focus on the main step in the transformation of a program from source code to functional representation: creating equations in Mathematica using an abstract syntax tree. This transformation is carried out by traversing the abstract syntax tree recursively, starting at its root and ending at its leaves. Each internal node causes recursive calls to be made to the appropriate subtrees. We call this function A2M (Abstract syntax tree to Mathematica). To keep the information of each node, we create a symbol table. The symbol table comprises four entries, as shown below. We keep a symbol table in such a way that, whenever we process a node of the tree, the symbol table contains all the variables that are active at that node.

There are two type of leaves in an abstract syntax tree: variable declarations and assignment statements.

7.1 Variable Declarations

When A2M is called on a leaf that represents a variable declaration, say *int x*, then an entry for x is added to the symbol table, and the call returns the following Mathematica equation: xP==Undef && yP==y. Where y represents all the variables in the symbol table prior to adding x, and Undef is an undefined value, which we use as an indication that variable x has no defined value.

Table 7.1 Symbol Table

Type	Name	Primed	2-Primed
int	x	xP	xPP

7.2 Assignment Statement

When A2M is called on a leaf that represents an assignment statement, say x=E(s), where x is a program variable and E is an expression that takes values that are compatible with the type of x, then:

A2M(x=E(s)) = defE(s) && xP==E(s) && yP==y,

where y represents all the variables in the symbol table other than x, and def E() is the predicate that holds for all s such that expression E(s) can be evaluated.

As an example for an assignment: A2M(y=x+z) = (xP==x && yP==x+z && zP==z).

7.3 Bracketed Scope

When A2M is applied to an internal node (a subtree, AST) that represents a bracketed scope (such as, e.g.: {int x; <statements>; }), we consider that it has two subtrees:

- A subtree, say AST.dec, which contains the variable declarations that are internal to the scope; we call them x1, x2, ... xk , and we use x to represent the aggregate <x1, x2, ...xk>.
- A subtree, say AST.code,which contains the executable code that is internal to the scope.

Then, application of A2M to AST yields the following Mathematica equation: A2M(AST)[s,sP] = (Exists x, xP: A2M(AST.code)[<s, xi >, <sP, xPi >]).

The space of the program inside the bracket consists of components s and x while the space of the bracketed block includes only component s.

7.4 Sequence

When A2M is applied to a node AST that represents a sequence, we consider that it has two subtrees, which represent the first term and the second term of the sequence. Even if the source code includes

a longer sequence than two statements, we assume that the AST arranges them in a binary tree structure.

A2M(AST) = (Exists(sPP: sP2sPP(A2M(AST.first)) && s2sPP(A2M(AST.second)))),

where sP2sPP() transforms a Mathematica formula written in terms of (s, sP) by replacing each instance of sP by sPP and s2sPP() transforms a Mathematica formula written in terms of (s, sP) by replacing each instance of s by sPP. The result is a formula that is written in terms of (s, sP) where s is the state of the program before the sequence and sP is the state of the program after the sequence; even if we have a sequence of 100 statements, we only need one instance of sPP, since the compositions are applied two at a time (through nesting). As for sP2sPP() and s2sPP, they are defined simply as:

sP2sPP(F) = Exists (sP: F(s,sP) && sP==sPP . s2sPP(F))

= Exists (s: F(s,sP) && s==sPP).

We can see this production in this example:

$x = y + 2;$

$y = x * 3;$

taken from the Java program of Figure 1. We find:

A2M(x=y+2) =(xP== y +2 && yP==y)

sP2sPP(A2M(x=y+2)) = (yPP==y && xPP== y+2)

A2M(y=x*3) = (yP=x*3 && xP==x)

s2sPP(A2M(y=x*3)) = (yP=xPP*3 && xP==xPP)

The Mathematica code for the A2M(x=y+2;y=x*3;), which is the production of two expressions, is:

```
Simplify[
  Reduce[Reduce[
    Exists[{xPP, yPP},
     Exists[{xP, yP},
       xP == xPP && yP == yPP &&
```

```
        Reduce[Exists[{xPP, yPP},
          Exists[{xP, yP},
            xP == xPP && yP == yPP && xP == x && yP == y] &&
           Exists[{x, y},
            x == xPP && y == yPP && xP == x + 2 && yP == y]], {xP,
          yP}, Backsubstitution -> True]] &&
      Exists[{x, y},
       x == xPP && y == yPP && yP == x*3 && xP == x]], {xP, yP},
    Backsubstitution -> True], {xP, yP}],
  Element[{x, y, xP, yP}, Integers]] /. ConditionalExpression :> And
```

Finally the result is:

(xP = y+2 && yP = 3*y + 6)

7.5 If-Then

We assume that a node AST that represents an if-then statement has two subtrees, one that represents the condition of the if and one that represents the then branch. For simplicity, we further assume that the condition of the if is written in the syntax of Mathematica (if not, we need to apply A2M to it as well). We write:

A2M(AST) = AST.cond && A2M(AST.then) || !AST.cond && s==sP

7.6 If-Then-Else

We assume that a node AST that represents an if-then-else statement has three subtrees, one that represents the condition of the if, one that represents the then branch, and one that represents the else branch. For simplicity, we further assume that the condition of the if is written in the syntax of Mathematica (if not, we need to apply A2M to it as well). We write: A2M(AST) = AST.cond &&

A2M(AST.then) || !AST.cond && A2M(AST.else).

7.7 While Loop

We assume that a node AST that represents a while-loop has two subtrees, one that represents the condition of the loop (AST.T) and one that represents the body of the loop(AST.B). For simplicity, we further assume that the condition of the loop is written in the syntax of Mathematica (if not, we need to apply A2M to it as well). We write:

A2M(AST) = ElemIR(AST.T, A2M(AST.B)) && invR(AST.T, A2M(AST.B) && !AST.T[sP]

Where ElemIR(T,B), where T is a unary predicate on space S and B is a binary predicate on space S, represents the elementary invariant relation discussed in previous chapter, and is defined as follows:

ElemIR(T,B) = (s==sP) || T(s) && Exists sPP: T(sPP) && B(sPP,sP).

Also, invR(T,B), where T is a unary predicate on S and B is a binary predicate, returns the invariant relation generated from the database of recognizers, according to the process depicted in previous chapter.

7.8 For Loop

We can make the for loop arbitrarily general, but for the sake of simplicity, we consider a for loop that iterates for some integer variable (i) ranging from a low value (l) to a high value (h) in increments of 1. If we let AST be the node that represents this loop and AST.B the subtree that represents its loop body, then we have:

A2M(AST) = Exists i, iP: i==l && iP==h+1 && invR(i $\leq$ h, iP=i+1 && A2M(AST.B)).

Note that variables i and iP are existentially quantified, so they are not part of the equation of A2M(AST), because in fact they are not part of the states s and sP, respectively (in most languages,

the index of a for loop is a local variable whose scope is limited to the loop). The invariant relation generator InvR() will tie the incrementation of i to the execution of the loop body (through the condition: iP==i+1 && A2M(AST.B)), and the condition (i==l && iP==h+1) will ensure that the loop body gets executed the right number of times.

CHAPTER 8

INVARIANT RELATIONS FOR AFFINE LOOPS

In this chapter, the semantics of loops whose loop body performs affine transformations on numeric variables are captured by two generic invariant relations. We claim that these two invariant relation (recognizer) are adequate to capture the semantics of any loop that contain affine equalities (also known as Linear Relations) on numeric variables, regardless of their number.

8.1 A Unary Recognizer

We consider a while loop w of the form w: {while (t) {b}} and we assume that the function of its guarded loop body (T ∩ B) is a subset of a relation of the form:

$B' = \{(s, s')|x' = ax + b\}$,

for some variables x, constants a and b such that a is different from 0 and 1. Then the following is an invariant relation for w:

$R0 = \{(s, s')|fr(log_{|a|}(|x' + \frac{b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|))\}$,

where fr(x) is the fractional part of x, $|x|$ is the absolute value of x, and $log_a(x)$ is the logarithm base a of x. In order to derive relation R0, we had to assume that a is different from 0 and 1.

by replacing the x with x′, we can check the correctness of R0.

$fr(log_{|a|}(|ax + b + \frac{b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

$fr(log_{|a|}(|\frac{(a-1)*(ax+b)+b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

$fr(log_{|a|}(|\frac{a^2x+ab-ax-b+b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

$fr(log_{|a|}(|\frac{a(ax+b-x)}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

$fr(log_{|a|}(|a|) + log_{|a|}(|\frac{(ax+b-x)}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

$fr(1 + log_{|a|}(|x + \frac{b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

Table 8.1 Unary Recognizer

ID	Formal Space, σ	Formal Clause, γ	Condition, α	Invariant Relation Template, ρ
1R0	`real x;` `const real a,b`	$x' = ax + b$	$a \neq 0$ $\wedge a \neq 1$	$fr(log_{\|a\|}(\|x + \frac{b}{a-1}\|))$ $= fr(log_{\|a\|}(\|x' + \frac{b}{a-1}\|))$

$fr(log_{|a|}(|x + \frac{b}{a-1}|)) = fr(log_{|a|}(|x + \frac{b}{a-1}|)$:

We know that fraction part of 1 is zero and has no effect. Hence we could prove that R0 is correct.

8.2 A Binary Recognizer

Now we consider a while loop w of the form w: {while (t) b} and we assume that the function of its guarded loop body (T ∩ B) is a subset of two affine relations of the form:

$B' = \{(s, s')|x0' = a0 \times x0 + b0\}$,

$B'' = \{(s, s')|x1' = a1 \times x1 + b1\}$,

for some variables x0, x1, and constants a0, a1 and b0, b1 such that a0, a1 are different from 0 and 1. Then the following is an invariant relation for w:

$R_1 = \{(s, s')|log_{|a_0|}(|x'_0 + \frac{b_0}{a_0-1}|) - log_{|a_1|}(|x'_1 + \frac{b_1}{a_1-1}|)$

$= log_{|a_0|}(|x_0 + \frac{b_0}{a_0-1}|) - log_{|a_1|}(|x_1 + \frac{b_1}{a_1-1}|)\}$

Extracting i from the second equation and replacing it in the first equation, we derive the following invariant relation:

$R''_1 = \{(s, s')|b_1 \times log_{|a_0|}(|x'_0 + \frac{b_0}{a_0-1}|) - x'_1 = b_1 \times log_{|a_0|}(|x_0 + \frac{b_0}{a_0-1}|) - x_1\}$. This is clearly a reflexive and transitive relation; to check that it is a superset of $(T \cap B)$, we can apply similar approach and replace x'_0 by $a_0x_0 + b_0$ and x'_1 by $a_1x_1 + b_1$ in the expression on the left and get the same expression on right side of the equation.

Table 8.2 Binary Recognizer

ID	Formal Space, σ	Formal Clause, γ	Condition, α	Invariant Relation Template, ρ
2R1	`real x0, x1; const int a0,a1,b0,b1`	$x_0' = a_0x_0 + b_0 \wedge x_1' = a_1x_1 + b_1$	$a_0, a_1 \neq 0 \wedge a_0, a_1 \neq 1$	$log_{\lvert a_0\rvert}(\lvert x_0 + \frac{b_0}{a_0-1}\rvert) - log_{\lvert a_1\rvert}(\lvert x_1 + \frac{b_1}{a_1-1}\rvert) = log_{\lvert a_0\rvert}(\lvert x_0' + \frac{b_0}{a_0-1}\rvert) - log_{\lvert a_1\rvert}(\lvert x_1' + \frac{b_1}{a_1-1}\rvert)$

```
While(x<y)
{
    x = 5*x + 3;
    y = 3*y + 5;
    w = 3*z + w;
    z = z + 6;
}
```

Figure 8.1 Source code to test the invariant generator tools

$$
\begin{aligned}
& fr(log_5(|x0 + \tfrac{3}{4}|)) = fr(log_5(|x + \tfrac{3}{4}|)) \\
& \wedge log_5(|x0 + \tfrac{3}{4}|) - log_3(|y0 + \tfrac{5}{2}|) = log_5(|x + \tfrac{3}{4}|) - log_3(|y + \tfrac{5}{2}|) \\
& \wedge w0 - \frac{3z0(z0-6)}{12} = w - \frac{3z(z-6)}{12} \\
& \wedge \frac{4x0+3}{5^{\frac{z0}{6}}} = \frac{4x+3}{5^{\frac{z}{6}}} \\
& \wedge ((x = x0 \wedge y = y0 \wedge z = z0 \wedge w = w0) \vee (x0 < y0 \wedge \tfrac{x-3}{5} < \tfrac{x-5}{3})).
\end{aligned}
$$

Figure 8.2 Our invariant relation for program in Figure 8.1.

8.3 Comparison With Other Tools

We attempted to run a number of publicly available loop invariant generating tools on the same loop for the sake of comparison to see how our findings compared to theirs. We ran these tools on the program in Figure 8.1 and compared their results with our invariant which is shown in Figure 8.2

1. **Aligator**. When we run Aligator [64] on the same program, it declares that there no P-solvable solutions to this search, and returns no result, because it searches for loop invariants as polynomials.

2. **Daikon**. Whereas our loop invariants can be parameterized in terms of the initial state, Daikon requires that we specify the initial values of the program variables. We initialize the program variables as follows: {x=0;y=90;z=0;w=1;}. Daikon returns the following loop invariant: $x < y \wedge x \neq w \wedge y > z \wedge y > w \wedge z \neq w$.

3. **Sting**. When we run Sting [94] on this loop, with the same initialization as above, we find: $x \leq 1003 \wedge y \geq 90 \wedge -185x + 403y \geq 36270$.

Multiple attempts to run other tools such as InvGen [52] and ESMBC [43], have failed.

CHAPTER 9

IMPLEMENTATION AND DEMOS

In this chapter, we talk describe how this tool is implemented. There are two main sections, the backend implementation and invariant relations semantic matching.

9.1 From Source Code to AST

As we explained in Section 3.3, there are three step to transform a source code into a function. We mentioned that Java Parser is the tool that we used to get the abstract syntax tree of a program. Now, we want to show more details about how Java Parser works and how information is organized and represented.

In the inner representation of Java Parser, every vertex of the AST is stored as a node. This presentation is not exactly same as the AST, but it contains some extra information about each node. For example, MethodDeclaration is considered as a node and it has children nodes that represent the methods name, its argument parameters (if it has them) and type, what it returns and it's body(Figure 9.1). There are other language keywords used to define a method e.g. final, static, abstract etc. These are defined as a modifier property on the MethodDeclaration itself. For the first step, the source code is loaded and parsed through Java Parser and the abstract syntax tree is returned and ready to be navigated. The output of parsed tree is returned as a *CompilationUnit* object and we can invoke its function to traverse the tree and sub-trees.

CompilationUnit cu = StaticJavaParser.parse(new FileInputStream(SourceCodeFilePath));

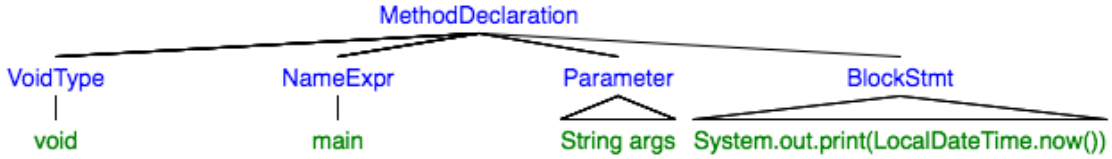


Figure 9.1 Method declaration in Java Parser.

9.2 From AST to Mathematica Equations

After we get the compilation unit which represent the AST, we are able to navigate it and evaluate each node. The compilation unit includes information about every single part of the source code, and provides a lot of information about each node (we don't need all of these information here). To navigate the AST, we used the Breadth First Search algorithm, included in Java parser framework.

```
expressionIterator = new Node.BreadthFirstIterator(cu);

while (expressionIterator.hasNext())
{
        Node node1 = expressionIterator.next();
        String nodeClass = node1.getClass().getSimpleName();

        if (nodeClass.equals("MethodDeclaration") &&
        ((MethodDeclaration)node1).getNameAsString().equals("main")
        == true)
        {
            ...
        }
        else if (...)
        {
            ...
        }
}
```

As we described in Chapter 7, for each type of node there is an method to evaluate the node. This method generates and returns the equivalent relation in Mathematica syntax. For instance, let's take a look at the method that we extract the variables from a variable declaration statement.

```
public static List<Expression> VisitDeclaration
(VariableDeclarationExpr expr,
    BlockStmt block, String addTo,String initValue)
    {
        NodeList<VariableDeclarator> list = expr.getVariables();
        List<Expression> decExpr = new ArrayList<Expression>();
        for (int i = 0; i < list.size(); i++)
        {
            block.variables.add( new Variable(list.get(i)
            .getNameAsString(),
            null,true));

            BinaryExpr bin = new BinaryExpr();
            NameExpr nm = new NameExpr(list.get(i).
            getNameAsExpression().
            asNameExpr().toString() + addTo);
            bin.setLeft(nm);
            bin.setRight(new NameExpr(initValue));
            bin.setOperator(BinaryExpr.Operator.EQUALS);
            decExpr.add(bin);
        }
    }
```

Figure 9.2 shows the Java Parser representation at runtime. Every node has a list of children nodes. Based on the type of he node, there are some properties and sublists. This makes our work very easy to find and navigate the expression under each node.

9.3 From Mathematica to Program Function

The last step is to collect and integrate all of the generated relations into one final relation. To solve and transform this final relation to program function, we need to connect to the Mathematica's services and use its Java API. To make this, Mathematica program must be installed on local machine and verified with a license. Figure 9.3 shows methods that we implemented to connect to Mathematica.

To solve the final relation, we used *Reduce* to solve the equations. Since we want to find the final state of variables (sP) based on initial state(s), we need to call Reduce along with a list

Figure 9.2 Java parser runtime representation.

parameters which includes the primed variables.

There is another parameter, *Backsubstitution* → *True* which gives explicit values for the variables and removed unnecessary dependencies between variables. We can see the final generated relation.

This complete relation is the input for the *SendExpressionToKernel* method.

```
Simplify[
  Reduce[Reduce[
    Exists[{kPP, nPP, fPP},
     Exists[{kP, nP, fP},
       kP == kPP && nP == nPP && fP == fPP && kP == k && nP == n &&
        fP == f] &&
      Exists[{k, n, f},
       k == kPP && n == nPP && f == fPP &&
        InvR[{k, n, f, kP, nP,
          fP}, {{k, kP, "int"}, {n, nP, "int"}, {f, fP, "int"}},
         kP != nP,
         Reduce[Exists[{kPP, nPP, fPP},
           Exists[{kP, nP, fP},
```

```
public connection()
{}
public static KernelLink ml;
public static  void connect(String exepath) throws MathLinkException
{

   String s = "-linkmode launch -linkname '" + exepath+ "'" ; //E:\\Software\\mathematica\\mathkernel.exe'";
   ml = MathLinkFactory.createKernelLink(s);
   ml.connect();
   // this is necessary to discard the very first packet,
   ml.discardAnswer();

}

public static String SendExpressionToKernel(String expr)
{
   String outform = ml.evaluateToOutputForm(expr, i: 0); // both are similar, small diffirence in the format
   String inform = ml.evaluateToInputForm(expr, i: 0);
   return inform;

}
```

Figure 9.3 Mathematica api connection.

```
            kP == kPP && nP == nPP && fP == fPP && fP == f*k &&
             kP == k && nP == n] &&
           Exists[{k, n, f},
            k == kPP && n == nPP && f == fPP && kP == k + 1 &&
             nP == n && fP == f]], {kP, nP, fP},
         Backsubstitution -> True]]]], {kP, nP, fP},
   Backsubstitution -> True], {kP, nP, fP}],
 Element[{k, n, f, kP, nP, fP}, Integers]] /.
ConditionalExpression :> And
```

9.4 A Full Sample

In this section, we provided a full sample that covers all of the functionalities in our tool. There are three conditional branches, two single loops and one nested loop. There is also a member function which is called from the main function. The target source code is shown below.

```
public class Main {
public static int f(int x) {
    x = 7 * x + 7;
    return x;
```

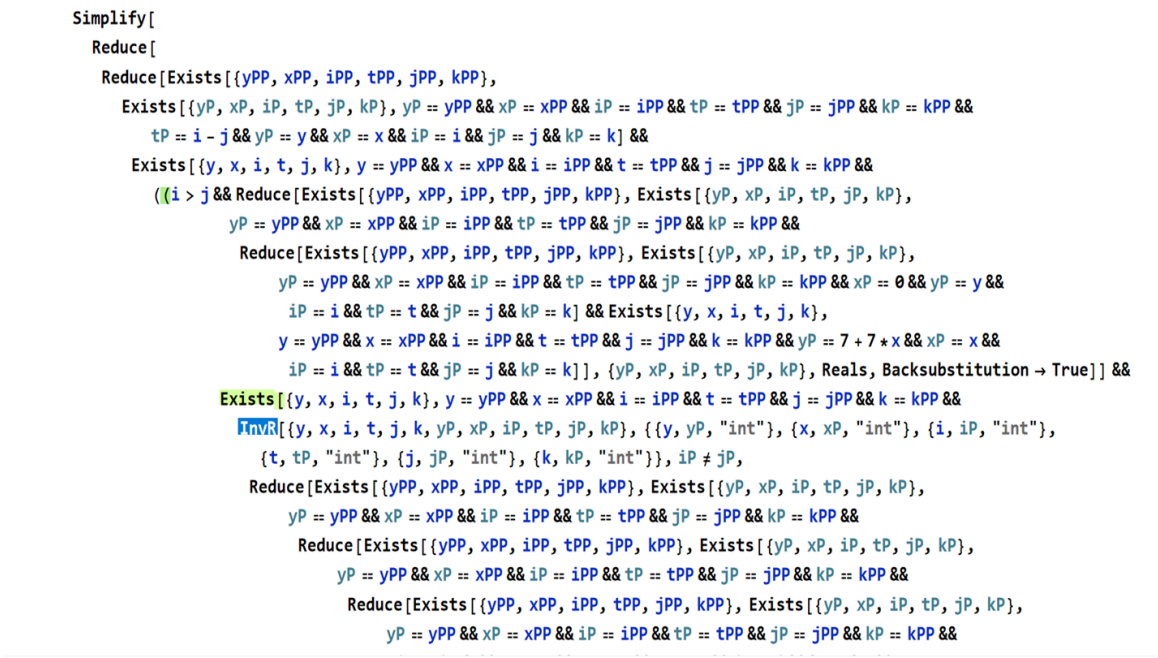
```
    }

    public static void main(String argv[]) {
        int x, y, t, i, j, k;
        //  read x, y, i, j, k, t;
        t = i - j;
        if (i > j) {
            x = 0;
            y = f(x);
            while (i != j) {
                i = i + k;
                k = k + 1;
                i = i - k;
                y = f(y);
            }
        } else if (j > i) {
            while (j != i) {
                j = j + k;
                k = k - 1;
                j = j - k;
                y = f(y);
            }
        } else {
            while (t != i) {
                for (int z = 0; z != y; z = z + 1) {
                    x = x + 1;
                }
                y = x - y;
                t = t + 1;
            }
            Label L5;
        }
        k = i + j;
        j = 2 * k;
        Label L6;
    }
}
```

Also, in figures 9.4, 9.5 and 9.6, we can see the Mathematica relations that we generate in our tool:

And finally, we have the function of this program:

```
(i<j&& yP == 1/6 (-7+7^(i-j) (7+6 y))&& xP==x && i==iP
&& tP==i-j && i==jP && kP>=k && kP = k + j - i)
||(j>=0 && i==j && yP == y Fibonacci[-1+j]+ x Fibonacci[j]
&& xP == y Fibonacci[j]+ x Fibonacci[1+j] && iP == j
```



```
Simplify[
  Reduce[
   Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP},
     Exists[{yP, xP, iP, tP, jP, kP}, yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
       tP == i - j && yP == y && xP == x && iP == i && jP == j && kP == k] &&
      Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP &&
        ((i > j && Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                 Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                     yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP && xP == 0 && yP == y &&
                      iP == i && tP == t && jP == j && kP == k] && Exists[{y, x, i, t, j, k},
                     y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP && yP == 7 + 7 * x && xP == x &&
                      iP == i && tP == t && jP == j && kP == k]], {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]] &&
              Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP &&
               InvR[{y, x, i, t, j, k, yP, xP, iP, tP, jP, kP}, {{y, yP, "int"}, {x, xP, "int"}, {i, iP, "int"},
                 {t, tP, "int"}, {j, jP, "int"}, {k, kP, "int"}}, iP ≠ jP,
                Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                    yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                     Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                         yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                          Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                              yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
```

Figure 9.4 Mathematica relation1.

```
&& tP == j && jP == j && kP==k)
```

```
                                    iP == i + k && yP == y && xP == x && tP == t && jP == j && kP == k] &&
                                  Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP &&
                                    k == kPP && kP == k - 1 && yP == y && xP == x && iP == i && tP == t && jP == j]],
                                 {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]] &&
                               Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP &&
                                 k == kPP && iP == i - k && yP == y && xP == x && tP == t && jP == j && kP == k]],
                              {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]] &&
                            Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP &&
                              yP == 7 + 7 * y && xP == x && iP == i && tP == t && jP == j && kP == k]], {yP, xP, iP, tP, jP, kP},
                           Reals, Backsubstitution → True]]]], {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]) ||
(! i > j && ((j > i && InvR[{y, x, i, t, j, k, yP, xP, iP, tP, jP, kP}, {{y, yP, "int"}, {x, xP, "int"},
        {i, iP, "int"}, {t, tP, "int"}, {j, jP, "int"}, {k, kP, "int"}}, jP ≠ iP,
      Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
          yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
           Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
              yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
               Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                  yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                   jP == j + k && yP == y && xP == x && iP == i && tP == t && kP == k] &&
                 Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP &&
                   k == kPP && kP == k + 1 && yP == y && xP == x && iP == i && tP == t && jP == j]],
                {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]] &&
```

Figure 9.5 Mathematica relation2.

```
                                  Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP &&
                                    k == kPP && jP == j - k && yP == y && xP == x && iP == i && tP == t && kP == k]],
                                 {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]] &&
                               Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP &&
                                 yP == 7 + 7 * y && xP == x && iP == i && tP == t && jP == j && kP == k]], {yP, xP, iP, tP, jP, kP},
                              Reals, Backsubstitution → True]]) || ((! j > i && InvR[{y, x, i, t, j, k, yP, xP, iP, tP, jP, kP},
                            {{y, yP, "int"}, {x, xP, "int"}, {i, iP, "int"}, {t, tP, "int"}, {j, jP, "int"}, {k, kP, "int"}},
                            tP ≠ iP, Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                                yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                                 Reduce[Exists[{yPP, xPP, iPP, tPP, jPP, kPP}, Exists[{yP, xP, iP, tP, jP, kP},
                                    yP == yPP && xP == xPP && iP == iPP && tP == tPP && jP == jPP && kP == kPP &&
                                     Exists[{z, zP}, z == 0 && Not[zP ≠ y] && InvR[{y, x, i, t, j, k, yP, xP, iP, tP, jP, kP},
                                        {{y, yP, "int"}, {x, xP, "int"}, {i, iP, "int"}, {t, tP, "int"}, {j, jP, "int"},
                                         {k, kP, "int"}, {z, zP, "int"}}, zP ≠ y, xP == x + 1 && yP == y && iP == i &&
                                         tP == t && jP == j && kP == k && (zP == z + 1)]]] && Exists[{y, x, i, t, j, k},
                                      y == yPP && x == xPP && i == iPP && t == tPP && j == jPP && k == kPP && yP == x - y && xP == x &&
                                       iP == i && tP == t && jP == j && kP == k]], {yP, xP, iP, tP, jP, kP}, Reals,
                                    Backsubstitution → True]] && Exists[{y, x, i, t, j, k}, y == yPP && x == xPP && i == iPP &&
                                   t == tPP && j == jPP && k == kPP && tP == t + 1 && yP == y && xP == x && iP == i && jP == j && kP == k]],
                                {yP, xP, iP, tP, jP, kP}, Reals, Backsubstitution → True]]))))]], {yP, xP, iP, tP, jP, kP},
  Reals, Backsubstitution → True], {yP, xP, iP, tP, jP, kP}],
 Element[{y, x, i, t, j, k, yP, xP, iP, tP, jP, kP}, Integers]] /. ConditionalExpression :→ And
```

Figure 9.6 Mathematica relation3.

CHAPTER 10

INGREDIENTS FOR SCALABILITY

To support the user in analyzing and verifying the correctness of a program, we provide an automated tool based on this research. The main goal of this tool is to provide an interactive environment for users to submit queries about the functional attributes of the program, as written. The tool is a web application consist of two windows: one window holds the source code, another one is like a console that user can write queries and commands. We will show how finding the function of a program could be used in multiple use-cases and scenarios.

10.1 Assume(), Capture(), Verify(), Establish()

Our tool is an interactive system which provides four functions to support the program analysis. Here we explain the functions. These functions could be applied at a specific line of the code, which we specified here as some labels, or invoked on a highlighted and named program part.

Assume():

This clause makes an assumption about the state of the program at some specific labels in the source code or about a hilighted and named program part.. One application for this clause could be to declare the precondition of a program (e.g., the conditions that we assume the inputs of a program part or subroutine to satisfy). Other possible uses include, e.g. to make an assumption about a function that is called in a program part, as a condition for proving a property about the program part.

Capture():

This statement may be used to characterize program states at a location in the source code (e.g., a loop invariant) or to compute the function of a program part (e.g., the function of a loop).

Verify():

This clause is used to verify functional properties of program states. for instance, we may want to verify that the states of a program at a specific position satisfy some conditions, given what we know from the Assume() statements; or we may want to verify that a program part satisfies (is correct with respect to) some specification.

Establish():

We want to employ program repair technology to give users the ability to fix a program portion in accordance with a local specification. To illustrate, if the user tries to confirm that a software component adheres to a specification but the Verify() query fails, the user can submit the Establish() question with the same inputs as Verify() to see whether the suggested fixes make sense. Although the technique for program repair struggles with massive programs [47], it could work well for little pieces of programs. this function is not implemented, but is part of our plan for future research.

10.2 A Use Case

To understand better how these functions could be integrated into our tool, we provided a demo that shows a series of commands and queries. The codes in this sample are in a simple Java program. The tool, as it is shown in Figure 10.1, loads a source code as the input of the software(left window). As we see there are three labels in the code, pre, post and inv. These points are marked and we can run our queries at those specific positions. We intend to investigate the correctness of $z = x * y$ at the label *post*. In figure 10.2 we run the first query to verify this equation. After running this command

the system returns False, which means the assertion is not verified. We realized that $x * y$ needs both variables to have value and be initialized. So as the next try in figure 10.3 we add an assumption for variables. The system assigned these values to x and y. Adding these assumption, we verify the equation $z = x * y$. However, the user receives the False as the verification result 10.4. There is a same problem with initialization of z. So in figure 10.5 we try to add assumption for z and again check the same verification. Adding the initialization could solve the problem and we have verified equation, but this time the system returns another message as the reachability is unverified. This message tells us that we may or may not reach to post label but if it reaches, then it is verified. The problem with previous unreachability is, the loop may never terminate. If the initial value for y is negative, then the condition $y \neq 0$ will always be true and the loop turns into an infinite loop. In order to prevent the program from any infinite loop, we add one more assumption for y in Figure 10.6. Finally the system returns True for that query. In the next try we added one of the assumption to the source code and remove it from the Assume() arguments. Running the same $Verify(z = x * y)$ returns True as we expect. We can see this new query in Figure 10.7.

In order to prevent and remove the possibility of getting an infinite loop, we need to change the source code and make the y variable positive. Therefore, we should not need to have any assumption like $y \geq 0$. We remove this assumption and run the verify function again. We expect to have verified query as rest of the code and other assumptions are same as previous example.

Having our query verified, we intend to find the state of the program at *inv* label. This can be achieved by calling the Capture() function. The output is what we call it the function of the program, but here it is the function at the specific position 10.9. This simply means, if we run the program while the initial value for x and y is x0 and y0, then what we will get in the end and after finishing the loop is same as the function that we got from Capture(), both have same answer.

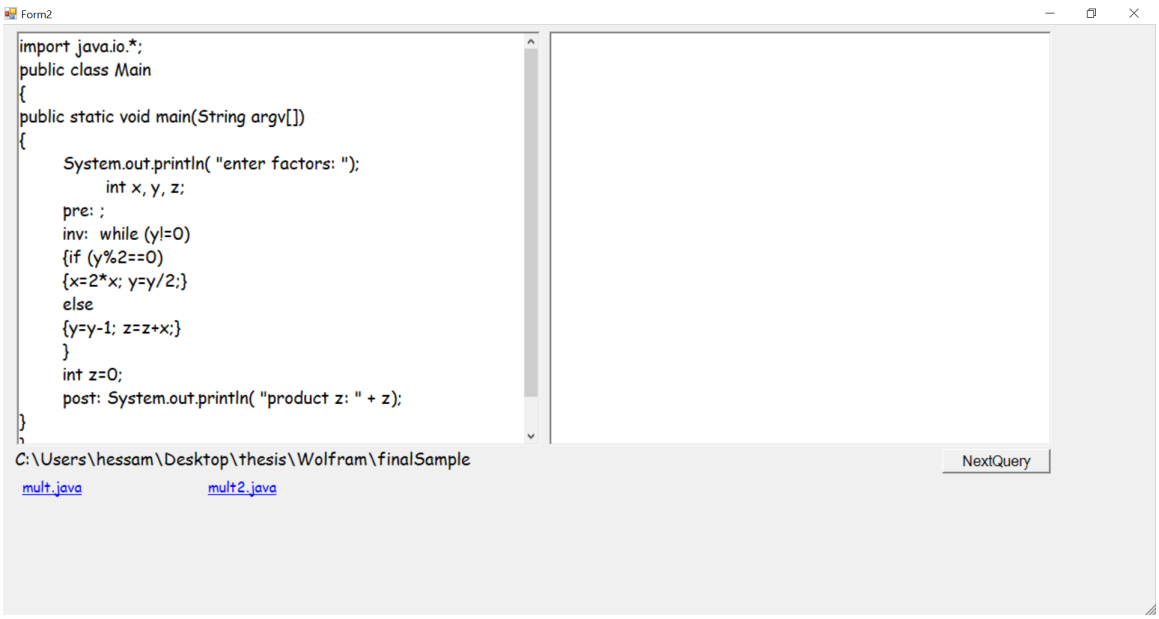


Figure 10.1 Use case 1, tool overview.

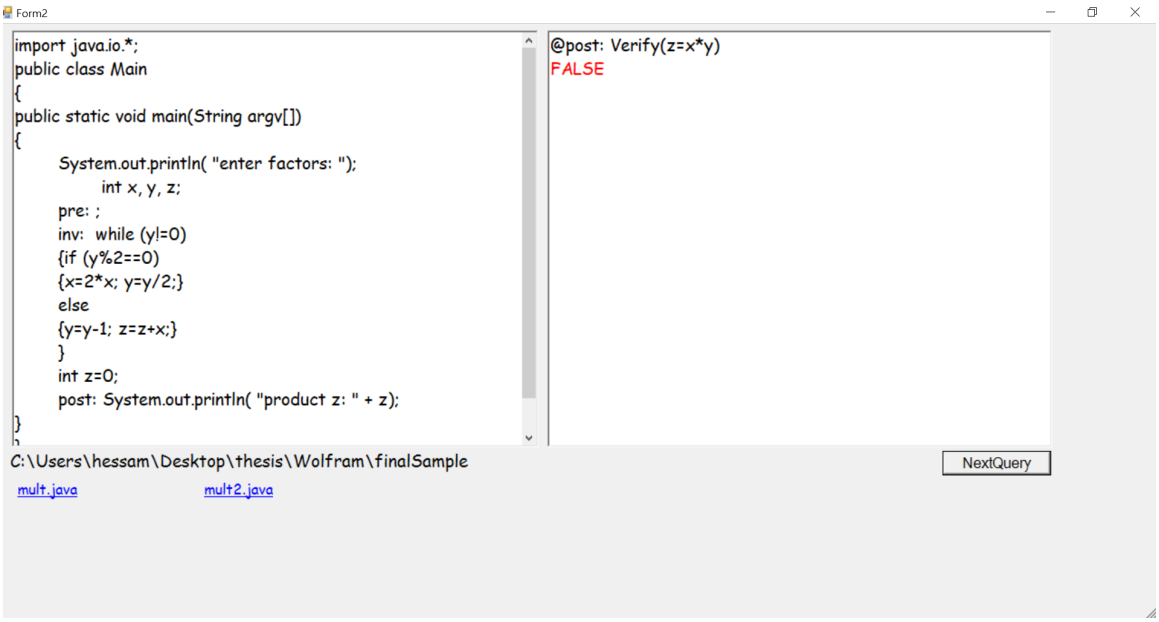


Figure 10.2 Use case 2.

10.3 Path and Path Function

In order to answer queries such as those, we discussed in the previous section, we may need to compute the function of programs, program parts, and execution paths. Hence to proceed, we

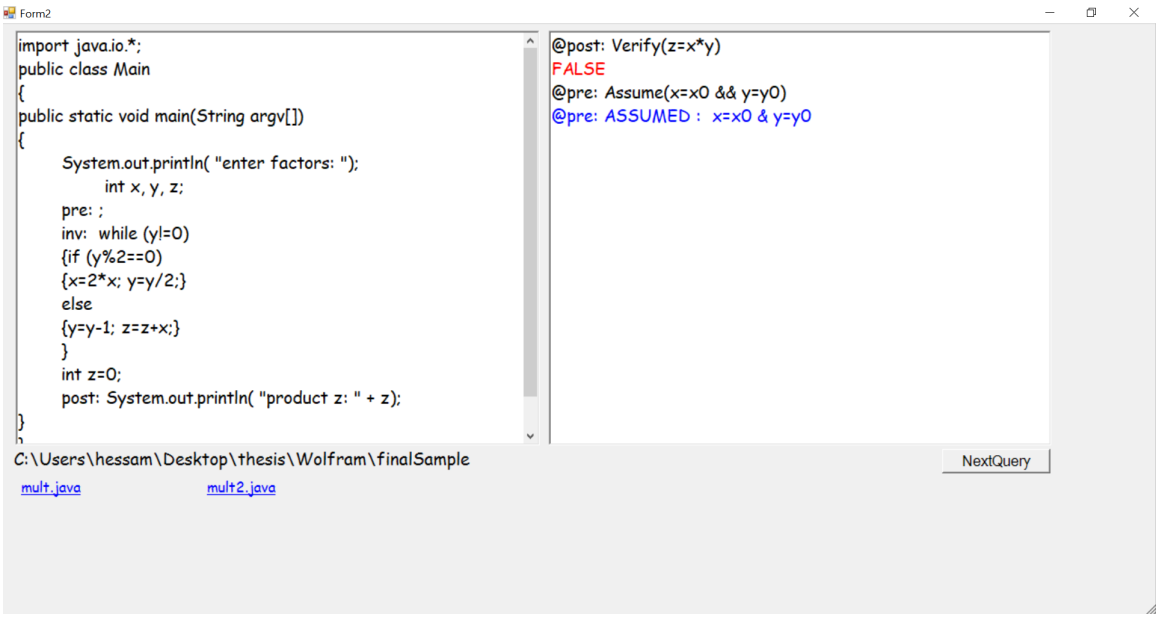


Figure 10.3 Use case 3.

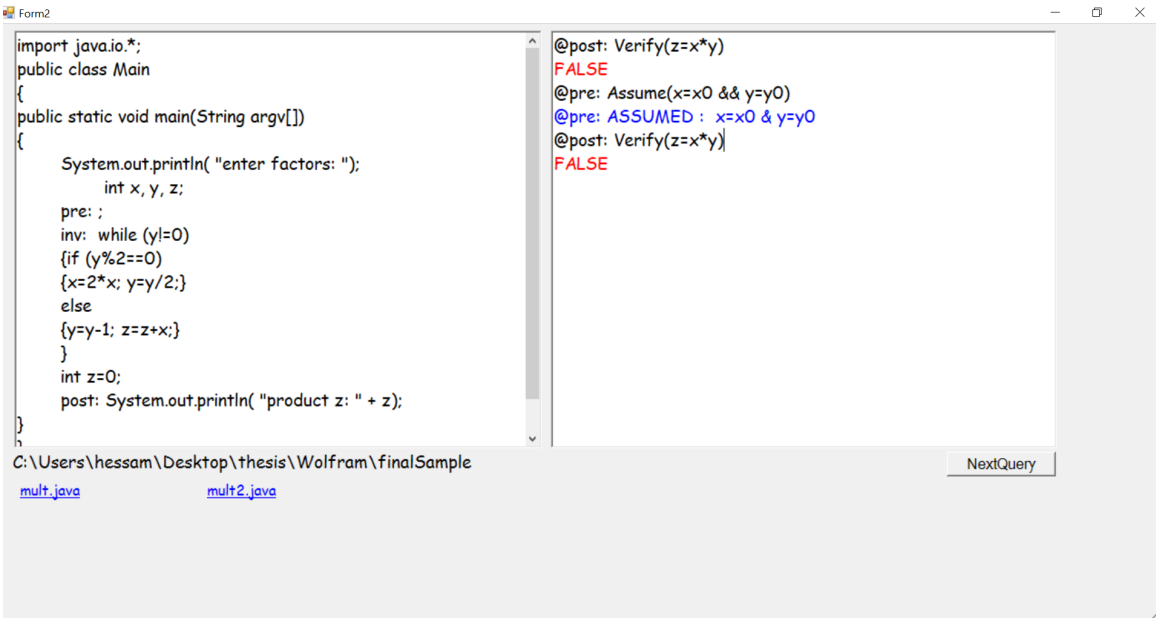


Figure 10.4 Use case 4.

must define paths and path functions. The path is simply defined as a series of statements which executed one by one, from a start point to end point. The statements could be routine programming statements such as, *variable declarations*, *assignment statements*, *Sequence statements*, *Conditional*

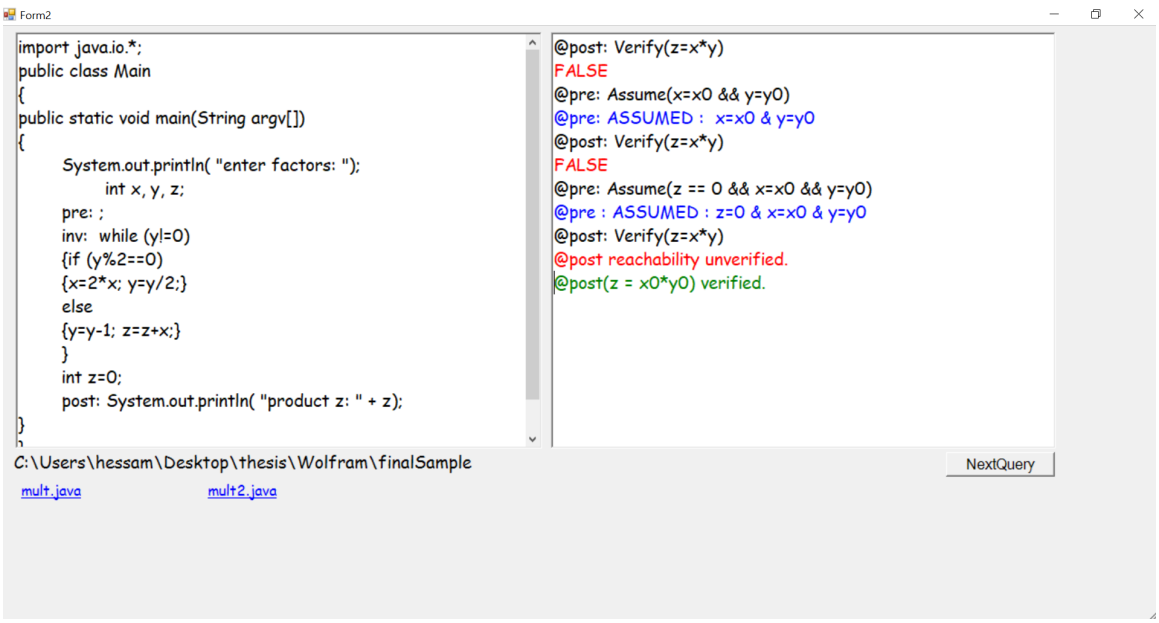


Figure 10.5 Use case 5.

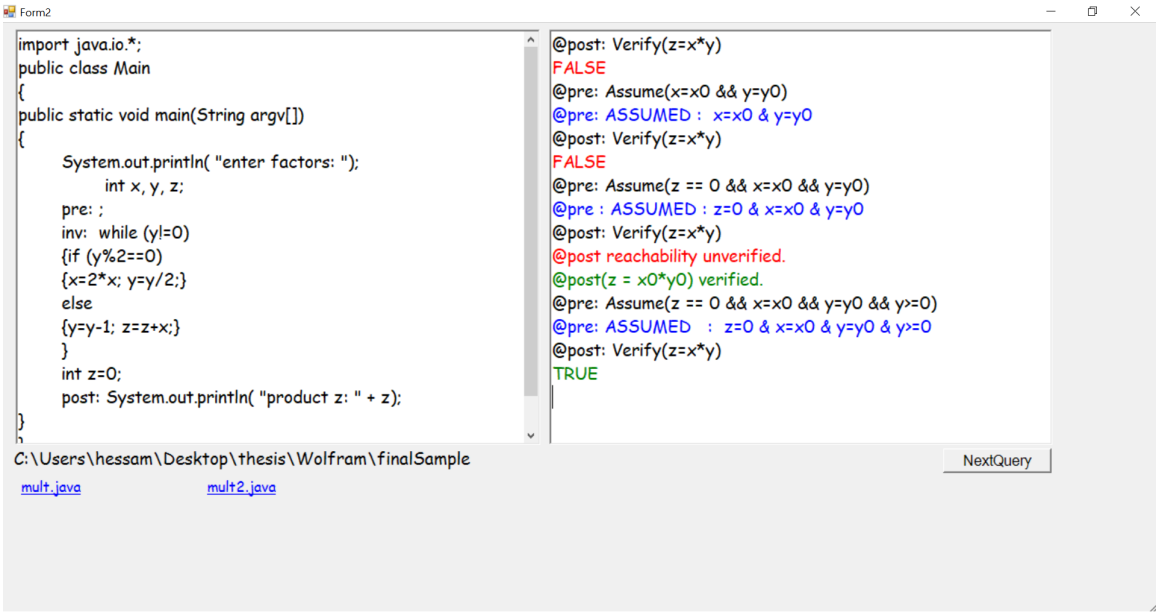


Figure 10.6 Use case 6.

statements, *alternation statements*, *iteration statements*, *Function calls* and *labeled statements*. the labels indicates the target position the code that we want to run our query on. For example, two labels can be marked as the start and end point of a unique path. Here we have some definitions for the

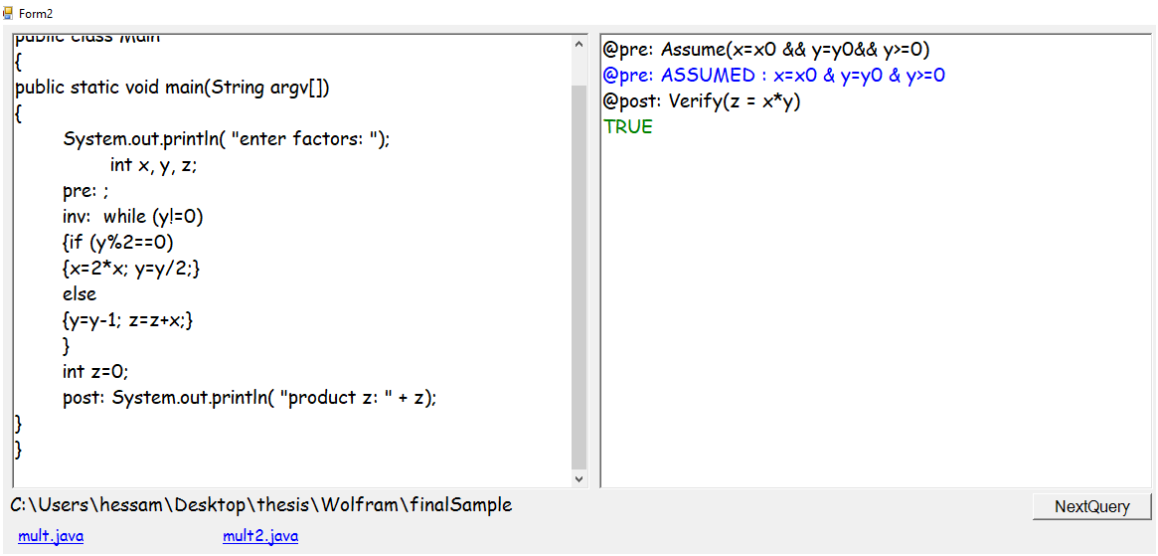


Figure 10.7 Use case 7.

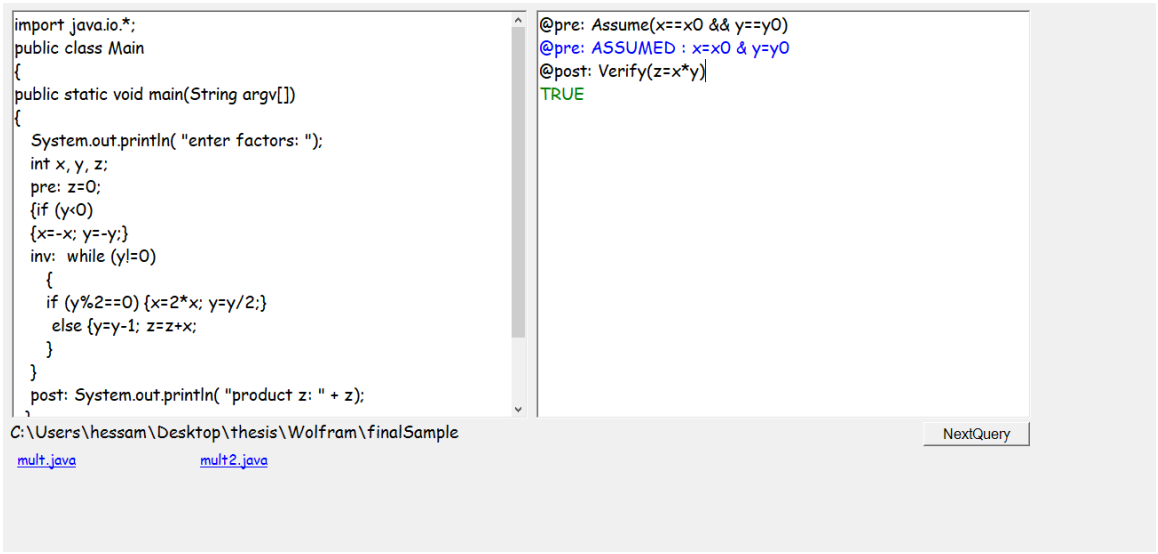


Figure 10.8 Use case 8.

path.

elementary statement: we define an elementary statement which could be any of the following:

- An assignment statement
- A function call

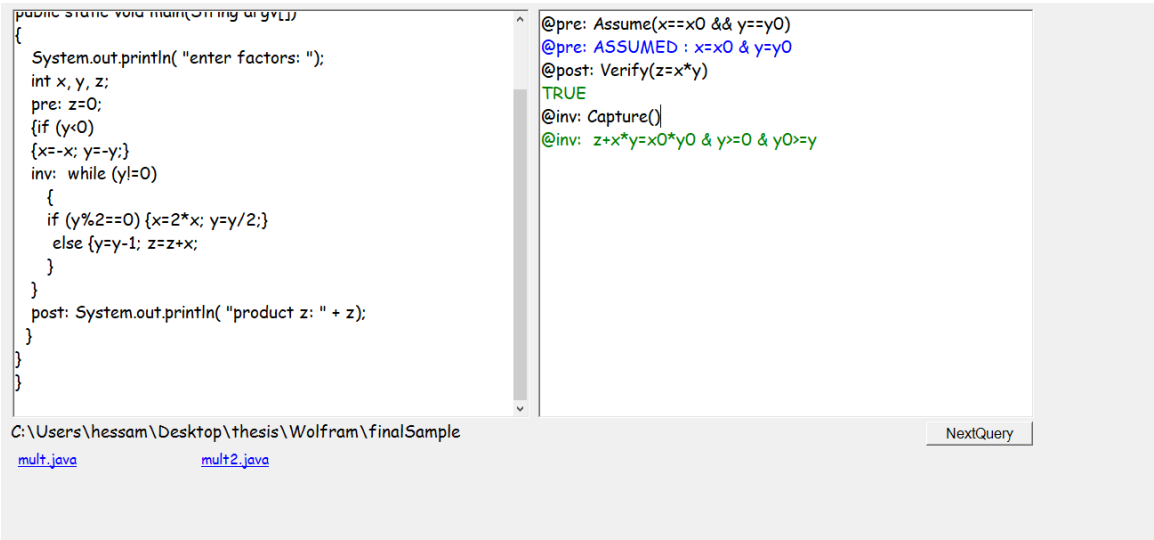


Figure 10.9 Use case 9.

- A condition test of the form:
  - (trueTest(condition C))
  - (falseTest(condition C))

With these definitions, we can now have more specific definition for a *path*. A path through a program is a sequence of elementary statements separated by semicolons such that whenever two elementary statements <es1> and <es2> follow each other in the path, then one of the following conditions holds:

- <es1> and <es2> follow each other in the program,
- <es2> is derived from the condition of a conditional statement or an alternative statement or an iterative statement which follows <es1> in the program
- <es1> is the last elementary statement of a conditional statement or an alternative statement that precedes <es2> in the program
- <es1> is derived from the condition of an iterative statement that precedes <es2> in the program
- <es1> has the form trueTest(C), where c is the condition of a conditional statement or an alternative statement, and <es2> is the first statement of the then-branch
- <es1> has the form falseTest(C), where c is the condition of an alternative statement and <es2> is the first statement of the else branch

- <es1> has the form trueTest(C), where c is the condition of an iterative statement and <es2> is the first statement of the loop body

As an application, we consider the following program.

```
{
int x; int y;
read(x); read(y); // assuming x>0, y>0
while (x!=y)
{
    if (x>y)
    {
        x=x-y;
    }
    else
    {
        y=y-x;
    }
}
write(x);
}
```

Sample paths through this program include the following:

- **p0**: int x; int y; read(x); read(y); ((x! =y)? false); write(x);
- **p1**: int x; int y; read(x); read(y); ((x! =y)? true); ((x>y)? true); x=x-y; ((x! =y)? false); write(x);
- **p2**: int x; int y; read(x); read(y); ((x! =y)? true); ((x>y)? false); y=y-x; ((x! =y)? false); write(x);
- **p3**: int x; int y; read(x); read(y); ((x! =y)? true); ((x>y)? true); x=x-y; ((x! =y)? true); ((x>y)? true); x=x-y; ((x! =y)? false); write(x);

Path Function: The function of a path is the function determined inductively using the semantics of statements. The conditions that are part of the path execution has to be true or false. It mean, if there are some conditions in a path from *L1* to *L2*, these conditions evaluated as true or false. The semantics of a condition is defined by the following equations:

$\{[c?true] = \{(s, s')|s = s' \wedge c(s)\}$

$\{[c?false] = \{(s, s')|s = s' \wedge \neg c(s)\}$

In order to achieve the function of the path, we have to find the function of each statement and do the product for the statements' functions consecutively. As an illustration, consider the following functions on a space S defined by integer variables x and y:

$F1 = \{(s, s')|x > y \wedge x' = 2x + y \wedge y' = 2y + x\}$

$F2 = \{(s, s')|x > 2y \wedge x' = 3x + 2y \wedge y' = 3y + 2x\}$

Then the product of these two functions yields the following result:

$F1 * F2 = \{(s, s')|x > y \wedge 2x + y > 2(2y + x) \wedge x' = 3(2x + y) + 2(2y + x) \wedge y = 3(2y + x) + 2(2x + y)\}$

10.4 Semantic Definition

Here we intend to to explore the semantic behind our the four functions.

@L: Assume(C): $\{(s, s')|s' = s \wedge C(s)\}$

@L: Capture():

Let P be the path from the first executable statement to L, we return two terms:

1. Reachability Condition: ($\exists$ sP: (s,sP) $\in$ P). This is the condition under which label is reached.
2. State Assertion: ($\exists$ sP: s==sP $\wedge$ ($\exists$ s: (s,sP) $\in$ P)). This is the strongest assertion we know to hold at label L.

@L: Verify(C): State Assertion $\implies$ C.

@L: Establish(C): change the code of the program in such a way that @L: Verify(C) returns TRUE.

10.5 Illustration and Demo

In this section, we show how the web application works. We run our functions on a simple program and test multiple queries for different paths.

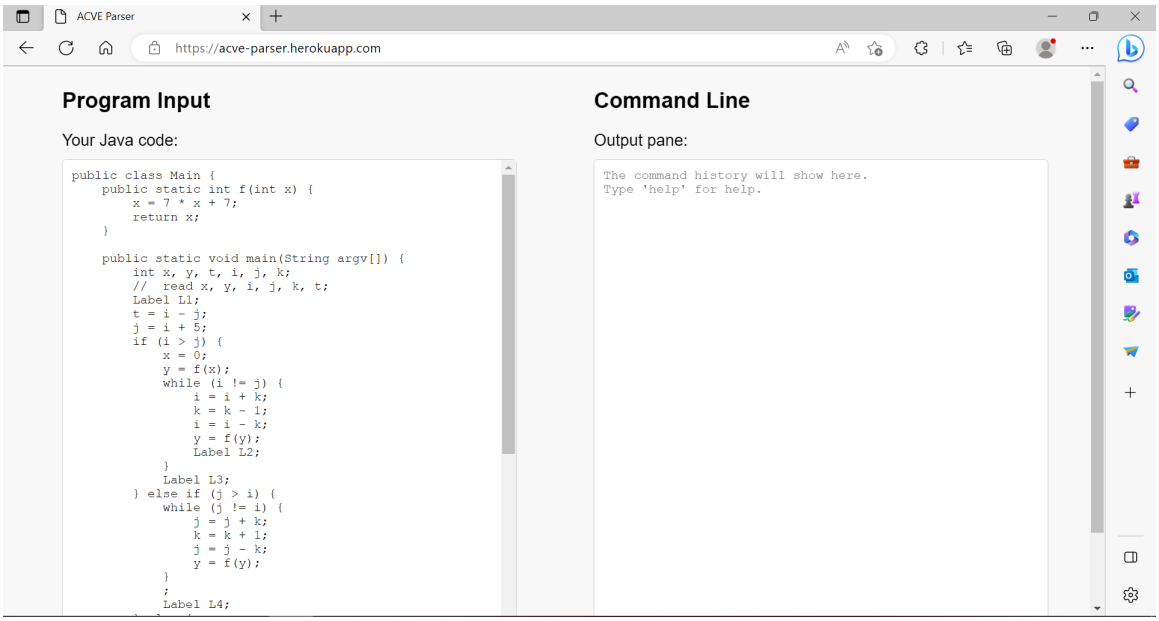


Figure 10.10 Path generator web application.

Figures 10.10 and 10.11 shows an overview of the path generator application and the tested program. The first query we want to run is add an assumption at label *L1*(Figure 10.12). We want to assume that at that position the value of the variable *i* is positive. Therefore, we run the first command as Assume($i \geq 0$).

Our second, try is to capture reachability condition and state of program at the label L2(Figure 10.14). We can see in figure 10.13 how this path is generated. Similarly, in the next commands, we invoke the capture for labels *L3 and L4*. We notice that the loop which precedes the L4 is an infinite loop and will never terminate. Hence, we never reach to L4 and the results for the capture at L4 is False. We can see these queries in figures 10.15 and 10.17. Moreover, we see the results of Verify function in figures 10.16 and 10.18. We checked the few expressions at those labels and the get the results.

```
public class Main {
    public static int f(int x) {
        x = 7 * x + 7;
        return x;
    }

    public static void main(String argv[]) {
        int x, y, t, i, j, k;
        //  read x, y, i, j, k, t;
        Label L1;
        t = i - j;
        j = i + 5;
        if (i > j) {
            x = 0;
            y = f(x);
            while (i != j) {
                i = i + k;
                k = k + 1;
                i = i - k;
                y = f(y);
                Label L2;
            }
            Label L3;
        } else if (j > i) {
            while (j != i) {
                j = j + k;
                k = k - 1;
                j = j - k;
                y = f(y);
            }
            ;
            Label L4;
        } else {
            while (t != i) {
                for (int z = 0; z != y; z = z + 1) {
                    x = x + 1;
                }
                y = x - y;
                t = t + 1;
            }
            Label L5;
        }
        k = i + j;
        j = 2 * k;
        Label L6;
    }
}
;
```

Figure 10.11 Full program

Figure 10.12 Assume query at label L1.

```
public class Main {

    public static void Assume(Boolean x) {
    }

    public static void FalseTest(Boolean x) {
    }

    public static void TrueTest(Boolean x) {
    }

    public static int f(int x) {
        x = 7 * x + 7;
        return x;
    }

    public static void main(String[] argv) {
        int x, y, t, i, j, k;
        Assume(i >= 0);
        t = i - j;
        TrueTest(i > j);
        x = 0;
        y = f(x);
        TrueTest(i != j);
        i = i + k;
        k = k + 1;
        i = i - k;
        y = f(y);
    }
}
```

Figure 10.13 Generated path at L2.

```
Reachability Condition:
j >= 0 && i > j
||
j < 0 && i >= 0

State Assertion:
 ( x == 0 && y == 56 && t > 0 && j >= 0 && i == -1 + j + t )
||
 ( x == 0 && y == 56 && t > 0 && -t <= j < 0 && i == -1 + j + t )
```

Figure 10.14 Result of running capture at L2.

```
Reachability Condition:
i >= 0 && j < 0
||
i > 0 && 0 <= j < i
||
i >= 0 && j < i

State Assertion:
 ( x == 0 && j < 0 && t >= -j && y == (7*(-1 + 7^(1 + t)))/6
 && i == j )
||
 ( x == 0 && j >= 0 && t > 0 && y == (7*(-1 + 7^(1 + t)))/6
 && i == j )
||
 ( x == 0 && ((j < 0 && t >= -j && y == (7*(-1 + 7^(1 + t)))/6
 && i == j) || (j >= 0 && t > 0 && y == (7*(-1 + 7^(1 + t)))/6
 && i == j)) )
```

Figure 10.15 Result of running capture at L3.

```
@L3: Verify(t+j>=0)

True

@L3: Verify( i != j)

False

@L3: Verify(j>0)

False
```

Figure 10.16 Running verify for few expression at L3.

```
@L4: Capture()

Reachability Condition:
False

State Assertion:
False
```

Figure 10.17 Capture the path at L4.

```
@L4: Verify(1 == 0)

True
```

Figure 10.18 Verify at L4.

CHAPTER 11

CONCLUSION

The symbolic execution of programs is a valuable capability, due to its interest in software testing, software maintenance, software verification, and broadly in any activity that involves software understanding (software reuse, component based software development, etc). In this dissertation, we presented our work developing and evolving a tool that takes source code and computes the function that it defines between its initial states and its final states. Although the current solution is focused on Java, we may create a version to support any other C-like language using a parser that converts the source code into an abstract syntax tree. We owe our success to two effective artifacts of automated software engineering: compiler generation technology and symbolic equation solvers;

our approach could be described in three steps:

1. **J2A:Mapping the Source Code onto an Abstract Syntax Tree(AST)** In order to run our functions on each part of the source code, we need to get the abstract syntax tree of the program. This is done by Java Parser, an open source framework which can parse a java program and return a well object-oriented format of AST.
2. **A2M: Mapping the Abstract Syntax Tree onto a Mathematica Equation** We covered a few type of node, such as control statements, assignments, loops and function calling. These nodes are transformed into Mathematica equation. The relations shows the state of the program before and after execution of the statement represented by each node.
3. **M2F: Solving the Mathematica Equation to Derive the Program Function** Once the equations are generated in a Mathematica syntax, we can call Mathematica API to solve the equations and return the final state of the program as a function of the initial state;

11.1 Contribution and Comparison With Previous Studies

The major differentiating feature of our approach is that, we capture the precise semantics of loops in full, to the extent that (under the condition that) we have the essentials and enough invariant

relations(recognizers) to match its code. To map and find the relevant invariant relation, we adopt a semantic matching pattern to search and find the correct invariants relation for the given loop. This approach is studied against the conventional syntactic matching strategy. The syntactic matching approach that is taken in [70], tries to find an identical match between formal patterns of recognizers and the actual output of (T ∩ B). The main problem of this idea is checking the invariant and code pattern with a pure string matching. For example, if there are two code pattern like $b = a + 1 - 1$ and $b = a$, these two are different and we need two different recognizer record. Each record has different code pattern and same invariant relation. While in semantic matching, we use *Imply* function to check if the invariant relation can be reached from a code pattern.

Another difference between string matching and our work is, we get the Abstract Syntax Tree(ATS) of the source code and then navigate it node by node. This is against the idea of processing the input as a simple string data type and process it line by line. The reason that gives more weight to our work is, having AST, we do not need to think about how complex the program is. We are able to analyze the source code with any number of nested conditional statements, any number of function call, scope, loop and expression. However, with the string matching strategy, they could only process some simple programs. To get the AST, we used Java Parser, a public tool that represent a java source code as a tree data structure. In the AST, each part of the program is specified as a node with complete information.

Another major difference between our work and [29] is, we generate a large Mathematica relation which reflects the full source code. We apply same algorithm without caring about whether the loop has too many nested conditions and scopes and expressions, or if it contains only few simple expressions. However, if the loop has some if-then-else statement, [29] tries to create a union of relations like below:

$(T \cap B) = B1 \cup B2 \cup B3 \cup ... \cup Bk$

Each union term reflects one conditional statement or scope. They try to run their pattern algorithm separately for each union term. This idea is not efficient and is prone to loss of information because it may not give us the smallest invariant relation. More limitation of this idea explained in [61].

In Chapter 10, we explained our tool development and features. The main functionality of the program is to take a source code and find the function of the program. Besides this, we have four functions, of which we have already implemented three, that helps to apply some queries and derive the function of specific path of the program. Assume() function can put some assumption at one particular line of the code, specified as a label. These assumptions could be initialized values for variables, or some conditions for them. The Capture() can take the function at specific label. Also, Verify() which checks and verify the given argument against the function at that label.

11.2 Future Work

There are some potential industry that could benefit from this work, such as sensitive systems, crypto currency and financial firms. Two major reason could be behind this claim. First, we try to capture what the program is intending to do. Hence, we are able to figure out the purpose of the program without running the that program. This can prevent the failure or sabotage coming from malicious codes. Second, we transform a program into a series of relations. This means, running time could significantly reduced. for example a loop with O(N) running time mapped to a equation which works in O(K), constant time.

Future prospects for this work involve enabling the analyzer to recognize and more complex data structures and transform them into Mathematica relations. Adding more data structures, we can cover and analyze wider range of programs. The second potential scale up is to enlarge the database of recognizers. But we argue that the scale we are up against is not the size of programs,

but rather the size of loops within programs. Hence, our scope of application is not limited by the size of candidate programs, but rather by the size of the largest loop in the program;

REFERENCES

[1] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 1982.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification-the key book. *Lecture Notes in Computer Science*, 2016.

[3] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Abstract interpretation, symbolic execution and constraints. In *Recent Developments in the Design and Implementation of Programming Languages*, 2020.

[4] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular atatic analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 2010.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 2018.

[6] Clark Barrett, Daniel Kroening, and Thomas Melham. Problem solving for the 21st century: Efficient solver for satisfiability modulo theories. *London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering*, 2014.

[7] Radim Belohlavek and Martin Trnecka. A new algorithm for boolean matrix factorization which admits overcovering. *Discrete Applied Mathematics*, 2018.

[8] Abraham I Beltzer. Engineering analysis via symbolic computation—a breakthrough. *ASME. Appl. Mech. Rev.*, 1990.

[9] Richard Bird et al. *A Calculus of Functions for Program Derivation*. Oxford University. Computing Laboratory. Programming Research Group, Oxford, England, 1987.

[10] Bruno Blanchet. Introduction to abstract interpretation. *Lecture Script*, 2002.

[11] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 1975.

[12] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 1992.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, 2008.

[14] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[15] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 1989.

[16] Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark D Ryan. *Modeling and Verification of Parallel Processes*. Springer, Berlin, Heidelberg, 2003.

[17] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Symposium on Security and Privacy*. IEEE, 2012.

[18] Thomas E Cheatham and Judy A Townley. Symbolic evaluation of programs: A look at loop analysis. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, 1976.

[19] Jan Chomicki and David Toman. Temporal databases. *Handbook of Temporal Reasoning in Artificial Intelligence*, 2005.

[20] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*. Springer, 1982.

[21] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 1994.

[22] Lori A Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, 1976.

[23] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*. Springer, 2003.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[25] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[27] Ewen Denney and Bernd Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006.

[28] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo F Frias, Ali Jaoua, and Ali Mili. Relational mathematics for relative correctness. In *Relational and Algebraic Methods in Computer Science: 15th International Conference, RAMiCS 2015, Braga, Portugal, September 28-October 1, 2015, Proceedings 15*, 2015.

[29] Nafi Diallo, Wided Ghardallou, Jules Desharnais, and Ali Mili. Convergence: Integrating termination and abort-freedom. *Journal of Logical and Algebraic Methods in Programming*, 2018.

[30] Edsger W Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 1968.

[31] Thomas W Doeppner Jr. Parallel program correctness through refinement. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[32] Douglas D Dunlop and Victor R Basili. A heuristic for deriving loop functions. *IEEE Transactions on Software Engineering*, 1984.

[33] educba. Matlab vs mathematica. `https://www.educba.com/mathematica-vs-matlab/`. Accessed 03-26-2023.

[34] Bernard Elspas, Karl N Levitt, Richard J Waldinger, and Abraham Waksman. An assessment of techniques for proving program correctness. *ACM Computing Surveys (CSUR)*, 1972.

[35] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*.

[36] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[37] Delores M Etter, David C Kuncicky, and Douglas W Hull. *Introduction to MATLAB*. Prentice Hall Hoboken, NJ, USA, 2002.

[38] facebook. SPARTA, Static Analyzer Library. `https://github.com/facebook/SPARTA`. Accessed 03-26-2023.

[39] Hantao Feng, Xiaotong Fu, Hongyu Sun, He Wang, and Yuqing Zhang. Efficient vulnerability detection based on abstract syntax tree and deep learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*.

[40] Gregor Fischer, J Lusiardi, and J Wolff Von Gudenberg. Abstract syntax trees-and their role in model driven software development. In *International Conference on Software Engineering Advances*, 2007.

[41] Kenneth R Foster and Haim H Bau. Symbolic manipulation programs for the personal computer. *Science*, 1989.

[42] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of logic and computation*. Springer, Berlin, Heidelberg, 2010.

[43] Mikhail R Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. Esbmc v6. 0: Verifying c programs using k-induction and invariant inference: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April, 2019*.

[44] John Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 1991.

[45] Xiang Gan, Jori Dubrovin, and Keijo Heljanko. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, 2014.

[46] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 2016.

[47] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[48] David Gil, Jose Luis Fernández-Alemán, Juan Trujillo, Ginés García-Mateos, Sergio Luján-Mora, and Ambrosio Toval. The effect of green software: A study of impact factors on the correctness of software. *Sustainability*, 2018.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[50] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[51] Susanne Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 1999.

[52] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification: 21st International Conference, CAV 2009*, 2009.

[53] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 2002.

[54] Eric CR Hehner. *A Practical Theory of Programming*. Berlin, Heidelberg: Springer Science and Business Media, 2012.

[55] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[56] Charles Antony Richard Hoare. Mathematics of programming. In *Program Verification*. Clarendon Press, University of California, California, USA, 1993.

[57] William E. Howden. Symbolic testing and ahe dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 1977.

[58] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Aligator. jl–a julia package for loop invariant generation. In *International Conference on Intelligent Computer Mathematics*, 2018.

[59] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Invariant generation for multi-path loops with polynomial assignments. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2018.

[60] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 2009.

[61] Lamia Labed Jilani, Olfa Mraihi, Asma Louhichi, Wided Ghardallou, Khaled Bsaies, and Ali Mili. Invariant functions and invariant relations: An alternative to invariant assertions. *Journal of Symbolic Computation*, 2013.

[62] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 1976.

[63] James C King. A new approach to program testing. *ACM Sigplan Notices*, 1975.

[64] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009.

[65] Laura Ildikó Kovács and Tudor Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, 2005.

[66] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. Loop summarization using abstract transformers. In *International Symposium on Automated Technology for Verification and Analysis*, 2008.

[67] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004.

[68] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, 2010.

[69] José Guillermo Sánchez León. *Mathematica® Beyond Mathematics: The Wolfram Language in The Real World*. 2017.

[70] Asma Louhichi, Wided Ghardallou, Khaled Bsaies, Lamia Labed Jilani, Olfa Mraihi, and Ali Mili. Verifying while loops with invariant relations. *International Journal of Critical Computer-Based Systems*, 2014.

[71] Hugo Daniel Macedo and José Nuno Oliveira. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming*, 2013.

[72] Zohar Manna. The correctness of programs. *Journal of Computer and System Sciences*, 1969.

[73] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc Hainaut. Testing database programs using relational symbolic execution. *Technical report, University of Namur*, 2014.

[74] Stephan Merz. Model checking: A tutorial overview. *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France,*, 2001.

[75] Albert R Meyer and Dennis M Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd National Conference*, 1967.

[76] Ali Mili, Shir Aharon, and Chaitanya Nadkarni. Mathematics for reasoning about loop functions. *Science of Computer Programming*, 2009.

[77] Ali Mili, Shir Aharon, Chaitanya Nadkarni, Lamia Labed Jilani, Asma Louhichi, and Olfa Mraihi. Reflexive transitive invariant relations: A basis for computing loop functions. *Journal of Symbolic Computation*, 2010.

[78] Ali Mili, Jules Desharnais, and Jean-Raymond Gagné. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Informatica*, 1985.

[79] Ali Mili and Fairouz Tchier. *Software Testing: Concepts and Operations*. John Wiley and Sons, Hoboken, NJ, USA, 2015.

[80] Quentin Monnet and Lynda Mokdad. Dos detection in wsns: Energy-efficient designs and modeling tools for choosing monitoring nodes. In *Modeling and Simulation of Computer Networks and Systems*. Elsevier, Amsterdam, Netherlands, 2015.

[81] Bruno Monsuez. System f and abstract interpretation. In *International Static Analysis Symposium*, 1995.

[82] Carroll Morgan. *Programming From Specifications*. Prentice-Hall, Inc., One Lake Street, Upper Saddle River, NJ, 7458, 1990.

[83] Olfa Mraihi. Calcul automatique des fonctions de boucles: Analyse des relations invariantes. Technical report, Technical report, Institut Superieur de Gestion, Bardo, Tunisie, 2014.

[84] Olfa Mraihi, Wided Ghardallou, Asma Louhichi, Lamia Labed Jilani, Khaled Bsaies, and Ali Mili. Computing preconditions and postconditions of while loops. In *Theoretical Aspects of Computing–ICTAC 2011: 8th International Colloquium, Johannesburg, South Africa, August 31–September 2, 2011. Proceedings 8*, 2011.

[85] Ahmed K Noor and CM Andersen. Computerized symbolic manipulation in structural mechanics—progress and potential. *Computers and Structures*, 1979.

[86] Jan Obdržálek and Marek Trtík. Efficient loop navigation for symbolic execution. In *International Symposium on Automated Technology for Verification and Analysis*, 2011.

[87] Peter Ørbæk. Can you trust your data. In *Colloquium on Trees in Algebra and Programming*, 1995.

[88] Jeffrey L Overbey and Ralph E Johnson. Generating rewritable abstract syntax trees: A foundation for the rapid development of source code transformation tools. In *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers 1*, 2009.

[89] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 2016.

[90] David Lorge Parnas. Really rethinking 'formal methods'. *Computer*, 2010.

[91] MN Pavlović. Symbolic computation in structural engineering. *Computers and structures*, 2003.

[92] David A Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 1981.

[93] Enric Rodríguez-Carbonell and Deepak Kapur. Program verification using automatic generation of invariants. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 2004.

[94] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004.

[95] Gunther Schmidt. A proposal for a multilevel relational reference language. *Journal of Relational Methods in Computer Science*, 2004.

[96] Gunther Schmidt. *Relational Mathematics*. Cambridge University Press, England, 2011.

[97] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 2016.

[98] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 2018.

[99] Jake Silverman and Zachary Kincaid. Loop summarization with rational vector addition systems. In *International Conference on Computer Aided Verification*, 2019.

[100] Moritz Sinn and Florian Zuleger. Loopus-a tool for computing loop bounds for c programs. In *WING@ ETAPS/IJCAR*, 2010.

[101] Jiri Slaby, Jan Strejček, and Marek Trtík. Compact symbolic execution. In *Automated Technology for Verification and Analysis*. Springer, Berlin, Heidelberg, 2013.

[102] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: Visited. *Leanpub, oct. de*, 2017.

[103] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 1990.

[104] Statanalytica. Matlab vs mathematica. `https://statanalytica.com/blog/matlab-vs-mathemtica/`. Accessed 03-26-2023.

[105] Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

[106] Arun Veeramani, Kausik Venkatesan, and K Nalinadevi. Abstract syntax tree based unified modeling language to object oriented code conversion. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, 2014.

[107] Steven C White and Sahra Sedigh Sarvestani. Comparison of security models: Attack graphs versus petri nets. In *Advances in Computers*. Elsevier, Amsterdam, Netherlands, 2014.

[108] Jeannette M. Wing. Chapter 9 - scenario graphs applied to network security. In Yi Qian, James Joshi, David Tipper, and Prashant Krishnamurthy, editors, *Information Assurance*. Morgan Kaufmann, Burlington, 2008.

[109] Wolfram. Mathematica. `https://reference.wolfram.com/language/tutorial/SymbolicCalculations.html`. Accessed 03-26-2023.

[110] Wolfram. MathematicaFeatures. `https://www.wolfram.com/mathematica/`. Accessed 03-26-2023.

[111] Wolfram. MathematicaInforSeries. `https://reference.wolfram.com/language/tutorial/SeriesLimitsAndResidues.html#3309`. Accessed on 03.26.2023.

[112] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.