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

TERMINATION, CORRECTNESS AND RELATIVE CORRECTNESS

by
Nafi Diallo

Over the last decade, research in verification and formal methods has been the subject of increased interest with the need of more secure and dependable software. At the heart of software dependability is the concept of software fault, defined in the literature as the adjudged or hypothesized cause of an error. This definition, which lacks precision, presents at least two challenges with regard to using formal methods: (1) Adjudging and hypothesizing are highly subjective human endeavors; (2) The concept of error is itself insufficiently defined, since it depends on a detailed characterization of correct system states at each stage of a computation (which is usually unavailable). In the process of defining what a software fault is, the concept of relative correctness, the property of a program to be more-correct than another with respect to a given specification, is discussed. Subsequently, a feature of a program is a fault (for a given specification) only because there exists an alternative to it that would make the program more-correct with respect to the specification. Furthermore, the implications and applications of relative correctness in various software engineering activities are explored. It is then illustrated that in many situations of software testing, fault removal and program repair, testing for relative correctness rather than absolute correctness leads to clearer conclusions and better outcomes. In particular, debugging without testing, a technique whereby, a fault can be removed from a program and the new program proven to be more-correct than the original, all without any testing (and its associated uncertainties/imperfections) is introduced. Given that there are orders of magnitude more incorrect programs than correct programs in use nowadays, this has the potential to expand the scope of proving methods significantly. Another technique, programming without refining, is also introduced.

The most important advantage of program derivation by correctness enhancement is that it captures not only program construction from scratch, but also virtually all activities of software evolution. Given that nowadays most software is developed by evolving existing assets rather than producing new assets from scratch, the paradigm of software evolution by correctness enhancements stands to yield significant gains, if we can make it practical.

TERMINATION, CORRECTNESS AND RELATIVE CORRECTNESS

by
Nafi Diallo

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 2016

APPROVAL PAGE

TERMINATION, CORRECTNESS AND RELATIVE CORRECTNESS

Nafi Diallo

Dr. Ali Mili, Dissertation Advisor — Date
Professor of Computer, New Jersey Institute of Technology

Dr. Narain Gehani, Committee Member — Date
Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Marek Rusinkiewicz, Committee Member — Date
Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Kurt Rohloff, Committee Member — Date
Associate Professor of Computer Science, New Jersey Institute of Technology

---

Dr. Thomas Wies, Committee Member — Date
Assistant Professor of Computer Science, New York University

Dr. Kazunori Sakamoto, Committee Member — Date
Assistant Professor of Information Systems Architecture Science Research Division, National Institute of Informatics

BIOGRAPHICAL SKETCH

Author: Nafi Diallo

Degree: Doctor of Philosophy

Date: May 2016

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2016
- Master of Science in Financial Mathematics,
  Worcester Polytechnic Institute, Worcester, MA, 2005
- Master of Science in Civil Engineering,
  Gunma University, Gunma, Japan, 2000
- Bachelor of Science in Applied Mathematics and Computer Science,
  Universite Gaston Berger de Saint-Louis, Saint-Louis, Senegal, 1997

Major: Computer Science

Presentations and Publications:

Submitted Journal Papers

J. Desharnais, N. Diallo, W. Ghardallou, and A. Mili, "Projecting programs on specifications: definition and implications" *(Submitted to Elsevier)*, 2016.

N. Diallo, W. Ghardallou, J. Desharnais, and A. Mili, "Convergence = Termination + Abort-Freedom", *(Submitted to Journal of Symbolic Computation)*, 2015.

Published Conference Papers

W. Ghadallou, N. Diallo, and A. Mili, "Software evolution by correctness enhancement", *The 28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco, CA, USA, July 1-4, 2016.

N. Diallo, W. Ghardallou, and A. Mili, "Program repair by stepwise correctness enhancement", *First International Workshop on Pre- and Post-Deployment Verification Techniques(PrePost)*, Reykjavk, Iceland, June 2016.

W. Ghardallou, N. Diallo, M. Frias, and A. Mili, "Debugging without testing", *International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, April 10-15, 2016.

J. Desharnais, N. Diallo, W. Ghardallou, M. Frias, A. Jaoua and A. Mili, "Relational Mathematics for Relative Correctness", *Relational Methods in Computer Science*, Braga, Portugal, September 2015, (Springer Verlag: Lecture Notes in Computer Science).

N. Diallo, W. Ghardallou and A. Mili, "Program derivation by correctness enhancements", *Refinement Workshop*, *Formal Methods* 2015, Oslo, Norway, June 2015.

N. Diallo, W. Ghardallou and A. Mili, "Absolute Correctness and Relative Correctness", *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy 2015.

W. Ghardallou, N. Diallo and A. Mili, "Merging Termination with Abort Freedom", *RISC Report Series No. 14-11*, *Symbolic Computation in Software Science (SCSS)*, Tunis, Tunisia, December 7-11, 2014.

N. Diallo and W. Ghardallou, "Work-In-Progress: Repairing a Loop by Constructive Transformation using Mutation Analysis", *RISC Report Series No. 14-11*, *Symbolic Computation in Software Science (SCSS)*, Tunis, Tunisia, December 7-11, Tunis, Tunisia 2014.

To my mother, Marie Ba, and all the little girls who are denied the opportunity of education because they have to stay home and help in household chores.

To my precious little girls and mommy's cheerleaders, Marie and Houleye Sow, may you dream big!

ACKNOWLEDGMENT

"Try not to become a person of success, but rather try to become a person of value." Albert Einstein

I thank Dr. Ali Mili for his invaluable, heartfelt guidance and support. His expertise and continuous feedback made this dissertation possible. His constant motivation and confidence in me have ensured the best work from me.

I thank all of my committee members, Dr. Narain Gehani, Dr. Marek Rusinkiewicz, Dr. Kurt Rohloff, Dr.Thomas Wies and Dr. Kazunori Sakamoto, for their time and effort.

I thank Dr.Wided Ghardallou for her contribution to the work in this dissertation. I also thank the many students who contributed their time and skills to the software projects that made this dissertation possible.

I thank Ms.Angel Butler and Ms.Kathy James for their wonderful support and encouragement, these women rock!

I also thank Ms. Clarisa Gonzalez-Lenahan for the constant support and useful advice provided over my years at NJIT. I cannot name all my good friends at NJIT, so I say thank you to all, especially Mr Nafize Paiker and his lab mates for the invaluable feedback during practice presentations.

My heartfelt thanks go to Amadou Tidiane Ba for his unconditional support and cheering, Maman Nene Thiam, for all the prayers.

Behind the success of a married mom is a great husband, I deeply thank Papa Ibrahima Sow for his unconditional support, love and pride in seeing me accomplish my passion.

Lastly, I would like to thank my daughters Marie Khadija and Houleye Sow, and all my family and friends for their support and encouragement over the course of my doctoral studies.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Motivation 1

1.2 Background 1

1.3 Research Problems 4

1.4 Contributions 5

1.5 Organization of this Dissertation 6

2 THEORETICAL FOUNDATIONS 8

2.1 Introduction 8

2.2 Relational Mathematics 8

2.2.1 Definitions and Notations 8

2.2.2 Operations on Relations 9

2.2.3 Properties of Relations 10

2.2.4 Relational Laws 11

2.3 A Refinement Calculus 13

2.3.1 Refinement Ordering 14

2.3.2 Refinement Lattice 15

2.4 Program Semantics 16

2.5 Conclusion 18

3 INVARIANT RELATIONS 19

3.1 Introduction 19

3.2 Invariant Relations 19

3.2.1 Invariant Functions 21

3.2.2 Invariant Assertions 23

3.2.3 Ordering Invariant Relations by Refinement 25

3.2.4 Comparative Analysis 26

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.3 Conclusion 32

4 CONVERGENCE 34

4.1 Introduction: The Case for Merger 34

4.1.1 Motivation 35

4.1.2 Illustration 36

4.1.3 Premises 38

4.1.4 Contributions and Limitations 39

4.2 Characterizing Convergence Conditions 40

4.2.1 A Necessary Condition of Termination 40

4.2.2 Abort Freedom 41

4.3 Applications 47

4.3.1 Simple Loops 47

4.3.2 Nested Loops 49

4.4 Related Work 50

4.4.1 Loop Termination 50

4.4.2 Pointer Semantics 53

4.5 Conclusion 55

5 THE FXLOOP TOOL 57

5.1 Introduction 57

5.2 From Source Code to Relational Representation 57

5.2.1 Invariant Relation Generation 58

5.2.2 Other Invariant Relations 59

5.2.3 Recognizers 60

5.3 Architecture 63

5.3.1 Web Interface 63

5.3.2 CCA Compiler Generator 64

TABLE OF CONTENTS
(Continued)

Chapter **Page**

5.3.3 Http Server 65

5.3.4 Modeler 67

5.4 Domain Coverage 67

5.5 FxLoop Tour 68

5.5.1 Main Interface 68

5.5.2 Examples 69

5.5.3 Invariant Relation Generator 69

5.5.4 Computing Convergence Conditions 71

5.5.5 Evaluating Correctness 72

5.6 Limitations 73

6 RELATIVE CORRECTNESS 74

6.1 Introduction 74

6.2 Absolute Correctness 76

6.3 Relative Correctness 78

6.3.1 Deterministic Programs 78

6.3.2 Non-Deterministic Programs 80

6.4 Validation of Relative Correctness 83

6.4.1 Litmus Tests 83

6.4.2 Passing the Tests 84

6.4.3 Absolute Correctness as the Culmination of Relative Correctness 84

6.4.4 Relative Correctness and Reliability 85

6.4.5 Relative Correctness and Refinement 86

6.5 Faults and Fault Removal 87

6.6 Implications of Relative Correctness 90

6.6.1 Counting Faults 90

6.6.2 A Bridge Between Testing and Proving 98

TABLE OF CONTENTS
(Continued)

Chapter **Page**

6.7 Testing for Relative Correctness 99

6.8 Related Work 100

6.9 Conclusion 103

7 APPLICATIONS OF RELATIVE CORRECTNESS 105

7.1 Introduction 105

7.2 Debugging Without Testing 105

7.2.1 Proving Correctness and Incorrectness of Loops 106

7.2.2 Proving Relative Correctness for Loops 108

7.2.3 Uninitialized Loops 111

7.2.4 Initialized While Loops 119

7.3 Mutation Based Program Repair 124

7.3.1 Illustration 1 124

7.3.2 Illustration 2 125

7.4 Programming Without Refinement 130

7.4.1 Producing A Correct Program 130

7.4.2 Producing A Reliable Program 134

7.5 Software Evolution 136

7.5.1 Program Merger 136

7.5.2 Program Upgrade 139

7.6 Software Maintenance 141

7.6.1 Corrective Maintenance 141

7.6.2 Adaptive Maintenance 142

7.7 Related Work 143

7.8 Conclusion 145

8 CONCLUSIONS AND FUTURE WORK 147

8.1 Summary and Assessment 147

TABLE OF CONTENTS
(Continued)

Chapter **Page**

8.1.1 Conditions of Convergence 147

8.1.2 Relative Correctness 149

8.2 Future Work 151

8.2.1 Condition of Convergence 151

8.2.2 Relative Correctness 152

BIBLIOGRAPHY 154

LIST OF TABLES

Table **Page**

3.1 Characterizing invariants 33

5.1 Example of 1-recognizer 62

5.2 Example 2-recognizer 62

5.3 Example of 3-recognizer 62

5.4 C/C++/Java grammar supported by CCA compiler generator 66

LIST OF FIGURES

Figure		Page
2.1	Lattice structure of refinement.	17
5.1	fxLoop architecture.	63
5.2	fxLoop main interface.	64
5.3	Numeric recognizer database.	68
5.4	Examples by feature.	69
5.5	Invariant relations.	71
5.6	Convergence condition options.	72
5.7	Correctness condition options.	72
6.1	Enhancing correctness without duplicating behavior: $P' \sqsupseteq_R P$.	79
6.2	Ordering candidate programs by relative correctness.	80
6.3	Relative correctness for non-Deterministic programs: $P' \sqsupseteq_R P$.	82
6.4	Monotonic and non-monotonic fault removals.	90
7.1	Ranking candidates by relative correctness.	124
7.2	Program repair by stepwise correctness enhancement.	126
7.3	Relative correctness-based repair: stepwise fault removal.	130
7.4	Alternative program derivation paradigms.	131
7.5	Merger and upgrade.	139
7.6	Corrective maintenance.	142
7.7	Adaptive maintenance.	143

CHAPTER 1

INTRODUCTION

1.1 Motivation

Today, ensuring software quality, dependability and safety is of paramount importance in software design and development[1, 2, 3]. Considerable responsibility rests on the hand of modern days software engineers [1]. Because software is pervasive in the life of modern societies [1, 4], it is important for software engineers to control the quality of software products that they produce and maintain. Because software is used in critical applications [1, 4], quality standards that are expected of software may be very high. Because software products are large and complex, and their size and complexity increase with time, controlling their quality is an ongoing challenge: the demands on software technology are putting relentless pressure on software research to deliver ever more capable methods, tools, and processes.

1.2 Background

Despite the emergence of multiple programming languages and paradigms, the vast majority of software being developed and maintained nowadays is written in C-like languages, where the bulk of complexity stems from iterative constructs. Most automated tools that are used today to analyze source code in C-like languages are unable to build the inductive argument that is required to analyze loops, hence, resort to unrolling the loop a limited (user-specified) number of times. In this dissertation, an orthogonal approach, based on the concept of invariant relations, is explored. Invariant relations enable the approximation of the function of a loop. The choice between capturing all the functional details of a loop whose iterations are bounded, and approximating its functions for all possible executions, can be viewed as a trade-off between knowing everything about some executions and knowing something

about all executions. This dissertation argues in favor of the latter approach on the grounds that knowing everything is not necessary (many properties of interest can be established with partial information) and that making claims about bounded executions is not sufficient (a property may hold for bounded executions and fail to hold for unbounded executions).

In this dissertation, invariant relations are used to compute or approximate its termination condition, to prove or disprove its correctness, and to prove or disprove that a modification to the loop makes it monotonically more-correct.

In [5], Mili et al. isolate invariant relations as a worthy concept that carries intrinsic value, rather than being an auxiliary to computing invariant assertions. Previous works exist where invariant relations, or special forms thereof, are computed and used in the analysis of while loops. In [6], Carrette and Janicki derive properties of numeric iterative programs by modeling the iterative program as a recurrence relation, then solving these equations. Typically, the equations obtained from the recurrence relations by removing the recurrence variable are nothing but the reflexive transitive relations that we are talking about. However, whereas the method of Carrette and Janicki is applicable only to numeric programs, ours is applicable to arbitrary programs, provided we have adequate recognizers to match their control structure, and adequate axiomatizations of their data structure. Recognizers are aggregate patterns that allows us to analyze the source code. In [7], Podelski and Rybalchenko introduce the concept of transition invariant, which is a transitive (but not symmetric) superset of the loop body's function. Of special interest are transitive invariants which are well-founded (hence, asymmetric); these are used to characterize termination properties or liveness properties of loops. Transitive invariants are related to invariant relations in the sense that they are both transitive supersets of the loop body. However, transition invariants and invariant relations differ on a very crucial attribute: whereas the latter are reflexive, and are used to capture what remains

invariant from one iteration to the next, the former are asymmetric, hence, not reflexive, and are used to capture what changes from one iteration to the next. In [8], Kovacs et. al. deploy an algorithm (Aligator) they have written in Mathematica to generate invariant assertions of while loops using Cousot's Abstract Interpretation [9, 10]. They proceed by formulating then resolving the recurrence relations that are defined by the loop body, much in the same way (except for the automation) as Carette and Janicki advocate [6]; once they solve the recurrence relations, they obtain a binary relation beween the initial states and the current state, from which they derive an invariant assertion by imposing the pre-conditions on the initial state. Typically, this operation correspond to the formula we propose in Chapter 2: the relation they find between initial and current states is an invariant relation (R); and initial conditions they dictate on the initial state correspond to vector C in proposition 8 of Chapter 2. Aligator does not take the converse of the invariant relation, nor does it have to, since R is typically symmetric. In [11], Furia and Meyer present an algorithm for generating loop invariants by generalizing the postcondition of the loop. To this effect, they deploy a number of generalization techniques, such as constant relaxation (replacing a constant by a variable), uncoupling (replacing two occurrences of the same variable by different variables), term dropping (removing a conjunct), and variable aging (replacing a variable by an expression it had prior to termination). This work differs from ours in many ways, obviously, including that it does not distinguish between what is dependent on the loop and what is dependent on its context (hence, must be redone for the same loop whenever the postcondition changes), that it is of a highly heuristic nature, and that it is highly syntactic (the same postcondition written differently may yield a different result). Another important distinction, of course, is that we analyze the loop as it is (without consideration of its specification), whereas Furia and Meyer analyze it as it should be (assuming it is correct).

1.3 Research Problems

Termination: Liveness and safety properties are important aspects of software security. The question of whether a program, most notably iterative, terminates or if a program executes without causing an abort(such as a division by zero, an array reference out of bounds, a reference to a nil pointer, or any other illegal operation) of a program, have both concurrently attracted much research[12, 13, 7, 14, 15, 16, 17, 18, 10]. They also have, consequently, used very different mathematical models. Yet knowing that a program terminates after 100 iterations when it aborts after 10 iterations is not sufficient. Also knowing that a program will abort after 100 iterations when the program terminates after 50 iterations is not necessary. So the research problem we set to investigate is concerned with the possibilities of merging these two aspects using the same mathematical model, namely invariant relations.

Debugging: The cost of software production is still largely consumed by debugging, which involves finding bugs in software and repairing them. While a great deal of research has been done on automated bug finding, the programmer is still left with the daunting task of repair while assuring that no regression is introduced. Recently, lots of researchers have turned their attention to automated repair [19, 2, 20, 21]. This dissertation explores the possibility of achieving debugging, namely localize a bug, suggest and validate a repair through static analysis of the code.

Program Derivation: Despite having been the subject of significant research effort [22, 23], program development approaches still struggle to deliver the promise of a formally derived software, that would ensure the quality of the end-product. Complexity is the enemy of security yet program derivation is fraught with difficulties; there are many steps between involved from analyzing the program to its efficient implementation [23], usually involving complex decisions. In times where software security is paramount due to ubiquitous use of software in modern societies, production of software that is formally guaranteed to be correct with respect to

its specification is highly needed. This dissertation explores the possibility of program derivation through correctness enhancements while ensuring that intermediate programs are executable.

1.4 Contributions

The contributions of this work are as follow:

Termination: For iterative programs, using invariant relations, we are able to capture both termination as finite number of iterations and absence of abort, to which we refer as convergence. The theoretical foundation of this work is summarized through two theorems. One theorem maps an invariant relation into a necessary condition of convergence while the other gives a general format of an invariant relation used to capture the absence of abort properties.

Through this work, a prototype tool is developed to automatically generate invariant relations and compute the convergence conditions. Source code in ($C/C++/Java$) is converted into an intermediate representational language of analysis. The generation of invariant relations is done against a database of recognizers. A recognizer is the aggregate made up of a code pattern and the corresponding invariant relation. Mathematica is used to both generate invariant relations and to analyze them for the purpose of computing convergence conditions.

Debugging: In [24, 25, 26], we explore the use of the concept of relative correctness introduced in [27] to support a number of software engineering processes. In particular, in [25], we present a relative correctness-based static analysis method that enables us to locate and remove a fault from a program, and prove that the fault has been removed all without testing. This technique, *which we call debugging without testing*, shows that we can apply static analysis to an incorrect program to prove that, although it may be incorrect, it is still more correct than another. Given that there are orders of magnitude more incorrect programs than there are correct

programs, the pursuit of this idea may expand the scope of static analysis methods. in [26], we explore the use of relative correctness in program repair. Specifically, we discuss how to perform program repair when we test candidate mutants for relative correctness rather than absolute correctness. We analyze current practice, try to show how a relative-correctness-based approach may offer better outcomes, and supporting our case with analytical arguments as well as a simple illustrative example.

Program Derivation: It is accepted wisdom that software quality should be part of the software development process rather than an after the fact concern. This is the motivation behind this part of out work, namely provide a model for deriving program with a built-in correctness proof.

In [28], we discuss how relative correctness can be used in the derivation of a correct program from a specification. Whereas traditional programming calculi derive programs from specifications by successive refinement-based correctness-preserving transformations starting from the specification, we show that we can derive a program by successive correctness-enhancing transformations (using relative correctness) starting from the trivial program *abort*. We refer to this technique as *programming without refining* [28]. Subsequently, we explore software evolution in [29] and envision expanding these methods as part of future research plans.

1.5 Organization of this Dissertation

Chapter 2 lays out the theoretical foundations of this dissertation. In particular, it presents the mathematical concepts to support the discussion of this work. It also describes the refinement calculus which is used in this work as well some program semantics. Chapter 3 also discusses the concept of invariant relation, fundamental to the approaches presented in this dissertation. Chapter 4 elaborates on the use of invariant relations to model and compute convergence conditions of while loops. Chapter 5 describes the implementation a tool to analyze while loops, automatically

generating invariant relations for the loop and computing convergence conditions, evaluating correctness with respect to a specification and computing invariant assertions based on pre/post conditions. Chapter 6 presents the concept of relative correctness, discusses how it relates to (absolute) correctness. The same chapter also discusses how to test for relative correctness and its implications in various software engineering practices. Some applications of relative correctness are presented in Chapter 7. Activities such as program repair stepwise by correctness-enhancement debugging without testing and programming without refining are illustrated by mean of illustrative examples. Chapter 8 describes some future work and concluding remarks.

CHAPTER 2

THEORETICAL FOUNDATIONS

2.1 Introduction

The foundations of this work lay in relational mathematics and set theory; we use the concept of relation to capture specifications, as well as program semantics and *sets* are used to represent the values of program variables. We assume the reader familiar with relational algebra, and we generally adhere to the definitions of [30, 31].

2.2 Relational Mathematics

2.2.1 Definitions and Notations

We represent *sets* using a programming-like notation, by introducing variable names and associated data types (sets of values). For example, if we represent set S by the variable declarations

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

then S is the Cartesian product $X \times Y \times Z$. Elements of S are denoted in lower case s, and are triplets of elements of X, Y, and Z. Given an element s of S, we represent its X-component by $x(s)$, its Y-component by $y(s)$, and its Z-component by $z(s)$. When no risk of ambiguity exists, we may write x to represent $x(s)$, and x' to represent $x(s')$.

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 $\phi = \{\}$.

Given a predicate t, we define the relation T : $T = \{(s, s')|t(s)\}$

2.2.2 Operations on Relations

Operations on relations (say, R and R') include the set theoretic operations of *union* ($R \cup R'$), *intersection* ($R \cap R'$), *difference* ($R \setminus R'$) and *complement* ($\overline{R}$).

They also include the *relational product*, denoted by ($R \circ R'$), or (RR', for short) and defined by:

$$RR' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}.$$

The *power* of relation R is denoted by R^n, for a natural number n, and defined by $R^0 = I$, and for $n > 0$,

$$R^n = R \circ R^{n-1}.$$

The *transitive closure* of relation R is denoted by R^+ and defined by

$$R^+ = \{(s, s')|\exists n > 0 : (s, s') \in R^n\}.$$

The *reflexive transitive closure* of relation R is denoted by R^* and defined by

$$R^* = \{(s, s')|\exists n \geq 0 : (s, s') \in R^n\}.$$

We admit without proof that $R^*R^* = R^*$ and that $R^*R^+ = R^+R^* = R^+$.

The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by

$$\widehat{R} = \{(s, s')|(s', s) \in R\}.$$

The nucleus of a relation R, denoted by $\mu(R)$, is defined as

$$\mu(R) = R\widehat{R}.$$

The *domain* of a relation R is defined as the set $dom(R) = \{s|\exists s' : (s, s') \in R\}$, and the *range* of relation R is defined as the domain of $\widehat{R}$.

The pre-restriction (resp. post-restriction) of relation R to predicate t is the relation defined by $\{(s, s')|t(s) \wedge (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \wedge t(s')\}$).

Operator precedence is applied as follows: unary operators evaluate first, followed by product, then intersection, then union.

To represent subsets of S in a relational form, we use vectors. A relation R is said to be a *vector* if and only if $RL = R$; a vector on space S is a relation of the form $R = A \times S$, for some subset A of S; we use vectors to represent subsets of S, and we may by abuse of notation write $s \in R$ to mean $s \in A$.

A non-empty vector p on S is said to be a point if and only if it satisfies the condition $p\widehat{p} \subseteq I$. Whereas a mere vector represents a subset of C, a point represents a singleton; the same symbol may be used to represent a point and the single element of S that defines it.

2.2.3 Properties of Relations

A relation R is said to be

- *reflexive* if and only if $I \subseteq R$,
- *symmetric* if and only if $R = \widehat{R}$,
- *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$,
- *asymmetric* if and only if $R \cap \widehat{R} = \phi$,
- *transitive* if and only if $RR \subseteq R$
- anti-reflexive if and only if $R \cap I = \emptyset$, i.e. it has no pairs of the form (s, s)

Relation R is said to be inductive if there exists a vector A such that $R = \overline{A} \cap \widehat{A}$

Also, a relation R is said to be *total* if and only if $I \subseteq \mu(R)$. We also represent this property as $RL = L$.

Finally a relation R is said to be *deterministic* (or: a *function*) if and only if $\mu(\widehat{R}) \subseteq I$. The following property is of special interest to this work: two functions f and f' are identical if and only if $f \subseteq f'$ and $f'L \subseteq fL$.

A relation is said to be a *partial ordering* if and only if it is reflexive, antisymmetric, and transitive.

An equivalence relation is a relation that is reflexive, symmetric and transitive. In particular, the nucleus of a deterministic relation f can be written as

$$\mu(f) = \{(s, s' | f(s) = f(s')\}.$$

2.2.4 Relational Laws

We briefly present some relational laws that will be needed throughout this dissertation. They are presented without proofs as they are typically straightforward consequences of definitions. We let R and Q be arbitrary relations, C be an arbitrary vector on S, p be a point on S, V be a total function on S, and F be an arbitrary

function on S. Then, the following stand:

$$(C \cap Q)R = C \cap QR, \tag{2.1}$$

$$\overline{C}L = \overline{C}, \tag{2.2}$$

$$L\widehat{C} = \widehat{C}, \tag{2.3}$$

$$\widehat{C}R = L(C \cap R), \tag{2.4}$$

$$RC = (R \cap \widehat{C})L, \tag{2.5}$$

$$\widehat{\overline{C}}C = \phi, \tag{2.6}$$

$$(Q \cap \widehat{C})R = Q(C \cap R), \tag{2.7}$$

$$C \cap R = (I \cap C)R = (I \cap \widehat{C})R, \tag{2.8}$$

$$\widehat{C} \cap R = R(I \cap \widehat{C}) = R(I \cap C), \tag{2.9}$$

$$I \cap RL = I \cap R\widehat{R} = I \cap L\widehat{R} \tag{2.10}$$

$$Q \subseteq RL \Leftrightarrow QL \subseteq RL \tag{2.11}$$

$$Q \subseteq LR \Leftrightarrow LQ \subseteq LR \tag{2.12}$$

$$R \subseteq I \Rightarrow \widehat{R} = R \tag{2.13}$$

$$R \neq \phi \Rightarrow LRL = L \tag{2.14}$$

$$\overline{Rp} = \overline{R}p \tag{2.15}$$

$$\widehat{\overline{pR}} = \widehat{p}\overline{R} \tag{2.16}$$

$$RV \subseteq Q \Leftrightarrow R \subseteq Q\widehat{V} \tag{2.17}$$

$$\widehat{V}R \subseteq Q \Leftrightarrow R \subseteq VQ \tag{2.18}$$

$$RF \subseteq Q \Leftrightarrow L\widehat{F} \cap R \subseteq Q\widehat{F} \tag{2.19}$$

$$\widehat{F}R \subseteq Q \Leftrightarrow FL \cap R \subseteq FQ \tag{2.20}$$

$$Q \cup R(R^*Q) = R^*Q \tag{2.21}$$

$$R \subseteq F \wedge FL \subseteq RL \Leftrightarrow R = F. \tag{2.22}$$

2.3 A Refinement Calculus

A fundamental concept in programming calculus is that of refinement ordering. The exact definition of refinement (the property of a specification to refine another) varies from one calculus to another [32, 33, 34, 35, 36, 37]. This section describes our version of refinement calculus as introduced in [38, 39].

Definition 1 *Given two relations R and R', we say that R'* refines *R (abbr.: $R' \sqsupseteq R$) if and only if:*

$$RL \cap R'L \cap (R \cup R') = R.$$

The following proposition provides an alternative characterization of refinement, which we may use in our proofs throughout this dissertation, as well as an intuitive interpretation of the refinement relation.

Proposition 1 *Given two relations R and R', R' refines R if and only if*

$$RL \subseteq R'L \wedge RL \cap R' \subseteq R.$$

Proof. *Proof of Sufficiency.* Let R and R' satisfy the conditions $RL \subseteq R'L$ and $RL \cap R' \subseteq R$. We compute:

$RL \cap R'L \cap (R \cup R')$

$=$ {hypothesis $RL \subseteq R'L$}

$RL \cap (R \cup R')$

$=$ {distributing pre-restriction}

$RL \cap R \cup RL \cap R'$

$=$ {identity, hypothesis $RL \cap R' \subseteq R$}

R.

Hence R' refines R.

Proof of Necessity. Let R' refine R; we must prove $RL \subseteq R'L$ and $RL \cap R' \subseteq R$. For the first clause, we proceed as follows:

RL

$=$ {hypothesis}

$RL \cap R'L \cap (R \cup R')L$

$=$ {distributivity, associativity}

$(RL \cap R'L) \cap (RL \cup R'L)$

$=$ {the first term is a subset of the second}

$RL \cap R'L$.

Hence (by set theory) $RL \subseteq R'L$. For the second clause, we proceed as follows:

$RL \cap R'$

$\subseteq$ {monotonicity}

$RL \cap (R \cup R')$

$=$ {since $RL \subseteq R'L$}

$RL \cap R'L \cap (R \cup R')$

$=$ {hypothesis}

R. **qed**

Interpretation: by proposition 1, R' refines R if and only if it has a larger domain and assigns fewer images to elements in the domain of R.

2.3.1 Refinement Ordering

The following proposition provides an important property of refinement. The proof can be found in [40].

Proposition 2 *The refinement relation is a partial ordering between relations on a space S.*

For the sake of readability, we do not include the proof of this proposition here, but place it in the appendix instead. Because the refinement relation is a partial

ordering, we may refer to it as the *refinement ordering*. The following proposition provides simple properties of refinement in two special cases.

Proposition 3 *Let R and R' be two relations on set S.*

- *If R and R' have the same domain, then $R' \sqsupseteq R$ if and only if $R' \subseteq R$.*
- *If R and R' are deterministic, then $R' \sqsupseteq R$ if and only if $R' \supseteq R$.*

Proof. If R and R' have the same domain, then the condition of refinement can be written as: $(R \cup R') = R$, which means $R' \subseteq R$.

To prove the second clause of the proposition, we consider the lemma introduced in the proof of proposition 2. The proof of sufficiency is trivial: if R and R' are functions and $R \supseteq R'$ then $R' = R'L \cap R$. As for the proof of necessity, let R and R' be functions such that $R \sqsupseteq R'$, and let R'' be defined as $R'L \cap R$. By hypothesis, $R'' \subseteq R'$; on the other hand,

$$
\begin{array}{ll}
 & R'L \\
\subseteq & \text{by hypothesis, and by construction} \\
 & RL \cap R'L \\
= & \text{by construction} \\
 & R''L.
\end{array}
$$

Hence $R'' = R'$, from which we infer (by construction of R'') that $R' \subseteq R$. **qed**

2.3.2 Refinement Lattice

Since refinement is a partial ordering between specifications, it is legitimate to ponder its lattice-like properties. Let $\mathbb{R} = \langle \mathcal{R}, \sqsubseteq \rangle$ be a structure in which $\mathcal{R}$ is the set of specifications on some space S, and $\sqsubseteq$ is the 'is-refined-by' relation. Then, from proposition 2, $\mathbb{R}$ is a partial ordering. The following Proposition, due to [38], summarizes the main findings with regards to the lattice of relational specifications.

- Any two relations R and R' have a greatest lower bound, denoted by $R \sqcap R'$ and referred to as the *meet* of R and R'. In particular,

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

- Two relations R and R' admit a least upper bound if and only if they satisfy the condition

$$RL \cap R'L = (R \cap R')L.$$

  This condition is called the *consistency condition*.

- Any two relations that satisfy the consistency condition admit a least upper bound, denoted by $R \sqcup R'$ and referred to as the *join* of R and R'. The *join* is expressed as:

$$R \sqcup R' = (\overline{RL} \cap R') \cup (\overline{R'L} \cap R) \cup (R \cap R').$$

  The join of two specifications (or programs) R and R' is very important, because it captures the specification that represents all the functional attributes of R, all the functional attributes of R', and nothing else. It is possible to capture all the functional attributes of R and R' only if R and R' do not contradict each other: this is what the consistency condition represents.

- Two relations R and R' have a least upper bound if and only if they have an upper bound. Consequently, if R and R' satisfy the consistency condition and R refines Q and R' refines Q' then a fortiori Q and Q' satisfy the consistency condition. Conversely, as R and R' are refined, it becomes less and less likely that they satisfy the consistency condition.

- The empty relation is the universal lower bound of this ordering.

- This ordering admits no universal upper bound. Total deterministic relations are the maximal elements of this ordering.

Figure 2.1 shows the overall structure of the lattice of specifications.

2.4 Program Semantics

We consider a simple programming notation that includes variable declarations, as well as a number of C-like executable statements. The semantics of variables declarations allows us to define the state space of the program as discussed Section

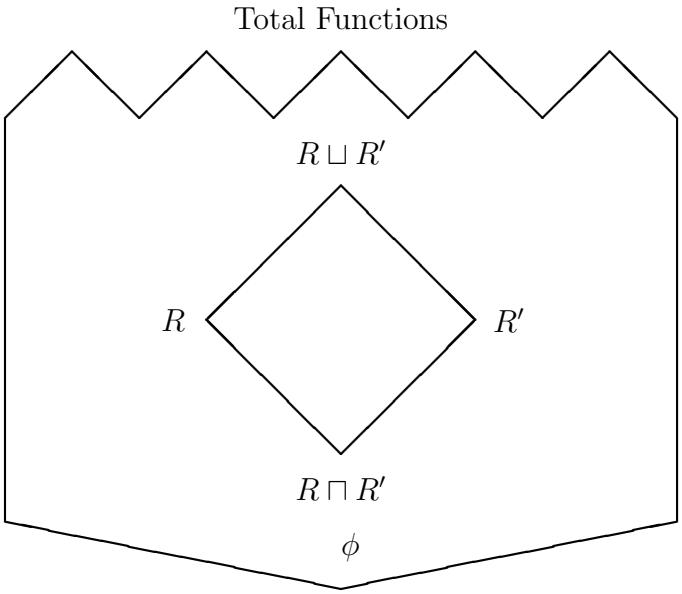


Figure 2.1 Lattice structure of refinement.

2.2.1. The semantics of executable statements are defined by means of a relation that captures the effect of the execution on the state of the program. Given a program or program part p, we let its semantics be represented by [p] (or by upper case P) and be defined by:

$$[p] = \{(s, s') | \text{if program } p \text{ executes on state } s \text{ then it terminates in state } s'\}.$$

We represent programs by means of C-like programming constructs, which we present below along with their semantic definitions:

- *Abort:* $[\texttt{abort}] \equiv \phi$.
- *Skip:* $[\texttt{skip}] \equiv I$.
- *Assignment:* $[s = E(s)] \equiv \{(s, s') | s \in \delta(E) \wedge s' = E(s)\}$, where $\delta(E)$ is the set of states for which expression E can be evaluated.
- *Sequence:* $[p_1; p_2] \equiv [p_1] \circ [p_2]$.

- *Conditional:* $[\texttt{if}\ (t)\ \{p\}] \equiv T \cap [p] \cup \overline{T} \cap I$, where T is the vector defined as: $T = \{(s, s') | t(s)\}$.

- *Alternation:* $[\texttt{if}\ (t)\ \{p\}\ else\ \{q\}] \equiv T \cap [p] \cup \overline{T} \cap [q]$, where T is defined as above.

- *Iteration:* $[\texttt{while}\ (t)\ \{b\}] \equiv (T \cap [b])^* \cap \widehat{\overline{T}}$, where T is defined as above.

- *Block:* $[\{x : X; p\}] \equiv \{(s, s') | \exists x, x' \in X : (\langle s, x\rangle, \langle s', x'\rangle) \in [p]\}$.

The semantic definition of a program written in this notation is a deterministic relation, i.e., a function, which we call the program's *function*. As a notational convention, lower case letters $[p]$ (possibly indexed) are used to represent the function of the program, and the same letters in upper case P to represent the relational semantic denotation of these programs. For the sake of readability, a program is sometimes identified with its function, i.e., we may use the program and its function interchangeably.

2.5 Conclusion

In this chapter, we present the theory needed to support the subsequent parts of this dissertation. The approach taken by this work relies on relational mathematics and uses a refinement calculus described above. The next chapter introduces invariant relations, a fundamental building block of our proposed methods.

CHAPTER 3

INVARIANT RELATIONS

3.1 Introduction

In the analysis and verification of loops, loop invariants also called invariant assertions have played a very important role [41, 6, 13, 42, 43, 44, 45, 46], since they were introduced by C.A.R. Hoare in his seminal work in 1969 [12]. Indeed, invariant assertions have been essential in proving correctness of indeterminate loops with respect to a pair of precondition and postcondition. In this chapter, we revisit the concept of invariant relation as introduced by [39]. The interest of invariant relations is that they lend themselves well for loop analysis and correctness verification.

3.2 Invariant Relations

Informally, an invariant relation of a while loop of the form w : `while (t) {b}` is a relation that contains all (but not necessarily only) the pairs of program states that are separated by an arbitrary number of iterations of the loop. Invariant relations are introduced in [47], their relation to invariant assertions is explored in detail in [40], and their applications are explored in [48]. This section is merely an excerpt from [48]. We refer to it for more details. We also present invariant assertions and invariant functions to highlight their relationship with invariant relations.

An invariant relation is defined formally as follows.

Definition 2 *Given a while loop of the form* w *:* `while (t) {b}` *on space* S*, we say that relation* R *is an* invariant relation *for* w *if and only if it is a reflexive and transitive superset of* $(T \cap B)$*.*

To illustrate the concept of invariant relation, we consider the following while loop on integer variables n, f, and k such that $0 < k \leq n$:

```
w:  while (k!=n) {k=k+1; f=f*k;}.
```

Listing 3.1 Factorial computation

We consider the following relation:

$$R = \left\{ (s, s') | \frac{f}{k!} = \frac{f'}{k'!} \right\}.$$

This relation is reflexive and transitive, since it is the nucleus of a function; to prove that it is a superset of $(T \cap B)$ we compute the intersection $R \cap (T \cap B)$ and easily find that it equals $(T \cap B)$. Other invariant relations include $R' = \{(s, s') | n' = n\}$, and $R'' = \{(s, s') | k \leq k'\}$.

The interest of invariant relations is that they are approximations of $(T \cap B)^*$, the reflexive transitive closure of $(T \cap B)$; smaller invariant relations are better, because they represent tighter approximations of the reflexive transitive closure; the smallest invariant relation is $(T \cap B)^*$. We quote the following theorem, due to [39], which we use as the semantic definition of a while loop.

Theorem 1 *We consider a while statement of the form w :* `while (t) {b}`*. Then its function W is given by:*

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

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

Also the following proposition stems readily from the definition.

Proposition 4 *Given a while loop of the form w :* `while (t) {b}` *on space S, we have the following results:*

1. *The relation $(T \cap B)^*$ is an invariant relation for w.*
2. *If R is an invariant relation for w, then $(T \cap B)^* \subseteq R$.*

3. If R_0 and R_1 are invariant relations for w then so is $R_0 \cap R_1$.

The main difficulty of analyzing while loops is that we cannot, in general, compute the reflexive transitive closure of $(T \cap B)$ for arbitrary values of T and B. Thus the motivation for using invariant relations.

3.2.1 Invariant Functions

Invariant functions are functions whose value remains unchanged by application of the loop body's function [49].

Definition 3 *Let w be a while statement of the form* {`while t do b`} *that terminates normally for all initial states in S, and let V be a total function on S. We say that V is an* invariant function *for w if and only if $(T \cap B)V \subseteq V$.*

In other words, an invariant function is a total function V on S if and only if $(T \cap B)$ preserves function V

The following Proposition provides an alternative characterization.

Proposition 5 *Let w be a while statement of the form* {`while t do b`} *that terminates normally for all initial states in S, and let V be a total function on S. Function V on S is an* invariant function *of w if and only if:*

$$T \cap BV = T \cap V.$$

Proof. *Sufficiency.* From $T \cap BV = T \cap V$ we infer (by identity 2.1) $(T \cap B)V = T \cap V$ from which we infer (by set theory) $(T \cap B)V \subseteq V$.

Necessity. Since $T \cap V$ is a function, we can prove $T \cap BV = T \cap V$ by proving $T \cap BV \subseteq T \cap V$ and $(T \cap V)L \subseteq (T \cap BV)L$. We proceed as follows: From $(T \cap B)V \subseteq V$ we infer (by identity 2.1) $T \cap BV \subseteq V$. Combining this with the set theoretic identity $T \cap BV \subseteq T$, we find (by set theory) $T \cap BV \subseteq T \cap V$.

As for proving that $(T \cap V)L$ is a subset of $(T \cap BV)L$, we note that because w terminates for all states in S, T is necessarily a subset of (or equal to) BL (if not any state in $T \setminus BL$ will cause the loop not to terminate), and we proceed as follows:

$(T \cap V)L \subseteq (T \cap BV)L$

$\Leftrightarrow$ { identity 2.1, applied twice }

$T \cap VL \subseteq T \cap BVL$

$\Leftrightarrow$ { totality of V, applied twice }

$T \cap L \subseteq T \cap BL$

$\Leftrightarrow$ { left: algebra; right: hypothesis $T \subseteq BL$ }

$T \subseteq T$

$\Leftrightarrow$ { set theory }

true.

qed

This condition can also be written using monotypes rather than vectors:

$$I(t) \circ V = I(t) \circ B \circ V.$$

To illustrate the concept of invariant function, we consider the loop of example 3.1, and propose the following function:

$$V \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} \frac{f}{(k-1)!} \\ 0 \\ 0 \end{pmatrix}.$$

This function is total, since its value can be computed for any state in S. To check the preservation condition, we compute:

$$
\begin{aligned}
& T \cap (BV) \\
\subseteq & \quad \{ \text{ substitution, simplification } \} \\
& \{(s, s') | n' = n \wedge f' = f \times k \wedge k' = k + 1\} \circ \{(s, s') | n' = \tfrac{f}{(k-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ relational product } \} \\
& \{(s, s') | n' = \tfrac{f \times k}{(k+1-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ simplification } \} \\
& \{(s, s') | n' = \tfrac{f}{(k-1)!} \wedge f' = 0 \wedge k' = 0\} \\
= & \quad \{ \text{ substitution } \} \\
& V.
\end{aligned}
$$

3.2.2 Invariant Assertions

Traditionally [14, 12, 50], an invariant assertion α for the while loop

$$w = \{ \textbf{while } t \textbf{ do } b\}$$

with respect to a precondition/ postcondition pair (P, Q) is defined as a predicate on S that satisfies the following conditions:

- $P \Rightarrow \alpha$.
- $\{\alpha \wedge t\} b \{\alpha\}$.
- $\alpha \wedge \neg t \Rightarrow Q$.

As defined, the invariant assertion is dependent not only on the while loop, but also on the loop's specification, in the form of a precondition/ postcondition pair. This precludes meaningful comparisons with invariant relations and invariant functions, which are dependent solely on the loop. Hence we redefine the concept of invariant assertion in terms of the second condition alone. Also, to represent

an invariant assertion, we map the predicate on S into a vector (a relation) on S. Specifically, we represent the predicate α by the vector A defined by

$$A = \{(s, s')|\alpha(s)\}.$$

Definition 4 *Given a while statement on space S of the form*

$$w = \{\texttt{while t do b}\}$$

that terminates for all initial states in S, and a vector A on S, we say that A is an invariant assertion *for w if and only if $(A \cap T \cap B) \subseteq \widehat{A}$.*

In other words, A is an invariant assertion for w if and only if $T \cap B$ preserves A. This is a straightforward interpretation, in relational terms, of the second condition of Hoare's rule,

$$\{\alpha \wedge t\}B\{\alpha\}.$$

For the running example, we claim that the following vector satisfies the condition of Definition 4:

$$A = \{(s, s')|f = (k-1)!\}.$$

To verify that α is an invariant assertion, we compute the left hand side of the definition:

Proof. $A \cap T \cap B$

$=$ { substitutions }

$\{(s, s')|f = (k-1)! \wedge k \neq n+1 \wedge n' = n \wedge f' = f \times k \wedge k' = k+1\}$

$\subseteq$ { deleting conjuncts }

$\{(s, s')|f = (k-1)! \wedge f' = f \times k \wedge k' = k+1\}$

$=$ { substitution }

$\{(s, s') | f = (k-1)! \wedge f' = (k'-1)! \wedge k' = k+1\}$

$\subseteq$ { deleting conjuncts }

$\{(s, s') | f' = (k'-1)!\}$

$=$ { substitution }

$\widehat{A}$. **qed**

Note that we have not proved that the assertion $f = (k-1)!$ holds after each iteration; rather we have only proved that if this assertion holds at one iteration, then it holds at the next iteration, hence, (by induction) after each iteration thereafter. This is in effect an inductive proof without a basis of induction.

3.2.3 Ordering Invariant Relations by Refinement

Invariant relations are ordered by refinement, which, as we have discussed in Section 2.3.2, has lattice-like properties. More refined relations give more information on loop behavior. Because they are by definition reflexive, invariant relations are total. If we consider the definition of refinement (Definition 1), we find that it can be simplified as follows

$RL \cap R'L \cap (R \cup R') = R'$

$\Leftrightarrow$ { R and R' are total }

$L \cap L \cap (R \cup R') = R'$

$\Leftrightarrow$ { Set Theory }

$R \cup R' = R'$

$\Leftrightarrow$ { Set Theory }

$R \subseteq R'$.

Hence for invariant relations, refinement is synonymous with set inclusion. We illustrate this ordering with a simple example. We leave it to the reader to check that the following two relations are invariant relations for our running sample program.

$$R_0 = \{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}.$$

$$R_1 = \{(s, s') | \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!} \wedge n = n'\}.$$

Invariant relation R_1 is a subset of, therefore (because they are both total) a refinement of, invariant relation R_0. Clearly, the latter also provides more information on loop properties than the former.

3.2.4 Comparative Analysis

Invariants and Loop Functions In this section, we put forth two propositions that elucidate the relation between invariant relations and loop functions. The first proposition shows us how to derive an invariant relation from the function of the loop.

Proposition 6 *Given a while loop w on space S of the form* {`while t do b`}*; we assume that w terminates for all s in S, and we let W be the function defined by w. Then $R = \mu(W)$ is an invariant relation for w.*

Proof. Relation $R = \mu(W)$ is reflexive and transitive because W is total and deterministic. According to Mills' Theorem, the loop function W satifies the condition $T \cap W = T \cap BW$. According to Proposition 5, this is equivalent to $T \cap BW \subseteq W$, which we rewrite (according to identity 2.1) as $(T \cap B)W \subseteq W$. By identity 2.17, this is equivalent to $(T \cap B) \subseteq W\widehat{W}$. **qed**

Proposition 6 shows us how to derive an invariant relation from the loop function; a much more useful result in practice is how to derive the function of the loop (or an approximation thereof) from an invariant relation. This is the subject of the following proposition. We refer to [51] for a detailed proof.

Proposition 7 *We consider a while loop w on space S of the form* {`w: while t do b`}*, which terminates for any element in S. If R is an invariant relation of w then W refines $R \cap \widehat{\overline{T}}$.*

The interest of this proposition is that it enables us to use any invariant relation to build a lower bound for the function of the loop. Because the function of the loop is total and deterministic, it is maximal in the lattice of refinement. Hence we can compute or approximate it using only lower bounds.

Hence, to summarize, we can derive an invariant relation from the loop function, and we can approximate (provide a lower bound of) the loop function from an invariant relation.

Invariants Relations and Invariant Assertions The first question that we raise in this section is: can an invariant relation be used to generate an invariant assertion? The answer is provided by the following proposition.

Proposition 8 *Let R be an invariant relation of $w =$* {`while t do b`} *on space S and let C be an arbitrary vector on S. Then $\widehat{R}C$ is an invariant assertion for w.*

Proof. Relation $\widehat{R}C$ is a vector since C is a vector. We must prove $\widehat{R}C \cap T \cap B \subseteq \widehat{\widehat{R}C}$. To do so, we proceed as follows:

$$
\begin{array}{ll}
 & \widehat{R}C \cap T \cap B \subseteq \widehat{\widehat{R}C} \\
\Leftarrow & \qquad \{ \text{ Since } T \cap B \subseteq R \ \} \\
 & \widehat{R}C \cap R \subseteq \widehat{C}R
\end{array}
$$

$\Leftarrow$ { since $\widehat{R}C \cap R \subseteq L(\widehat{R}C \cap R)$ }

$L(\widehat{R}C \cap R) \subseteq \widehat{C}R$

$\Leftrightarrow$ { vector identity 2.4 }

$\widehat{C}RR \subseteq \widehat{C}R$

$\Leftarrow$ { monotonicity }

$RR \subseteq R$,

which holds by virtue of the transitivity of R. **qed**

This proposition is interesting to the extent that it shows how to derive an invariant assertion from an invariant relation, but also because it shows that an invariant relation can generate (potentially) an infinity of invariant assertions, one for each vector C. We consider the following example, pertaining to the sample loop in listing 3.1. where we take an invariant relation

$$R = \{(s, s') | \tfrac{f}{(k-1)!} = \tfrac{f'}{(k'-1)!}\},$$

and a set of vectors, say

$$C_0 = \{(s, s') | f(s) = 1 \wedge k(s) = 1\}$$

$$C_1 = \{(s, s') | f(s) = 1 \wedge k(s) = 2\}$$

$$C_2 = \{(s, s') | f(s) = 2 \wedge k(s) = 3\}$$

$$C_3 = \{(s, s') | f(s) = 6 \wedge k(s) = 4\}$$

$$C_4 = \{(s, s') | f = (k-1)!\}$$

$$C_5 = \{(s, s') | f(s) = 2 \wedge k(s) = 5\}.$$

The invariant assertion that stems from vector C_0 is:

$$A = \{(s, s') | \exists s'' : \frac{f}{(k-1)!} = \frac{f"}{(k"-1)!} \wedge f" = 1 \wedge k" = 1\},$$

which can be simplified to

$$A = \{(s, s') | f = (k-1)!\}.$$

We leave it to the reader to check that the invariant assertions derived from vectors C_1, C_2, C_3 and C_4 are the same as the relation given above, since in all cases we have $\frac{f"}{(k"-1)!} = 1$. The invariant assertion that stems from vector C_5 is:

$$A_5 = \{(s, s') | \exists s'' : \frac{f}{(k-1)!} = \frac{f"}{(k"-1)!} \wedge f" = 2 \wedge k" = 5\},$$

which can be simplified to:

$$A = \{(s, s') | \frac{f}{(k-1)!} = \frac{1}{12}\},$$

or to

$$A = \{(s, s') | f = \frac{(k-1)!}{12}\}.$$

Through this example, we want to illustrate the idea that in the formula of invariant assertion provided by Proposition 8, the term $\widehat{R}$ pertains to the while loop alone, whereas C pertains to the context in which the loop is placed, viz its initalization. So that if the same loop is used with different initializations, we change C but maintain R.

We now consider the question of generating an invariant relation from an invariant assertion, for which we have the following proposition.

Proposition 9 *Given an invariant assertion A for while loop $w =$* {`while to do b`} *on space S, the relation $R = \overline{A} \cup \widehat{A}$ is an invariant relation for w.*

Now that we find that we can derive an invariant assertion from any invariant relation and an invariant relation from any invariant assertion, we need to ask the following questions: can any invariant assertion be derived from an invariant relation, and can any invariant relation be derived from an invariant assertion? The answers are provided below.

Proposition 10 *Given an invariant assertion A, there exists an invariant relation R and a vector C such that $A = \widehat{R}C$.*

Proof. If A is empty then this proposition holds vacuously for $C = \phi$. Given a non-empty invariant assertion A, we let $R = \overline{A} \cup \widehat{A}$ and $C = A$, and we prove that $A = \widehat{R}C$; we already know, by Proposition 9 that R is an invariant relation. What remains to prove:

$\widehat{R}C$

$=$ { substitutions }

$\widehat{(\overline{A} \cup \widehat{A})}A$

$=$ { distributing the converse operation }

$(\widehat{\overline{A}} \cup A)A$

$=$ { distributivity }

$\widehat{\overline{A}}A \cup AA$

$=$ { identity 2.6 }

AA

$=$ { definition of a vector }

$ALAL$

$=$ { associativity, and identity 2.14 ($A \neq \phi$) }

AL

$=$ { definition of a vector }

A.

qed

As to the matter of whether any invariant relation can be generated from an invariant assertion, the answer appears to be no, though we have a substitute: any invariant relation can be generated as an intersection of elementary invariant assertions. To formulate this result, we recall the concept of *point*, which is a special type of vector. While a vector is defined by a subset of S, a point is defined by a singleton. As an illustration, we consider the set S defined by natural variables n, f and k, and we write a few vectors and a few points, to illustrate the distinction.

$C_0 = \{(s, s') | f = 1\}$,

$C_1 = \{(s, s') | f = (k-1)!\}$,

$C_2 = \{(s, s') | f = 1 \wedge k = 1\}$,

$p_0 = \{(s, s') | n = 6 \wedge f = 1 \wedge k = 1\}$,

$p_1 = \{(s, s') | n = 6 \wedge f = 120 \wedge k = 7\}$,

$p_2 = \{(s, s') | n = 9 \wedge f = 2 \wedge k = 5\}$.

To conclude, from an invariant relation, we can derive as many invariant assertions as there are vectors on S (infinitely many, if S is infinite); and it takes a large number (possibly infinity) of invariant assertions (one for each element of S) to produce an invariant relation; furthermore, any invariant assertion stems from an invariant relation.

Summing up: Figure 3.1, borrowed from [40], compiles in tabular form the distinguishing characteristics of invariant assertions, invariant relations, and invariant functions.

3.3 Conclusion

In this chapter, we introduce the concept of invariant relations. In particular, we showed how they relate to invariant assertions. The next chapter introduces our approach to integrating termination with abort-freedom for while loops.

Table 3.1 Characterizing invariants

Criterion	Invariant Assertion	Invariant Function	Invariant Relation
Name, Attribute	A, vector	V, total deterministic	R, reflexive transitive
Condition	$A \cap T \cap B \subseteq \widehat{A}$	$T \cap BV \subseteq V$	$(T \cap B) \subseteq R$
Example	$A = \{(s, s') \mid f = (k-1)!\}$	$V \begin{pmatrix} n \\ f \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{f}{(k-1)!} \\ 0 \end{pmatrix}$	$R = \{(s, s') \mid \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\}$
Supersets of $(T \cap B)$	$(\overline{A} \cup \widehat{A})$	$(V\widehat{V})$	R
Relation to Loop Function	$A = WC$ for arbitrary vector C	$V = W$	$R = W\widehat{W}$
Ordering	Implication Inclusion	Injectivity	Refinement
Weakest	**true**	Constant functions	L
Optimal / Adequate Invariants	$A = \{(s, s') \mid W(s) = W(s_0)\}$ for some $s_0 \in S$	$V = W$	$R = (T \cap B)^*$

CHAPTER 4

CONVERGENCE

4.1 Introduction: The Case for Merger

The condition (on initial states) under which a computation terminates, and the question of whether a computation terminates for a given initial state, have been the focus of much research interest since the early days of computing. The question of termination arises, by definition, in the context of iterative programs. Traditionally, researchers have analyzed iterative programs by means of two constructs: they use *invariant assertions* [12] to capture functional properties of iterative programs, and *variant functions* (also referred to as *ranking functions*) [52, 53] or *well founded orderings* [14] to model operational properties, including termination. We argue that the derivation of a ranking function of a loop is amenable to the derivation of a transitive asymmetric superset of the function of the loop body, which Podelski and Rybalchenko introduce under the name of *transition invariant* [**?**]. What makes the derivation of ranking functions or, equivalently, transition invariants, very difficult is the fact that the transitive closure of a union of relations is not the union of the transitive closures of the individual relations; so that whenever the function of the loop body is structured as a union of relations, it is not sufficient to compute a transitive superset of each term of the union; this has been the driving motivation behind much of the work on the generation of composite ranking functions [54, 55, 56, 57] and composite transition invariants [58].

Non-termination is not the only issue we have to worry about with regards to the execution of a program; we also have to worry about the possibility that the program encounters an exceptional condition, such as an array reference out of bounds, an arithmetic overflow, the attempt to execute an illegal arithmetic operation (such as

a division by zero, the square root of a negative number, an arithmetic overflow, a reference to a nil pointer, etc). We refer to all these events as *abort*s, and we refer to the property of a program that avoids them as *abort freedom*. Most other authors refer to this property as *safety*, but we prefer to be compatible with the terminology of Avizienis et al. [59], where *safety* refers to correctness with respect to high stakes requirements. Traditionally, abort-freedom has been investigated separately from termination, and has, consequently, used totally distinct mathematical models, such as abstract interpretation [17, 18, 60].

When a program terminates without causing an abort, we say that it *converges*; and we use the term *convergence* to refer to the property of a program that terminates without causing an abort. When a program fails to converge, we say that it *diverges*.

4.1.1 Motivation

One of our main contributions in this chapter is that we want to capture termination and abort freedom by a single model; to explain the motivation for this decision, we consider a while loop whose execution may lead to an abort, and discuss why it is advantageous to compute the condition under which this loop terminates without causing an abort (as opposed to computing separately the condition under which it terminates, and the condition under which it causes no abort).

- Knowing that a loop does not exceed 100 iterations does not help us if it turns out that it will cause an abort at the 10th iteration. Hence the condition of termination is insufficient unless we also know the condition of abort-freedom.

- Knowing that a loop does not cause an abort for the next 100 iterations is not necessary if it turns out that the loop exits after only 10 iterations. Hence the condition of abort-freedom is unnecessary unless we also know the condition of termination.

- The condition of convergence of a loop is <u>not</u> the conjunction of the condition of termination with the condition of abort-freedom. As we will see throughout this chapter, the condition of convergence weaves conditions of termination and conditions of abort freedom in non-trivial ways.

4.1.2 Illustration

We consider the following loop on integer variables i, x and y, and we wish to compute the condition under which this loop terminates without attempting a division by zero; in other words, we want the condition under which this loop terminates after a finite number of iteration, and such that no single iteration will fail to execute properly.

```
1   while (i!=0)
2   {
3     i=i+2;
4     x=x-5;
5     y=y-y/x;
6   }
```

The abort condition we are concerned about in this loop is the possibility of a division by zero in the statement `{y=y-y/x;}`. Application of our analytical approach (which we discuss later in this chapter) to the source code of this loop yields the following condition of convergence:

$$(i = 0) \vee (i < 0 \wedge i \bmod 2 = 0 \wedge (x < 5 \vee (5 < x < \frac{-5 \times i}{2} \wedge x \bmod 5 \neq 0) \vee x > \frac{-5 \times i}{2})).$$

If we analyze this condition, we find that it stipulates that either $(i = 0)$ (in which case the loop does not iterate at all) or $(i < 0 \wedge i \bmod 2 = 0)$ (in which case the number of iterations is finite —note that if i is odd, then it will skip over zero and never terminate) then either $x < 5$ (in which case x never takes value 0 as it is decremented by 5 at each iteration) or $(5 < x < \frac{-5 \times i}{2} \wedge x \bmod 5 \neq 0)$ (in which case x flies over zero on its way down but does not hit zero) or $(x > \frac{-5 \times i}{2})$ (in which case i reaches 0 and terminates the loop before x gets near zero). If $(i > 0)$ or if $(i < 0 \wedge i \bmod 2 \neq 0)$ then this loop does not terminate since i never hits 0 as it is incremented by 2 at each iteration.

As additional illustrations, we consider the following loops and analyze their condition of convergence:

ID	Loop	Convergence Condition
P1	`for (int j=-100;j<=100;j++) {i=j; x=x0; y=y0;` `  while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	False
P2	`for (int j=0;j<=100;j++) {i=j; x=x0; y=y0;` `  while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	False
P3	`for (int j=1;j<=100;j++) {i=j; x=x0; y=y0;` `  while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	False
P4	`for (int z=10;z<=100;z++) {x=z; i=i0; y=y0;` `  while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	$i = 0 \vee i = -2$
P5	`for (int z=-100;z<=100;z++) {x=z; i=i0; y=y0;` `  {while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	$i = 0$
P6	`for (int z=0;z<=100;z++) {x=z; i=i0; y=y0;` `  {while (i!=0) {i=i+2; x=x-5; y=y-y/x;}}`	$i = 0$

Experimentation bears this analysis out, in the following sense:

- Execution of programs $P1$, $P2$, and $P3$ fails to converge for any initial value $x0$ of x and $y0$ of y; if the initial value of x is a positive multiple of 5, then execution of these programs fails due to an abort (our run-time system announces: `Floating Exception`); if the initial value of x is negative or is not a multiple of five, then the program fails to terminate. In both cases, we simply say that it fails to converge.

- Execution of program $P4$ converges only for initial values 0 and -2 of variable i; for all other initial values of i, it fails because it attempts a division by zero (hence, the execution yields the `Floating Exception`). For $i = 0$ it converges because the inner loop does not iterate at all; for $i = -2$ it converges because, even though the inner loop executes, it exits before x becomes 0.

- Execution of programs $P5$ and $P6$ converge only for initial value 0 of variable i. For positive initial values of i, and for negative initial values that are not divisible by 2, the programs fail to converge because they fail to terminate; for other values (also different from -2 and 0), the programs fail to converge because they attempt a division by zero.

4.1.3 Premises

There is a vast literature on termination analysis, and on abort-freedom analysis; we discuss some of the relevant work in Section 4.4. In this section, we characterize our approach by a number of premises, which ought to elucidate how our work differs from related literature; we characterize our work by its unique ends, then by its unique means.

Ends As our foregoing discussions makes it clear, our goal is to compute the condition under which a program, in particular an iterative program, terminates without causing an abort; we argue that computing the termination condition separately or the abort-freedom condition separately produces incomplete results, and that taking the conjunct of these conditions computed separately does not lead to the complete convergence condition. Our goal can further be characterized by the premise that we are interested in computing the condition of convergence of a program, rather than merely proving that a program does indeed converge. Finally, our analysis is focused on uninitialized loops rather than initialized loops, on the grounds that computing the convergence condition of an unitialized loop enables us to infer the converge (or divergence) of the loop for any initialization, whereas analyzing the convergence of an initialized loop produces a result for a single initialization.

Means Whereas termination is usually analyzed by means of variant functions or transition invariants, we analyze it by means of invariant relations. This is a fitting choice, from our standpoint, because we model the condition of convergence of any program as the condition under which an initial state of the program is in the domain of the program's function. Since we use invariant relations to compute or approximate loop functions, it is only natural that we use the same artifact to compute the domain of loop functions, since the domain of a function is an integral part of the definition of the function, rather than an orthogonal attribute. Like

invariant relations, transition invariants [61] are required to be transitive supersets of the function of the guarded loop body; but whereas transition invariants must be asymmetric (and well founded) because they aim to capture termination properties, invariant relations can be reflexive and symmetric, since they aim to capture all the functional properties of while loops (including equivalence relations between inputs and outputs). The case we make in this chapter is that invariant relations can take a wide range of forms, therefore, can be used to model a wide range of properties, including termination and abort-freedom.

4.1.4 Contributions and Limitations

The core idea of this approach can be summed up in two theorems: Theorem 2 maps any given invariant relation into a necessary condition of convergence; and Theorem 3 gives a general format of invariant relations that capture abort-freedom properties. Because invariant relations can be arbitrarily large, hence, capture arbitrarily little functional information of the loop, it is only fitting that Theorem 2 produces necessary, but not necessarily sufficient, conditions of convergence. We could appeal to another theorem, Theorem 1, to characterize necessary and sufficient conditions of convergence, but this theorem offers little guidance in practice; thus, we resort to heuristics, which we discuss in section 8.1.1, that enable us to compute sufficient conditions of termination using partial (but still sufficient) information about the loop. Note that because the intersection of invariant relations is an invariant relation, we do not distinguish between invariant relations that capture termination and invariant relations that capture abort-freedom; rather the set of invariant relation forms a continuum, where the same relation can capture the two aspects to varying degrees. The main limitation of our work is that it offers ideas and algorithms, but does not offer an integrated operational tool that we could match up against existing tools; then again, given that no tool we know of computes the condition of

convergence per se, all we can do is compare our approach to tools that compute termination conditions (or prove termination) and tools that compute abort-freedom conditions (or warn of possible abort occurrences).

In Section 4.2 we discuss a general framework for analyzing the convergence of programs, which we then specialize to iterative programs, by means of a necessary condition of convergence. In Section 4.3, we consider several conditions of abort avoidance and apply the necessary condition of convergence to them, then we discuss in section 8.1.1 under what condition the computed necessary conditions can be deemed sufficient. Finally in Section 4.5 we summarize our findings, compare them to related work, and sketch directions of future research.

4.2 Characterizing Convergence Conditions

The purpose of this section is to lay a foundation for the analysis of loop convergence by means of two theorems: the first gives a general formula for mapping any invariant relation into a necessary condition of convergence; and the second theorem gives guidance on how to generate invariant relations to target specific abort freedom properties.

4.2.1 A Necessary Condition of Termination

We consider a while loop w of the form w: `while (t) {b}` on space S, and we are interested to compute its domain, which we represent by the vector WL (where W is the function of w and L is the universal relation). The following theorem, due to [48], gives a necessary condition of convergence.

Theorem 2 *We consider a while loop w of the form w:* `while (t) {b}` *on space S, and we let R be an invariant relation for w. Then $WL \subseteq R\overline{T}$.*

This Theorem converts an invariant relation of w into a necessary condition of convergence; we seek to derive the smallest possible invariant relations, in order to

approximate or achieve the necessary and sufficient condition of convergence. The proof of this Theorem is given in [48]; it stems readily from Theorem 1, and from relational identities. In practice, we compute the convergence condition of a loop by means of the following steps:

- Using the invariant relation generator, we generate all the invariant relations we can recognize; whenever a code pattern of the loop matches a recognizer pattern from our recognizer database, we generate the corresponding invariant relation. These relations are represented in Mathematica syntax (©Wolfram Research).

- We compute the intersection of the invariant relations we are able to generate, by merely taking the conjunct of their Mathematica representation.

- Given R the aggregate invariant relation computed above, we simplify the following logical formula, which is the logical representation of the formula of Theorem 2.
$$\exists s' : (s, s') \in R \wedge \neg t(s').$$
The result is a logical expression in s, which represent a necessary condition of convergence of the loop.

As an illustration of this Theorem, we consider the sample factorial loop discussed earlier, namely:

$$w\text{: } \texttt{while (k!=n) \{k=k+1; f=f*k;\}}.$$

We consider the following invariant relation of w: $R = \{(s, s') | k \leq k'\}$. Application of Theorem 2 to this invariant relation yields the following necessary condition: $k \leq n$. Indeed, this condition is necessary to ensure that the number of iterations of the loop is finite.

4.2.2 Abort Freedom

Theorem 2 converts any invariant relation into an approximation of (more precisely: a superset of) the domain of the while loop; in logical terms, this produces a necessary condition of convergence. The domain of W is limited by failure of the

loop to terminate, as well as failure of abort-prone statements to execute successfully; Theorem 2 applies equally well to either of these circumstances. Depending on our choice of invariant relations, we can capture one aspect of non-convergence or the other, or a combination thereof. In this subsection, we present a general format of invariant relations that enable us to capture arbitrary aspects of abort-freedom (freedom from: array reference out of bounds, nil pointer reference, division by zero, arithmetic overflow, etc).

The following discussion builds an intuitive argument for the proposed theorem, and explains how we derived it. As a general rule, a program convergence whenever it is applied to a state within its domain, and fails to convergence otherwise. Hence, at a macro-level, the condition of convergence of program g can merely be written as:

$$s \in dom(G).$$

If g is a sequence of two subprograms, say `g = g1; g2` then this condition can be rewritten as:

$$s \in dom(G_1) \wedge G_1(s) \in dom(G_2).$$

We can prove by induction that if g is written as a sequence of arbitrary length, say `g = (g1; g2; g3; ...; gn)`, then the condition of convergence can be written as:

$$s \in dom(G_1) \wedge G_1(s) \in dom(G_2) \wedge G_2(G_1(s)) \in dom(G_3) \wedge ... \wedge$$

$$G_{n-1}(G_{n-2}(...(G_3(G_2(G_1(s))))...)) \in dom(G_n),$$

or, equivalently, as:

$$\forall h : 0 \leq h < n : G_h(G_{h-1}(...(G_3(G_2(G_1(s))))...)) \in dom(G_{h+1}). \tag{4.1}$$

If we specialize this equation to while loops, where all the G_i's are instances of the loop body, we find the following equation:

$$\forall h : 0 \leq h < n : (T \cap B)^h(s) \in dom(B). \tag{4.2}$$

In practice it is difficult to compute $(T \cap B)^h$ for arbitrary values of h; fortunately, it is not necessary to compute them either, as usually only a small set of program variables (and only some of their functional properties) are involved in characterizing convergence. Hence, we substitute in the above equation the term $(T \cap B)$ by a superset thereof (which we call B'), that captures only the transformation of convergence-relevant variables. This equation can then be written as:

$$\forall h : 0 \leq h < n : B'^h(s) \in dom(B). \tag{4.3}$$

We want to change this formula from a quantification on the number of iterations to a quantification on intermediate states; to this effect, we use the change of variables: $u = (T \cap B)^h(s)$, and we represent the initial state (that corresponds to $h = 0$) by s and the final state (that corresponds to $h = n$) by s'. With these change of variables, the inequality $0 \leq h$ can be written as $(s, u) \in B'^*$, and the inequality $(h < n)$ can be written as $(u, s') \in B'^*$. Equation 4.3 can then be written as:

$$\forall u : (s, u) \in B'^* \wedge (u, s') \in B'^+ \Rightarrow u \in dom(B). \tag{4.4}$$

Interestingly, this equation defines an invariant relation between s and s'; this is the object of Theorem 3. Before we present this theorem and its proof, we write the proposed invariant relation in algebraic form.

$$\begin{aligned} & R \\ = & \qquad \{ \text{ denotation } \} \\ & \{(s, s') | \forall u : (s, u) \in B'^* \wedge (u, s') \in B'^+ \Rightarrow u \in dom(B)\} \end{aligned}$$

$=$ { rewriting $u \in dom(B)$ }

$\{(s,s') | \forall u : (s,u) \in B'^{*} \wedge (u,s') \in B'^{+} \Rightarrow (u,s') \in BL\}$

$=$ { De Morgan }

$\overline{\{(s,s') | \exists u : (s,u) \in B'^{*} \wedge (u,s') \in B'^{+} \wedge (u,s') \notin BL\}}$

$=$ { Associativity }

$\overline{\{(s,s') | \exists u : (s,u) \in B'^{*} \wedge (u,s') \in (B'^{+} \cap \overline{BL}\}}$

$=$ { Relational Product }

$\overline{B'^{*}(B'^{+} \cap \overline{BL})}$.

This discussion introduces, though it does not prove, the following theorem; its proof is given below.

Theorem 3 *We consider a while loop w of the form w:* `while (t) {b}` *on space S, and we let B' be a superset of $(T \cap B)$. If B' satisfies the following conditions:*

- *B'^{+} is anti-reflexive.*
- *The following relation $Q = \overline{B'^{*}(B'^{+} \cap V)}$ is transitive, for an arbitrary vector V.*
- *$T \cap B \cap B'^{+}B' = \phi$.*

then $R = \overline{(B'^{*}(B'^{+} \cap \overline{BL}))}$ is an invariant relation for w.

This theorem provides, in effect (when applied in conjunction with Theorem 2), that if the loop converges for initial state s (i.e. s is in $dom(W)$), then any intermediate state s' generated from s by an arbitrary number of iterations of the loop causes no abort at the next iteration (i.e. s' is in $dom(B)$). It is in this sense that this theorem links $dom(W)$ and $dom(B)$.

Proof. We have to show three properties of R, namely reflexivity, transitivity, and invariance (i.e. that R is a superset of $(T \cap B)$).

Reflexivity. In order to show that I is a subset of R, we show that $I \cap \overline{R} = \phi$. We find:

$I \cap \overline{R}$

$=$ { substitution }

$I \cap (B'^*(B'^+ \cap \overline{BL})$

$\subseteq$ { monotonicity }

$I \cap B'^* B'^+$

$=$ { relational identity }

$I \cap B'^+$

$=$ { anti-reflexivity of B'^+ }

ϕ.

Transitivity. Transitivity is a trivial consequence of the second condition of the theorem, by taking $V = \overline{BL}$.

Invariance. In order to prove that $(T \cap B) \subseteq R$, it suffices (by set theory) to prove that $(T \cap B) \cap \overline{R} = \phi$. To this effect, we analyze the expression $(T \cap B) \cap \overline{R}$. But first, we introduce a lemma to the effect that for any relation C, $C^+C = C^+C^+$. Indeed, C^+C^+ can be written CC^*C^*C by decomposing C^+ as CC^* then as C^*C. Now, C^*C^* is equal to C^*: $C^*C^* \subseteq C^*$ because of transitivity, and $C^* \subseteq C^*C^*$ (because $I \subseteq C^*$). Hence $C^+C^+ = CC^*C = C^+C$. Now, we consider the expression $(T \cap B) \cap \overline{R}$.

$(T \cap B) \cap \overline{R}$

$=$ { substitution, double complement }

$(T \cap B) \cap (B'^*(B'^+ \cap \overline{BL}))$

$=$ { decomposing the reflexive transitive closure }

$(T \cap B) \cap (I \cup B'^+)(B'^+ \cap \overline{BL})$

$=$ { distributing the union over the product }

$((T \cap B) \cap B'^+ \cap \overline{BL}) \cup ((T \cap B) \cap B'^+(B'^+ \cap \overline{BL}))$

$=$ { associativity, and relational identity: $B \cap \overline{BL} = \phi$ }

$(T \cap B) \cap B'^{+}(B'^{+} \cap \overline{BL})$

$\subseteq$ { monotonicity }

$(T \cap B) \cap B'^{+}B'^{+}$

$=$ { lemma above }

$(T \cap B) \cap B'^{+}B'$

$=$ { by hypothesis }

ϕ. **qed**

The first condition of this theorem ensures that B' captures variant properties of $(T \cap B)$, hence, does not revisit the same state after a number of iterations; we refer to this as the *anti-reflexivity condition*. The second condition ensures that the resulting relation is transitive (a necessary condition to be an invariant relation); this condition involves B' and the structure of R, but does not involve B; we refer to this as the *transitivity condition*. The third condition ensures that B', while approximating $(T \cap B)$, remains in unison with it, i.e. does not iterate faster than $(T \cap B)$; this condition is needed to ensure that R is a superset of $(T \cap B)$; we refer to it as the *concordance condition*. Note that there is a one-to-one correspondence between the properties of B' and the resulting properties of R: The anti-reflexivity of B'^{+} yields the reflexivity of R; the transitivity of $(B'^{*}(B'^{+} \cap V))$ yields the transitivity of R and the concordance of B' yields the invariance of R (i.e. the property that $(T \cap B)$ is a subset of R).

The interest of this theorem is that it captures, in the form of an invariant relation, the property of abort-freedom of a while loop (as we illustrate subsequently). To understand how it does that, consider the logical form of such invariant relations:

$$R = \{(s, s') | \forall u : (s, u) \in B'^{*} \wedge (u, s') \in B'^{+} \Rightarrow u \in dom(B)\},$$

where B' is a superset of B. In practice, we use B' to approximate B, by focusing on the variables that are of interest to us (that are involved in abort-prone statements) and recording how B transforms them. As for $dom(B)$, we use it to capture/ represent the abort condition we are paranoid about: for example, if we want to model the condition that arithmetic operations in the loop body do not cause overflow, then we let $dom(B)$ include a clause to the effect that all operations produce a result within the range of representable values; if we want to model the condition that no division by zero arises in the execution of the loop body, then we include a condition in $dom(B)$ that ensures that all divisors in B are non-zero; if we want to avoid nil pointer references, then we capture in $dom(B)$ the condition that all dereferenced pointer variables are non-nil, etc. So that relation R, as written above, provides that all intermediate states generated by successive iterations of B cause no abort conditions. When we apply Theorem 2 using invariant relations generated by Theorem 3 (for various choices of B' and various possible characterizations of $dom(B)$), we find conditions on the initial states of the loop, that ensure a terminating abort-free execution.

As we have discussed in Section 3.2, smaller invariant relations are better. If we consider the template of invariant relations generated by Theorem 3,

$$R = \overline{B'^{*}(B'^{+} \cap \overline{BL})},$$

we find that R grows smaller (better) when B' grows larger (i.e. provides a looser approximation of B) and when BL (i.e. the domain of B) grows smaller (i.e. we capture more and more abort conditions).

4.3 Applications

4.3.1 Simple Loops

As an illustration, we consider the following loop on integer variables i, j, and k.

```
while (i>1) {j=j+1; i=i+2*j-1; k=k-1;}
```

The parameters of this loop are:

- $T = \{(s, s')|i > 1\}$.
- $B = \{(s, s')|j' = j + 1 \wedge i' = i + 2j + 1 \wedge k' = k - 1\}$.

We derive the following invariant relations (using recognizers from our existing database [62]):

- The elementary invariant relation, $R_0 = I \cup T(T \cap B)$.
- Symmetric invariant relations: $R_1 = \{(s, s')|j+k = j'+k'\}$, $R_2 = \{(s, s')|i-j^2 = i' - j'^2\}$.
- Antisymmetric invariant relations (one of them suffices, given that we already have R_1, but we write them both): $R_3 = \{(s, s')|j' \geq j\}$, $R_4 = \{(s, s')|k' \leq k\}$.

Taking their intersection $R = R_0 \cap R_1 \cap R_2 \cap R_3 \cap R_4$, and applying Theorem 2 to R, we find the following convergence condition:

$$(i \leq 1) \vee (i > 1 \wedge j \leq -\sqrt{i-1}).$$

This condition is provably a necessary condition of termination; we believe that it is also a sufficient condition of convergence, because the invariant relations we have used to generate it capture all the relevant information for termination: relation R_0 captures relevant boundary conditions; relation R_3 captures the progression of the program state; relation R_2 links variable j which counts the number of iterations and variable i, which is used in the loop condition. Note that relations R_1 and R_4 were redundant for our purposes, and are not needed to compute the convergence condition, if we have R_0, R_2 and R_3. As an illustration, we consider a data sample that satisfies the convergence condition, e.g., $i = 10 \wedge j = -5$ and a data sample that does not satisfy the condition, e.g., $i = 10 \wedge j = 0$, and verify that the first sample yields to convergence and the second leads to divergence.

4.3.2 Nested Loops

So far, we have focused on one loop at a time, and considered how to compute necessary (and possibly sufficient) conditions to ensure that the loop terminates without raising an abort condition. In this section we briefly review how to analyze nested loops: Let w be a loop of the form:

```
w:  while (t) {...  ...  ...  w':  while (t') {b';}; ... ... ...}
```

where w and w' are labels in the source code (to identify the loops). To analyze this nested loop, we first consider the inner loop and derive its convergence condition, which we call $C'(s)$. Then we apply Theorem 3 to the outer loop, using $C'(s)$ for $s \in dom(B)$, assuming no other source of abort exist in the loop body of w (if other sources did exist, we just take their conjunct with $C'(s)$). The rationale for this process is very straightforward: when we apply Theorem 3 to a loop, we capture in $dom(B)$ the condition under which the loop body is assured to converge; in the case of a nested loop, that condition is precisely $C'(s)$ (if no other cause of divergence existed). As an illustration of this approach, consider again the example of programs P1 to P6 presented in Section 4.1.2: Given that the condition of convergence of the inner loop was found to be

$$C'(i,x,y) \Leftrightarrow$$

$$(i=0)\vee(i<0\wedge i \bmod 2 = 0\wedge(x<5\vee(5<x<\frac{-5\times i}{2}\wedge x \bmod 5 \neq 0)\vee x>\frac{-5\times i}{2})),$$

the condition of convergence of P4 (for example) stems from simplifying the expression

$$\forall x, 10 \leq x \leq 100 : C'(i,x,y).$$

The result is $i = -2 \vee i = 0$.

4.4 Related Work

4.4.1 Loop Termination

Analysis of termination is a very active research area for which there is a vast bibliography; it is impossible to do justice to all the relevant work in this area, so we will just discuss some work that has influenced our research.

Boyer and Moore [63] propose a technique based on semi-automatic theorem proving where termination arguments have to be user-supplied. The work of Gupta et al.[64] uses templates to identify recurrent sets, but for the sole purpose of characterizing infinite loops; also focused on non termination is the work of Velroyen and Ruemmer[65]. In these two cases, the analysis is restricted to linear programs. Linear programs are also the focus of other researchers, such as [66, 67, 68, 52]. In [69], Burnim et. al. propose a dynamic approach to detecting infinite loops, based on concolic executions (a combination of concrete execution and symbolic analysis); the technique is generally incomplete, in the sense that the iterative analysis may lack the resources needed to solve complex constraints. In [70] Falke et. al. critique existing approaches to the analysis of termination of iterative program, on the grounds that treating bitvectors and bitvector arithmetic as integers and integer arithmetic is unsound and incomplete; also, they propose a novel method for modeling the wrap-around behavior of bitvector arithmetic, and analyze loop termination within this model.

In [53], Podelski and Rybalchenko propose a complete method for computing linear ranking functions; their approach is complete in the sense that if the loop can be bound by a linear ranking function, one such a function will be found by their method; Lee et al.[71] use the results of Podelski and Rybalchenko[53, 72] and propose an approach based on algorithmic learning of Boolean formula in order to compute disjunctive, well founded, transition invariants; the technique appears to be particularly effective when dealing with simple programs dealing

with linear arithmetic. In [16], Cook et. al. give a comprehensive survey of loop termination, in which they discuss transition invariants; whereas invariant relations are approximations of $(T \cap B)^*$, transition invariants are in fact approximations of $(T \cap B)^+$; this slight difference of form has a significant impact on the properties and uses of these distinct concepts. Whereas transition invariants are used by Cook et al. to characterize the well founded property of $(T \cap B)^+$, we use invariant relations to approximate the function of a loop, and its domain.

In [73], Chawdhary et. al. use abstract interpretation to synthesize ranking functions; their technique is subsequently improved by Tsitovitch et. al.[74], where loop summaries allow them to increase the scalability of the technique. In [75], Cook et. al. propose to under approximate weakest liberal preconditions in order to synthesize simpler predicates that still enable them to prove termination in cases where other tools would return a spurrious warning of possible non-termination. In [65], Velroyen and Ruemmer propose to synthesize invariants from a set of prerecorded invariant templates, and deploy a theorem prover to prove that the final states characterized by the invariants is unreachable, hence, disproving termination; because it provides a necessary condition of termination, our work can be used to disprove termination: whenever the necessary condition is violated, the loop does not terminate. In [76], Cook et al. introduce a technique for proving the non-termination of non-linear, non-deterministic and heap-based programs. Their approach is based on an over-approximation of non-linear behaviors by means of non-deterministic behaviors, and is based on the concept of closed recurrence set. We are interested in this approach because of its analogy with our work: an invariant relation is an overapproximation of the program's function, and Theorem 2 maps each invariant relation into a necessary condition of termination, whose negation is a sufficient condition of non-termination.

Abstract interpretation[77, 9, 10] is a broad scoped technique that aims to infer properties of programs by successive approximations of their execution traces; as such, it bears some resemblance to our invariant relations-based approach (which infer properties of while loops by approximations of the transitive closure $(T \cap B)^*$). Also, abstract interpretation has been used to, among others, analyze the properties of abort freedom of arbitrary programs [78]. The work on abstract interpretation has given rise to a widely used automated tool that analyzes programs and issues reports pertaining to their correctness, termination, abort-freedom, etc [18, 17]. In [79], Ancourt et. al. analyze loops by some form of abstract interpretation, but they dispense with the fix-point semantics of loops by attempting to approximate the transitive closure of the loop body abstraction. While the calculation of transitive closures is complex in general, the authors attempt it using affine approximations of the loop body transformations, which they define in terms of affine equalities and inequalities of state variables. Using techniques of discrete differentiation and integration, they derive an algorithm that computes affine invariant assertions from this analysis, and use the generated assertions to monitor abort-freedom conditions on the state of the program. They illustrate their algorithm by running it on many published sample loops. Overall, it is fair to say, perhaps, that all the work on ensuring termination by means of ranking functions and well founded orderings is an attempt to approximate (i.e. find a superset of) the transitive closure of the loop body, i.e. $(T \cap B)^+$.

In summary, we can characterize our approach (and contrast it with other approaches) by means of the following premises: unlike all other approaches, we compute an integrated convergence condition rather than merely a termination condition; we use the same artifact, namely invariant relations, to capture functional properties and operational properties (termination, abort-freedom) of iterative programs; we can handle any data type (not limited to numeric types) and any

numeric transformation (not limited to linear transformations). Limitations of our approach include: we can only handle programs for which we have pre-stored recognizers; and we can only ensure that the conditions we generate are necessary conditions of convergence. Our future work aims to address these weaknesses.

4.4.2 Pointer Semantics

Heap data structures manipulate potentially unbounded data structures, which do not lend themselves to simple modeling; as such, they represent one of the biggest challenges to scalable and precise software verification. In order to model the property that a loop causes no illegal pointer reference, we have to capture some aspects of pointer semantics; in our work, we use invariant relations to represent unbounded pointer references, and to reason about them. In this section, we review some of the alternative approaches to pointer semantics, and compare them to ours; we have been able to classify it into five broad categories, which we review in turn below.

- *Shape Analysis.* These approaches proceed by identifying some structure into the pattern of pointers between nodes. In [80] Sagiv et. al. use three-valued logic as a foundation for a parameterized framework for carrying out shape analysis; the framework is instantiated by supplying predicates that capture different relationships between nodes, and by supplying the functions that specify how the predicates are updated by particular assignments. In [81], Bhargav et. al. propose a new shape analysis algorithm, which is presented as an inference system for computing Hoare triplets summarizing heap manipulation programs. These inference rules are used as a basis for a bottom-up shape analysis of data structures.

- *Path-Length Analysis.* In [82], Spoto et al. prove the termination of programs written in Java Bytecode by mapping them into a constraint logic program which is built on the basis of a path-length analysis of the original program. The proof is based on the proposition that the termination of the logic constraint program is a sufficient condition to the termination of the original program. The path-length analysis of a Java bytecode program derives an upper bound of the maximal length of a path of pointers that can be followed from each variable of the program; the concepts of maxDepth and maxHeight presented in this chapter bear some resemblance to Spoto et al.'s path-length function, and the overapproximations derived for the path-length function bears some

resemblance to the type of approximations that are produced by invariant relations. But while Spoto et al. are interested in proving program termination, we are interested in computing termination conditions; while Spoto et al. are interested in termination as the property that the program executes a finite number of steps, we are interested to model termination as well as abort freedom; while Spoto et al.'s approach is focused primarily on the data structure of the program, our approach is focused primarily on its control structure.

- *Alias Analysis.* This approach focuses on determining whether two pointers refer to the same heap cell [83]. In [84], Hackett and Aiken use a combination of predicate abstraction, bounded model checking, and procedure summarization to compute a precise path-sensitive and context-sensitive pointer analysis. Alias analysis is only useful for reasoning about explicitly named heap cells, and cannot model general unbounded data structures.

- *Separation Logic.* This approach makes it possible to reason about heap manipulation programs [85] by extending Hoare logic [12] with two operators, namely separation conjunction and separation implication; these operators are used to formulate assertions over disjoint parts of the heap. In [86], O'hearn et. al. define a logic for reasoning about programs that alter data structures; to this effect they define a low-level storage model based on a heap with associated access operations, along with axiomatizations for these operations. The resulting model supports local reasoning, whereby only those cells that a program accesses are referenced in specifications and proofs.

- *Reachability Predicates.* This approach defines and uses predicates that characterize reachable nodes in an arbitrary data structure [87]. Indexed predicate abstraction [88] and Boolean heaps [89] generalize the predicate abstraction domain so that it enables the inference of universally quantified invariants. In [90], Gulwani et. al. show how to combine different abstract domains to obtain universally quantified domains that capture properties of linked lists. Craig interpolation has also been used to find universally quantified invariants for linked lists [91]. In [92], Mehta and Nipkow model heaps as mappings from addresses to values, and pointer structures are mapped to higher level data types for the verification of inductively defined data types like lists and trees. In [93], Filliatre and Marche introduce a method for proving that a program satisfies its specification and is free of null pointer referencing and out-of-bounds array access. Their approach is based on Burstall's model for structures extended to arrays and pointers. Similar tools have been developed for C-like languages, including Astree [17], Caveat [94], and SDV [95], but they are bounded to specific provers. In [96, 97], Meyer presents a comprehensive theory for modeling pointer-rich object structures and proving their properties; the model proposed by Meyer comes in two versions, a coarse-grained version that supports the analysis of the overall properties of the object structures, and

a fine-grained version, that analyzes object structures at the level of individual fields. Meyer's approach is represented in Eiffel syntax, and uses simple discrete mathematics.

Our interest in pointer semantics is much more recent than all these authors, and is driven by (and limited to) our interest in capturing conditions of abort avoidance as they pertain to illegal pointer references. Whereas we had thought initially that we could produce invariant relations that represent the scope equation of pointer references in loops for arbitrary data structures, we have subsequently resolved to generate invariant relations for well known data structures instead, for several reasons: First, generating invariant relations for the general case is very difficult; second, many authors whose work we have reviewed above appear to focus on well-known data structures rather than to arbitrary pointer-based structures; third, existing algorithms of shape analysis give us confidence that we can proceed by first analyzing the shape of our data, then deploying specialized invariant relations according to the shape that has been identified.

4.5 Conclusion

In this chapter, we present our approach to computing convergence conditions that takes a purely semantic approach to defining the condition of convergence. We say that a program converges for an initial state s if and only if the program can produce a final state s' as an image of s by the program function. Whether the program fails to produce a final state because it fails to terminate or because it fails to apply an intermediate function in its finite execution sequence does not matter to us, as it is a syntactic distinction, not a semantic distinction. In keeping with this premise, our definition of convergence applies to iterative programs as much as it applies to non-iterative programs; also, as far as while loops are concerned, our approach provides a way to map any given invariant relation of the loop onto a necessary condition of termination. We can generate many invariant relations for the loop,

each capturing a specific aspect of convergence, and obtain a convergence condition that ensures freedom from all causes of non-termination; to the best of our knowledge, our approach is unique in this feature.

CHAPTER 5

THE FXLOOP TOOL

5.1 Introduction

Chapters 2 and 4 introduced invariant relations and the model for integrating termination with abort-freedom, what we dubbed as convergence. In the first part of this chapter, the approach taken to generate invariant relations, for a given loop, is described. The second part presents the fxLoop software tool. This tool is dedicated to implementing the methods presented so far to allow the analysis of a while loop.

This chapter is organized into two sections as follows: Section 5.2.1 presents the method used for invariant relation generation, describing in particular the representation of the knowledge in the form of recognizers in Section 5.2.1. Section 5.3 describes the tool, its architecture, its input and output, its functionalities, each associated with an example.

5.2 From Source Code to Relational Representation

The first step in our approach is to transform the source code into a notation that allows to achieve two purposes:

- to have a uniform representation of knowledge, that is independent of the programming language, so that subsequent steps can be reused and,
- to prepare for the generation of invariant relations which are our basis for loop analysis.

Because invariant relations are supersets of the function of the loop body, it is advantageous to write the function of the loop body as an intersection of terms;

$$T \cap B = B_1 \cap B_2 \cap B_3 \cap B_4 \cap ..... \cap B_m$$

then, any superset of a term of the intersection is a superset of $T \cap B$, any superset of a pair of terms of the intersection is a superset of $T \cap B$, any superset of a triplet of terms of the intersection is a superset of $T \cap B$, etc. If we consider a loop body that is made up of a sequence of assignment statements, we can rewrite it as an intersection by eliminating the sequential dependencies between statements and writing, for each program variable, the cumulative effect of all relevant assignment statements. We obtain what is called concurrent assignments, or more generally conditional concurrent assignments (abbreviated: CCA) [98, 99]. The following 2 listings illustrate the transformation from $C/C++$ code to CCA notation. Notice the semi-colon separators in the $C/C++$ code and the comma separators in the CCA code.

```
1  int x = 0, y=0;
2  const int n=100;
3  while ( x!= n ){
4  x=x+1;
5  y=y+x; }
```

Listing 5.1 C/C++ source code example

```
1  int x = 0, y=0;
2  const int n=100;
3  while ( x!= n ){
4  n=n,
5  x=x+1,
6  y=y+x+1}
```

Listing 5.2 CCA code example

5.2.1 Invariant Relation Generation

The elementary invariant relation Given a while loop, the first (free) invariant relation can be derived following this proposition from [40].

Proposition 11 *Let w:* `while (t) {b}` *be a while loop on space S. The relation* $R = I \cup T(T \cap B)$ *is an invariant relation for w.*

This relation can be computed constructively from T and B, and includes pairs (s, s') such that $s' = s$ (case when no iterations are executed) and pairs (s, s') such that s verifies t and s' is in the range of $(T \cap B)$ (case when one or more iterations are executed). We refer to it as the *elementary invariant relation* of w, and in practice we generate it systematically whenever we analyze a loop.

5.2.2 Other Invariant Relations

For all other invariant relations, we have to inspect and analyze the loop in detail. The key ideas that underpin our algorithm are the following:

- Under the hypotheses of our study, we are interested in while loops written in a (deterministic) C-like programming language that terminate for all initial states.

- Because invariant relations are supersets of the loop body's function, we can prepare the loop for the extraction of invariant relations by writing its function as an intersection of terms; once it is written as an intersection, any superset of any term or combination of terms is a superset of the function of the loop body.

- When the loop body includes if-then-else statements, the outer structure of its function is a union rather than an intersection; in that case, we find a superset for each term of the union, then we merge them using a specially programmed function. The role of this function is to take several reflexive transitive relations (which represent the invariant relations corresponding to each branch of the loop body) and find a (preferably the smallest) reflexive transitive superset thereof (while the union of reflexive relations is reflexive, the union of transitive relations is not necessarily transitive). The current version of fxLoop does not support this feature.

- The invariant relations of individual branches are generated by pattern matching, using patterns that are developed off-line by means of invariant functions. These patterns, which we call *recognizers*, described in Section 5.2.3, capture all the programming knowledge and domain knowledge that is needed to analyze the loop.

Having invariant relations, we can carry out the analysis of the loop and compute artifacts such as convergence conditions, invariant assertions and correctness. Hence

invariant relations are derived from the source code in a 2-phase process and artifacts computed through a 3^{rd} step as follow:

1. **sourcecode2cca**: The first step is to transform the source code onto a form that represents the function of the loop body as an intersection, or as a union of intersections. For this purpose, we use CCA (*conditional concurrent assignments*) notation. This step is carried out by a compiler generator described in 5.3.2

2. **cca2mat**: Using the database of recognizers, we search for patterns in the CCA code, for which we have a corresponding pattern of an invariant relation; whenever a match is successful, we generate the corresponding invariant relation, which we represent as a set of Mathematica equations between initial states and final states. This step is carried out by a component, described in 5.3.4, that converts CCA code into Mathematica equations, involving unprimed program variables (representing initial states) and primed program variables (representing final states). In its current version, this program proceeds by performing semantic match of the source code against code patterns of pre-stored recognizers. This enables us to do more with fewer, more generic, recognizers.

3. **mat2nb**. The equations stemming from the lower bounds are submitted to Mathematica for resolution; they are solved in the output values as a function of the input values, yielding an explicit expression of the function of the loop.

5.2.3 Recognizers

The aggregate made up of a code pattern and the corresponding invariant relation pattern is called a *recognizer*. We distinguish between 1-recognizers, whose code pattern includes a single statement, 2-recognizers, whose code pattern includes two statements, and 3-recognizers, whose code pattern includes three statements; to keep combinatorics under control, we seldom use recognizers of more than 3 statements. For some recognizers, the invariant relations are dependent on some conditions on the variables they involved. We called such recognizers, conditional recognizers.

1-Recognizer 1-recognizers involve a single variables. They provide us information about the function of the loop body when this single statement is executed an arbitrary number of times. Table 5.1 shows an illustration of 1-recognizers.

2-Recognizer 2-recognizers involve two variables. They provide us information about the function of the loop body when these two statement are executed an arbitrary number of times. Table 5.2 shows an illustration of 1-recognizers.

3-Recognizer 3-recognizers involve three variables. They provide us information about the function of the loop body when these three statement are executed an arbitrary number of times. Table 5.3 shows an illustration of 1-recognizers.

Table 5.1 Example of 1-recognizer

Level	***Variables***	***Constants***	***Condition***	***Pattern***	***Invariant Relation***
1	x:int	a:int	true	$x = x + a$	$\{(s, s')\|ax \leq ax'\}$
1	x:int	a:int	x%2=0	$x = x/a$	$\{(s, s')\|frac(log_a(x)) = frac(log_a(x'))\}$

Table 5.2 Example 2-recognizer

Level	***Variables***	***Constants***	***Condition***	***Pattern***	***Invariant Relation***
2	x,y:int	a:int	true	$x = ax, y = ay$	$\{(s, s')\|xy' == x'y\}$
2	x,y:int	a,b:int	true	$x = ax, y = bx + y$	$\{(s, s')\|y + b(x/(1 - a)) == y' + b(x'/(1 - a))\}$

Table 5.3 Example of 3-recognizer

Level	***Variables***	***Constants***	***Condition***	***Pattern***	***Invariant Relation***
3	x,y,z:int	a,b:int	true	$x = x - a, y = y + bz, z = z$	$\{(s, s')\|z == z' \wedge bxz + ay == bx'z' + ay'\}$
3	x,y,z:real	a,b:int	true	$x = x + a, y = zy, z = z$	$\{(s, s')\|z == z' \wedge\ alog(\|y\|) - xlog(\|z\|) == alog(\|y'\|) - x'log(\|z'\|)\}$

5.3 Architecture

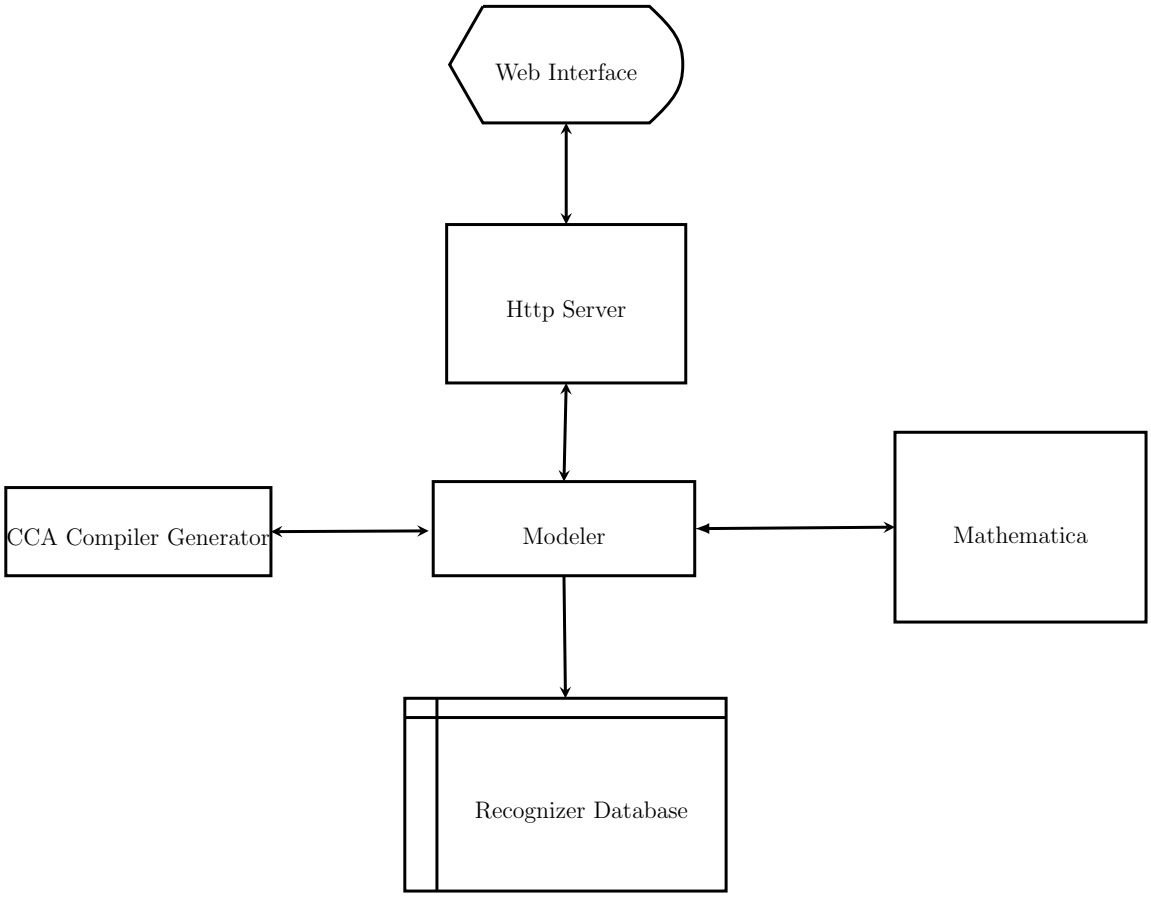


Figure 5.1 fxLoop architecture.

fxLoop is designed as a client server application since we rely on Mathematica, a commercial software as a solver. It is made of multiple components as shown in Figure 5.3. The tool is built in C++, mainly using the Boost C++ Libraries, a peer-reviewed, open collaborative development effort.

In the following sections , we describe each of the components in detail.

5.3.1 Web Interface

The web interface is designed with HTML and PHP. It's a very thin client with the following functions:

- Collect the input file (currently C/C++/Java source code) and user selected options,
- Form an http request and sends it to the server,
- Wait for response from the server,
- Display server response to the screen.

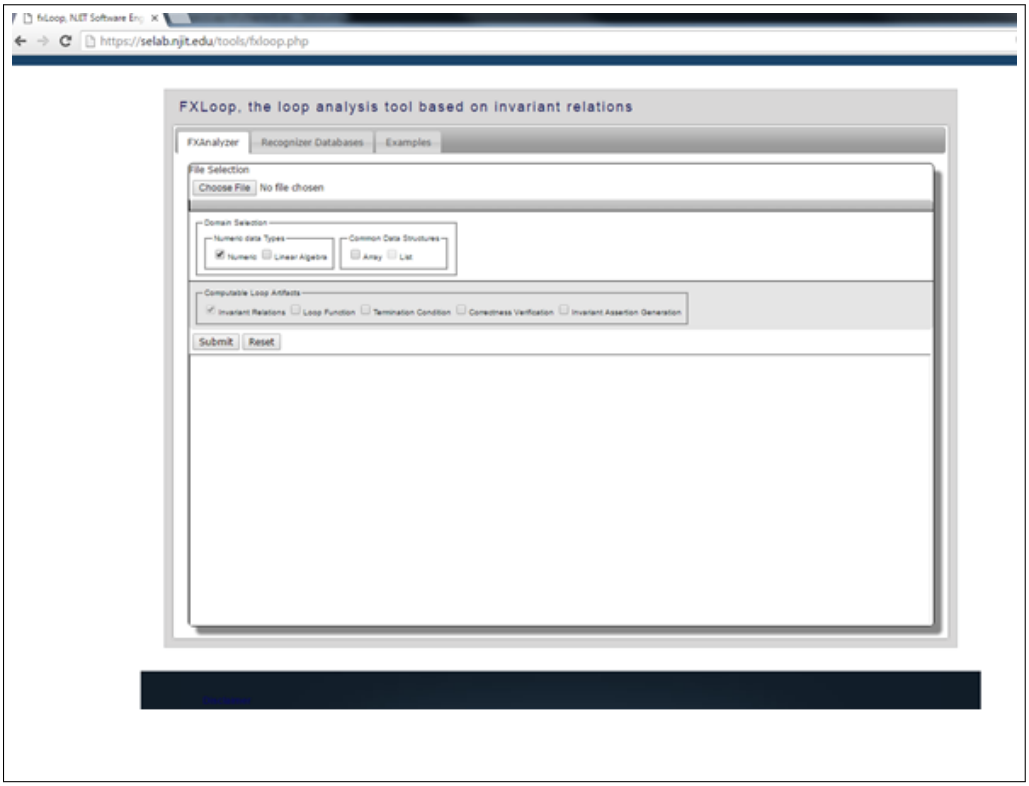


Figure 5.2 fxLoop main interface.

5.3.2 CCA Compiler Generator

This component allows to generate, from the source code, the Conditional Concurrent Assignment (CCA) code, which is the our language of analysis. This has the advantage to make the analysis programming language independent. We can add support for new languages by just updating the CCA compiler generator.

The compiler is built using the Boost C++ Spirit, a set of C++ libraries for parsing and output generation implemented as Domain Specific Embedded Languages (DSEL) using Expression templates and Template Meta-Programming. The Spirit libraries enable a target grammar to be written exclusively in C++. The compiler currently supports the C/C++ and Java programming languages. The grammar currently supported for each language is shown in Figure 5.4. In the lexical and syntax rules given in the table,

- Alternatives are separated by vertical bars: i.e., 'a | b' stands for "a or b".
- Square brackets indicate optionality: '[a]' stands for an optional a, i.e., "a | epsilon" (here, epsilon refers to the empty sequence).
- Curly braces indicate repetition: '{a}' stands for " a | aa | aaa | ..."

The source code must include a 'main' function and at least one while loop. However currently, we only support analysis of a simple while loop, as described in Section 5.2.2 . The implementation for nested loop is planned as part of the future work.

5.3.3 Http Server

The Boost C++ libraries provide a ready made http server that serves as the basis of this component. The source code was customized to fit the need of fxTool. In particular, the following stream of actions are done by the server :

- Parsing the incoming request,
- Notifying the CCA compiler to produce the CCA code,
- then passing the handle to the modeler,
- sending the reply, in the form of an XML structure, to the front end after completion of the task, an error message otherwise.

Table 5.4 C/C++/Java grammar supported by CCA compiler generator

prog	:	$\{dcl';' \mid func\}$		assg	:	id ['[' expr ']'] = expr
dcl	:	type var_decl { ',' var_decl }		expr	:	'' expr
	|	[extern] type id '(' parm_types ')' { ',' id '(' parm_types ')' }			|	'!' expr
var_decl	:	id ['[' intcon ']']			|	expr binop expr
modifier	:	'public' | 'protected' | 'private'		static_mod	:	'static'
type	:	'String[]' | 'void' | 'char' | 'short' | 'int' | 'long' | 'float'| 'double' |'signed' | 'unsigned			|	expr relop expr
parm_types	:	type id ['[' ']'] { ',' type id ['[' ']'] }			|	expr logical_op expr
func	:	type id '(' parm_types ')' '{' { type var_decl { ',' var_decl } ';' } { stmt } '}'			|	id ['(' [expr { ',' expr }] ')' | '[' expr ']']
stmt	:	if '(' expr ')' stmt [else stmt]			|	'(' expr ')'
	|	while '(' expr ')' stmt			|	intcon
	|	for '(' [assg] ';' [expr] ';' [assg] ')' stmt			|	charcon
	|	return [expr] ';'			|	stringcon
	|	assg ';'		binop	:	+
	|	id '(' [expr { ',' expr }] ')' ';'			|	
	|	'{' { stmt } '}'			|	*
	|	';'			|	/
relop	:	==		logical_op	:	&&
	|	!=			|	||
	|	<=				
	|	<				
	|	>=				
	|	>				

5.3.4 Modeler

This component is implemented in C/C++, using the Boost C++ string processing libraries. It handles the following tasks:

- Generating invariant relations through semantic matching of CCA code against the recognizer databases using Mathematica
- Computing the termination condition
- Evaluating correctness with respect to a specification
- Deriving invariant assertions for a given pre/post conditions

The modeler uses ©Mathematica as a solver through the *C* language API provided by ©Mathematica.

5.4 Domain Coverage

Even though we have recognizers for the numeric data type domain, the linear algebra domain and some advanced data types such as list, currently, the tool offer support for the numeric and the linear algebra domains only. The semantic recognizers used for each domain are listed on the Recognizer Database tab.

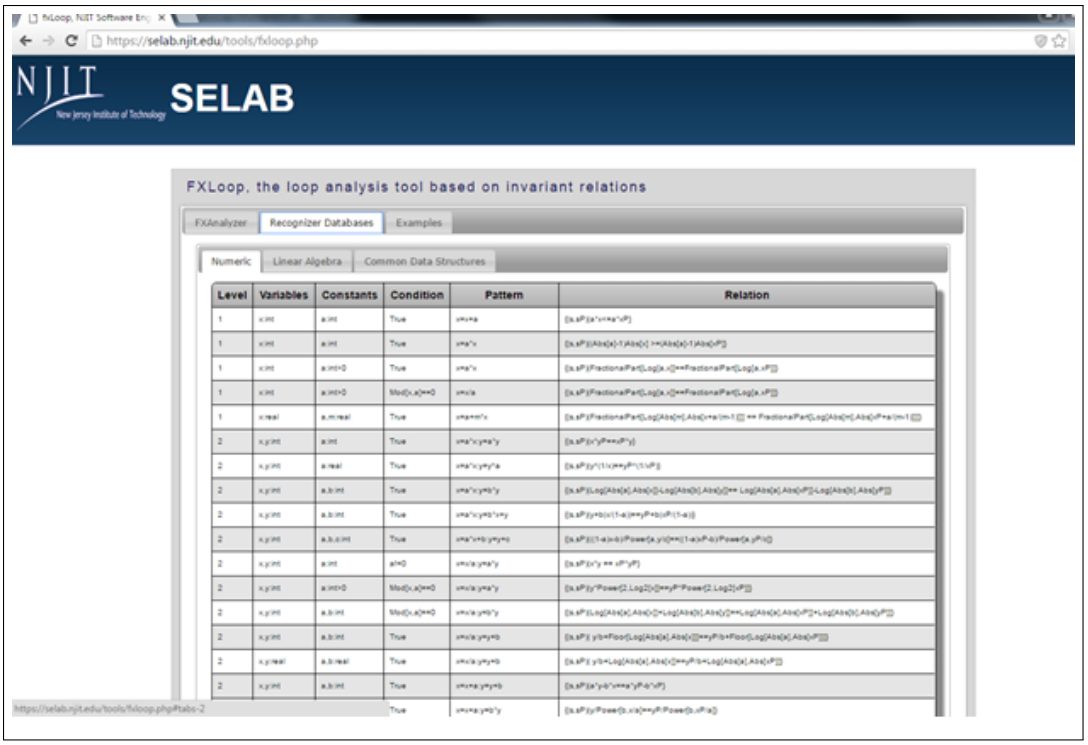


Figure 5.3 Numeric recognizer database.

5.5 FxLoop Tour

5.5.1 Main Interface

The main interface is the *Analyzer* which is the portal to application. This is the main interface of the tool. The interface is divided in two sections. The top portion is used to get user selections and the bottom part is used to display the result of the analysis.

To use the tool, you can download the examples (Figure 5.4). The tool computes various loop artifacts. The default option is to only generate the invariant relations. Termination condition, loop function can also be computed and correctness verification can be done.

Convergence can be computed as finite iteration or in combination with abort. There are 3 abort-freedom options:

- Arithmetic Overflow,

- Array out of Bound and,
- Illegal Arithmetic Operation

Correctness verification is done in conjunction with a specification file provided in Mathematica format.

A domain needs to be selected to guide which recognizers to use. Currently we only support Numeric data types, which are the default option.

5.5.2 Examples

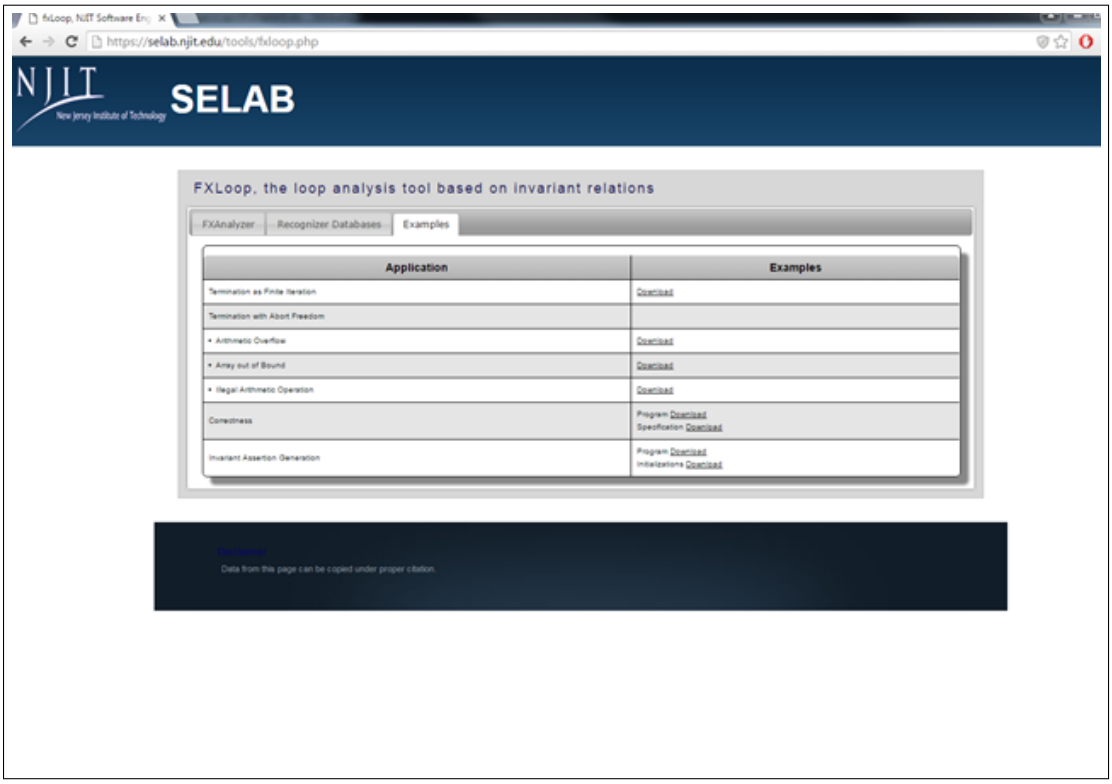


Figure 5.4 Examples by feature.

5.5.3 Invariant Relation Generator

The semantic match against the recognizer database is done via ©Mathematica using the following function written in ©Mathematica scripting language:

```
SemanticMatch[scode_, rcode_, svar_, rvar_, ovar_,IR_] :=Module[{
    matchResults, theSystem, semMatch, isResList, svarP, rvarP,
    svarNotZero, rvarNotZero},
(**Build primed variables **)
svarP = f @@@ List /@ svar; rvarP = f @@@ List /@ rvar;
theSystem =(scode == rcode) && (svar == rvar);
matchResults = SolveAlways[Eliminate[FullSimplify[Reduce[
    theSystem, Join[svar, rvar]], TimeConstraint->60], rvar],
    Union[svar,ovar]];
(*check if results is in the form of a list *)
semMatch = Eliminate[Simplify[theSystem /. matchResults,
    TimeConstraint->60], svar]; If[TrueQ[semMatch], IR /.
    matchResults /.Inner[Rule, Join[rvar, rvarP], Join[svar, svarP
    ], List], False]]//Simplify
```

Listing 5.3 Semantic matching script written in ©Mathematica scripting language

When this script is executed through the ©Mathematica provided API, a list is returned with invariant relations instantiated with the correct constant values if there is a match, otherwise false is returned.

The following are an C/C++ and a Java sample source code files , which when submitted to the invariant relation generator, lead to the output of figure 5.5. The C/C++/Java grammar is currently supported is described in table 5.4.

```
int main(){
  /*comment*/
  int i,j,k;
  while( i > 1) {
    j=j+1;
    i=i+2*j-1;
    k=k-1;
  }
  return 0;}
```

Listing 5.4 C/C++ invariant relation generation example

```
public static void main(String[] args){
  /** comment */
  int i,j,k;
  while( i > 1) {
    j=j+1;
    i=i+2*j-1;
    k=k-1;
  }}
```

Listing 5.5 Java invariant relation generation example

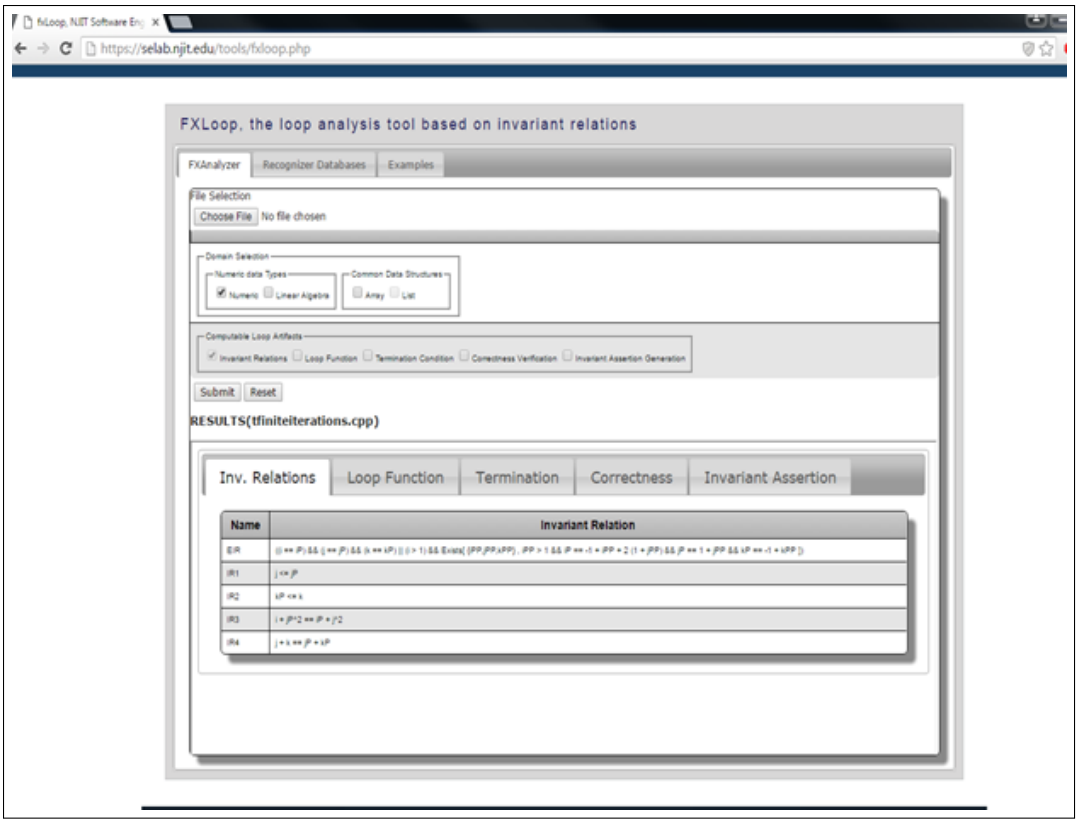


Figure 5.5 Invariant relations.

5.5.4 Computing Convergence Conditions

When the user checks the termination condition, more options open to computing convergence conditions, integrating a condition of abort of interest. Figure 5.6 shows the options currently available. For more details, refer to [100].

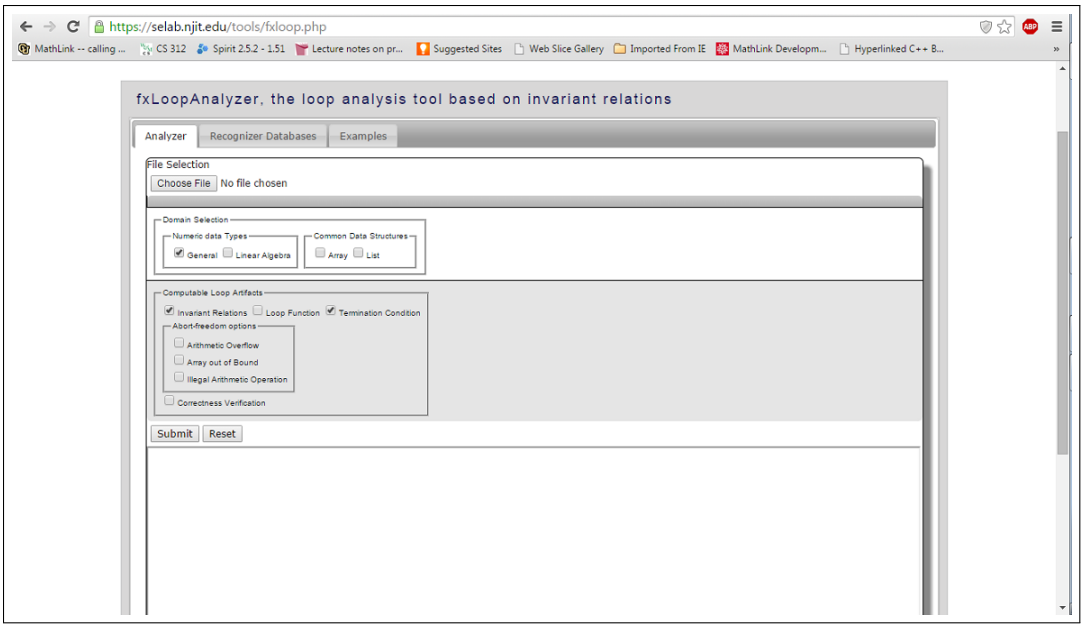


Figure 5.6 Convergence condition options.

5.5.5 Evaluating Correctness

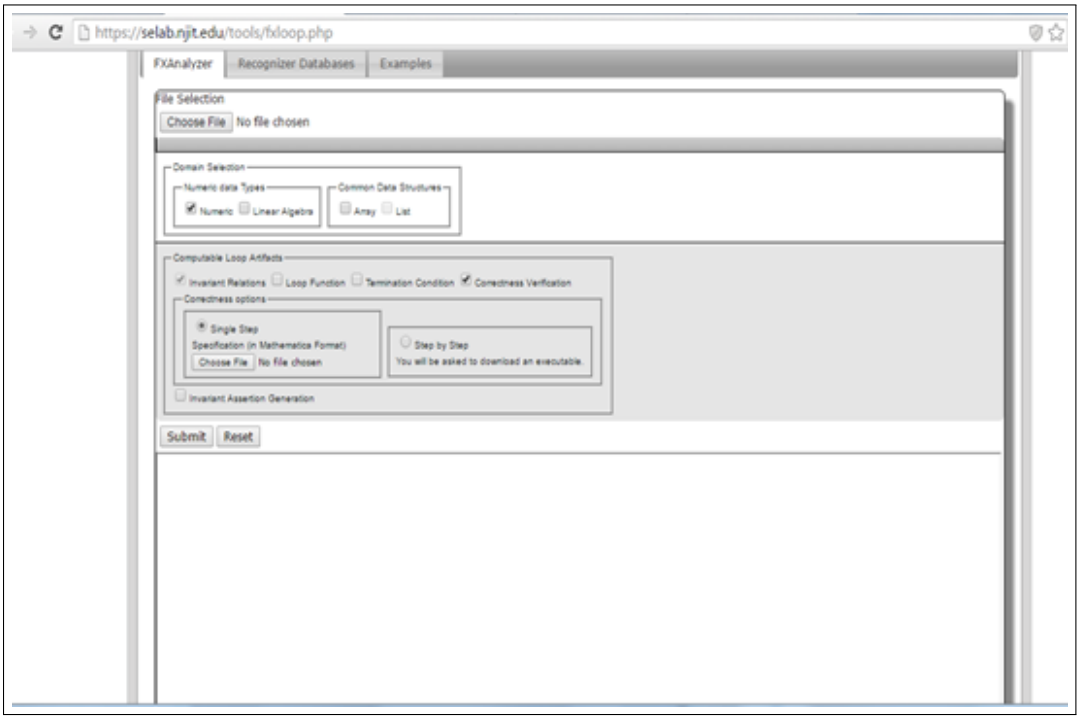


Figure 5.7 Correctness condition options.

5.6 Limitations

One may argue that our approach lacks generality because it depends on a pre-coded database of recognizers. We put forth the following observations:

- It is impossible to build a system to analyze programs without codifying the programming knowledge and the domain knowledge that are needed for this task; we argue that the recognizers are our way to capture the relevant programming knowledge and domain knowledge.

- We are currently exploring ways to do away with pre-coded recognizers for simple numeric calculations; indeed, many of our numeric invariant relations can be generated automatically from the source code by converting the code to recurrence relations (according to the work of Janicki and Carrette [6]) and eliminating the recurrence variable.

- The focus of this dissertation is the generation of convergence conditions from invariant relations; we deploy some automated tools in the process of analyzing while loops, but these tools are not integrated, and they require some level of human intervention.

CHAPTER 6

RELATIVE CORRECTNESS

6.1 Introduction

In [101, 59, 102, 103] Laprie et al. define the hierarchy of faults, errors and failures as part of the conceptual basis of dependable computing. In this hierarchy, faults are defined as *the adjudged or hypothesized cause of an error* [59]; we argue that, as far as software is concerned, this definition is not sufficiently precise, first because adjudging and hypothesizing are highly subjective human endeavors, and second because the concept of error is itself insufficiently defined, since it depends on a detailed characterization of correct system states at each stage of a computation (which is usually unavailable). We further argue that a formal/unambiguous definition of faults is indispensable, given that faults play a crucial role in the study of software dependability, that they are the basis of the classification of methods of dependability (fault avoidance, fault removal, fault tolerance), and that they play an important role in several software engineering concepts, such as fault density, fault proneness, and fault forecasting. But defining software faults is fraught with difficulties:

- *Discretionary determination.* Usually we determine that a program part is faulty because we think we know what the designer intended to achieve in that particular part, and we find that the program does not fulfill the designer's intent; clearly, this determination is only as good as our assumption about the designer's intent.

- *Contingent determination.* The same faulty behavior of a software product may be repaired in more than one way, possibly involving more than one location; hence, the determination that one location is a fault is typically contingent upon the assumption that other parts are not in question.

- *Tentative determination.* The determination that a program part is faulty is usually made in conjunction with a substitution that would presumably repair

the program; clearly, this determination is valid only to the extent that the substitution is an adequate repair.

- *Inconclusive determination.* Usually, we determine that a fault has been removed from a program if upon substituting the allegedly faulty part by an allegedly correct part, we find that the program runs satisfactorily on some test data T. In fact, the successful execution of the program on test data T is neither a necessary condition nor a sufficient condition to the actual removal of the fault.

In order to overcome the difficulties raised above, we resolve to proceed as follows:

- We introduce a concept of *relative correctness*, i.e. the property of a program to be more correct than another program with respect to a specification [27].

- We define a fault in a program as any program part (be it a simple statement, a lexical token, an expression, a compount statement, a block of statements, a set of non-contiguous statements, etc.) for which there exists a substitution that would make the program more-correct than the original with respect to a relevant specification [27].

With such a definition, we address all the difficulties raised above, namely:

- *A Fault as an Intrinsic Attribute.* The definition of a fault is not dependent on any design assumptions, but involves only the (incorrect) program, the faulty program part, and the specification with respect to which correctness (and failure) is defined.

- *A Fault as a Definite Property.* If we let a fault be any program part that admits a substitution that makes the program more-corect, then the designation of a fault is no longer contingent on any hypothesis; we need not make any assumption on whether other parts of the program are faulty or not.

- *A Fault as an Opportunity for Correctness Enhancement.* By definition, every fault represents an opportunity to make the program more-correct, i.e. closer to being correct; the challenge of the tester is to find an appropriate substitution, knowing that one does exist.

- *Fault Removal as a Verifiable Process.* Whether a fault has been removed is not dependent on the program's behavior on some (partial) test data, but rather on a formally verifiable property.

In order to reap all these benefits, we introduce a definition of relative correctness; this is the subject of Section 6.3. Whereas absolute correctness characterizes a program with respect to a specification, relative correctness ranks two programs with respect to a specification; in order to discuss the latter, we first review the former in 6.2, to see how it is defined in our notation. In Section 6.4, we consider in turn several properties that we would want a concept of relative correctness to satisfy, and we prove that our proposed definition does satisfy all of them; the goal of this section is to give the reader a measure of confidence in the soundness of the proposed definition, as a prelude to the subsequent discussions. We then discuss about the implications of relative correctness. Then the next issue we wish to address is: how do we establish relative correctness, i.e. how to build the case that a program is more-correct than another with respect to a specification; this is the subject of Section 6.7. Section 6.9 summarizes and assesses our findings.

6.2 Absolute Correctness

We define program correctness using the refinement ordering introduced in Chapter 2.

Definition 5 *Let p be a program on space S and let R be a specification on S.*

- *We say that program p is* correct *with respect to R if and only if P (the function defined by program p on space S) refines R.*

- *We say that program p is* partially correct *with respect to specification R if and only if P refines $R \cap PL$.*

Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*. This definition is consistent with traditional definitions of partial and total correctness [12, 14, 104, 32, 33]. The following proposition gives a simple characterization of correctness, and sets the stage for the definition of relative correctness.

Proposition 12 *Program g is correct with respect to specification R if and only if $(G \cap R)L = RL$.*

Proof. Proof of necessity: The condition $(G \cap R)L \subseteq RL$ stems readily from set theory, hence, we focus on proving the condition $RL \subseteq (G \cap R)L$. Given that g is correct with respect to R, we know that G refines R. By virtue of the lemma introduced in the proof of proposition 2, we have the hypotheses: $RL \subseteq GL$ and $RL \cap G \subseteq R$. Let s be an element of the domain of R; by the first clause, it is necessarily an element of the domain of G. By virtue of the second clause, $(s, G(s))$ is necessarily an element of R. Because $(s, G(s))$ is an element of R and G (by definiton), it is an element of $(G \cap R)$; hence, s is an element of the domain of $(G \cap R)$.

Proof of sufficiency: From $RL = (G \cap R)L$ (hypothesis) and $(G \cap R)L \subseteq GL$ (set theory) we infer $RL \subseteq GL$. Let (s, s') be an element of $(RL \cap G)$; then s is in the domain of R and $s' = G(s)$. By hypothesis, we know that s is in the domain of $(G \cap R)$, which means that $(s, G(s))$ is in R. Since $G(s) = s'$, we infer that (s, s') is in R. **qed**

In [105], Mills et al. define correctness of a program p with function P with respect to the specification R, by the formula $(R \cap P)L = RL$; hence, this proposition is inspired by their definition (though for us it is a proposition rather than a definition because we define correctness by means of refinement). Note that we could likewise characterize partial correctness by the formula: $(R \cap P)L = RL \cap PL$; but since relative correctness is a generalization of (total) correctness rather than partial correctness, proposition 12 only talks about (total) correctness.

In this dissertation, we are only interested in total correctness, to which we refer by *correctness*. The following definition introduces the concept of *relative correctness*; to contrast correctness to relative correctness, we may refer to the former as *absolute correctness*.

6.3 Relative Correctness

6.3.1 Deterministic Programs

Definition 6 *Let R be a specification on space S and let p and p' be two deterministic programs on space S whose functions are respectively P and P'.*

- *We say that program p' is* more-correct *than program p with respect to specification R (denoted by: $P' \sqsupseteq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$.*

- *Also, we say that program p' is* strictly more-correct *than program p with respect to specification R (denoted by: $P' \sqsupset_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.*

Interpretation: $(R \cap P)L$ represents (in relational form) the set of initial states on which the behavior of P satisfies specification R. We refer to this set as the *competence domain* of program P. Relative correctness of P' over P with respect to specification R simply means that P' has a larger competence domain than P. Whenever we want to contrast correctness (given in Definition 5) with relative correctness, we may refer to it as *absolute correctness*. Note that when we say *more-correct* we really mean *more-correct or as-correct-as*; we use the shorthand, however, for convenience. Note also that in order for program p' to be more-correct than program p, it does not need to duplicate the behavior of p over the competence domain of p; see Figure 6.3.1. In the example shown in this figure, we have:

$(R \cap P)L = \{1, 2, 3, 4\} \times S$,

$(R \cap P')L = \{1, 2, 3, 4, 5\} \times S$,

where $S = \{0, 1, 2, 3, 4, 5, 6\}$. Hence p' is more-correct than p with respect to R.

In order to highlight the contrast between relative correctness and absolute correctness, we consider the specification R on space $S = nat$

$$R = \{(s, s') | s^2 \leq s' \leq s^3\},$$

and we consider the following programs, where along with each program we indicate its function, then its competence domain with respect to R:

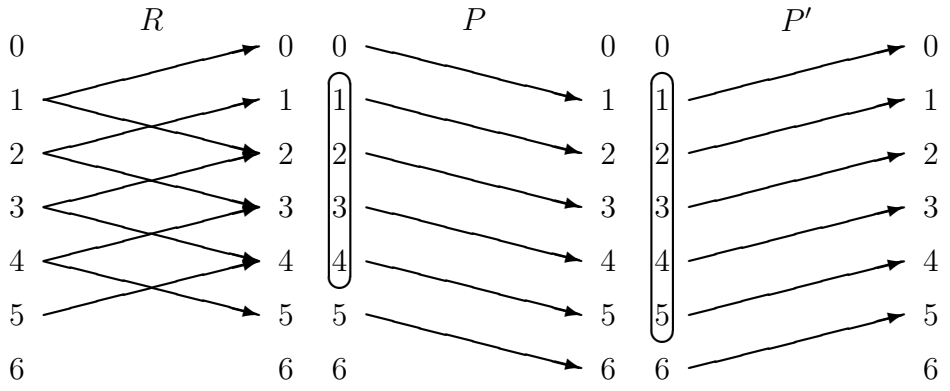


Figure 6.1 Enhancing correctness without duplicating behavior: $P' \sqsupseteq_R P$.

p0: {`abort`}. $P_0 = \phi$. $CD_0 = \emptyset$.

p1: {`s=0;`}. $P_1 = \{(s, s') | s' = 0\}$. $CD_1 = \{0\}$.

p2: {`s=1;`}. $P_2 = \{(s, s') | s' = 1\}$. $CD_2 = \{1\}$.

p3: {`s=2*s**3-8;`}. $P_3 = \{(s, s') | s' = 2s^3 - 8\}$. $CD_3 = \{2\}$.

p4: {`skip;`}. $P_4 = I$. $CD_4 = \{0, 1\}$.

p5: {`s=2*s**3-3*s**2+2;`}. $P_5 = \{(s, s') | s' = 2s^3 - 3s^2 + 2\}$. $CD_5 = \{1, 2\}$.

p6: {`s=s**4-5*s;`}. $P_6 = \{(s, s') | s' = s^4 - 5s\}$. $CD_6 = \{0, 2\}$.

p7: {`s=s**2;`}. $P_7 = \{(s, s') | s' = s^2\}$. $CD_7 = S$.

p8: {`s=s**3;`}. $P_8 = \{(s, s') | s' = s^3\}$. $CD_8 = S$.

p9: {`s=(s**2+s**3)/2;`}. $P_9 = \{(s, s') | s' = \frac{s^2+s^3}{2}\}$. $CD_9 = S$.

Figure 6.2 shows how these ten programs are ordered according to their relative correctness with respect to R; in this sample, programs P_7, P_8, P_9 are (absolutely) correct while programs $P_0, P_1, P_2, P_3, P_4, P_5, P_6$ are incorrect because their competence domain is strictly smaller than the domain of R.

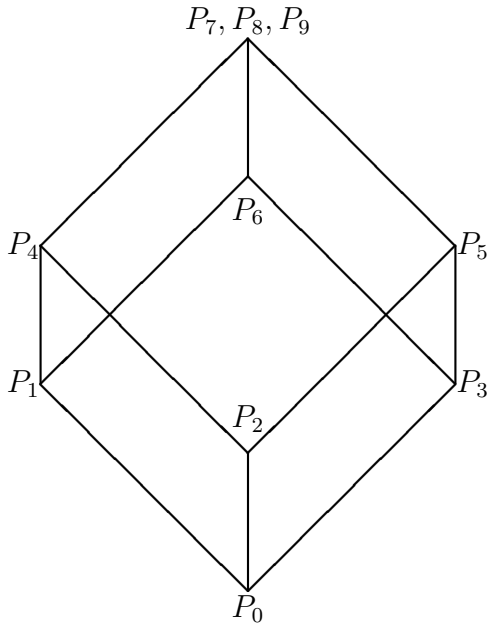


Figure 6.2 Ordering candidate programs by relative correctness.

6.3.2 Non-Deterministic Programs

In this section we extend the definition of relative correctness to non deterministic program. There are several reasons why this is important/ useful:

- Non-determinacy is a convenient means to model deterministic programs whose detailed behavior is difficult to capture, unknown, or irrelevant to a particular analysis.

- We may want to reason about the relative correctness of deterministic programs without having to compute their function is all its minute details.

- We may want to apply relative correctness, not only to finished software products, but also to partially defined intermediate artifacts, such as designs.

We submit the following definition.

Definition 7 *We let R be a specification on set S and we let P and P' be (possibly non-deterministic) programs on space S. We say that P' is more-correct than P with respect to R (abbrev: $P' \sqsupseteq_R P$) if and only if:*

$$(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P' \subseteq P.$$

Interpretation: P' is more-correct than P with respect to R if and only if it has a larger competence domain, and for the elements in the competence domain of P, program P' has fewer images that violate R than P does. As an illustration, we consider the set $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and we let R, P and P' be defined as follows:

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

$P = \{(0,2), (0,3), (1,3), (1,4), (2,0), (2,1), (3,1), (3,2), (4,1), (4,2), (5,2), (5,3)\}$

$P' = \{(0,2), (0,3), (1,2), (1,3), (2,0), (2,3), (3,1), (3,4), (4,2), (4,5), (5,2), (5,3)\}$

From these definitions, we compute:

$R \cap P = \{(2,1), (3,2)\}$,

$(R \cap P)L = \{2,3\} \times S$,

$R \cap P' = \{(1,2), (2,3), (3,4), (4,5)\}$

$(R \cap P')L = \{1,2,3,4\} \times S$

$(R \cap P)L \cap P' = \{(2,0), (2,3), (3,1), (3,4)\}$

$(R \cap P)L \cap \overline{R} \cap P' = \{(2,0), (3,1)\}$

By inspection, we do find that $(R \cap P)L = \{2,3\} \times S$ is indeed a subset of $(R \cap P')L = \{1,2,3,4\} \times S$. Also, we find that $(R \cap P)L \cap \overline{R} \cap P' = \{(2,0), (3,1)\}$ is a subset of P. Hence the two clauses of Definition 7 are satisfied. Figure 6.3 represents relations R, P and P' on space S. Program P' is more-correct than program P with respect to R because it has a larger competence domain ($\{2,3\}$ vs. $\{1,2,3,4\}$, highlighted in Figure 6.3) and because on the competence domain of P ($=\{2,3\}$), program P' generates no incorrect output unless P also generates it ($\{(2,0), (3,1)\}$).

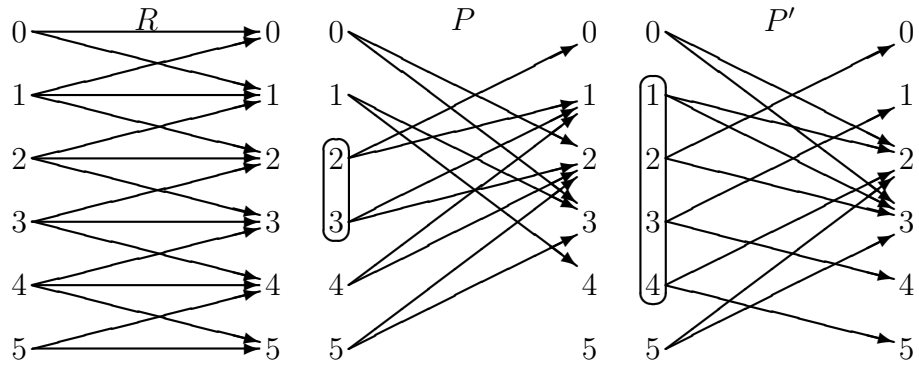


Figure 6.3 Relative correctness for non-Deterministic programs: $P' \sqsupseteq_R P$.

We show in [106] that if P' is deterministic, then the conditions $(R \cap P)L \subseteq (R \cap P')L$ and $P' \sqsupseteq_R P$ are logically equivalent, which means that Definition 7 can be used in general for relative correctness.

How do we know that our definition of relative correctness is valid? We have reviewed a number of properties that we would want a definition of relative correctness to have, and found that our definition features them all:

- *Relative Correctness Culminates in Absolute Correctness.* Indeed, it is very easy to see, from the definitions of correctness and relative correctness that if a program p is correct with respect to R, then it is more-correct than any program with respect to R.

- *Relative Correctness Implies Higher Reliability.* The probability of successful execution of a randomly chosen initial state is equal to the integral of the probability distribution over the competence domain of the program; hence, the larger the competence domain, the higher the probability.

- *Relative Correctness as Point-wise Refinement.* We have found in [27] that if and only if a program p refines a program p', then p is more-correct than p' with respect to any specification R. We will see in Section 7.4 that this property has an implication on program design.

6.4 Validation of Relative Correctness

6.4.1 Litmus Tests

Now that we have defined the concept of relative correctness, how do we know that our definition is sound? To answer this question, we list in this section some properties that, we believe, a definition of relative correctness ought to meet; then, in the next section, we check that our definition does indeed meet these conditions. For each property cited below, we discuss why we believe that a definition of relative correctness needs to satisfy this property.

- *Reflexivity and Transitivity, and non-Antisymmetry.* Referring to the loose version of relative correctness (more-correct-than-or-as-correct-as), we feel that this property ought to be transitive and reflexive, for obvious reasons. We also feel that it must not be antisymmetric: In other words, two programs may be mutually more-correct (each is more-correct than the other), and still be distinct (not only syntactically distinct, but computing different functions as well). In particular, we want to maintain the possibility that a given specification admit more than one correct program: all the correct programs are more-correct than one another, without necessarily being identical.

- *Absolute Correctness as the Culmination of Relative Correctness.* Relative correctness ought to be defined in such a way that if a program keeps getting more and more-correct with respect to a specification, it will eventually be (absolutely) correct. Alternatively, we want relative correctness to be defined in such a way that a correct program is more-correct than any candidate program.

- *Relative Correctness as a Sufficient Condition of Higher Reliability, but not a Necessary Condition Thereof.* If program p' is more-correct than program p, then of course we want p' to be more reliable than p; but we do not want *more-correct* to be equivalent to *more reliable*, as the former is a logical/functional property, whereas the latter is a stochastic property.

- *Refinement Implies Relative Correctness with respect to any Specification.* When program p' refines program p, we interpret that to mean that whatever p can do, p' can do as well or better; in particular, it means that p' is more-correct than (or as-correct-as) p with respect to R, for any specification R.

- *Relative Correctness with respect to Arbitrary Specifications Implies Refinement.* The only way for a program p' to be more-correct than a program p with respect

to all possible specifications is for p' to refine p; that way, we are assured that whatever p can do, p' can do as well or better.

6.4.2 Passing the Tests

In this section, we review in turn the desirable properties we have listed above, and show that our definition of relative correctness satisfies every one of them.

Reflexivity, Transitivity, and Non-Antisymmetry Program p' is more-correct than program p if and only if $(R \cap P')L \supseteq (R \cap P)L$. Transitivity and reflexivity stem readily from the definition, as does non-antisymmetry: Indeed, two functions P and P' may satisfy $(R \cap P)L = (R \cap P')L$ while P and P' are distinct. Consider $R = \{(0,1),(0,2)\}$, $P = \{(0,1)\}$ and $P' = \{(0,2)\}$.

6.4.3 Absolute Correctness as the Culmination of Relative Correctness

A program is more-correct than another if it has a larger competence domain, where the competence domain of a program p with respect to specification R is defined as $(R \cap P)L$. By set theory, the competence domain of a program p is necessarily a subset of RL; when it actually equals RL, the program is correct.

Proposition 13 *Let R be a specification on space S and let p be a program on S. Then p is correct with respect to R if and only if p is more-correct with respect to R than any program on S.*

Proof. Proof of necessity: Let p' be correct with respect to R; then, according to proposition 12, $RL = (R \cap P')L$. Let p be an arbitrary program on space S; by set theory, we have $RL \supseteq (P \cap R)L$. Hence p' is more-correct with respect to R than p.

Proof of sufficiency: Let p' be more-correct with respect to R than any candidate program p on S. Let p'' be a correct program with respect to R; then $(R \cap P'')L = RL$. Since p' is more correct with respect to R than p'', $(R \cap P')L \supseteq (R \cap P'')L$, hence,

$(R \cap P') \supseteq RL$, which is equivalent to $(R \cap P')L = RL$ since the inverse inclusion is a tautology. **qed**

This proposition provides in effect that (absolute) correctness is the ultimate form of relative correctness: to be correct with respect to a specification a candidate program must be more-correct than any candidate program. We write this as:

$$\boxed{P' \sqsupseteq R \Leftrightarrow (\forall P : P' \sqsupseteq_R P)}.$$

6.4.4 Relative Correctness and Reliability

The next proposition links the concept of relative correctness to a familiar property: reliability.

Proposition 14 *Let p and p' be two programs on space S and let R be a specification on S. If program p' is more-correct than program p with respect to specification R then p' is more reliable than p.*

Proof. We interpret *more reliable* to mean *less likely to fail*. Reliability is usually estimated with respect to a probability distribution over the input domain (specifically, the domain of specification R), which reflects the likelihood of occurrence of each element of the domain. Given a candidate program p and a probability distribution θ on the domain of R, the probability that a random execution of p succeeds is the integral of θ over the competence domain of p; clearly, the larger the competence domain (with respect to inclusion), the bigger the probability of successful execution. **qed**

If a program is more-correct than another, then it is more reliable. The reverse is not true, of course: a program may be more reliable than another without being

more-correct; it may be more reliable because it runs successfully on more frequently occuring input states, and fails on seldom occuring input states. We write:

$$\boxed{P' \sqsupseteq_R P \Rightarrow \int_{(R \cap P')L} \theta(s)ds \geq \int_{(R \cap P)L} \theta(s)ds}.$$

6.4.5 Relative Correctness and Refinement

The following proposition links relative correctness with the concept of refinement, by casting relative correctness as a form of pointwise refinement.

Proposition 15 *Let p and p' be programs on space S. Then p' refines p if and only if p' is more-correct than p with respect to any specification R on S.*

Proof. Proof of necessity: We have seen in proposition 3 that if P and P' are two functions that P' refines P if and only if $P' \supseteq P$. The condition $(P' \cap R)L \supseteq (P \cap R)L$ stems readily, by monotonicity (from set theory).

Proof of sufficiency: Let p' be more-correct than p with respect to any specification R on S. Then p' is more-correct than p with respect to specification $R = P$. This can be written as: $(P \cap P')L \supseteq (P \cap P)L$, which we simplify as: $(P \cap P')L \supseteq PL$. On the other hand, we have, by construction, $(P \cap P') \subseteq P$. Combining the two conditions, we obtain: $(P \cap P') = P$, from which we infer (by set theory) $P' \supseteq P$ and, by proposition 3, $P' \sqsupseteq P$. **qed**

This proposition provides in effect that traditional refinement is the ultimate form of relative correctness: whereas relative correctness is a tripartite relation linking two programs and a specification, refinement is a bipartite relation linking two programs when one is systematically more-correct than the other regardless of the specification being considered. We can write this as:

$$\boxed{P' \sqsupseteq P \Leftrightarrow (\forall R : P' \sqsupseteq_R P)}.$$

6.5 Faults and Fault Removal

Now that we have a definition for relative correctness, we are ready to define faults, and to characterize monotonic fault removal (i.e., fault removal that makes the program provably better, rather than to make the program work successfully for some inputs only to find that this was done at the expense of other inputs). In these definitions, we use the term *program part* to refer to any set of lexemes of the program's source code; these may range from a single lexeme (e.g. a comparison operator) to an expression to a simple statement to a compound a statement to a set of non-contiguous lexemes or statements spread across the source code of the program.

Definition 8 Faults, and Fault Removals. *Let p be a program on space S and R be a specification (relation) on S;*

- *let f be a program part of p. We say that f is a* fault *if and only if there exists a substitution f' of f such that the program p' obtained from p by substituting f by f' is strictly more-correct than p.*

- *Let f be a fault in p and let f' be a substitute for f. We say that the pair (f, f') is a* (monotonic) fault removal *if and only if the program p' obtained from p by substituting f by f' is strictly more-correct than p.*

Whereas *fault* is an intrinsic property of a program part, *monotonic fault removal* is a binary property involving a fault and a corresponding substitution. We argue that the qualifier *monotonic* is redundant (though we may still use it, for emphasis), as no substitution ought to be considered a fault removal unless it does make the program strictly more-correct than the original.

For illustration, we consider the following program, say p, taken from [107] (with some modifications):

```
1 #include <iostream>  ...  ...  ...
2 void main (char q[]){
3   int let, dig, other, i, l;
4   char c;
5   i=0;let=0;dig=0;other=0;l=strlen(q);
6   while (i<l) {
7     c = q[i];
8     if ('A'<=c && 'Z'>c) let+=2;
9     else if ('a'<=c && 'z'>=c) let+=1;
10    else if ('0'<=c && '9'>=c) dig+=1;
11    else other+=1;
12    i++;
13  }
14  printf ("%d %d %d\n",let,dig,other);
15 }
```

Listing 6.1 Initial program with modifications

To define the space of this program, we introduce the following notations:

- $\alpha_A =' A' \ldots ' Z'$.
- $\alpha_a =' a' \ldots ' z'$.
- $\nu =' 0' \ldots ' 9'$.
- $\sigma = \{' +', ' -', ' =', \ldots ' /'\}$, the set of all the ascii symbols.

We let $list\langle T\rangle$ denote the set of lists of elements of type T, and we let $\#_A$, $\#_a$, $\#_\nu$ and $\#_\sigma$ be the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols; also, we let $\#_\alpha$ be defined as $\#_\alpha(l) = \#_a(l) + \#_A(l)$, for an arbitrary list l. We let the space of this program be defined by all the variables declared in line 2. Also, by virtue of the `include` statement of line 1, we add a variable of type `stream`, that serves as the stream variable of the output file (in the parlance of C++). We let this variable be named *os* (for output stream), we assume (for the purposes of our example) that the stream is a sequence of natural numbers. Using these notations, we write the following specification on S:

$$R = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle \wedge os' = os \oplus \#_\alpha(q) \oplus \#_\nu(q) \oplus \#_\sigma(q)\},$$

where $\oplus$ represents the concatenation operator. We introduce the following programs, which are derived from p by some modifications of its source code:

p_{01} The program obtained from p when we replace (`let+=2`) by (`let+=1`).

p_{10} The program obtained from p when we replace (`'Z'>c`) by (`'Z'>=c`).

p_{11} The program obtained from p when we replace (`let+=2`) by (`let+=1`) and (`'Z'>c`) by (`'Z'>=c`).

We compute the expression $(R \cap P)L$ for each candidate program, and find the following:

- $(R \cap P)L = \{(s, s') | q \in list\langle \alpha_a \cup \nu \cup \sigma \rangle\}$.
- $(R \cap P_{01})L = \{(s, s') | q \in list\langle (\alpha_A \setminus \{'Z'\}) \cup \alpha_a \cup \nu \cup \sigma \rangle\}$.
- $(R \cap P_{10})L = \{(s, s') | q \in list\langle \alpha_a \cup \nu \cup \sigma \rangle\}$.
- $(R \cap P_{11})L = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle\}$.

Figure 6.4 illustrates the 'more-correct-than' relationships between programs p, p_{01}, p_{10}, and p_{11}, with respect to specification R; each ordering relationship is labeled with the corresponding substitution. From this Figure, we can make the following observations:

- The statement (`let+=2`) is a fault in p, and its substitution by (`let+=1`) is a monotonic fault removal, yielding the more-correct program p_{01}.
- The statement (`'Z'>c`) is a fault in p_{01}, and its substitution by (`'Z'>=c`) is a monotonic fault removal, yielding the more-correct program p_{11}.

- The program part defined by the two statements (`let+=2`) and (`'Z'>c`) is a fault in p, and its substitution by (`let+=1`) and (`'Z'>=c`) is a monotonic fault removal, yielding the more-correct program p_{11}.

- The program p_{11} is correct with respect to R, hence, it has no faults.

Note that the statement (`'Z'>c`) is a fault in p_{01} but it is not a fault in p; also note that the statement (`'Z'>c`), *in combination with* the statement (`let+=2`) forms a program part which is a fault in p, but it not a fault in p by itself. The reason is that program p fails for all upper case letters due to the fault (`let+2`), so whether it tests correctly for 'Z' or not does not matter.

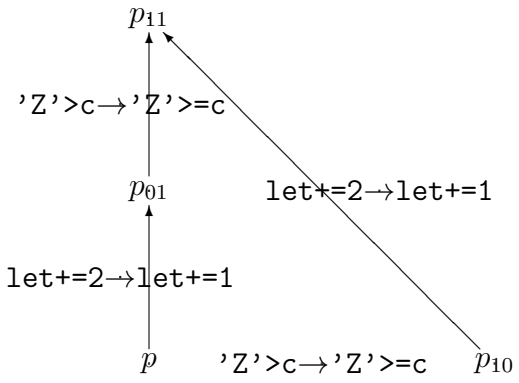


Figure 6.4 Monotonic and non-monotonic fault removals.

6.6 Implications of Relative Correctness

6.6.1 Counting Faults

In [59], Laprie et al. go to great lengths to define concepts and terminology pertaining to system dependability. In particular, they define the standard hierarchy of *fault*, *error*, and *failure*: a fault is the adjudged or hypothesized cause of an error; an error is the state of the system that may lead to its subsequent failure; a failure is the event whereby the system fails to meet its specification. In the context of software, it is fairly straightforward to characterize a failure: it is the event when the

program produces an output that violates the specification. It is much more difficult to characterize errors, because to do so assumes that we have a clear definition of what state the program must be in at any stage in its execution; it is even more difficult to characterize faults, because to do so assumes that we can trace every error to a single feature of the program. In fact, the same program failure can be remedied in more than one way, involving more than one location in the program, and possibly involving more than one type of remedy (adding statements, removing statements, altering existing statements, etc). Hence in practice, neither the number, nor the location, nor the nature of faults can be uniquely defined.

With our definition of relative correctness as a partial ordering that ranks candidate programs by how close they are to being (totally) correct with respect to a given specification, we define program faults: A fault in a given program with respect to a given specification is a program part (which can be a statement, a block of statements, or a set of non contiguous statements) which can be altered in such a way as to produce a more-correct program with respect to the specification.

Elementary faults Let p be a program on space S and R be a specification on S; let $f1$ be a fault in p, $f1'$ be a monotonic substitution of $f1$, let p' be the program obtained from p by substituting $f1$ by $f1'$, and let $f2$ be a fault in p'. We argue that the program part made up of $f1$ and $f2$ is a fault in p, since there exists a substitution of $(f1, f2)$ that would make p more-correct: we can substitute $f1$ by $f1'$ to obtain program p', and since by hypothesis $f2$ is a fault in p', there exists a substitution $f2'$ of $f2$ that would produce a program p'' that is more-correct than p. This raise the question: how many faults do we count in program p, one fault (program part $(f1, f2)$), two faults (program part $f1$ and program part $f2$), or three faults (program parts $f1$, $f2$ and $(f1, f2)$). In order to settle this matter, we have to introduce the following definition.

Definition 9 *Let f be a fault in program p on space S with respect to specification R on S. We say that f is an* elementary fault *in p if and only if no part of f is a fault in p with respect to R.*

So that if we are going to count faults, we need to count elementary faults rather than arbitrarily large faults. If we consider the program we introduced in Section 6.5, we find that the statement {`let+2`} is an elementary fault with respect to R, and that the program part ({`'Z'>c`}, {`let+2`}) is a fault but is not an elementary fault.

```
#include <iostream>  ...  ...  ...
void main (char q[]){
  int let, dig, other, i, l;
  char c;
  i=0;let=0;dig=0;other=0;l=strlen(q);
  while (i<l) {
    c = q[i];
  if ('A'<=c && 'Z'>c) let+=2;
  else if ('a'<=c && 'z'>=c) let+=1;
  else if ('0'<=c && '9'>=c) dig+=1;
  else  other+=1;
  i++;
  }
  printf ("%d %d %d\n",let,dig,other);
}
```

Listing 6.2 Program from Section 6.5

Multi-site faults The foregoing discussion about elementary (and non-elementary) faults may leave the reader with the impression that elementary faults are merely single-site faults, i.e. faults that involve a single statement (or, more broadly, a single contiguous program part). The purpose of this section is to dispel this notion, and to characterize multi-site elementary faults. The distinction between elementary multi-site faults and multiple elementary faults is important from the standpoint of multiple mutation generation.

We consider the following space S, specification R, and program p:

- Space, S: {`x: real; i: int; a: array [0..N] of real;`}.

- Specification, $R = \{(s, s') | x' = \sum_{i=1}^{N} a[i]\}$.

- Program p: {x=0; i=0; while (i<=N-1) {x=x+a[i]; i=i+1;}}

We compute the function of this program, then its competence domain:

$P = \{(s, s') | a' = a \wedge i' = N \wedge x' = \sum_{j=0}^{N-1} a[j]\}$.

$(R \cap P) = \{(s, s') | a' = a \wedge i' = N \wedge a[0] = a[N]\}$.

$(R \cap P)L = \{(s, s') | a[0] = a[N]\}$.

Since $(R \cap P)L$ is not equal to RL, which is L, this program is not correct; indeed, it computes the sum of the array from 0 to N_1 while the specification mandates computing the sum between indices 1 and N. One way to correct this program is to change {`i=0`} to {`i=1`} and to change {`i<=N-1`} to {`i<=N`}. The question that we raise here is: do we have two faults here ({`i=0`} and {`i<=N-1`}) or just one fault that spans two sites? To answer this question we consider the proposed substitutions and check whether they produce more-correct programs. We find:

- $p_{01} = \{x = 0; i = 1; while(i <= N - 1)\{x = x + a[i]; i = i + 1; \}\}$.
  $P_{01} = \{(s, s') | a' = a \wedge i' = N \wedge x' = \sum_{j=1}^{N-1} a[j]\}$.
  $(R \cap P_{01}) == \{(s, s') | a[N] = 0 \wedge a' = a \wedge i' = N \wedge x' = \sum_{j=1}^{N} a[j]\}$.
  $(R \cap P_{01})L == \{(s, s') | a[N] = 0\}$.

- $p_{10} = \{x = 0; i = 0; while(i <= N)\{x = x + a[i]; i = i + 1; \}\}$.
  $P_{10} = \{(s, s') | a' = a \wedge i' = N + 1 \wedge x' = \sum_{j=0}^{N} a[j]\}$.
  $(R \cap P_{10}) = \{(s, s') | a[0] = 0 \wedge a' = a \wedge i' = N + 1 \wedge x' = \sum_{j=1}^{N} a[j]\}$.
  $(R \cap P_{10})L = \{(s, s') | a[0] = 0\}$.

Since the competence domain of p is not a subset of the competence domains of p_{01} and p_{10}, neither p_{01} nor p_{10} is more-correct than p. We extrapolate: no substitution to {`i=0`} can cause program p to include cell $a[N]$ in the sum it is computing in x, and no substitution to {`i<=N-1`} can preclude program p from including cell $a[0]$ in the sum it is computing in x. Hence neither program part {`i=0`} nor program part {`i<=N-1`} is a fault in program p with respect to R, but program part ({`i=0`}, {`i<=N-1`}) is a fault in program p with respect to R, since substitution of this fault

by ({`i=1`}, {`i<=N`}) produces a more-correct (and actually correct) program. We say about this fault

Fault density and fault depth It is common for software researchers and practitioners to talk about fault density of a program as a measure of program quality and/or as a measure of the effort that it takes to transform the program into a correct program. In this section we show that fault density reflects neither quality nor fault removal effort: we can show a simple example of a program with a single fault that may still go through several monotonic fault removals before it is correct; also, we can show an example of a program that has several faults, but can be corrected in one *elementary* fault removal. In this discussion, we use the term *fault density* to mean: the number of faults in a program; strictly speaking, fault density is the number of faults per line of code, but for a given program size, these quantities are linearly related.

As an example of a program with a single fault but many fault removals, consider the following space S, specification R, and propgram p:

- Space, S: `int i; float a[0..N];` // $N \geq 2$.
- Specification, $R = \{(s, s') | \forall j : 0 \leq j \leq N : a'[j] = 0\}$.
- Program, p: {`i=2; while (i<=N) {a[i]=0; i=i+1;}`}

Clearly, $RL = L$. To determine correctness, we must compute $(R \cap P)L$ and compare it to RL. We find:

$P = \{(s, s') | a[0] = a'[0] \wedge a[1] = a'[1] \wedge \forall j : 2 \leq j \leq N : a'[j] = 0\}$.

$R \cap R = \{(s, s') | a[0] = a'[0] \wedge a[1] = a'[1] \wedge \forall j : 2 \leq j \leq N : a'[j] = 0\}$.

$(R \cap P)L = \{(s, s') | a[0] = 0 \wedge a[1] = 0\}$.

Hence p is not correct with respect to R. We can check easily that {`i=2`} is a fault in p with respect to R, and we show that the substitution of {`i=2`} by {`i=1`} produces a more-correct program:

- p': {`i=1; while (i<=N) {a[i]=0; i=i+1;}`},
- $P' = \{(s, s') | a[0] = a'[0] \wedge \forall j : 1 \leq j \leq N : a'[j] = 0\}$.
- $(R \cap P') = \{(s, s') | a[0] = a'[0] \wedge \forall j : 1 \leq j \leq N : a'[j] = 0 \wedge \forall j : 0 \leq j \leq N : a'[j] = 0\}$
- $(R \cap P')L = \{(s, s') | a[0] = 0\}$.

Since RL is (still) L, program P' is not correct with respect to R. We perform another fault removal when we replace {`i=1`} by {`i=1; a[0]=0;`}. We find:

- p'': {{`i=1; a[0]=0;`} `while (i<=N) {a[i]=0; i=i+1;}`}.
- $P'' = \{(s, s') | \forall j : 0 \leq j \leq N : a'[j] = 0\}$.
- $(R \cap P'') = \{(s, s') | \forall j : 0 \leq j \leq N : a'[j] = 0\}$.
- $(R \cap P'')L = L$.

Hence the same fault {`i=2`} has been corrected twice, first when we replaced it by {`i=1`} then we replaced {`i=1`} by {`i=1; a[0]=0;`} (we could have replaced {`i=1`} by {`i=0`}, though that would have looked far too artificial).

As for an example of a program with several faults that can be corrected with a single elementary fault removal, consider the following space, specification, and program:

- Space, S: {`x: real; i: int; a: array [0..N] of real;`}.
- Specification, $R = \{(s, s') | x' = \sum_{i=1}^{N} a[i]\}$.
- Program p: {`x=0; i=0; while (i<=N-1) {x=x+a[i]; i=i+1;}`}

We compute the function of this program then its competence domain with respect to R:

$P = \{(s, s') | a' = a \wedge i' = N \wedge x' = \sum_{j=0}^{N-1} a[j]\}$.

$(R \cap P) = \{(s, s') | a' = a \wedge i' = N \wedge a[0] = a[N]\}$.

$(R \cap P)L = \{(s, s') | a[0] = a[N]\}$.

Since this is not equal to RL (which is L), we conclude that this program is not correct with respect to R. We see at least two faults in this program, i.e. two program parts that admit substitutions that would make the program more-correct:

- The multi-site fault that we had identified in section 6.6.1, namely the program part ({`i=0`},{`i<=N-1`}).

- The single-site fault {`a[i]`} in the body of the loop; this is a fault since replacing it by {`a[i-1]`} yields a correct program.

If we do substitute {`a[i]`} by {`a[i-1]`} then we obtain a program p' where the multi-site program part ({`i=0`},{`i<=N-1`}) is not a fault, hence, the same fault removal action has removed more than one fault. This leads us to introduce the following definition, as a possible alternative metric to fault density.

Definition 10 *We consider a specification R on space S and a program p on space S. The* fault depth *of program p with respect to specification R is the minimal number of elementary fault removals that are required to transform p into a correct program.*

In light of this definition, we find that the depth of the array sum program above is 1, the depth of the array initialization program is also 1, and the depth of the character-counting program (Section 6.1) is 2. Note that the array sum program has a fault density of two, but a fault depth of 1; interestingly, the fault density, which is usually perceived as indicative of a flaw, in this case appears to be a sign of quality, since it gives us two distinct opportunities to correct the program. Hence not only is fault density a poor measure of program unsoundness, it may sometimes

reflect quite the opposite: it may measure the range of possibilities for making the program correct.

We argue that this metric is a more meaningful reflection of program quality, and certainly more directly related to the effort required to make the program correct; also, unlike fault density, this metric does decrease by one whenever we remove a fault (provided the fault is in the minimal path). Given a faulty program p and a program p' obtained from p by monotonic fault removal; if the fault removal is in a minimal sequence of faults removals to a correct program, then we can write:

$$\boxed{depth(p) = 1 + depth(p').}$$

If p'' is obtained from p by an arbitrary monotonic fault removal then all we can claim about the depths of p and p' is:

$$\boxed{depth(p) \leq 1 + depth(p'').}$$

Revisiting the sample program above, we find that if we remove the multi-site fault in the original program p, we find the following program p', which is also correct:

p'': `{x=0; i=1; while (i<=N) {x=x+a[i]; i=i+1;}}`

Interestingly, note also that if we proceed to remove both faults at once, this yields the following program, which is *not* correct:

p''': `{x=0; i=1; while (i<=N) {x=x+a[i+1]; i=i+1;}}`

This program has a fault density of 2 and a fault depth of 1; removing one fault makes it correct; but removing two faults makes it incorrect. The only meaningful characterization of fault density is whether the program has no faults (if it is correct) or whether it has at least one fault (if it is not correct); once we determine that it has at least one fault, then we need to remove that fault then ask the same question about the new program; the answer to that question depends on what substitution we have used to remove the first fault. In other words, fault density can take only

two meaningful values: 0, or > 0; fault depth, by contrast, takes its values in the set of natural numbers.

6.6.2 A Bridge Between Testing and Proving

Whereas traditionally we distinguish between two categories of candidate programs for a given specification R, namely correct programs and incorrect programs, relative correctness enables us to arrange candidate programs over a partial ordering structure, whose maximal elements are the correct programs, and all non-maximal elements are incorrect.

Also, traditionally, proving methods and testing methods have been used on different sets of programs:

- Proving methods are deployed on correct programs to prove their correctness; they are useless when deployed on incorrect programs because even when a proof fails, we cannot conclude that the program is incorrect, since we cannot tell whether the proof failed because the program is incorrect or because it was improperly documented (re: invariant assertions, intermediate assertions, etc).

- Testing methods are deployed on incorrect programs to detect, locate and remove their faults; they are useless when deployed on correct programs, because no matter how often a program runs failure-free under test, we can never (in practice) conclude with certainty that it is correct.

We argue that consideration of relative correctness (rather than traditional absolute correctness) has the potential to have a significant impact on both proving methods and testing methods:

- Once we have a formal definition of relative correctness, we can deploy proving methods to an incorrect program to prove that while it may be incorrect, it is still more-correct than another. In particular, we can take a faulty program (say P), remove a fault from it (to obtain a program P') and prove that the fault has been removed by showing that P' is more-correct than P. Given that there are orders of magnitude more incorrect programs than there are correct programs, the ability to apply proving methods to incorrect programs expands the scope of these methods significantly. This approach is discussed in Section 7.2 in Chapter 7.

- Relative Correctness can also alter the practice of software testing by recognizing the difference between testing for relative correctness and testing for (traditional) absolute correctness. Indeed, when we remove a fault from a program, we ought to test it for relative correctness rather than absolute correctness, unless we have reason to believe (how do we ever?) that the fault we have just removed is the last fault of the program. Yet in the current practice of software testing, programs are routinely tested for absolute correctness, even when we have no reason whatsoever to believe that they are correct (due to the presence of other faults). This matter is discussed in Section 6.7.

- It has long been a cornerstone of software engineering wisdom that programs should not be developed then checked for correctness, but should instead be developed hand-in-hand along with their proof, with the proof leading the way [104]; echoing David Gries, Carrol Morgan talks about developing programs by calculation from their specification, in the same way that a mathematician solves an equation by computing its root [36]. The favorite paradigm for developing programs from specifications has always been that of refinement, whereby a program is derived from a specification through a sequence of correctness-preserving transformations based on refinement. In Chapter 7 , Section 7.4 we present an alternative paradigm based on relative correctness, illustrate it with a simple example, then briefly discuss some of its advantages.

In Chapter 2 we introduce the mathematical background that is needed to carry out our discussions.

6.7 Testing for Relative Correctness

The usual process of software debugging proceeds as follows: We observe a failure of the program; we analyze the failure and formulate a hypothesis on its cause; we modify the source code on the basis of our hypothesis; and finally we test the new program to ensure that it is now correct. But there is a serious flaw in this process: when we remove a fault from an incorrect program, we have no reason to expect the new program to be correct, unless we know (how do we ever?) that the fault we have just removed is the last fault of the program; hence, when a fault is removed from a program, the new program ought to be tested for relative correctness over the original program, rather than for absolute correctness. When a doctor treats a patient for one condition (e.g. bacterial infection) then tests him/her for another (e.g.

diabetes) we would consider that a serious case of medical malpractice and grounds for career-ruining medical malpractice lawsuit. But when a software engineer removes a fault from a program then tests it for absolute correctness, we consider that as routine professional practice, even though it is essentially the same type of misconduct (incompatibility between the treatment and the test).

This raises the question: how do we test a program for relative correctness over another program, and how does that differ from testing it for absolute correctness. We argue that testing a program for relative correctness has an impact on three aspects of testing, namely test data selection, test oracle design, and test coverage assessment.

- *Test data selection.* The problem of test data selection can be summarized as follows: We are given a large or infinite test space S, and we must select a small subset thereof T such that the behavior of candidate programs on T is a faithful predictor of their behavior on S. The difference between absolute correctness and relative correctness is that for absolute correctness with respect to specification R, the test space S is $dom(R)$ whereas for relative correctness over P with respect to R the test space S is $dom(R \cap P)$.

- *Test oracle design.* Let $\Omega(s, s')$ be the test oracle for absolute correctness derived from specification R. Because relative correctness over program P tests a candidate program P' for Ω only for those states on which P is successful, the oracle for relative correctness $\omega(s, s')$ can be written as:
  $\omega(s, s') \equiv \Omega(s, P(s)) \Rightarrow \Omega(s, s')$.

- *Test Coverage Assessment.* It is not sufficient to know that some program P' has executed successfully on a test data set of size N using oracle $\omega(s, s')$; it is also necessary to know what percentage of the test data set satisfy the precondition $\omega(s, P(s))$.

In Chapter 7, we show examples of the difference between testing for relative correctness and testing for absolute correctness .

6.8 Related Work

In [108], Logozzo et al. introduce a technique for extracting and maintaining semantic information across program versions: specifically, they consider an original program P

and a variation (version) P' of P, and they explore the question of extracting semantic information from P, using it to instrument P' (by means of executable assertions), then pondering what semantic guarantees they can infer about the instrumented version of P'. The focus of their analysis is the condition under which programs P and P' can execute without causing an abort (due to attempting an illegal operation), which they approximate by sufficient conditions and necessary conditions. They implement their approach in a system called VMV (*Verification Modulo Versions*) whose goal is to exploit semantic information about P in the analysis of P', and to ensure that the transition from P to P' happens without regression; in that case, they say that P' is *correct relative to* P. The definition of relative correctness of Logozzo et al. [108] is different from ours, for several reasons: whereas [108] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance (where P and P' may be subject to distinct requirements), we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct) in the context of corrective maintenance with respect to a fixed requirements specification; whereas [108] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states to specify the standards against which absolute correctness and relative correctness is defined; whereas [108] represent program executions by execution traces (snapshots of the program state at assertion sites), we represent program executions by functions mapping initial states into final states; finally, whereas Logozzo et al. define a successful execution as a trace that satisfies all the relevant assertions, we define a successful as simply an initial state/ final state pair that falls with the specification (relation).

In [109], Lahiri et al. introduce a technique called *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. Lahiri et al. explore applications of this technique as a

tradeoff between soundness (which they concede) and lower costs (which they hope to achieve). Like the approach of Logozzo et al. [108] (from the same team), the work of Lahiri uses executable assertions as specifications, represents executions by execution traces, defines successful executions as traces that satisfy all the executable assertions, and targets abort-freedom as the main focus of the executable assertions. Also, they define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smallest set of unsuccessful traces than P; and they introduce relative specifications as specifications that capture functionality of P' that P does not have. By contrast, we use input/ output (or initil state/ final state) relations as specifications, we represent program executions by functions from initial states to final states, we characterize correct executions by initial state/ final state pairs that belong to the specification, and we make no distinction between abort-freedom (a.k.a. safety, in [109]) and normal functional properties. Indeed, for us the function of a program is the function that the program defines between its initial states and its final states; the domain of this function is the set of states for which execution returns terminates normally and returns a well-defined final state. Hence execution of the program on a state s is abort free if and only if the state is in the domain of the program function; the domain of the program function is part of the function rather than being an orthogonal attributes; hence, we view abort-freedom as a special form of functional attribute, rather than being an orthogonal attribute. Another important distinction with [109] is that we do not view relative correctness is a compromise that we accept as a substitute for absolute correctness; rather we argue that in many cases, we ought to test programs for relative correctness rather than absolute correctness, regardless of cost.

In [110], Logozzo and Ball introduce a definition of relative correctness whereby a program P' is correct relative to P (*an improvement* over P) if and only if P' has more good traces and fewer bad traces than P. Programs are modeled with

trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. Logozzo and Ball introduce an algorithm that specializes in deriving program repairs from a predefined catalog that is targeted to specific program constructs, such as: contracts, initializations, guards, floating point comparisons, etc. Like the work cited above ([108, 109]), Logozzo and Ball model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction, and model the two aspects with the same relational framework. Logozzo and Ball deploy their approach in an automated tool based on the static analyzer cccheck, and assess their tool for effectiveness and efficiency.

6.9 Conclusion

In this chapter, we have introduced the concept of relative correctness, used it to propose a definition for program faults, then explored the implications of these two concepts on a variety of aspects of testing and fault removal. Particularly, we presented the following:

- A definition of relative correctness, and an analysis of the proposed definition to ensure that it meets all the properties that one wants to see in such a concept.

- A definition of fault and fault removal, and the analysis of monotonic fault removal, as a process that transforms a faulty program into a correct program by a sequence of correctness-enhancing transformations.

- An analysis of mutation-based program repair, highlighting that when repair candidates are evaluated by testing them for absolute correctness rather than

relative correctness, one runs the risk of selecting programs that are not adequate repairs, and rejecting programs that are.

- A critique of the concept of fault density, and the introduction of fault depth as perhaps a more meaningful measure of the degree of imperfection of a faulty program; also the observation that for a given fault depth, the higher the fault density the better (which is the opposite of what fault density purports to represent).

- An analysis of techniques for testing that a program is more-correct than another with respect to a specification, and discussion of the difference between testing a program for relative correctness and testing it for absolute correctness.

- A study of techniques for proving, by static analysis, that a program is more-correct than another with respect to a given specification, as well as techniques for decomposing a proof of relative correctness with respect to a compound specification into proofs of relative correctness with respect to its building components

CHAPTER 7

APPLICATIONS OF RELATIVE CORRECTNESS

7.1 Introduction

In Chapter 6, we discuss about the concept of relative correctness, i.e., the property of a program to be more-correct than another with respect to a given specification. In this chapter, we explore the impact of relative correctness in software engineering, in software testing, and in software design. Further, we build upon our results on program design by showing that relative correctness can be used not only for program development from scratch, but also for various forms of program evolution. We argue, in fact, that virtually all software evolution is nothing but an effort to make some program more-correct with respect to some specification. Given that today most software is developed, not from scratch, but rather by evolving existing software products, we feel that exploration of this avenue may yield substantial returns across software engineering practice. Our purpose, in doing so, is not to offer polished/ validated/ scalable solutions; rather, it is merely to highlight some of the opportunities that are opened by relative correctness in the field of software evolution.

7.2 Debugging Without Testing

It is so inconceivable to debug a program without testing it that these two words are used nearly interchangeably. Yet we argue that using the concept of relative correctness, as described in Section 7.2.2, we can indeed remove a fault from a program and prove that the fault has been removed, by proving that the new program is more correct than the original. This is a departure from the traditional roles of proving and testing methods, whereby static proof methods are applied to a correct program to prove its correctness, and dynamic testing methods are applied to an incorrect program to expose its faults Broadly speaking, this method has the same advantages

and disadvantages as traditional methods for proving correctness by static analysis: namely that it offers the confidence and certainty of formally provable results, at the cost of mathematical formalisms and limited scalability. At the same time as we present the method, we also discuss means to capitalize on its advantages while mitigating its disadvantages. In the following , we present illustrative examples of the method.

7.2.1 Proving Correctness and Incorrectness of Loops

The analysis of while loops by means of invariant relations provides a way to infer the relative correctness of iterative programs from partial semantic information.

Invariant Relations and Absolute Correctness In [112], Mili et al. present a method to prove the correctness or incorrectness of a loop with respect to a specification, using invariant relations. This method is based on the following two propositions, which give, respectively, a sufficient condition and a necessary condition of correctness of a (uninitialized) while loop with respect to a specification R. Even though correctness is defined in terms of the program function, invariant relations enable us to rule on correctness or incorrectness long before we have collected all the necessary information to compute the loop function.

Proposition 16 Sufficient condition of correctness. *Given a while loop w of the form* `while (t) {b}` *that terminates for all states in its space S, and given a specification C on S, if an invariant relation V of w satisfies the condition*

$$V\overline{T} \cap RL \cap (R \cup V \cap \widehat{\overline{T}}) = R$$

then w is correct with respect to R.

This proposition provides, in effect, that if an invariant relation R meets this condition, then it contains sufficient information to subsume the specification, and to prove the correctness of the loop with respect to C.

Proposition 17 Necessary condition of correctness. *Let w be a while loop of the form $w =$* `while (t) {b}` *that terminates for all states in S, let V be an invariant relation for w, and let R be a specification on S. If w is correct with respect to R then*

$$(R \cap V)\overline{T} = RL.$$

This proposition provides, in effect, that any while loop that while this is a necessary condition of correctness, it is best to interpret it by considering that its negation is a sufficient condition of incorrectness. This proposition provides in effect that any while loop that admits an invariant relation V that does not satisfy this condition could not possibly be correct with respect to R. In other words, any while loop that admits an invariant relation V that satisfies the condition (note the change from $=$ to $\neq$)

$$(R \cap V)\overline{T} \neq RL.$$

is necessarily incorrect with respect to R. Any invariant relation V that satisfies this condition is said to be incompatible with respect to specification R. Any invariant relation that is not incompatible is said to be compatible. In [113], Mili et al. present an algorithm for proving the correctness or incorrectness of a loop with respect to a specification, which proceeds as follows:

- Using an invariant relations generator, we generate invariant relations one by one, and test the sufficient condition and necessary condition.
- If the aggregate of invariant relations found so far satisfy the sufficient condition then we conclude that the loop is correct, and we exit.
- If one of the invariant relations proves to be incompatible with R, we conclude that the loop is incorrect, and we exit.
- If we run out of invariant relations before we reach the conclusion that the loop is correct or that the loop is incorrect, then we conclude that we do not know

enough about the loop to rule on its correctness (hence, we must upgrade our invariant relations generator), and we exit.

In the next section we discuss how we can use a variation of this algorithm to establish relative correctness, rather than absolute correctness.

7.2.2 Proving Relative Correctness for Loops

In this section, we explore how they can be used to prove relative correctness of a loop over another with respect to a given specification. Given a while loop w of the form **while (t) {b}** on space S and a specification R on S, we are interested to determine whether w is correct with respect to R, and if not how we can locate and remove a fault in w. Ideally, we want to support all the steps in this process, namely:

- Determine that the loop is incorrect (for else there is no fault to remove).
- Determine the location of the fault.
- Determine what to replace the fault with.
- Prove that the substitution constitutes a monotonic fault removal.

To this effect, we consider the following proposition, which we give without proof.

Proposition 18 *Let R be a specification on space S and let w be a while loop on S of the form, w:* `{while (t) {b}}` *which terminates for all s in S. Let Q be an invariant relation of w that is incompatible with R, i.e. such that $(R \cap Q)\overline{T} \neq RL$; and let C be the largest invariant relation of w such that $W = (C \cap Q) \cap \widehat{\overline{T}}$. Let w' be a while loop that has C as an invariant relation, terminates for all s in S, and admits an invariant relation Q' that is compatible with R and satisfies the condition $W' = (C \cap Q') \cap \widehat{\overline{T}}$. Then w' is strictly more-correct than w.*

Interpretation: This proposition provides that if we change the loop in such a way as to replace an incompatible invariant relation (Q) with a compatible invariant relation (Q') of equal strength (so that $((C \cap Q') \cap \widehat{\overline{T}})$ is deterministic, just as $((C \cap Q) \cap \widehat{\overline{T}})$), while preserving all the other invariant relations (C), then we obtain a more-correct while loop.

Proof. By hypothesis, Q is incompatible with R, hence, we write:

$$(R \cap Q)\overline{T} \neq RL$$

$\Rightarrow$ { by set theory $(R \cap Q)\overline{T} \subseteq (R \cap Q)L \subseteq RL$ }

$$(R \cap Q)\overline{T} \subset RL$$

$\Rightarrow$ { By hypothesis, Q' is compatible }

$$(R \cap Q)\overline{T} \subset (R \cap Q')\overline{T}$$

$\Rightarrow$ { Taking the intersection with C on both sides }

$$(R \cap Q \cap C)\overline{T} \subset (R \cap Q' \cap C)\overline{T}$$

$\Rightarrow$ { For any vector v and relation R, $Rv = (R \cap \widehat{v})L$ }

$$(R \cap Q \cap C \cap \widehat{\overline{T}})L \subset (R \cap Q' \cap C \cap \widehat{\overline{T}})L$$

$\Rightarrow$ { associativity }

$$(R \cap (Q \cap C \cap \widehat{\overline{T}}))L \subset (R \cap (Q' \cap C \cap \widehat{\overline{T}}))L$$

$\Rightarrow$ { substitution }

$$(R \cap W)L \subset (R \cap W')L.$$

Hence w' is strictly more-correct than w with respect to R. **qed**

Using this Proposition, we propose the following algorithm for fault removal in while loops:

1. *Determination that the loop is faulty.* Given the specification R and the while loop w, we generate all the invariant relations we can, and place them in two separate columns, one for compatible relations and one for incompatible relations.

- If the *incompatible* column has at least one invariant relation, then the loop is incorrect, hence, it has a fault.

- If the *incompatible* column is empty and the intersection of all the compatible invariant relations satisfies the sufficient condition of correctness, then the loop is correct.

- If neither of the conditions above hold, then we cannot rule on the correctness of the loop, and the algorithm fails (the invariant relations generator needs to be upgraded).

2. *Localization of the Fault.* We consider the incompatible column and select from it an invariant relation that involves the fewest possible variables; for the same number of variables, we select the invariant relation (say Q) whose variables are involved in the smallest number of statements in the loop. We select one of these statements as the feature that we want to correct.

3. *Guidance to modify the selected statement.* We need to modify the selected statement in such a way as to replace the current incompatible invariant relation (Q) with a compatible invariant relation (Q). But we want to do so without affecting the compatible invariant relations. This constraint is used to generate a condition that guides us in the modification process. Let C be the intersection of all the compatible invariant relations, let x_1, x_2, x_3, ... x_n be the variables of the program, and let x_1 and x_2 be the two variables that appear in Q. Then, to preserve the compatible invariant relations of the loop, variables x_1, x_2, x'_1, x'_2 must satisfy the following constraint:

$$\exists x_3, x_4, ...x_n, x'_3, x'_4, ...x'_n : \begin{pmatrix} x_1 & x'_1 \\ x_2 & x'_2 \\ ... & ... \\ x_n & x'_n \end{pmatrix} \in C.$$

   We refer to this condition as the condition of compatibility preservation.

4. *Verification of Fault Removal.* Once we have changed the selected statement in such a way as to preserve the compatible invariant relations, we recompute the invariant relations and ensure that the selected incompatible invariant relation is now replaced by a compatible invariant relation. This ensures that we now have a more-correct program than we did before. This sends us back to step 1, to check whether the loop has now become correct (if its compatible relations subsume the specification) or whether it is still incorrect (if the incompatible column is still not empty).

We now illustrate this approach on initialized and uninitialized while loops.

7.2.3 Uninitialized Loops

Illustration 1 We reconsider example 6.1 from Chapter 6 with some modifications. We recall that the space of the specification is defined by the following variable declarations:

`char q[]; int let, dig, other, i, l; char c;`.

We let the specification R that we use for relative correctness be:

$R = \{(s, s') | q \in list(\alpha_A \cup \alpha_a \cup \vartheta \cup \sigma) \wedge let' = let + \#_a(q) + \#_A(q) \wedge dig' = dig + \#_\vartheta(q) \wedge other' = other + \#_\sigma(q)\}$

where $list < T >$ denotes the set of lists of elements of type T, $\#_A$, $\#_a$, $\#_\vartheta$ and $\#_\sigma$ the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols. The (faulty) program that we consider is w:

```
1  #include <iostream>  ...  ...  ...
2  void main (char q[]){
3    int let, dig, other, i, l;
4    char c;
5    i=0;let=0;dig=0;other=0;l=strlen(q);
6    while (i<l) {
7      i++;
8      c = q[i];
9      if ('A'<=c && 'Z'>=c) let=let-1;
10     else if ('a'<=c && 'z'>=c) let=let-1;
11     else if ('0'>c && '9'>=c) dig=dig+1;
12     else other=other+1;
13   }
14 }
```

Listing 7.1 Uninitialized loop: Initial program with modifications

We find the following invariant relations of this while loop:

- $V_0 = \{(s, s') | q = q'\}$
- $V_1 = \{(s, s') | i \leq i'\}$
- $V_2 = \{(s, s') | dig \leq dig'\}$

- $V_3 = \{(s, s') | other \leq other'\}$
- $V_4 = \{(s, s') | let \geq let'\}$
- $V_5 = \{(s, s') | let - \#_{a \cup A}(q[i..l-1]) = let' - \#_{a \cup A}(q'[i'..l-1])\}$
- $V_6 = \{(s, s') | dig + \#_{\sigma 1}(q[i..l-1]) = dig' + \#_{\sigma 1}(q'[i'..l-1])\}$
- $V_7 = \{(s, s') | other + \#_{\sigma 2 \cup \vartheta}(q[i..l-1]) = other' + \#_{\sigma 2 \cup \vartheta}(q'[i'..l-1])\}$

The following table table shows which of these invariant relations are compatible, and which are incompatible.

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3	V_4, V_5, V_6, V_7

Because the *incompatible* column is non-empty, we conclude that the program is incorrect with respect to R, hence, we must enhance its correctness. To this effect, we select the incompatible invariant relation V_4 for remediation, which leads us to focus on variable *let* for fault removal. Preservation of the compatible invariant relations mandates that *let* be modified under the following condition: $let \leq let'$. We propose: `let=let+1;`. The generation of invariant relations of the new loop yields the following table:

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3, V_4', V_5'	V_6, V_7

Application of the same process one more time yields the following program:

```
1 #include <iostream>  ...  ...  ...
2 void main (char q[]){
3   int let, dig, other, i, l;
4   char c;
5   i=0;let=0;dig=0;other=0;l=strlen(q);
6   while (i<l) {
7     i++;
8     c = q[i];
9     if ('A'<=c && 'Z'>=c) let=let+1;
```

```
10      else if ('a'<=c && 'z'>=c) let=let+1;
11      else if ('0'<=c && '9'>=c) dig=dig+1;
12      else other=other+1;
13    }
14  }
```

Listing 7.2 Uninitialized loop: program with faults removed

Analysis of this program produces 8 invariant relations, which are all compatible.

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3, V_4', V_5', V_6', V_7'	

This does not prove that the program is now correct, all it proves is that we have no evidence (in the forms of an incompatible invariant relation) that it is incorrect. To establish correctness, we must ensure that the intersection of all the available invariant relations satisfies the sufficient condition provided by Proposition 16, which it does. All the faults have been removed; we now have a correct program.

Illustration 2 As an illustrative example, we consider the state space S defined by the following variable declarations:

```
1  const float upsilon = 0.00001;
2  const float a= 0.15;
3  const float b= 0.08;
4  //  we always have:  0<b<a<1.0;
5  float r, p, n, x, m, l, k, y, w, y, z, v, u, d;  int t;
```

Listing 7.3 initialized loop: program with modifications

and we consider program w on a state space S defined by:

```
1  p1: while (abs(r-p)>upsilon)
2  {
3    t=t+1;
4    n=n+x;
5    m=m-l;
6    l=(1+b)*l;
7    k=k+1000;
8    y=n+k;
9    w=w+z;
10   z=(1+a)+z;
11   v=w+k;
```

```
12    r=(v-y)/y;
13    u=(m-n)/n;
14    d=r-u;
15 }
```

The invariant relations generator produces fourteen invariant relations:

- $V_1 = \{(s, s')|x' = x\}$.
- $V_2 = \{(s, s')|t \leq t'\}$.
- $V_3 = \{(s, s')|k \leq k'\}$.
- $V_4 = \{(s, s')||l| \leq |l'|\}$.
- $V_5 = \{(s, s')|z \leq z'\}$.
- $V_6 = \{(s, s')|k - 1000t = k' - 1000t'\}$.
- $V_7 = \{(s, s')|l(1 + b)^{-z} = l'(1 + b)^{-z'}\}$.
- $V_8 = \{(s, s')|l(1 + b)^{(-k/1000)} = l'(1 + b')^{(-k'/1000)}\}$.
- $V_9 = \{(s, s')|l(1 + b)^{-z/(1+a)} = l'(1 + b')^{-z'/(1+a)}\}$.
- $V_{10} = \{(s, s')|z - (1 + a)t = z' - (1 + a)t'\}$.
- $V_{11} = \{(s, s')|1000z - (1 + a)k = 1000z' - (1 + a)k'\}$.
- $V_{12} = \{(s, s')|m + l/b = m' + l'/b\}$.
- $V_{13} = \{(s, s')|w + z(z - 1 - a)/(2(1 + a)) = w' + z'(z' - 1 - a)/(2(1 + a))\}$.
- $V_{14} = \{(s, s')|1000n - kx = 1000n' - k'x'\}$.

We consider the following specification R on space S:

$$R = \{(s, s')|b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1 - (1 + a)^{t'-t}}{a} \wedge m' \geq 0 \wedge l' \geq 0$$

We review all the invariant relation for compatibility with respect to R; this is done using Mathematica (©Wolfram Research), by writing a logical formula that corresponds to the condition of compatibility discussed above. We find:

Compatible	Incompatible
V_1, V_2, V_3, V_4, V_5, V_6, V_{11}, V_{14}	V_7, V_8, V_9, V_{10}, V_{12}, V_{13}

We select invariant relation V_7 for remediation; the variables that appear in this relation are l and z. We compute the condition of compatibility preservation, and we find:

$$|l| \leq l' \wedge z \leq z'$$

We focus on variable z, consider the statement where this variable is modified, and consider alternative statements that satisfy the constraint. For each alternative, we recompute the new invariant relation that stems from the new statement and check for compatibility. We find the following substitute:

```
z=(1+a)*z;
```

Hence the new program:

```
1  p2: while (abs(r-p)>upsilon){
2      t=t+1;
3      n=n+x;
4      m=m-l;
5      l=(1+b)*l;
6      k=k+1000;
7      y=n+k;
8      w=w+z;
9      z=(1+a)*z;
10     v=w+k;
11     r=(v-y)/y;
12     u=(m-n)/n;
13     d=r-u;
14 }
```

Listing 7.4 Uninitialized loop: 2 faults removed

We do not know whether this program is correct, but we know that it is more-correct than the original program; if we test it and it fails, it will not be because our fault removal was wrong; rather it will be because it has other faults. When we run the invariant relations generator on this program, we find the following list.

- $V_1 = \{(s, s') | x' = x\}$.
- $V_2 = \{(s, s') | t \leq t'\}$.
- $V_3 = \{(s, s') | k \leq k'\}$.
- $V_4 = \{(s, s') | |l| \leq |l'|\}$.
- $V_5 = \{(s, s') | z \leq z'\}$.
- $V_6 = \{(s, s') | k - 1000t = k' - 1000t'\}$.
- $V_7 = \{(s, s') | l \leq l'\}$.
- $V_8 = \{(s, s') | 1000l - (1 + b)k = 1000l' - (1 + b)k'\}$.
- $V_9 = \{(s, s') | (1 + b)z - (1 + a)l = (1 + b)z' - (1 + a)l'\}$.
- $V_{10} = \{(s, s') | 1000z - (1 + a)k = 1000z' - (1 + a)k'\}$.
- $V_{11} = \{(s, s') | 1000n - kx = 1000n' - k'x'\}$.
- $V_{12} = \{(s, s') | (1 + b)n - xl = (1 + b)n' - x'l'\}$.
- $V_{13} = \{(s, s') | z(1 + a)^{-t} = z'(1 + a)^{-t'}\}$.
- $V_{14} = \{(s, s') | z(1 + a)^{-k/1000} = z'(1 + a)^{-k'/1000}\}$.
- $V_{15} = \{(s, s') | w - \frac{z}{a} = w - \frac{z}{a}\}$.
- $V_{16} = \{(s, s') | m + \frac{l}{b} = m + \frac{l}{b}\}$.

Checking these invariant relations for compatibility against specification R, we find the following classification:

Compatible	Incompatible
V_1, V_2, V_3, V_4, V_5, V_6,V_7, V_8, V_9, V_{10} V_{11}, V_{12}, V_{13}, V_{14} ,V_{15}	V_{16}

Note that the same fault removal can turn several incompatible relations into compatible relations; also, when we change a statement in a loop, our invariant relations generator may have to use different code patterns to generate invariant relations. Relation V_{16} refers to variables l and m, hence, these are the variables we may modify. We generate the condition on variables l and m under which modification of these variables does not affect compatible invariant relations, and find the following:

$$((l = 0 \wedge l' = 0) \cup (l \leq l' \wedge (l \geq 0 \cup l + l' \geq 0)))$$

Looking at the statement that updates variable l, we find that it meets (the second clause of) this condition as it is; hence, if we do not change it, we are assured not to affect any compatible invariant relation. We focus on variable m, and we suggest to change statement $(m = m - l)$ into $(m = m + l)$. This yields the following program:

```
1  p3: while (abs(r-p)>upsilon){
2      t=t+1;
3      n=n+x;
4      m=m+l;
5      l=(1+b)*l;
6      k=k+1000;
7      y=n+k;
8      w=w+z;
9      z=(1+a)*z;
10     v=w+k;
11     r=(v-y)/y;
12     u=(m-n)/n;
13     d=r-u;
14 }
```

Listing 7.5 Uninitialized loop: 3 faults removed

We compute the invariant relations of this program and find:

- $V_1 = \{(s, s')|x' = x\}$.
- $V_2 = \{(s, s')|t \leq t'\}$.
- $V_3 = \{(s, s')|k \leq k'\}$.
- $V_4 = \{(s, s')||l| \leq |l'|\}$.
- $V_5 = \{(s, s')|z \leq z'\}$.
- $V_6 = \{(s, s')|k - 1000t = k' - 1000t'\}$.
- $V_7 = \{(s, s')|l \leq l'\}$.
- $V_8 = \{(s, s')|1000l - (1 + b)k = 1000l' - (1 + b)k'\}$.
- $V_9 = \{(s, s')|(1 + b)z - (1 + a)l = (1 + b)z' - (1 + a)l'\}$.
- $V_{10} = \{(s, s')|1000z - (1 + a)k = 1000z' - (1 + a)k'\}$.
- $V_{11} = \{(s, s')|1000n - kx = 1000n' - k'x'\}$.
- $V_{12} = \{(s, s')|(1 + b)n - xl = (1 + b)n' - x'l'\}$.
- $V_{13} = \{(s, s')|z(1 + a)^{-t} = z'(1 + a)^{-t'}\}$.
- $V_{14} = \{(s, s')|z(1 + a)^{-k/1000} = z'(1 + a)^{-k'/1000}\}$.
- $V_{15} = \{(s, s')|w - \frac{z}{a} = w - \frac{z}{a}\}$.
- $V_{16} = \{(s, s')|m - \frac{l}{b} = m - \frac{l}{b}\}$.

When we check these invariant relations against specification R for compatibility, we find that they are all compatible.

Compatible	Incompatible
V_1, V_2, V_3, V_4, V_5, V_6,V_7, V_8, V_9, V_{10} V_{11}, V_{12}, V_{13}, V_{14} ,V_{15}, V_{16}	

This does not mean that program p3 is correct. All it means is that program $p3$ is more-correct than programs $p2$ and $p1$; the absence of incompatible relations is not sufficient to ensure correctness; all it means is that we did not prove the program incorrect). We do find that program $p3$ is correct with respect to R, by virtue of the proposition of sufficient correctness, because we find that relation V, the intersection of all the invariant relations of p3, satisfies the sufficiency condition:

$$V\overline{T} \cap RL \cap (R \cup V \cap \widehat{\overline{T}}) = R$$

7.2.4 Initialized While Loops

As a second illustrative example, we consider the following program that purports to compute Fibonacci numbers; its space is defined by the following declarations:

```
1 const int cN =     ;
2 int i, j, fb, nc, np;
```

The source code of the loop w is:

```
1 g: while (j!=cN){
2   j=j+i;
3   nc=fb;
4   i=i+1;
5   fb=np+nc;
6   np=nc;
7   j=j-i;
8   }
```

Listing 7.6 Initialized loop: program with modifications

Deployment of the invariant relations generator produces the following invriant relations (where F is the Fibonacci function):

- $V_1 = \{(s, s') | i \leq i'\}$.
- $V_2 = \{(s, s') | j \geq j'\}$.
- $V_3 = \{(s, s') | i + j = i' + j'\}$.
- $V_4 = \{(s, s') | np' = fb \times F(i' - i) + np \times F(i' - i - 1)\}$.

- $V_5 = \{(s, s') | fb' = fb \times F(i' - i + 1) + np \times F(i' - i)\}$.

We consider the following specification:

$$R = \{(s, s') | j > cN \wedge fb' = F(j + 2 - cN) \wedge$$
$$nc' = F(j + 1 - cN) \wedge np' = F(j + 1 - cN) \wedge i' = i + j \wedge j' = cN\}$$

In the table below, we show how the invariant relations listed above are classified between compatible relations and incompatible relations with respect to specification R.

Compatible	Incompatible
V_1, V_2, V_3	V_4,V_5

Because we have found invariant relations that are incompatible with specification R, we infer that this loop is incorrect with respect to R; hence, there is a fault.

A theorem by H.D. Mills[114] provides a condition under which a function W can be computed by an uninitialized while loop:

$$(LW \cap I)W = (LW \cap I).$$

In [115], Mili et al. generalize this result to give a condition on a relation R to admit an uninitialized while loop as a correct program (i.e. a condition under which specification R can be refined by a function W that satisfies Mills condition, above):

$$RL \subseteq R(R \cap I)L.$$

Interestingly, we find that our relation R given above does not satisfy this condition. Indeed, we find:

$$RL = (s, s') | j > cN.$$

On the other hand, we find

$$R \cap I = \{(s, s') | s' = s \wedge j > cN \wedge fb' = F(j + 2 - cN) \wedge$$
$$nc' = F(j + 1 - cN) \wedge np' = F(j + 1 - cN) \wedge i' = i + j \wedge j' = cN\}$$

This relation is empty, since it is a subset of

$$(s, s') | j > cN j = cN,$$

which is itself empty. Hence R(RI)L is empty, and the condition

$$RL \subseteq R(R \cap I)L.$$

does not hold. So that specification R cannot be satisfied by an uninitialized while loop; in other words, even though w is incorrect with respect to R (as shown by the existence of incompatible relations), there is nothing we can do to w to correct it; instead, any correction must be outside the loop, say in the initialization. In light of this example, we may want to refine the algorithm discussed above (in Section 7.2.2) by adding a step where we check the condition $RL \subseteq R(R \cap I)L.$ before attempting to remedy the loop; indeed, if this condition is not satisfied, then no loop can satisfy specification R, hence, the focus of fault removal ought to divert away from the loop (e.g. towards its initialization). To get some guidance for how to initialize this loop, we compute its competence domain with respect to R. To this effect, we calculate the function of w from its invariant relations using a formula provided by [5]; this calculation is done automatically, using the computer algebra program Mathematica (Wolfram Research). We find:

$$W = \{(s, s') | j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge$$
$$np' = np \times F(j - cN - 1) + fb \times F(j - cN)$$
$$\wedge nc' = np' \wedge fb' = np \times F(j - cN) + fb \times F(j - cN + 1)\}.$$

The competence domain of w can be computed in ©Mathematica by simplifying the following logical expression (where each relation is represented by its characteristic predicate):

$$\exists s' : R(s, s') \wedge W(s, s').$$

We find:

$$CD = (s, s')|j > cN \wedge ((fb = 1 \wedge np = 1) \cap (fb \times (1 + 5) + 2 \times np = 3 + 5)).$$

Because variables fb and np are of type integer, this competence domain can be written simply as:

$$CD = (s, s')|j > cN \wedge fb = 1 \wedge np = 1.$$

In order for w to behave according to specification R, variables fb and np have to be 1; this suggests that the required initialization is

$$fb = 1; np = 1;$$

We find (as shown below) that these initializations ensure that the program is now correct with respect to R. Interestingly, we also find that doing only one of these two initializations produces more-correct (albeit not absolutely correct) programs, as we show below. Let $p1$ be the program obtained from w by adding the initialization $fb = 1$; We find

$$\begin{aligned} P1 = \{(s, s')|j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge \\ np' = np \times F(j - cN - 1) + F(j - cN) \\ \wedge fb' = np \times F(j - cN) + F(j - cN + 1) \wedge nc' = np'\} \end{aligned}$$

From which we infer the competence domain of p1 as:

$$CD1 = \{(s, s')|j > cN \wedge np = 1\}$$

Likewise, we compute the function then competence domain of $p2$, obtained by adding $np = 1;$ to the while loop, and we find:

$$P2 = \{(s, s')|j \geq cN \wedge i' = i + j - cN \wedge$$
$$j' = cN \wedge np' = F(j - cN - 1) + fb \times F(j - cN)$$
$$\wedge fb' = F(j - cN) + fb \times F(j - cN + 1) \wedge nc' = np'\},$$

Whence,,

$$CD2 = \{(s, s')|j > cN \wedge fb = 1\}.$$

Finally, we compute the function and competence domain of the program $p3$ obtained from w by adding the two initializations,$fb = 1; np = 1;$, and we find

$$P3 = \{(s, s')|j \geq cN \wedge i' = i + j - cN \wedge$$
$$j' = cN \wedge np' = F(j - cN + 1) \wedge fb' = F(j - cN + 2) \wedge nc' = np'\},$$

Whence,

$$CD3 = \{(s, s')|j > cN\}.$$

Hence to summarize:

$$CD = (s, s')|j > cN \wedge fb = 1np = 1$$
$$CD1 = (s, s')|j > cN \wedge np = 1$$
$$CD2 = (s, s')|j > cN \wedge fb = 1$$
$$CD3 = (s, s')|j > cN$$
$$RL = (s, s')|j > cN$$

This is reflected in the following figure:

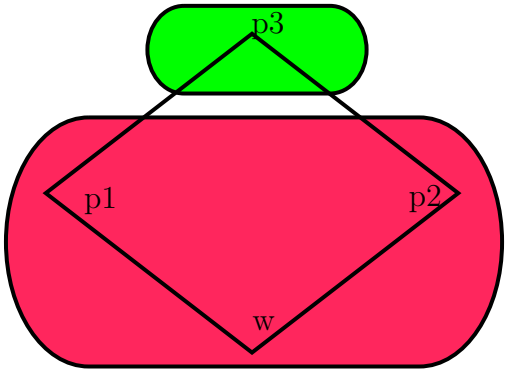


Figure 7.1 Ranking candidates by relative correctness.

7.3 Mutation Based Program Repair

7.3.1 Illustration 1

To illustrate the difference between absolute correctness and relative correctness, we consider the same program as Section 7.2.3, and we resolve to remove its faults not by static analysis, as we did there, but by testing for relative correctness after each fault removal. To this effect, we proceed iteratively as follows, starting from the original program:

1. Using muJava [116], we generate mutants of the program, and submit each mutant to three tests:

   - A test for absolute correctness, using oracle $\Omega(s, s')$ derived from specification R.
   - A test for relative correctness, using oracle $\omega(s, s')$ derived from $\Omega(s, s')$.
   - A test for strict relative correctness, which in addition to relative correctness also ensures that there is at least one state on which the mutant satisfies Ω whereas the base program fails it.

2. We select those mutants which prove to be strictly more-correct than the base program, make each one of them a base program on which we apply recursively the same procedure, starting from step 1 above.

We invoke muJava with the option of mutating statements and conditions and we test every mutant for relative correctness, strict relative correctness and absolute correctness using randomly generated test data of size 1000. Every invokation of muJava generates exactly 64 mutants, which we label by indices 1 through 64; hence, for example $m4.53.8$ is mutant 8 of mutant 53 of mutant 4 of the original program. The outcome of this experiment is illustrated in the graph in Figure 7.2. The arcs represent relative correctness relationships; at the bottom of this graph is the original program, and at the top is the corrected version of the program. Note that the test for absolute correctness kept coming empty-handed every time except whenever muJava produced the correct program P'. i.e. as many times as there are arcs pointing to P' (six times). The test for relative correctness returned *true* for every arc in Figure 7.2 i.e. 25 times; it enabled us to remove faults one at a time, and to follow the path of increased relative correctness in a stepwise manner. Note also that many mutations prove to be perfectly commutative, i.e. they can be applied in an arbitrary order; such is the case for 4, 8 and 53. Note further that, if we assume for the sake of argument that our test is exhaustive, then the number of arcs emerging from each program represents the number of faults in that program. For example, program P has four faults even though it is three fault removals away from being correct (P, $m4$, $m4.8$, $P' = m4.8.53$); we say that P has a *fault density* of 4 and a *fault depth* of 3.

7.3.2 Illustration 2

Experimental Setup To illustrate the distinction between program repair by absolute correctness and by relative correctness, we consider a program that performs the Fermat decomposition of a natural number, in which we introduce three changes (that we find, subsequently, to be three faults). The space of a Fermat decomposition is defined by three natural variables, n, x and y and the specification is defined as

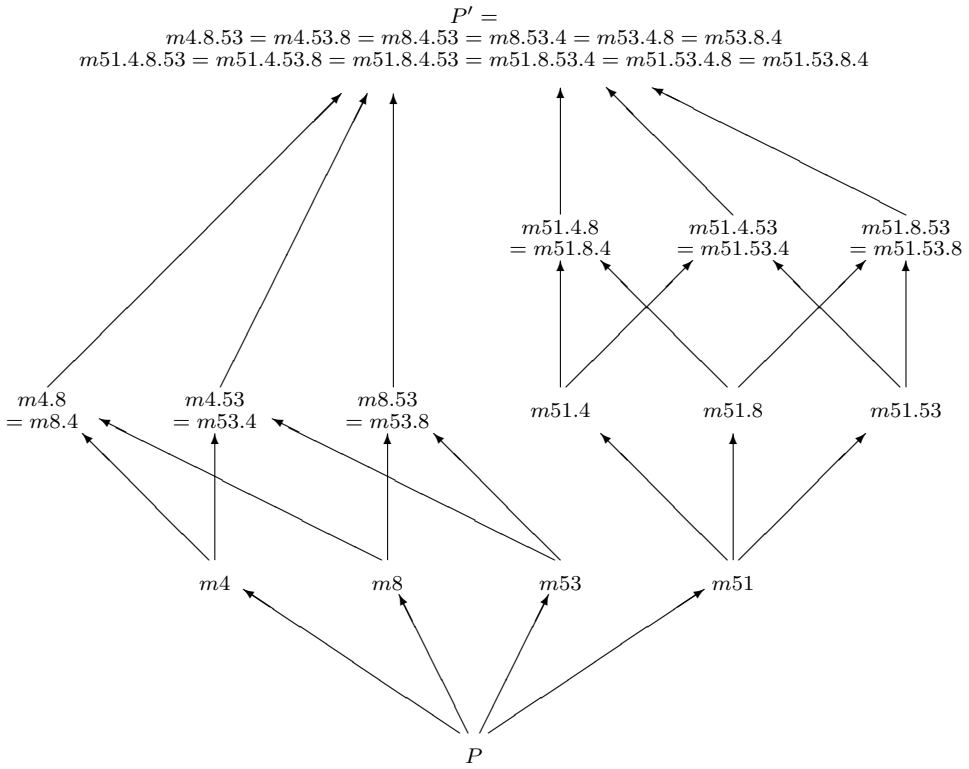


Figure 7.2 Program repair by stepwise correctness enhancement.

follows:

$$R = \{(s, s') | ((n \textbf{ mod } 2 = 1) \vee (n \textbf{ mod } 4 = 0)) \wedge n = x'^2 - y'^2\}.$$

A correct Fermat program (which we call p') is:

```
1 void fermatFactorization() {
2   int n, x, y; // input/output variables
3   int r; // work variable
4   x = 0; r = 0;
5   while (r < n) {r = r + 2 * x + 1; x = x + 1; }
6   while (r > n) {int rsave; y = 0; rsave = r;
7   while (r > n) {r = r - 2 * y - 1; y = y + 1; }
8   if (r < n) {r = rsave + 2 * x + 1; x = x + 1; }}}
```

Listing 7.7 Fermat factorization

The three changes we introduce in this program are shown below; we do not call them faults yet because we do not know whether they meet our definition of a fault (Chapter 6, Definition 8). We call this program p:

```
void basep(int& n, int& x, int& y) {
  int r; x = 0; r = 0;
  while (r < n) {r = r + 2 * x - 1; /*change in r*/ x =x+1;}
  while (r > n) {int rsave; rsave = r; y = 0;
  while (r > n) {r =r-2*y+1; /*change in r*/ y =y+1;}
  if (r < n) {r =rsave+2*x-1; /*change in r*/ x =x+1;}}}
```

Listing 7.8 Fermat factorization with 3 modifications

To repair this program, we apply muJava to generate mutants using the single mutation option with the AORB operator (Arithmetic Operator Replacement, Binary). Whenever a set of mutants are generated, we subject them to three tests:

- A test for absolute correctness, using the oracle $\Omega(s, s')$.
- A test for relative correctness, using the oracle $\omega(s, s')$.
- A test for strict relative correctness, which in addition to relative correctness checks the presence of at least one state in the competence domain of the mutant that is not in the competence domain of the base program.

The main iteration of the test driver is given below.

```
int main (){
  for (int mutant =1; mutant<= nbmutants; mutant++)
  {// test mutant vs spec. R for abs and rel correctness
    bool cumulabs=true; bool cumulrel=true; bool cumulstrict=
   false;
    while (moretestdata){
    int n,x,y; int initn,initx,inity; //initial, final states
    bool abscor, relcor, strict;
    initn=td[tdi]; tdi++; // getting test data
    n=initn; x=initx; y=inity;   // saving initial state
    callmutant(mutant, n, x, y);
    abscor = absoracle(initn, initx, inity, n, x, y);
    cumulabs = cumulabs && abscor;
    n=initn; x=initx; y=inity;  //  re-initializing
    basep(n, x, y);
    relcor = ! absoracle(initn, initx, inity, n, x, y) || abscor;
    strict = ! absoracle(initn, initx, inity, n, x, y) && abscor;
    cumulrel = cumulrel && relcor;
    cumulstrict = cumulstrict || strict;
  }}}
bool R (int initn, int initx, int inity, int n, int x, int y){
  return ((initn%2==1) || (initn%4==0)) && (initn==x*x-y*y);
```

```
22   }
23 bool domR (int initn, int initx, int inity){
24   return ((initn%2==1) || (initn%4==0));
25   }
26 bool absoracle (int initn,int initx,int inity,int n,int x,int y){
27   return  (! (domR(initn, initx, inity)) || R(initn, initx, inity
       , n, x, y));
28   }
```

Listing 7.9 Test driver

The main program includes two nested loops; the outer loop iterates over mutants and the inner loop iterates over test data. For each mutant and test datum, we execute the mutant and the base program on the test datum and test the mutant for absolute correctness (`abscor`), relative correctness (`relcor`) and strict relative correctness (`strict`); these boolean results are cumulated for each mutant in variables `cumulabs`, `cumulrel` and `cumulstrict`, and are used to diagnose the mutant. As for the Boolean functions `R`, `domR` and `absoracle`, they stem readily from the definition of R and from the oracle definitions given in Section 6.7.

Experimental Results Starting with program p, we apply muJava repeatedly to generate mutants, taking mutants which are found to be strictly more-correct as base programs and repeating until we generate a correct program. This proceeds as follows:

- When muJava is executed on program p, it produces 48 mutants, of which two ($m12$ and $m44$) are found to be strictly more-correct than p, and none are found to be absolutely correct with respect to R; we pursue the analysis of $m12$ and $m44$.

- *Analysis of* $m44$. When we apply muJava to $m44$, we find 48 mutants, none of them prove to be absolutely correct, nor relatively correct, nor strictly relatively correct.

- *Analysis of* $m12$. We find by inspection that $m12$ reverses one of the modifications we had applied to p' to find p; since $m12$ is strictly more-correct than p with respect to R, we conclude that the feature in question was in fact a fault in p with respect to R. When we apply muJava to $m12$, it generates 48 mutants, three of which prove to be strictly more-correct than $m12$: we

name them $m12.19$, $m12.20$ and $m12.28$. All the other mutants are found to be neither absolutely correct with respect to R, nor more correct than $m12$.

- *Analysis of* $m12.19$. When we apply muJava to $m12.19$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.19$, but one ($m12.19.24$) proves to be identical to $m12.20$ and is more-correct than (but not strictly more-correct than, hence, as correct as) $m12.19$.
- *Analysis of* $m12.20$. When we apply muJava to $m12.20$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.20$, but one ($m12.20.24$) proves to be identical to $m12.19$ and is more-correct than (but not strictly more-correct than, hence, as correct as) $m12.20$.
- *Analysis of* $m12.28$. We find by inspection that $m12.28$ reverses a second modification we had applied to p' to obtain p; since $m12.28$ is strictly more-correct than $m12$, this feature is a fault in $m12$; whether it is a fault in p we have not checked, as we have not compared $m12.28$ and p for relative correctness. When we apply muJava to $m12.28$, we find a single mutant, namely $m12.28.44$ that is absolutely correct with respect to R, more-correct than $m12.28$ with respect to R, and strictly more-correct than $m12.28$ with respect to R.
  - *Analysis of* $m12.28.44$. We find by inspection that $m12.28.44$ is nothing but the original Fermat decomposition program we have started out with: p'.

The results of this analysis are represented in Figure 7.3.

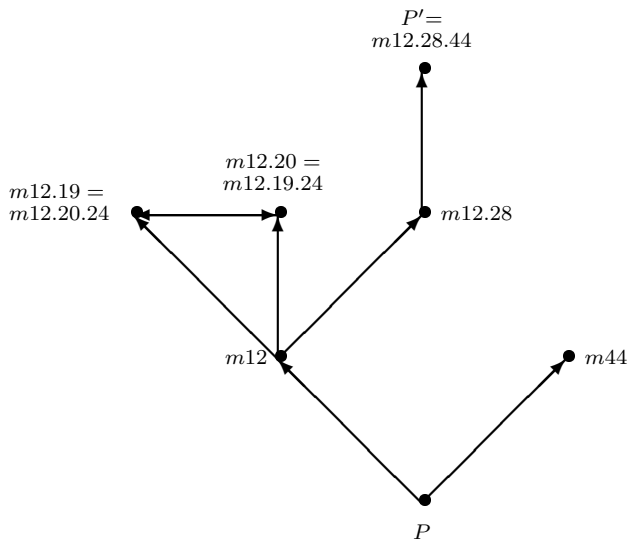


Figure 7.3 Relative correctness-based repair: stepwise fault removal.

7.4 Programming Without Refinement

The paradigm of program derivation by relative correctness is shown in Figure 7.4; in this section, we illustrate this paradigm on a simple example, where we show in turn, how to conduct the transformation process until we find a correct program or (if stakes vs cost considerations warrant) until we reach a sufficiently reliable program.

7.4.1 Producing A Correct Program

We let space S be defined by three natural variables n, x and y, and we let specification R be the following relation on S (borrowed from [117]):

$$R = \{(s, s') | n = x'^2 - y'^2 \wedge 0 \leq y' \leq x'\}.$$

Candidate programs must generate x' and y' (if possible) for a given n. The domain of R is the set of states s such that $n(s)$ is either odd or a multiple of 4; indeed, a multiple of 2 whose half is odd cannot be written as $n = x'^2 - y'^2$, since this equation is equivalent to $n = (x' - y') \times (x' + y')$, and these two factors ($(x' - y')$ and $(x' + y')$) have the same parity, since their difference ($x' + y' - x' + y' = 2 \times y'$)

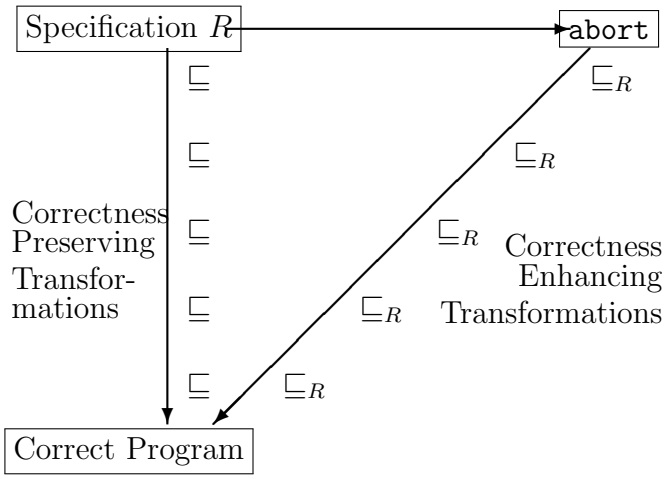


Figure 7.4 Alternative program derivation paradigms.

is even. Hence we write:

$$RL = \{(s, s') | n \mod 2 = 1 \vee n \mod 4 = 0\}.$$

Starting from the initial program P_0 =`abort`, we resolve to let the next program P_1 be the program that finds this factorization for $y' = 0$:

```
1 void p1(){
2   nat n, x, y;    //  input/output variable;
3   nat r;          //  work variable
4   x=0; y=0; r=0;
5   while (r<n) {r=r+2*x+1; x=x+1;}
6 }
```

Listing 7.10 Fermat factorization, computing perfect squares

We compute the function of this program by applying the semantic rules given in Section 2.4, and we find:

$$P_1 = \{(s, s') | n' = n \wedge y' = 0 \wedge x' = \lceil \sqrt{n} \rceil\}.$$

Whence, we compute the competence domain of P_1 with respect to R:

$(R \cap P_1)L$

$=$ {substitution, simplification}

$\{(s, s')|n = x'^2 \wedge n' = n \wedge y' = 0\} \circ L$

$=$ {taking the domain}

$\{(s, s')|\exists x'' : n = x''^2\}$.

In other words, P_1 satisfies specification R, whenever n is a perfect square.

We now consider the case where r exceeds n by a perfect square, making it possible to fill the difference with y^2; this yields the following program:

```
1 void p2(){
2   nat n, x, y;    //  input/output variables
3   nat r;          //  work variable
4   x=0; r=0;
5   while (r<n) {r=r+2*x+1; x=x+1;}
6   if (r>n) {y=0; while (r>n) {r=r-2*y-1; y=y+1;}}
7   if (r!=n) {abort;}
8 }
```

Listing 7.11 Fermat factorization, factoring perfect squares and more

This program preserves n, places in x the ceiling of the square root of n, and places in y the integer square root of the difference between n and x'^2, and fails if this square root is not an integer. We write its function as follows:

$$P_2 = \{(s, s')|n' = n \wedge x' = \lceil\sqrt{n}\rceil \wedge y'^2 = x'^2 - n \wedge y' \geq 0\}.$$

We compute the competence domain of P_2 with respect to R:

$(R \cap P_2) \circ L$

$=$ {Substitutions}

$\{(s, s')|n = x'^2 - y'^2 \wedge 0 \leq y' \leq x' \wedge n' = n \wedge x' = \lceil\sqrt{n}\rceil \wedge y'^2 = x'^2 - n$

$\wedge y' \geq 0\} \circ L$

$=$ {Simplifications}

$\{(s, s')|n' = n \wedge x' = \lceil\sqrt{n}\rceil \wedge y'^2 = x'^2 - n \wedge y' \geq 0\} \circ L$

$=$ {Computing the domain}

$\{(s, s')|\exists n'', x'', y'' : n'' = n \wedge x'' = \lceil\sqrt{n}\rceil \wedge y''^2 = x''^2 - n \wedge y'' \geq 0\}$

$=$ {Simplifications}

$\{(s, s') | \exists y'' : y''^2 = \lceil \sqrt{n} \rceil^2 - n\}$.

In other words, the competence domain of P_2 is the set of states s such that $n(s)$ satisfies the following property: the difference between $n(s)$ and the square of the ceiling of the square root of $n(s)$ is a perfect square. For example, a state s such that $n(s) = 91$ is in the competence domain of P_2, since $\lceil \sqrt{91} \rceil^2 - 91 = 10^2 - 91 = 9$, which is a perfect square. The competence domain of P_2 is clearly a superset of the competence domain of P_1, hence, the transition from P_1 to P_2 is valid.

The next program is derived from P_2 by resolving that if the ceiling of the integer square root of n does not exceed n by a square root, then we try the next perfect square (whose root we assign to x) and we check whether the difference between that perfect square and n is now a perfect square; we know that this process converges, for any state s for which $n(s)$ is odd or a multiple of 4. This yields the following program:

```
1  void p3(){
2    nat n, x, y;    //  input/output variables
3    nat r;          //  work variable
4    x=0; r=0;
5    while (r<n) {r=r+2*x+1; x=x+1;}
6    while (r>n) {
7      int rsave; y=0; rsave=r;
8    while (r>n) {r=r-2*y-1; y=y+1;}
9    if (r<n) {r=rsave+2*x+1; x=x+1;}
10   }
11 }
```

Listing 7.12 Final Fermat factorization

This program preserves n, places in x the smallest number whose square exceeds n by a perfect square and places in y the square root of the difference between n and x^2. If we let $\mu(n)$ be the smallest number whose square exceeds n by a perfect square, we write the function of P_3 as follows:

$$P_3 = \{(s, s') | n' = n \wedge x' = \mu(n) \wedge y' = \sqrt{\mu(n)^2 - n}\}.$$

We compute the competence domain of P with respect to R:

$(R \cap P_3) \circ L$

$=$ {Substitutions}

$\{(s, s') | n = x'^2 - y'^2 \wedge 0 \leq y' \leq x' \wedge n' = n \wedge x' = \mu(n) \wedge y' = \sqrt{\mu(n)^2 - n}\} \circ L$

$=$ {Simplifications}

$\{(s, s') | n = x'^2 - y'^2 \wedge n' = n \wedge x' = \mu(n)\} \circ L$

$=$ {Computing the domain}

$\{(s, s') | \exists n'', x'', y'' : n = x''^2 - y''^2 \wedge n'' = n \wedge x'' = \mu(n)\}$

$=$ {Simplifications}

$\{(s, s') | \exists x'', y'' : n = x''^2 - y''^2\}$

$=$ {By inspection}

RL.

Hence P_3 is correct with respect to R (by proposition 12) hence, it is more-correct than P_2 with respect to R. Hence we do have:

$$P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3.$$

Furthermore, we find that P_3 is correct with respect to R; this concludes the derivation.

7.4.2 Producing A Reliable Program

We interpret the reliability of a program as the probability of a successful execution of the program on some initial state selected at random from the domain of R according to some probability distribution θ. Given a probability distribution θ on $dom(R)$, the reliability of a candidate program P is then the probability that an element of $dom(R)$ selected according to the probability distribution θ falls in the competence domain of P with respect to R. Clearly, the larger the competence domain, the higher the probability. Hence the sequence of programs that we generate in the

proposed process feature higher and higher reliability. So that if we are supposed to derive a program under a reliability requirement, we can terminate the stepwise transformation process as soon as we obtain a program whose estimated reliability matches or exceeds the specified threshold. So far this is a theoretical proposition, but an intriguing possibility nevertheless. The sample program developed in the previous subsection may be used to illustrate this idea, though it does not show a uniform reliability growth. For the sake of argument, we suppose that n ranges between 1 and 10000, and we estimate the reliability of each of the programs generated in the transformation process.

- P_0: The reliability of P_0 is zero, of course, since it never runs successfully.

- P_1: If n takes values between 1 and 10000, then the domain of R has 7500 elements (since 1 out of four is excluded: even numbers whose half is odd are not decomposable); out of these 7500 elements, only 100 are perfect squares (1^2 to 100^2). Hence the reliability of P_1 under a uniform probability distribution is $\frac{100}{7500} = 0.01333$.

- P_2: The competence domain of P_2 includes all the elements n that can be written as: $n = \lceil\sqrt{n}\rceil^2 - y^2$ for some non-negative value y. To count the number of such elements, we consider all possible values of x (between 1 and 100) and all possible values of y such that $(x-1)^2 < x^2 - y^2 \leq x^2$. By inverting the inequalities and adding x^2 to all sides, we obtain:

  $$0 \leq y^2 < 2x - 1.$$

  Hence the number of elements in the competence domain of P_2 can be written as

  $$100 + \sum_{x=1}^{100} \sqrt{2x-1}.$$

  We find this quantity to be equal to 996, which yields a probability of $\frac{996}{7500} = 0.1328$.

- P_3: Because the competence domain of P_3 is all of $dom(R)$, the reliability of this program is 1.0.

We obtain the following table.

Program	Reliability
P_0	0.0000
P_1	0.0133
P_2	0.1328
P_3	1.0000

7.5 Software Evolution

In [29], we build upon our results of [28] by showing that relative correctness can be used not only for program development from scratch, but also for various forms of program evolution. In the following, we use relative correctness to model several aspects of software evolution, including: merging programs, the upgrade of a program with a new feature; the removal of a fault from a program (corrective maintenance); and the transformation of a program to satisfy a new specification (adaptive maintenance).

7.5.1 Program Merger

We consider a specification R and two candidate programs P_1 and P_2 (i.e. programs that are written to satisfy R –they may or may not satisfy it in fact), each of which fulfills the requirements of R to some limited extent, but not necessarily to the full extent. We are interested to merge programs P_1 and P_2 into a program that fulfills the requirements of R to the extent that P_1 fulfills them, and to the extent that P_2 fulfills them. We submit the following definition.

Definition 11 *Given a specification R and two candidate programs P_1 and P_2, a* merger *of P_1 and P_2 with respect to R is any program P' that is more-correct than P_1 and more-correct than P_2 with respect to R.*

We mandate that a merger program be merely more-correct than programs P_1 and P_2, rather than to refine them, for the following reasons:

- *Refinement is Unnecessary.* When we resolve to refine a program, we commit to refine all its functional attributes, those that are mandated by the specification as well as those that stem from design decisions. But we have no reason to preserve design decisions of P_1 and P_2 that do not advance the cause of relative correctness.

- *Relative Correctness is Sufficient.* If program P' is more-correct than P_1 and P_2 with respect to specification R, then it delivers all the specification-mandated behavior of P_1 and all the specification-mandated behavior of P_2.

- *Refinement may be Impossible.* Not only is it unnecessary to refine the design-related information of P_1 and P_2, it may actually be impossible: whereas the specification-mandated information of P_1 and P_2 is bounded by R, hence, (according to Section 2.3) can be combined by the least upper bound operation, the design-related information of P_1 and P_2 may be incompatible, hence, cannot be combined.

We consider the space S defined by three variables x, y and z of type integer, and we let R be the following specification: $R = \{(s, s')|x' = x + y \wedge z' \geq z + 2\}$. Let p_1 and p_2 be the following candidate programs for specification R:

```
p1:  {z=z+2; while (y!=0) {y=y-1; x=x+1;}}
p2:  {z=z+3; while (y!=0) {y=y+1; x=x-1;}}
```

The functions of these programs are, respectively:

$P_1 = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 2\}$

$P_2 = \{(s, s')|y \leq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 3\}$.

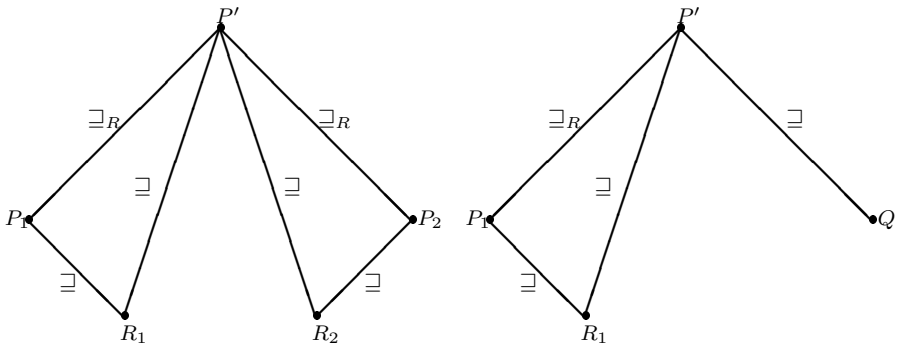
Indeed, the first program terminates only for initial y greater than or equal to zero, and when it terminates, the final value of x contains $x + y$, the final value of y is zero, and z is incremented by 2. As for the second program, it terminates only for non-positive y, and when it does terminate, the final value of y is zero, z is increased by 3 and x contains $x+y$. So that each program does some of what R asks, but neither is correct. A merger of these two programs is any program P' that is more-correct than P_1 and more-correct than P_2 with respect to R. We omit the systematic derivation of the merger of two programs and content ourselves with presenting a candidate program then showing that it satisfies the definition of a merger. We propose:

```
p':  {z=z+4;
      if (y>0) {while (y!=0) {y=y-1; x=x+1;}}
        else {while (y!=0) {y=y+1; x=x-1;}}}
```

As far as x and y are concerned, this program imitates the behavior of P_1 for non-negative values of y, and the behavior of P_2 for non-positive values of y; as far as z is concerned, this program overrides the behavior of both P_1 and P_2 and increments z by 4. We argue that this program is more-correct than P_1 and more-correct than P_2 with respect to R. The function of this program is:

$P' = \{(s, s') | x' = x + y \wedge y' = 0 \wedge z' = z + 4\}$.

Space restrictions preclude us from showing details, but it is easy to verify that the competence domain of P' ($(R \cap P)L$) is equal to L, hence, P' is more-correct than P_1 and P_2. Note that while we found a program that is more-correct than P_1 and P_2, we could not find a program that refines P_1 and P_2. Indeed we can easily check that P_1 and P_2 do not sarisfy the consistency condition, hence, they admit no joint refinement. Indeed, no program can simultaneously increase z by 2 (to refine P_1) and by 3 (to refine P_2). This discrepancy between what P_1 does and what P_2 does precludes P_1 and P_2 from having a joint refinement, but does not preclude them from having a program P' that is more-correct than them. The reason is: the statements {z=z+2} (in P_1) and {z=z+3} (in P_2) are not mandated by the specification (which only requires $\{z' \geq z + 2\}$) but stem instead from arbitrary design decisions; hence, both can be overridden by the merger program P'. The difference between refinement and relative correctness is that the former attempts to refine all the behavior of a program, regardless of its source, whereas the latter only refines the behavior that is mandated by the specification. As we see in this simple example, refining all the behavior of P_1 and all the behavior of P_2 is not only unnecessary, it is actually impossible. See Figure 7.5 (a), where R_1 and R_2 represent the specification-mandated behavior of P_1 and P_2 (we have explicit formulae for these).



(a)
Merger of P_1 and P_2

(b)
Upgrading P_1 with Feature Q

Figure 7.5 Merger and upgrade.

7.5.2 Program Upgrade

We are given a specification R and a candidate program P, and we are interested to augment program P with a new feature that is specified by some relation Q. Typically, P may be a large, complex, comprehensive application that delivers a wide range of services, and Q is a punctual additional function or service that we want to incorporate into P (for example, P is a sprawling corporate data processing application, and Q specifies an additional report to be delivered, or an additional output screen, or an additional statistic on corporate transactions, etc). In transforming P into P', we have every expectation that P' refines Q, because Q is a fairly simple requirement and because it is the main goal of the operation. But we have no expectation that P' refine P, because the implementation of Q may require that some of the behavior of P be altered. Nor do we expect that P' refines R, because in fact we are not even sure P refines R (P is typically incorrect, i.e. it fails to correctly deliver all the required services in all circumstances). While we do not expect P' to refine P nor R, we most certainly expect P' to be more-correct than P with respect to R; in other words, we do not want that in the process of adding feature Q to P, we degrade the correctness of P with respect to R.

Definition 12 *Given a specification R and a candidate program P, and given a feature Q that we want to add to P, an* upgrade *P' of P with feature Q is any program that refines Q and is more-correct than P with respect to R.*

Given a specification R on space S defined by integer variables x and y, $R = \{(s, s')|x' = x + y\}$ and given the following candidate program,

```
p1:  {x=x+10; while (y!=10) {y=y-1; x=x+1;}}
```

we consider the problem of upgrading program P_1 with feature Q defined by: $Q = \{(s, s')|y > 0 \wedge y' = 0\}$. The function of program `p1` is:

$P_1 = \{(s, s')|y \geq 10 \wedge x' = x + y \wedge y' = 10\}$.

Note that P_1 and Q do not satisfy the consistency condition, since P_1 sets y to 10 while Q mandates that we set it to 0 (for positive values of y). Therefore it is impossible to fulfill requirement Q without altering the behavior of P_1. Fortunately, the feature of P_1 that precludes us from refining Q, namely the clause $y' = 10$, is not a specification-mandated requirement, but stems instead from the specific design of P_1. Hence while it is impossible for the upgrade program P' to refine P, it is not impossible for P' to be more-correct than P with respect to R. We consider the following program:

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

The function of this program is:

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

It is easy to check that P' does refine Q. On the other hand, we can easily check that the competence domain of P_1 is $\{(s, s')|y \geq 10\}$ whereas the competence domain of P' is $\{(s, s')|y \geq 0\}$. Hence P' is more-correct than P_1 with respect to R. While it is not possible to satisfy Q while preserving all the behavior of P_1, it is possible, and sufficient, to satisfy Q while enhancing the correctness of P_1; this is what P' does. See Figure 7.5 (b).

7.6 Software Maintenance

7.6.1 Corrective Maintenance

We argue in this section that corrective maintenance is nothing but an instance of program transformation by relative correctness: in fact it is merely a step in the process we have outlined for program derivation by correctness enhancement; it starts at the current program (rather than `abort`) and it ends a step later (rather than necessarily at a correct program). See Figure 7.6. As an illustration, we reconsider the program 6.1 from Chapter 6:

```
p: #include <iostream>  ...  ...  ...
void main {(char q[])
{
  int let, dig, other, i, l;
  char c;
  i=0; let=0; dig=0; other=0; l=strlen(q);
  while (i<l) {
    c = q[i];
    if ('A'<=c && 'Z'>c) let+=2;
    else  if ('a'<=c && 'z'>=c) let+=1;
    else  if ('0'<=c && '9'>=c) dig+=1;
    else  other+=1;
    i++;
    printf ("%d %d %d\n",let,dig,other);
}

```

We define the following sets: $\alpha_A = \{'A' \ldots 'Z'\}$. $\alpha_a = \{'a' \ldots 'z'\}$. $\nu = \{'0' \ldots '9'\}$. $\sigma = \{'+','-','=', \ldots '/'\}$, the set of all the ascii symbols. We let $list\langle T \rangle$ denote the set of lists of elements of type T, and we let $\#_A$, $\#_a$, $\#_\nu$ and $\#_\sigma$ be the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits, and symbols. We consider the following specification on S:

$R = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle \wedge let' = \#_a(q) + \#_A(q) \wedge dig' = \#_\nu(p) \wedge other' = \#_\sigma(q)\}$.

The competence domain of P is:

$(R \cap P)L = \{(s, s') | q \in list\langle \alpha_a \cup \nu \cup \sigma \rangle\}$.

This is different from the domain of R, which is

$RL = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle\}$,

hence, P is not correct with respect to R. If we let P' be the program obtained from P by changing {`let=+2`} into {`let=+1`}, we find:

$(R \cap P')L = \{(s, s') | q \in list\langle (\alpha_A \setminus \{'Z'\}) \cup \alpha_a \cup \nu \cup \sigma \rangle\}$.

Clearly, $(R \cap P')L \supset (R \cap P)L$. Hence statement {`let+=2`} is a fault in P with respect to specification R and the substitution of {`let+=2`} by {`let+=1`} is a fault removal in P with respect to R.

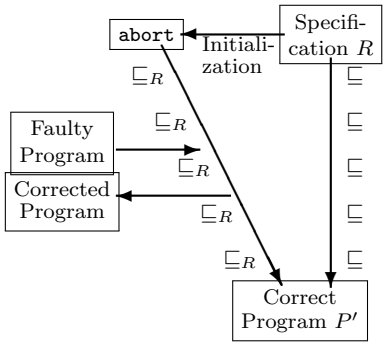


Figure 7.6 Corrective maintenance.

7.6.2 Adaptive Maintenance

Adaptive maintenance consists in taking a program P which was originally developed to satisfy some specification R and changing it to make it satisfy some new specification R'. We view this as simply trying to make P more-correct with respect to R' than it is in its current form. Clearly, one does this if one believes that P is close enough to satisfy R' that it is more economical to evolve P than to start from `abort`. Be that as it may, we argue that adaptive maintenance is again a process of making a program more-correct with respect to a given specification. See Figure 7.7.

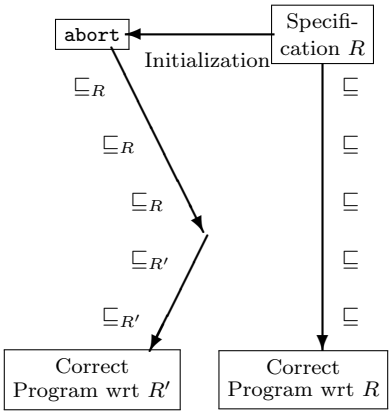


Figure 7.7 Adaptive maintenance.

7.7 Related Work

In [118], Nguyen et al. present an automated repair method based on symbolic execution, constraint solving, and program synthesis; they call their method SemFix, on the grounds that it performs program repair by means of semantic analysis. This method combines three techniques: fault isolation by means of statistical analysis of the possible suspect statements; statement-level specification inference, whereby a local specification is inferred from the global specification and the product structure; and program synthesis, whereby a corrected statement is computed from the local specification inferred in the previous step. The method is organized in such a way that program synthesis is modeled as a search problem under constraints, and possible correct statements are inspected in the order of increasing complexity. When programs are repaired by SemFix, they are tested for (absolute) correctness against some predefined test data suite; as we argue throughout this chapter, it is not sensible to test a program for absolute correctness after a repair, unless we have reason to believe that the fault we have just repaired is the last fault of the program (how do we ever know that?). By advocating to test for relative correctness, we enable the tester to focus on one fault at a time, and ensure that other faults do not interfere with our assessment of whether the fault under consideration has or has not been repaired

adequately. In [119], Weimer et al. discuss an automated program repair method that takes as input a faulty program, along with a set of positive tests (i.e. test data on which the program is known to perform correctly) and a set of negative tests (i.e. test data on which the program is known to fail) and returns a set of possible patches. The proposed method proceeds by keeping track of the execution paths that are visited by successful executions and those that are visited by unsuccessful executions, and using this information to focus the search for repairs on those statements that appear in the latter paths and not in the former paths. Mutation operators are applied to these statements and the results are tested again against the positive and negative test data to narrow the set of eligible mutants. While, to the best of our knowledge, our work is the first to apply relative correctness to program derivation, it is not the first to introduce a concept of relative correctness. In [108], Logozzo discusses a framework for ensuring that some semantic properties are preserved by program transformation in the context of software maintenance. In [109], Lahiri et al. present a technique for verifying the relative correctness of a program with respect to a previous version, where they represent specifications by means of executable assertions placed throughout the program, and they define relative correctness by means of inclusion relations between sets of successful traces and unsuccessful traces. Logozzo and Ball [110] take a similar approach to Lahiri et al. in the sense that they represent specifications by a network of executable assertions placed throughout the program, and they define relative correctness in terms of successful traces and unsuccessful traces of candidate programs. Our work differs significantly from all these works in many ways: first, we use relational specifications that address the functional properties of the program as a whole, and are not aware of intermediate assertions that are expected to hold throughout the program; second, our definition of relative correctness involves competence domains (for deterministic specifications) and the sets of states that candidate programs produce in violation of the specification

(for non-deterministic programs); third we conduct a detailed analysis of the relations between relative correctness and the property of refinement.

Also related to our work are proposals by Banach and Pempleton [120] and by Prabhu et al. [121, 122, 123] to find alternatives for strict refinement-based program derivation. In [120], Banach and Pempleton introduce the concept of *retrenchment*, which is a property linking two successive artifacts in a program derivation, that are not necessarily ordered by refinement; the authors argue that strict refinement may sometimes be inflexible, and present retrenchment as a viable substitute, that trades simplicity for strict correctness preservation, and discuss under what conditions the substitution is viable. In [121, 122, 123] Prabhu et al. propose another alternative to strict refinement, which is *approximate refinement*. Whereas strict refinement defines a partial ordering between artifacts, whereby a concrete artifact is a correctness-preserving implementation for an abstract artifact, approximate refinement defines a topological distance between artifacts, and considers that a concrete implementation is acceptable if it is close enough (by some measure of distance) to the abstract artifact. Retrenchment and Approximate refinement are both substitutes for refinement and are both used in a correctness-preserving transformation from a specification to a program; by contrast, relative correctness offers an orthogonal paradigm that seeks correctness enhancement rather than correctness preservation.

7.8 Conclusion

In this chapter, we present the applications of the concept of relative to various software engineering practices.

We show that it pervades software evolution, and is potentially more flexible, without being less effective, than refinement-based program transformations. In particular, we find that this concept can provide a formal model for a wide range of software evolution activities, including software design, corrective maintenance,

adaptive maintenance, software upgrade, program merger, etc. As a consequence, we argue that by evolving a technology of program transformation with relative correctness, we stand to enhance a wide range of software engineering activities. The study of this concept has led us to highlight the distinction between two sources of functional attributes of a program: functional properties that are dictated by the specification that we are trying to satisfy; and functional properties that stem from decisions we have taken as we design the program. This distinction is important because to make a program more-correct we must preserve or enhance the former but not the latter; and we may arbitrarily alter the latter in the process.

We also discuss how we can use the concept of relative correctness to refine the technique of program repair by mutation testing. We argue that when we remove a fault from a program, in the context of program repair, we have no reason to expect the resulting program to be correct unless we know (how do we ever?) that the fault we have just removed is the last fault of the program. Therefore we should, instead, be testing the program for relative correctness rather than absolute correctness. We have found that testing a program for relative correctness rather than absolute correctness has an impact on test data selection as well as oracle design, and have discussed practical measures to this effect. As an illustration of our thesis, we take a simple example of a faulty program, which we can repair in a stepwise manner by seeking to derive successively more-correct mutants; by contrast, the test for absolute correctness keeps excluding all the mutants except the last, and fails to recognize that some mutants, while being incorrect, are still increasingly more correct than the original. We are not offering a seamless validated solution as much as we are seeking to draw attention to some opportunities for enhancing practices.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Summary and Assessment

8.1.1 Conditions of Convergence

In this dissertation, we present a framework to integrate termination as finite iterations with absence of abort from conditions such as illegal arithmetic operations, array out of bound. We define this as the concept of convergence. For this purpose, we build upon the result of [39] to use invariant relations as a unifying model to compute or approximate the termination condition of a while loop. Invariant relations are reflexive transitive superset of the loop function. They represent the set of initial/final states separated by zero or more iterations of the loop body. The advantage of using invariant relations is two-fold:

- They can be used to model conditions of abort-freedom of interest
- They can be combined through the intersection operation to obtain tighter approximation of convergence conditions of a loop.

To the best of our knowledge, our work is the only approach to computing convergence conditions that interprets convergence in the general sense of: ending in a well-defined final state. We say that a program converges for an initial state s if and only if the program can produce a final state s' as an image of s by the program function. Whether the program fails to produce a final state because it fails to terminate or because it fails to apply an intermediate function in its finite execution sequence does not matter to us.

In keeping with this premise, our definition of convergence applies to iterative programs as much as it applies to non-iterative programs; also, as far as while loops

are concerned, our approach provides a way to map any given invariant relation of the loop onto a necessary condition of convergence. We can generate many invariant relations for the loop, each capturing a specific aspect of convergence, and obtain a convergence condition that ensures normal termination in a well-defined state; to the best of our knowledge, our approach is unique in this feature. Traditionally, the analysis of loop termination is studied separately from the analysis of its functional properties, with the latter relying on invariant assertions and the former relying on variant functions. By contrast, we use the same concept, namely invariant relations, to characterize the termination conditions and the functional properties of loops. From a conceptual viewpoint, we find it appealing to use the same approach/ means to analyze the function of the loop and the convergence condition of the loop, as the domain of a function is an integral part of the function, rather than an orthogonal attribute.

Condition of Sufficiency: In Chapter 4, we have considered several examples of programs for which we have given a necessary condition of termination, and claimed that we thought the condition was sufficient, in addition to being provably necessary. In this section, we discuss two questions, namely: why can't we derive a provably sufficient condition of termination? How can we claim that our necessary conditions are sufficient? We address these questions in turn, below.

- *Why can't we derive a sufficient condition?*

  It is hardly surprising that arbitrary (arbitrarily large) invariant relations can only generate necessary conditions, since they capture arbitrarily partial information about the loop, hence, cannot be used to make claims about a global property of the loop. Yet strictly speaking, we can formulate a sufficient condition of termination, but it is of little use in practice. A sufficient condition of termination would read as follows: Given a while loop of the form w: `while (t) {b}`, and given the invariant relation $R = (T \cap B)^*$, then $R\overline{T} \subseteq WL$.

  As we recall from Proposition 4, $R = (T \cap B)^*$ is an invariant relation of the loop, and is in fact the smallest invariant relation of the loop. In practice, it is very difficult to compute this reflexive transitive closure for arbitrary T and B. One of the main interests of invariant relations is in fact that:

– First they enable us to compute or approximate the reflexive transitive closure of $(T \cap B)$.

– Second and perhaps most importantly, they enable us to dispense with the need to compute the reflexive transitive closure of $(T \cap B)$; in particular, one of the main motivations for using invariant relations is that they enable us, with relatively little scrutiny of the loop, to answer many questions pertaining to the loops.

Hence requiring that we compute the strongest possible invariant relation to secure a sufficient condition of termination defeats the purpose of using invariant relations.

- *How can we claim sufficiency?.* We are currently developing heuristics that enable us to recognize when an invariant relation is small enough to ensure that the formula of Theorem 2 provides a sufficient condition of termination. As far as ensuring that the number of iterations is finite, we can proceed by identifying the variables that intervene in the loop condition, and generating all the invariant relations that involve these variables, and any variable that affects their value (through assignment statements). As for ensuring freedom from aborts, we also want to include any invariant relation that links the variables identified above with the variables that are involved in the abort condition (array indices, denominators of fractions, arithmetic expressions, etc). Another heuristic that we are considering is to define a set of recognizers that specialize in computing a sufficient condition of termination, by focusing on termination-related details; for example, if the loop body includes a clause of the form $x' = x + a[i]$ for some real variable x, real array a, and index (integer) variable i, then the complete recognizer would generate the invariant relation $\{(s, s') | x + \Sigma a = x' + \Sigma a'\}$ whereas the termination-related recognizer would merely record that array a has been accessed at index i. A final heuristic, invoked in [62] for the purpose of minimizing the number of invariant relations generated by our tool, involves generating just enough invariant relations to link all the statements of the loop body into a connected graph.

8.1.2 Relative Correctness

In Chapter 7 was discussed how we can use the concept of relative correctness to refine the technique of program repair by mutation testing. We argue that when we remove a fault from a program, in the context of program repair, we have no reason to expect the resulting program to be correct unless we know (how do we ever?)

that the fault we have just removed is the last fault of the program. Therefore we should, instead, be testing the program for relative correctness rather than absolute correctness. We have found that testing a program for relative correctness rather than absolute correctness has an impact on test data selection as well as oracle design, and have discussed practical measures to this effect. As an illustration of our thesis, we take a simple example of a faulty program, which we can repair in a stepwise manner by seeking to derive successively more-correct mutants; by contrast, the test for absolute correctness keeps excluding all the mutants except the last, and fails to recognize that some mutants, while being incorrect, are still increasingly more correct than the original. We are not offering a seamless validated solution as much as we are seeking to draw attention to some opportunities for enhancing the practice of software testing. Our research agenda includes further exploration of the technique proposed in this paper to assess its feasibility and effectiveness on software benchmarks, as well as techniques to streamline test data selection to enhance the precision of relative-correctness-based program repair. This dissertation is founded on the following work: In [27], we introduce relative correctness for deterministic programs, and explore the mathematical properties of this concept; in [106], we generalize the concept of relative correctness to non-deterministic programs and study its mathematical properties. In [28]. (Programming without Refinement) we argue that while we generally think of program derivation as the process correctness preserving transformations using refinement, it is possible to derive programs by correctness-enhancing transformations using relative correctness; one of the interesting advantages of relative correctness-based correctness enhancing transformations is that they capture, not only the derivation of programs from scratch, but also virtually all software maintenance activities. We can argue in fact that software evolution and maintenance is nothing but an attempt to enhance the correctness of a software product with respect to a specification. In [25], (Debugging

without Testing) we show how relative correctness can be used to define faults and fault removals, and that we can use these definitions to remove a fault from a program and prove that the fault has ben removed, all by static analysis, without testing. This work is clearly in its infancy; it includes the definition of a new concept, the premise that this concept can be used for a provably monotonic fault removal process, and some initial results that enable us to apply this concept with some automated support, and without getting involved into the minute functional details of the program and the specification. The question that arises with this type of work is, of course, whether it scales up to programs of realistic size and complexity. We argue that relative correctness scales up to the same degree as absolute correctness. The fact that it cannot be readily employed to software products of arbitrary size and complexity does not make it any less worthy of investigation, just as the same constraints do not make absolute correctness less worthy of study; it is still useful as a logical reasoning framework; and it can be applied in practice with the proper balance of formality, expressiveness, and usability, and with judicious automated support where possible. Also, we argue that in software quality assurance as in other endeavors, the law of diminishing returns advocates the use of diverse methods and tools to maximize impact; the use of relative correctness to support fault diagnosis and removal stands to play an important role as a tool in the engineers toolbox.

8.2 Future Work

8.2.1 Condition of Convergence

All the heuristics discussed in Section 8.1.1 are intended to enable us to claim sufficiency of our termination condition without having to generate all the invariant relations of the loop; we envision to organize these heuristics into a cohesive algorithm, as part of our future research plans.

On the automation side, we envisage to expand the tool to cover more data types, provide more support for $Java/C/C++$. We also plan on working on ways of scaling our implementation so that we are able to support the analysis of large scale program input.

8.2.2 Relative Correctness

This work is clearly in its infancy; We envision to continue exploring applications of relative correctness in fault removal, to enhance and integrate our tool support, and to consider other results (theorems) that enable us to streamline the verification of relative correctness. We also envision exploring automation of the methods exposed in this dissertation, wherever applicable.

One area that we are exploring concerns the projection of a program on a specification so as to find which part of the program is relevant to a specification. The concept of projection of a program on a specification comes about as a byproduct of the definition of relative correctness. The definition and implications are discussed in [124].

Projecting Programs on Specifications: Given a specification R and a program P that is written to satisfy R, we refer to P as a *candidate* program for specification R, and we refer to R as the *target* specification of program P, regardless of whether P does or does not satisfy specification R. Given a specification R and a candidate program P, the program P could well be falling short of some of the requirements of R, while at the same time exceeding (i.e. doing more than needed) on some other requirements. There is no shortage of reasons why a program may fall short of the requirements mandated by the specification, but there are also ample reasons why a program may do more than required: these include cases where excess functionality is a byproduct of normal design decisions, cases where it stems from programming language constructs, and more generally the need to bridge the gap between a non-

deterministic specification and a deterministic program. Consider for example the following relational specification on space S defined by integer variables x and y:

$$R = \{(\langle x, y\rangle, \langle x', y'\rangle) | x' = x + y\},$$

and consider the following program on the same space:

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

For non-negative values of y, this program computes the sum of x and y in x while placing 0 in y; for negative values of y, it fails to terminate. Hence its function can be written as:

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

This program does not do everything that specification R requires, since it fails to compute the sum of x and y into x for negative values of y; on the other hand, it puts 0 in y even though the specification did not ask for it (but this is a side effect of the algorithm we have chosen to satisfy the specification). Hence looking at specification R and program P, we would like to think that the functionality of P that is relevant to (mandated by) R is captured by the following relation:

$$\pi = \{(\langle x, y\rangle, \langle x', y'\rangle) | y \geq 0 \wedge x' = x + y\}.$$

Whatever else P does (e.g. it sets y to 0) is not relevant to specification R; on the other hand, whatever else the specification mandates (computing the sum of x and y into x for negative values of y), program P is not delivering. In other words, relation π fails to specify $y' = 0$ because that is not mandated by the specification, and it fails to specify the case $y < 0$ because that is not delivered by the candidate program P.

BIBLIOGRAPHY

[1] E. J. Braude and M. E. Bernstein, *Software engineering: modern approaches*. Hoboken, NJ: J. Wiley Sons, 2016.

[2] C. LeGoues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[3] L. Feinbube, P. Tröger, and A. Polze, "The landscape of software failure cause models," *arXiv preprint arXiv:1603.04335*, 2016.

[4] C. Kolias, A. Stavrou, J. Voas, I. Bojanova, and R. Kuhn, "Learning internet-of-things security "hands-on"," *IEEE Security Privacy*, vol. 14, no. 1, pp. 37–46, Jan 2016.

[5] A. Mili, S. Aharon, and C. Nadkarni, "Mathematics for reasoning about loop," *Science of Computer Programming*, pp. 989–1020, 2009.

[6] J. Carette and R. Janicki, "Computing properties of numeric iterative programs by symbolic computation," *Fundamentae Informatica*, vol. 80, no. 1-3, pp. 125–146, March 2007.

[7] A. Podelski and A. Rybalchenko, "Transition invariants," in *Proceedings, 19th Annual Symposium on Logic in Computer Science*, 2004, pp. 132–144.

[8] M. T. C. Group, "Demo of Aligator," Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, Tech. Rep., 2010.

[9] P. Cousot and R. Cousot, "Automatic synthesis of optimal invariant assertions: Mathematical foundations," vol. 12, no. 8. ACM, 1977, pp. 1–12.

[10] P. Cousot, "Abstract interpretation," Ecole Normale Superieure, Paris, France, Tech. Rep. www.di.ens.fr/c̃ousot/AI/, August 2008, accessed: 12-10-2012.

[11] C. A. Furia and B. Meyer, "Inferring loop invariants using postconditions," in *Festschrift in honor of Yuri Gurevich's 70th birthday*, ser. Lecture Notes in Computer Science, N. Dershowitz, Ed. Springer-Verlag, August 2010.

[12] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576 – 583, Oct. 1969.

[13] M. A. Colon, S. Sankaranarayana, and H. B. Sipma, "Linear invariant generation using non linear constraint solving," in *Proceedings, Computer Aided Verification, CAV 2003*, ser. Lecture Notes in Computer Science, vol. 2725. Springer Verlag, 2003, pp. 420–432.

[14] Z. Manna, *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[15] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, "Proving conditional termination," in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 328–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70545-1-32

[16] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Communications of the ACM*, vol. 54, no. 5, 2011.

[17] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival, "The astree static analyzer," Ecole Normale Superieure, http://www.astree.ens.fr/, Tech. Rep., 2012.

[18] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival, "Varieties of static analyzers: A comparison with astree," in *TASE*, 2007, pp. 3–20.

[19] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 459–470.

[20] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 515–524.

[21] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 162–168.

[22] J. Cai and R. Paige, "Program derivation by fixed point computation," *Science of Computer Programming*, vol. 11, no. 3, pp. 197 – 261, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167642388900330

[23] H. Partsch and R. Steinbrüggen, "Program transformation systems," *ACM Comput. Surv.*, vol. 15, no. 3, pp. 199–236, Sep. 1983. [Online]. Available: http://doi.acm.org/10.1145/356914.356917

[24] N. Diallo, W. Ghardallou, and A. Mili, "Correctness and relative correctness," in *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy, May 20–22 2015.

[25] W. Ghardallou, N. Diallo, M. Frias, and A. Jaoua, "Debugging without testing," in *Proceedings, ICST 2016*, 2016.

[26] N. Diallo, W. Ghardallou, and A. Mili, "Program repair by stepwise correctness enhancement," in *First International Workshop on Pre- and Post-Deployment Verification Techniques*, Reykjavk, Iceland, June 2016.

[27] A. Mili, M. Frias, and A. Jaoua, "On faults and faulty programs," in *Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science*, ser. Lecture Notes in Computer Science, P. Hoefner, P. Jipsen, W. Kahl, and M. E. Mueller, Eds., vol. 8428. Marienstatt, Germany: Springer, April 28–May 1st 2014.

[28] N. Diallo, W. Ghardallou, and A. Mili, "Program derivation by correctness enhancements," in *Refinement 2015*, Oslo, Norway, June 2015.

[29] W. Ghadallou, N. Diallo, and A. Mili, "Software evolution by correctness enhancement," in *The 28th International Conference on Software Engineering and Knowledge Engineering, San francisco, CA, USA, July 1-4, 2016.*

[30] G. Schmidt and T. Stroehlein, *Relationen und Graphen.* Berlin, Germany: Springer-Verlag, 1990.

[31] C. Brink, W. Kahl, and G. Schmidt, *Relational Methods in Computer Science.* Springer Verlag, January 1997.

[32] E. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.

[33] E. Hehner, *A Practical Theory of Programming.* Prentice Hall, 1992.

[34] C. Hoare and J. He, "The weakest prespecification," *Fundamentae Informaticae*, vol. IX, pp. Part I: pp 51–58. Part II: pp 217–252, 1986.

[35] R. Linger, H. Mills, and B. Witt, *Structured Programming.* Addison Wesley, 1979.

[36] C. Morgan, *Programming from Specifications*, ser. International Series in Computer Sciences. London, UK: Prentice Hall, 1998.

[37] D. L. Parnas, "Precise description and specification of software," in *Software Fundamentals*, D. M. Hoffman and D. M. Weiss, Eds. Addison Wesley, 2001, ch. 5.

[38] N. Boudriga, F. Elloumi, and A. Mili, "The lattice of specifications: Applications to a specification methodology," *Formal Aspects of Computing*, vol. 4, pp. 544–571, 1992.

[39] A. Mili, S. Aharon, and C. Nadkarni, "Mathematics for reasoning about loop," *Science of Computer Programming*, pp. 989–1020, 2009.

[40] O. Mraihi, A. Louhichi, L. L. Jilani, J. Desharnais, and A. Mili, "Invariant assertions, invariant relations, and invariant functions," *Science of Computer Programming*, vol. 78, no. 9, pp. 1212–1239, September 2013. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2012.05.006

[41] E. R. Carbonnell and D. Kapur, "Program verification using automatic generation of invariants," in *Proceedings, International Conference on Theoretical Aspects of Computing 2004*, vol. 3407. Lecture Notes in Computer Science, Springer Verlag, 2004, pp. 325–340.

[42] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2006.

[43] J. Fu, F. B. Bastani, and I.-L. Yen, "Automated discovery of loop invariants for high assurance programs synthesized using ai planning techniques," in *HASE 2008: 11th High Assurance Systems Engineering Symposium*, Nanjing, China, 2008, pp. 333–342.

[44] L. Kovacs and T. Jebelean, "Automated generation of loop invariants by recurrence solving in theorema," in *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, Eds. Timisoara, Romania: Mirton Publisher, 2004, pp. 451–464.

[45] L. I. Kovacs and T. Jebelea, "An algorithm for automated generation of invariants for loops with conditionals," in *Symbolic and Numeric Algorithms for Scientific Computing, 2005. SYNASC 2005. Seventh International Symposium on.* IEEE, 2005, pp. 5–pp.

[46] S. Sankaranarayana, H. B. Sipma, and Z. Manna, "Non linear loop invariant generation using Groebner bases," in *Proceedings, ACM SIGPLAN Principles of Programming Languages, POPL 2004*, 2004, pp. 381–329.

[47] A. Mili, S. Aharon, C. Nadkarni, O. Mraihi, A. Louhichi, and L. L. Jilani, "Reflexive transitive invariant relations: A basis for computing loop functions," *Journal of Symbolic Computation*, vol. 45, pp. 1114–1143, 2009.

[48] W. Ghardallou, O. Mraihi, A. Louhichi, L. L. Jilani, K. Bsaies, and A. Mili, "A versatile concept for the analysis of loops," *Journal of Logic and Algebraic Programming*, vol. 81, no. 5, pp. 606–622, May 2012.

[49] A. Mili, J. Desharnais, and J. R. Gagne, "Strongest invariant functions: Their use in the systematic analysis of while statements," *Acta Informatica*, April 1985.

[50] R. Floyd, "Assigning meaning to programs," *Proceedings of the American Mathematical Society Symposium in Applied mathematics*, vol. 19, pp. 19–31, 1967.

[51] A. Louhich, W. Ghardallou, K. Bsaies, L. L. Jilani, O. Mraihi, and A. Mili, "Verifying while loops with invariant relations," *Submitted to a Special Issue of a Journal*, 2012. [Online]. Available: http://web.njit.edu/~mili/ccb.pdf

[52] M. Colón and H. Sipma, "Practical methods for proving program termination," in *Proc. International Conference on Computer Aided Verification*, ser. CAV '02. London, UK, UK: Springer-Verlag, 2002, pp. 442–454.

[53] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *VMCAI*, 2004, pp. 239–251.

[54] A. Bradley, Z. Manna, and H. Sipma, "The polyranking principle," in *Proceedings, ICALP 2005*, 2005.

[55] A. R. Bradley, Z. Manna, and H. B. Sipma, "Linear ranking with reachability," in *Computer Aided Verification*. Springer, 2005, pp. 491–504.

[56] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *Proceedings, TACAS 2013: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Verlag, 2013, pp. 47–61.

[57] V. D'Silva and caterina Urban, "Complexity of bradley-manna-sipma lexicographic functions," in *CAV 2015: Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., no. 9206, San Francisco, CA, USA, July, 18-24 2015.

[58] D. Kroening, N. Sharygina, S. Tonetta, A. L. Jr, S. Potiyenko, and T. Weigert, "Loopfrog: Loop summarization for static analysis," in *Proceedings, Workshop on Invariant Generation: WING 2010*, Edimburg, UK, July 2010.

[59] A. Avizienis, J. C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[60] P. Cousot, "Abstract interpretation," Ecole Normale Superieure, Paris, France, Tech. Rep. www.di.ens.fr/c̃ousot/AI/, August 2008.

[61] A. Podelski and A. Rybalchenko, "Transition invariants and transition predicate abstraction for program termination," in *TACAS*, 2011, pp. 3–10.

[62] L. L. Jilani, O. Mraihi, A. Louhichi, W. Ghardallou, K. Bsaies, and A. Mili, "Invariant relations and invariant functions: An alternative to invariant assertions," *Journal of Symbolic Computation*, vol. 48, pp. 1–36, May 2013.

[63] R. Boyer and J. Moore, *A Computational Logic Handbook*. Academic Press inc., 1988.

[64] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," in *POPL*, 2008, pp. 147–158.

[65] H. Velroyen and P. Rümmer, "Non-termination checking for imperative programs," in *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, ser. lncs, B. Beckert and R. Hähnle, Eds., vol. 4966. spv, 2008, pp. 154–170.

[66] K. Durant, W. Visser, and C. Pasareanu, "Investigating termination of affine loops with jpf," in *Java PathFinder Workshop*, Lawrence, KS, 2012.

[67] A. R. Bradley, Z. Manna, and H. B. Sipma, "Termination analysis of integer linear loops," in *CONCUR*, 2005, pp. 488–502.

[68] A. Tiwari, "Termination of linear programs," in *CAV*, 2004, pp. 70–82.

[69] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *ASE*, 2009, pp. 161–169.

[70] S. Falke, D. Kapur, and C. Sinz, "Termination analysis of imperative programs using bitvector arithmetic," in *VSTTE*, 2012, pp. 261–277.

[71] W. Lee, B.-Y. Wang, and K. Yi, "Termination analysis with algorithmic learning," in *CAV*, 2012, pp. 88–104.

[72] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 415–426. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134029

[73] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang, "Ranking abstractions," in *ESOP*, 2008, pp. 148–162.

[74] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *Proc.International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011, pp. 81–95.

[75] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, "Proving conditional termination," in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 328–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70545-1-32

[76] B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn, "Disproving termination with overapproximation," in *Proceedings, FMCAD*, Lausanne, CH, October 2014.

[77] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings, Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, 1977.

[78] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Astrée: Proving the absence of runtime errors," in *Embedded Real Time Software and Systems (ERTS^2 2010)*, May 2010, pp. 1–9.

[79] C. Ancourt, F. Coelho, and F. Irigoin, "A modular static analysis approach to affine loop invariants detection," *Electronic Notes on Theoretical Computer Science*, vol. 267, no. 1, pp. 3–16, 2010.

[80] S. Sagiv, T. W. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Transactions on Programming Logics and Systems*, vol. 24, no. 3, pp. 217–298, 2002.

[81] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori, "Bottom up shape analysis using lisf," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 5, 2011.

[82] F. Spoto, F. Mesnard, and E. Payet, "A termination analyzerfor java bytecode based on path length," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 3, 2010.

[83] S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[84] B. Hackett and A. Aiken, "How is aliasing used in systems software," in *Proceedings, 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006, pp. 69–80.

[85] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings, LICS*, 2002, pp. 55–74.

[86] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings, CSL*, 2001, pp. 1–19.

[87] G. Nelson, "Verifying reachability invariants of linked structures," in *Proceedings, POPL 1983: Principles of Programming Languages*, 1983, pp. 38–47.

[88] S. Lahiri and R. Bryant, "Constructing quantified invariants via predicate abstraction," in *Proceedings, VMCAI*, 2004, pp. 267–281.

[89] A. Podelski and T. Wies, "Boolean heaps," in *Procedings, SAS*, 2005, pp. 267–282.

[90] S. Gulwani, B. McCloskey, and A. Tiwari, "Lifting abstract interpreters to quantified logic domains," in *35th ACM Symposium on Principles of Programming Languages.* ACM, january 2008, pp. 235–246.

[91] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *Proceedings, TACAS*, 2008, pp. 413–427.

[92] F. Mehta and T. Nipkow, "Proving pointer programs in higher order logic," *Inf. Comput.*, vol. 199, no. 1-2, pp. 200–277, 2005.

[93] J. Filliatre and C. Marche, "Multi prover verification of c programs," in *Procedings, ICFEM*, 2004, pp. 15–29.

[94] C. team, "caveat project," Commissariat a l'Energie Atomique, http://www-drt.cea.fr/Pages/List/Ise/LSL/Caveat/, Tech. Rep., 2012.

[95] B. Cook, "Static driver verifier," Microsoft Inc., http://www.microsoft.com/whdc/devtools/, Tech. Rep., 2012.

[96] B. Meyer, "Proving pointer program properties. part i: Context and overview," *Journal of Object Technology*, vol. 2, no. 2, pp. 87–108, 2003.

[97] M. Bertrand, "Proving pointer program properties. part 2: The overall object structure." *Journal of Object Technology*, vol. 2, no. 3, pp. 77–100, 2003.

[98] R. W. Collins, G. H. Walton, A. R. Hevner, and R. C. Linger, "The CERT function extraction experiment: Quantifying FX impact on software comprehension and verification," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2005-TN-047, December 2005.

[99] A. R. Hevner, R. C. Linger, R. W. Collins, M. G. Pleszkoch, S. J. Prowell, and G. H. Walton, "The impact of function extraction technology on next generation software engineering," Software Engineering Institute, Tech. Rep. CMU/SEI-2005-TR-015, July 2005.

[100] N. Diallo, "fxloop analyzer," Software Engineering Lab, New Jersey Institute of Technology, https://selab.njit.edu/tools/fxloop.php, Tech. Rep., 2015.

[101] J. Laprie, "Dependability —its attributes, impairments and means," in *Predictably Dependable Computing Systems*. Springer Verlag, 1995, pp. 1–19.

[102] J. C. Laprie, *Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese*. Heidelberg: Springer Verlag, 1991.

[103] J.-C. Laprie, "Dependable computing: Concepts, challenges, directions," *COMPSAC-NEW YORK-*, pp. 242–243, 2004.

[104] D. Gries, *The Science of programming*. Springer Verlag, 1981.

[105] H. Mills, V. Basili, J. Gannon, and D. Hamlet, *Structured Programming: A Mathematical Approach*. Boston, Ma: Allyn and Bacon, 1986.

[106] J. Desharnais, N. Diallo, W. Ghardallou, M. Frias, A. Jaoua, and A. Mili, "Mathematics for relative correctness," in *Relational and Algebraic Methods in Computer Science, 2015*, Lisbon, Portugal, September 2015.

[107] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *proceedings, Automated Software Engineering*, Lawrence, KS, 2011.

[108] F. Logozzo, S. Lahiri, M. Faehndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *Proceedings, PLDI*, 2014, p. 32.

[109] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proceedings, ESEC/ SIGSOFT FSE*, 2013, pp. 345–455.

[110] F. Logozzo and T. Ball, "Modular and verified automatic program repair," in *Proceedings, OOPSLA*, 2012, pp. 133–146.

[111] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings, ICSE*, 2013, pp. 772–781.

[112] A. Louhichi, W. Ghardallou, K. Bsaies, L. L. Jilani, O. Mraihi, and A. Mili, "Verifying loops with invariant relations," *International Journal of Critical Computer Based Systems*, vol. 5, no. 1/2, pp. 78–102, 2014.

[113] A. Louhichi, O. Mraihi, L. L. Jilani, and A. Mili, "Invariant assertions, invariant relations and invariant functions," in *Proceedings, 2nd International Workshop on Invariant Generation*, York, UK, 2009.

[114] H. Mills, "The new math of computer programming," *Communications of the ACM*, vol. 18, no. 1, January 1975.

[115] A. Mili, J. Desharnais, and F. Mili, "Relational heuristics for the design of deterministic programs," *Acta Informatica*, vol. 24, no. 3, pp. 239–276, 1987.

[116] Y. S. Ma, J. Offutt, and Y. R. Kwon, "Mu java: An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[117] G. Dromey, "Program development by inductive stepwise refinement," University of Wollongong, Australia, Tech. Rep. Working Paper 83-11, 1983.

[118] H. D. T. i. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.

[119] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.

[120] R. Banach and M. Poppleton, "Retrenchment, refinement and simulation," in *ZB: Formal Specifications and Development in Z and B*, ser. Lecture Notes in Computer Science. Springer, December 2000, pp. 304–323.

[121] A. Ghosal, M. Jurdzinski, R. Majumdar, and V. Prabhu, "Approximate refinement for hybrid systems," University of California at Berkeley, Tech. Rep.

[122] J. V. Deshmukh, R. Majumdar, and V. Prabhu, "Quantifying conformance using the skorokhod metric," in *Proceedings, CAV: Computer Aided Verification*. Springer Verlag, 2015, pp. 234–250.

[123] K. Chaterjee and V. S. Prabhu, "Quantitative temporal simulation and refinement distancess for timed systems," *IEEE Transactions for Automatic Control*, vol. 60, no. 9, pp. 2291–2306, 2015.

[124] J. Desharnais, N. Diallo, Ghadallou, and A. Mili, "Projecting programs on specifications: Definition and implications," NJIT, Newark, NJ, http://web.njit.edu/~mili/prj.pdf, Tech. Rep., 2016.