ABSTRACT

A PRACTICAL APPROACH TO AUTOMATED SOFTWARE CORRECTNESS ENHANCEMENT

by
Aleksandr Zakharchenko

To repair an incorrect program does not mean to make it correct; it only means to make it more-correct, in some sense, than it is. In the absence of a concept of relative correctness, i.e. the property of a program to be more-correct than another with respect to a specification, the discipline of program repair has resorted to various approximations of absolute (traditional) correctness, with varying degrees of success. This shortcoming is concealed by the fact that most program repair tools are tested on basic cases, whence making them absolutely correct is not clearly distinguishable from making them relatively more-correct. In this research a theory of relative correctness is used to implement an instance of a generic algorithm of program repair, whose core idea is to enhance relative correctness until absolute correctness is achieved. Analytical and empirical results pertaining to the approach and its high performance parallel implementation are presented in this work.

A PRACTICAL APPROACH TO AUTOMATED SOFTWARE CORRECTNESS ENHANCEMENT

by
Aleksandr Zakharchenko

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

December 2021

APPROVAL PAGE

A PRACTICAL APPROACH TO AUTOMATED SOFTWARE CORRECTNESS ENHANCEMENT

Aleksandr Zakharchenko

Dr. Ali Mili, Dissertation Advisor — Date
Professor of Computer Science and Associate Dean of Academic Affairs, NJIT

Dr. James M. Calvin, Committee Member — Date
Professor and Associate Chair of Computer Science, NJIT

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

Dr. Kurt R. Rohloff, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Zhenjiang Hu, Committee Member — Date
Professor of Computer Science, Peking University, Peking, People’s Republic of China

BIOGRAPHICAL SKETCH

Author:	Aleksandr Zakharchenko
Degree:	Doctor of Philosophy
Date:	December 2021
ORCID:	0000-0002-5665-4658

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2021
- Master of Business Administration in Finance,
  Saint Peter's University, Jersey City, NJ, 2018
- Bachelor of Science in Information Systems with Concentration in Programming,
  Strayer University, Washington, D.C., 2012

Major: Computer Science

Presentations and Publications:

Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2021. The Bane of Generate-and-Validate Program Repair: Too Much Generation, Too Little Validation. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press. 113-126. DOI: https://doi.org/10.3233/FAIA210013

Aleksandr Zakharchenko, Besma Khaireddine, and Ali Mili. 2021. A Massively Parallel Approach to Automated Software Correctness Enhancement in Java. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press. 141-154. DOI: https://doi.org/10.3233/FAIA210015

Aleksandr Zakharchenko. 2020. Implementing AI Models Across International Trading Systems. *ValleyML AI Expo 2020*, Virtual Event.

Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2020. A Semantic Definition of Faults and Its Implications. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. Macau, China. IEEE. 14-21. DOI: https://doi.org/10.1109/QRS51102.2020.00015

Besma Khaireddine, Marwa Ben AbdelAli, Lamia Labed Jilani, Aleksandr Zakharchenko, and Ali Mili. 2020. Correctness Enhancement: A Pervasive Software Engineering Paradigm. *International Journal of Critical Computer-Based Systems* 10 (1), 37-73.

Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2019. Fault Density, Fault Depth and Fault Multiplicity: The Reward of Discernment. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 532-533, DOI: https://doi.org/10.1109/QRS-C.2019.00110.

Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2017. A Generic Algorithm for Program Repair. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE '17)*. IEEE Press, 65–71.

Aleksandr Zakharchenko and James Geller. 2016. Expansion of the Hierarchical Terminology Auditing Framework Through Usage of Levenshtein Distance-Based Criterion. *Studies in Health Technology and Informatics* vol. 228, 491-495. DOI: https://doi.org/10.3233/978-1-61499-678-1-491, PMID: 27577431.

Aleksandr Zakharchenko and James Geller. 2015. Auditing of SNOMED CT's Hierarchical Structure using the National Drug File-Reference Terminology. *Studies in Health Technology and Informatics* vol. 210, 130-134. DOI: https://doi.org/10.3233/978-1-61499-512-8-130, PMID: 25991116.

Christopher Ochs, Zheng Ling, Gu Huanying, Yehoshua Perl, James Geller, Joan Kapusnik-Uner, and Aleksandr Zakharchenko. 2015. Drug-drug Interaction Discovery Using Abstraction Networks for "National Drug File–Reference Terminology" Chemical Ingredients. In *American Medical Informatics Association Annual Symposium Proceedings*. vol. 2015, 973-982, PMID: 26958234, PMCID: PMC4765653.

Submitted Publications:

Besma Khaireddine, Aleksandr Zakharchenko, Matias Martinez, and Ali Mili. 2021. Towards a Theory of Program Repair, *Acta Informatica*. 2021. Revision submitted for review.

Dedicated to my parents, grandparents, and a long list of people

who helped me become who I am today.

ACKNOWLEDGMENT

I thank Dr. Ali Mili for his guidance. His expertise and continuous feedback made the work in this dissertation possible.

I thank committee member Dr. James M. Calvin, committee member Dr. Iulian Neamtiu, committee member Dr. Kurt R. Rohloff and committee member Dr. Zhenjiang Hu for their time and effort.

I thank the fellow researchers Besma Khaireddine, Lamia Labed Jilani, Marwa Ben AbdelAli and Matias Martinez for their contributions to the work in this dissertation.

Lastly, I thank my parents and grandparents for their support and encouragement over the entire course of this multi-year journey.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION. …………………………………………………….………… 1

1.1 Motivation. ..………………………………………………….………... 1

1.2 Dissertation Overview. ……….……………….………………….……... 5

2 THE STATE OF THE ART IN PROGRAM REPAIR. …..…...…….….…………..…. 7

2.1 A Brief History of Program Repair. ….…………………………………….. 7

2.1.1 Hardware Limitations. ……………………………………………... 7

2.1.2 Evolution of Software Development Approaches. ……………………….. 8

2.2 Automatic Software Repair - A Survey of Current Practice. ………………… 10

2.3 A Focus on Faults. ..……………………………………….…………...…. 14

2.4 Bane of Program Repair: Too Much Generation, Too Little Validation. ..……. 16

2.5 Premises of the Relative Correctness-based Approach. ……………………… 20

3 BACKGROUND FOR A THEORETICAL APPROACH. ……………………… 21

3.1 A Critique: The Need for the Theory of Relative Correctness. ..……………… 21

3.2 Mathematics for Program Repair. …...………………………………………… 24

3.2.1 Relational Mathematics. ……………………..………………………... 24

3.2.2 Program Semantics and Correctness. ………...…………………………... 25

3.3 Absolute Correctness and Relative Correctness. ……………………………… 28

4 INGREDIENTS OF A THEORY BASED PROGRAM REPAIR ALGORITHM. . 31

4.1 Faults and Elementary Faults. ………………………………………….…. 31

4.2 Fault Density, Depth and Multiplicity. …………..…………………….…. 34

4.3 Fault Repair vs. Failure Remediation. ………………..……………………... 35

TABLE OF CONTENTS
(Continued)

Chapter **Page**

5 A GENERIC ALGORITHM FOR PROGRAM REPAIR. ………………………… 39

5.1 General Principle. .....………………………………………………………...… 39

5.2 An Infrastructure of Oracles. ……………………………………………… 39

5.2.1 Absolute Correctness With Respect to a Specification R. ……………… 39

5.2.2 Relative Correctness Over a Program P With Respect to a Specification R. ………………………………………………………………………… 40

5.2.3 Strict Relative Correctness Over a Program P With Respect to a Specification R. ……………………………………………...……………... 41

5.3 A Generic Algorithm. ……………………...……………………………… 41

5.4 Assessment of Precision and Recall. ………………………………….… 43

5.5 Introducing Parallelism. ………………..……………………………………… 44

6 AN INSTANCE OF THE GENERIC ALGORITHM: CORRECTNESS ENHANCER. ………………………………………………………………………. 49

6.1 Specification of Correctness Enhancer. ...……………………………...… 49

6.1.1 Design Goals. …………………...………………………………………... 49

6.1.2 Specifications in Practice. …………………………………………………... 50

6.2 Design of Correctness Enhancer. ……………………………………………… 51

6.2.1 A Practical Implementation Analysis - Approaching the Issue. ………… 51

6.2.2 Programming Language Choice. ………………………………………. 51

6.2.3 Program Repair Approach. ……………………………..……………… 53

6.2.4 MuJava. ………………………………………………………………... 53

6.2.5 The New Patch Validation. ……………………………………………….. 54

TABLE OF CONTENTS
(Continued)

Chapter Page

6.2.6 Adjusting to Changes in Project Structures With Levenshtein Distance-based Criterion. ……………………………………………………...……………... 55

6.3 Implementation of Correctness Enhancer: Introducing Parallelism. ………… 56

6.3.1 Reasons for Parallelism. ………………………………………………… 56

6.3.2 Implementation of Parallelism. ………………………………………... 58

7 EMPIRICAL ASSESSMENT. …………...…………………………………...…. 69

7.1 Performance on Standard Benchmarks. ……………………………………..… 69

7.2 Comparison: Fitness Function vs. Relative Correctness. …..…………………..…. 69

7.3 Comparison: Correctness Enhancer vs. Other Tools. ………………………… 72

8 LESSONS LEARNED. ………………..……………………………………………. 74

8.1 Theoretical Lessons: The Need for Theoretical Foundations. ………………. 74

8.2 Practical Lessons: The Use / Impact of Computing Power. ……………….….. 76

8.3 Outstanding Research Questions. …………………..……………………….... 78

9 CONCLUSION. ……………..…………………………………………………... 80

9.1 Summary. ………………………………...……………………………… 80

9.2 Assessment. …………………………………………………………………..… 80

9.3 Prospects. ………………………..……………………………………… 81

APPENDIX A PROOF OF PERFECT PRECISION AND PERFECT RECALL. 83

A.1 Proof of Perfect Recall. …………………………………………………… 83

A.2 Proof of Perfect Precision. ………………………………………………… 89

APPENDIX B CORRECTNESS ENHANCER USER MANUAL. ………......…....... 93

TABLE OF CONTENTS
(Continued)

Chapter **Page**

APPENDIX C APACHE DERBY LAUNCHER AND DATABASE TABLES. ……. 95

REFERENCES. ……………………………………………………………………….. 98

LIST OF TABLES

Table **Page**

7.1 Defects4j Results Comparison With Other Published Tools …….……………. 72

8.1 Results Variation …………...…..…...………………..…………………………. 75

LIST OF FIGURES

Figure **Page**

1.1 Demonstration of stages of a typical Agile software development process. 4

2.1 Number of papers published on automated program repair from 1996 to 2016. ... 10

2.2 An example of Generate and Validate process. ……………………………….. 13

2.3 An example of Semantic-driven repair process. ………………………………… 14

2.4 Absolute Correctness mandates larger spaces. ………………………………… 17

2.5 Relative Correctness enables step-wise validation. …………………………….... 18

4.1 An elementary fault of multiplicity 2. …………………………………………… 34

4.2 Fault Density vs. Fault Depth. ………………………………………………… 35

4.3 A generic example of correctness enhancement. ………………………………... 37

4.4 Failure Remediation vs. Fault Repair. …………………………………………... 38

5.1 Schematic drawing of a possible parallel implementation. …………...………… 47

5.2 Expanded view of validate stage flow for a single execution path, when launched on HPC Grid. ………………………………….……………………..... 48

6.1 Correctness Enhancer has retained MuJava's mutation mode screen in order to allow triggering mutant generation from UI locally, in addition to being able to do it through the console. …………………………………………………..... 59

6.2 Correctness Enhancer has mostly retained the original UI interface of muJava for local execution on a machine, adding option to control parallelism on validation and using "Live Mutants" to list results that are more correct than the original, however, main mode of operation is through console. …………………... 60

7.1 A screenshot of a portion of the results demonstrating the checks applied. ...…... 71

8.1 P to P' surface of JFreeCharts Charts1b variation. …………...…….….………... 77

LIST OF SYMBOLS

©	Copyright
$\forall$	For all
$\exists$	Exists (at least one)
$\in$	Belonging to
$\wedge$	Logical AND
v	Logical OR
$\cap$	Intersection
$\cup$	Union
$\neg$	Not (Logical)
$\supseteq$	Superset (For example, B is a superset of A is written as B $\supseteq$ A)
$\subseteq$	Subset
$\supset$	Proper superset
$\subset$	Proper subset
$\emptyset$	Empty set
$\sqsubseteq$	Is refined by
$\sqsupseteq$	Refines
dom(R)	Competence domain of R
$_{T}\backslash R$	The pre-restriction of R to T

LIST OF DEFINITIONS

Fault	Given a specification R, a program P and a feature f in P, we say that f is a fault in P with respect to R if and only if there exists a feature f′ such that the program P' obtained from P by replacing f by f' is strictly more-correct than P with respect to R.
Refining	Given two relations R and R', we say that R' refines R (abbrev: $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R$.
Absolute Correctness	A deterministic program p on space S is said to be correct (or absolutely correct) with respect to specification R on S if and only if its function P refines R.
Relative Correctness	Given a specification R and two deterministic programs P and P′, we say that P' is more-correct than P with respect to R if and only if $(R \cap P')L \supset (R \cap P)L$.
Strict Relative Correctness	Given a specification R and two deterministic programs P and P′, we say that P' is strictly more-correct than P with respect to R if and only if $(R \cap P')L \supset (R \cap P)L$.
Fault Removal	Given a program P and a specification R, a pair of features (f, f') is said to be a fault removal in P with respect to R if and only if f is a fault in P and program P' obtained from P by replacing f by f' is strictly more-correct than P.
Elementary fault	Given a program P, a specification R, and a fault f in P with respect to R, we say that f is an elementary fault in P with respect to R if and only if f has a single syntactic atom, or it has more than one syntactic atom, but no subset thereof is a fault.
Fault Density	Given a program P and a specification R, the fault density of P with respect to R is the number of elementary faults in P.
Fault Multiplicity	Number of syntactic atoms that form an elementary fault
Fault Depth	The fault depth of P with respect to R is the minimal number of elementary fault removals that are needed to transform P into an absolutely correct program (for the selected set of atomic changes).

Oracle of Absolute Correctness	Given a specification R on space S, the oracle of absolute correctness derived from R is denoted by $\Omega(s, s')$ and defined by: $\Omega(s, s') \equiv (s \in dom(R) \Rightarrow (s, s') \in R)$
Oracle of Relative Correctness	Given a specification R on space S and a program P on S, the oracle of relative correctness over P with respect to R is denoted by $\omega(s, s')$ and defined by: $\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s'))$
Oracle of Strict Relative Correctness	Given a specification R on space S, a subset T of S and a program P on S, the oracle of strict relative correctness over P with respect to ${}_T\backslash R$ is denoted by σ_T () and defined by: $\sigma_T (P') \equiv (\forall s \in T : \omega(s, P'(s))) \wedge (\exists s \in T : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$
Syntactic atom	Unit of source code at selected level of granularity
Syntactic feature	Aggregate of one or more syntactic atoms
Atomic Change Operator	A mutation operator that is applied to a syntactic atom to produce a different unit of code
Levenshtein Distance	String metric for measuring distance between two sequences, as the minimum number of single character edits required to change one sequence into the other.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Developing a modern software product is a complex multistep process [1], which involves multiple parties. This process sequentially goes through the steps of collecting the requirements, coming up with the design ideas to create synergy between the business definition of the product and its desired technical characteristics in order to form the product specifications and using the latter as an input for the chain of steps involving breakdown into individual development requirements, prioritization, implementation of the requirements in code, quality assurance and testing and finally culminating in release of the end product followed by post-release monitoring and support (Figure 1.1). The product that is released, however, is subject to two types of maintenance - adaptive maintenance stemming from adjustments to user and business requirements as a part of normal business activities and corrective maintenance that results from imperfect implementation of original specifications and is manifested as a difference between the expected and actual software behavior. While both types of maintenance are a source of major expenses requiring highly skilled resources to change the software product, the corrective maintenance is an especially painful one, as unlike adaptive maintenance it is normally not capitalized and is not bringing any new business value per se, while still introducing a risk of business disruption with patch application going wrong and leading to a continued accrual of significant costs of ownership of the target product.

Any costly labor intensive business process ends up often being considered as a candidate for automation. The idea of automating the process of program repair is not an exception with mentions of it being encountered as early as 1973 [2, 3]. However, high computational cost of localizing and addressing faults, combined with insufficient computational capacity has ensured that for many decades little progress has been made in practical addressing of the problem at hand. As time passed by, the hardware and the software running on it have rapidly evolved both in complexity and in the performance that they offer, bringing in the renewed interest in the topic of automated program repair. According to a detailed research survey by Gazzola et al. [4], based on the increase in the number of papers being published every year on this topic, the interest in the field of automated program repair has been growing steadily since the middle of the first decade (from 2005), with certain specific aspects being the focus of the early applications. A large variety of tools has been presented to the market, supporting different programming languages [5-30] with a few of them like GenProg setting a high watermark in the industry. Nevertheless, up until now, the amount of computations required to approach the general case of a problem of correcting a piece of code that does not conform to its specifications has remained several orders of magnitude higher than what could be efficiently processed on an average user desktop, resulting in the proposed tools resorting to artificial limitations on the search space, reducing the scope of the core algorithm to specific narrow cases of software faults that could be addressed with an absolute correctness-driven hit-or-miss enhancement algorithm without having to do a search across the entire search space.

This dissertation describes improvements to existing theoretical framework of relative correctness-driven correctness enhancement [31, 32] and dives into practical considerations in implementations for automated adjustments to project structure changes and massively parallel execution. Additionally, tools for the automatic program repair in general case are introduced. Four important research areas are described:

1. Programming-language specific code structure analysis.

2. Parallelization and scalability of generate and validate approaches for automated program repair

3. Relative correctness-based optimization in control of validation process.

4. Software tools for automated program repair and correctness enhancement suggestions.

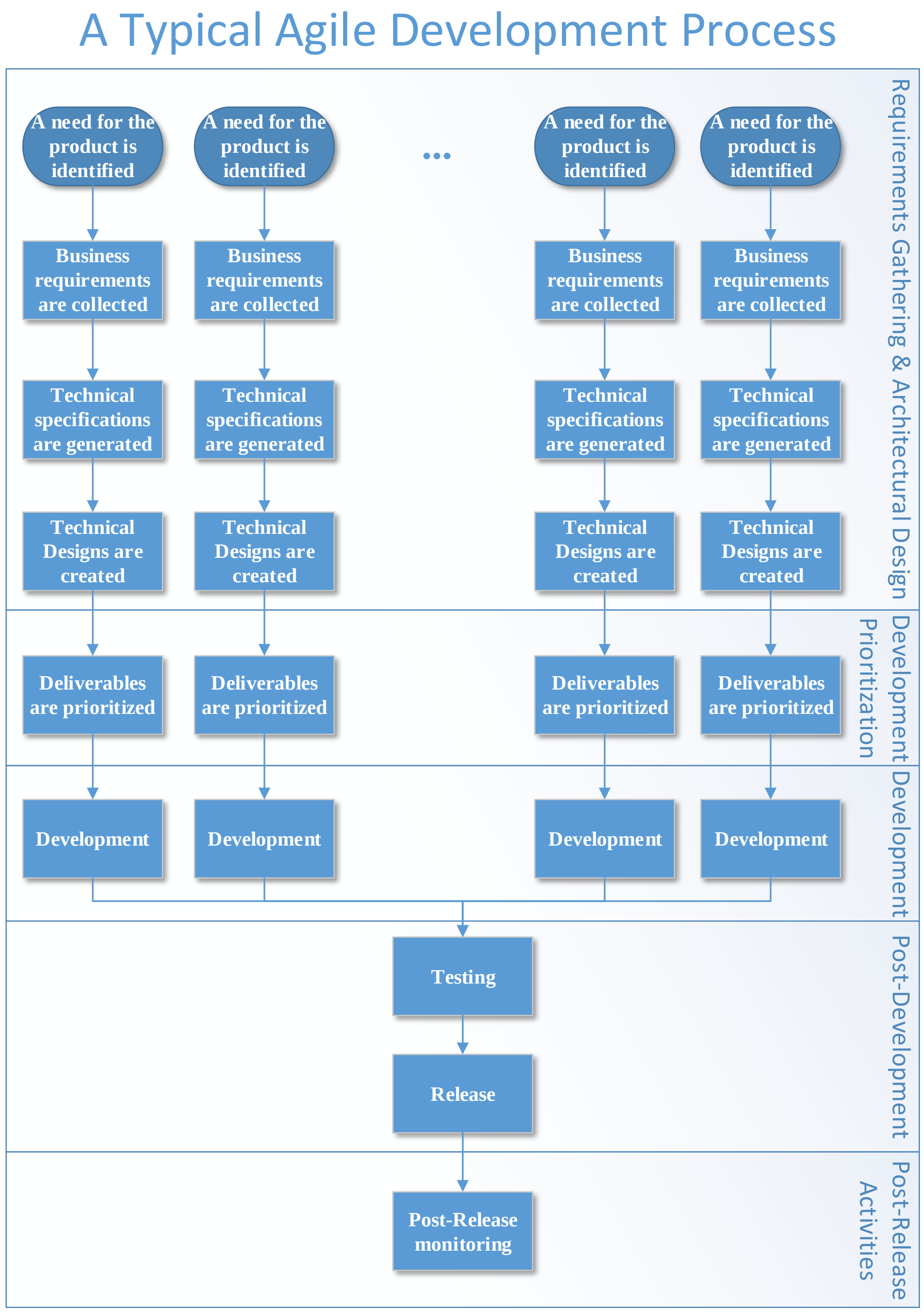

A Typical Agile Development Process
A need for the product is identified
...
Business requirements are collected
Technical specifications are generated
Technical Designs are created
Deliverables are prioritized
Development
Testing
Release
Post-Release monitoring
Requirements Gathering & Architectural Design
Development Prioritization
Development
Post-Development
Post-Release Activities

Figure 1.1 Demonstration of stages of a typical Agile software development process.

1.2 Dissertation Overview

Chapter 2 provides a brief overview of historical background of the field of program repair. It also provides information on the state of the art in the industry of program repair and discussed the premises behind this research. Specifically, Section 2.1 does a brief overview of the field history. Next, Section 2.2 walks over the survey of the current practices in the industry. Section 2.3 highlights some of the research works in the field that did not fall under the survey. Section 2.4 explains the conceptual inefficiencies of existing methods. Section 2.5 discusses the premises behind the new approach capable of addressing these inefficiencies. Chapter 3 provides a foundational theoretical background necessary for explanation of the theoretical framework being introduced. Specifically, Section 3.1 describes the need for a solid theoretical foundation. Section 3.2 explains the basics of relational mathematics. Section 3.3 explains the concepts of relative correctness and absolute correctness. Chapter 4 builds up on the concepts of chapter 3 to provide the explanation of the foundational elements specific to the new program repair framework. Section 4.1 provides the definition of fault and elementary fault. Section 4.2 explains the concepts of fault depth, fault density and fault multiplicity. Section 4.3 discusses the connection between fault repair and failure remediation and the differences between fault repair-driven and failure remediation-driven approaches. Chapter 5 connects the elements discussed in the previous chapter offering the generic algorithm of program repair. Section 5.1 explains the general principles behind such algorithm. Section 5.2 introduces different types of oracles and explains the difference between them. Section 5.3 brings the concepts together to provide the layout of a generic algorithm of program repair using relative correctness framework. Section 5.4 provides a comparative analysis of precision and recall of this new algorithm. Section 5.5 reworks the algorithm for optimal execution on parallel machines. Chapter 6 offers an overview of Correctness Enhancer – a new, massively-parallel implementation of the described algorithm and explores the solutions to practical issues that arise with implementation. Section 6.1 dives into the design goals and specifications of the tool. Section 6.2 discusses the functional design of Correctness Enhancer. Section 6.3 focuses on the aspects of parallelism in the implementation and researches practical considerations of such implementation. Chapter 7 discusses the performance of the implementation against standard benchmarks and other tools. Section 7.1 describes the experiment setup. Section 7.2 details the functional components of the hybrid approach that was applied to generate results. Section 7.3 provides the results of comparing the code execution against other tools. Chapter 8 looks into the lessons learned from the experiment and outlines pathways for further research. Section 8.1 assesses the impact of the new theoretical approaches. Section 8.2 focuses on the practical side of things, specifically on the impact of computing power that made some of the newly applied approaches feasible. Section 8.3 details possible pathways for further improvement. Lastly Chapter 9 provides the concluding remarks. Section 9.1 provides a summary of this dissertation work and its importance for the field. Section 9.2 assesses the threats to validity. Section 9.3 provides general suggestions for further expansion and applicability of the approach in the field.

CHAPTER 2

THE STATE OF THE ART IN PROGRAM REPAIR

2.1 A Brief History of Program Repair

2.1.1 Hardware Limitations

The ideas behind the field of automated program repair have been around for many decades [2, 3], however, active practical build-up of the field has only started gaining traction recently. The primary reason behind the perceived delay is due to the lack of hardware capabilities required to support even the most basic general purpose implementations. While over the course of the last five decades, the ongoing increase in hardware capabilities was roughly following Moore’s Law [33] even today, the amount of time required to execute the program repair algorithms remains one of the key considerations in assessing their efficiency and usability.

The hardware evolution over the years didn’t follow a straight path. Over time it switched from focusing on maximizing the execution speed of a single thread, to focusing on heavily distributed parallelized execution [34], including processing beyond the CPU [35]. That shift in trends has led to changes in development paradigms in order to maximize the usage of hardware capabilities, however, mainly due to complexity and coordination overhead, application of these changes has lagged behind in many areas. A portion of this dissertation focuses on the key considerations behind application of massive parallelism in automated program repair and their practical implementation.

2.1.2 Evolution of Software Development Approaches

The rapid growth of hardware capabilities over the years combined with popularization of personal computers and later smartphones and computerized wearables has cleared the pathway for an increase in overall software surface, becoming a catalyst for ongoing digital transformation and automation of mundane tasks, allowing computerization to penetrate every area of human lives. The resulting increase in the amount of code being written, as well as in the complexity of projects being created has led to an evolution in the software development process, encompassing program management, development, testing and post-production activities.

The program management approaches evolved over the years from pure Waterfall software development model with in-deep pre-planning of the development process that was first presented in early 50s to increasingly more fine-grained and controllable Spiral, Extreme and finally Agile practices [36-42].

The software development evolution led to a change in software development language preferences, eventually switching the mainstream preferences to higher-level languages like C#, Java and Java-based languages like Scala, which, through additional layers of optimization and compilation to intermediary language, make a tradeoff between slight decrease in efficiency of hardware resource utilization and ease of writing code in these languages, as compared to machine code-compilable languages like C, C++, Delphi or Fortran, which dominated the field of software development before them. In addition to programming language choice, the need for developing and maintaining large amounts of code led to a shift from in-house built tools and software solutions to open-source ones [43, 44], which, by leveraging shared model of support come with a better and cheaper maintenance, as well as make it easier to find talent, skilled in working with them.

Evolution in the field of testing was multistage, from creation of a testing theory [45] to popularization of Test-Driven Development [46] and active introduction of continuous integration and continuous development approaches (further referred to as CI/CD) [47], making the specifications provided to code for machine-processable, followed by underlining the importance of creating them first before development begins and popularizing the practice of doing so and, finally, creating the systems to automatically control, whether new code being developed still matches the specifications that were provided initially and signaling in case the contract has been breached. Creation of test scripts and test cases, although rather simplistic, is a coding task on its own and as such, the evolution is currently going towards behavior-driven development (further referred to as BDD) [48], in order to make creation of machine-processable and machine-verifiable program specifications from original business specifications more automatic and requiring less development and QA resources to create. The latest trends in the field of software testing look to cover post-production testing as well, by designing synthetic monitoring systems [49], which, through means similar to regression testing provide real-time coverage of system behavior in production and allow to detect issues proactively, before real client traffic is impacted.

A combination of all these factors allowed evolution of automated program repair approaches.

2.2 Automatic Software Repair - A Survey of Current Practice

With hardware and software evolution in place the renewed interest to automated program repair was the logical next step to follow. A detailed industry survey by Gazzola et al. [4], by looking at the number of papers, published on the topic of automated program repair, identifies the breaking point being somewhere around 2005 and with the field being the focus of much recent research and the number of papers growing ever since (see Figure 2.1)

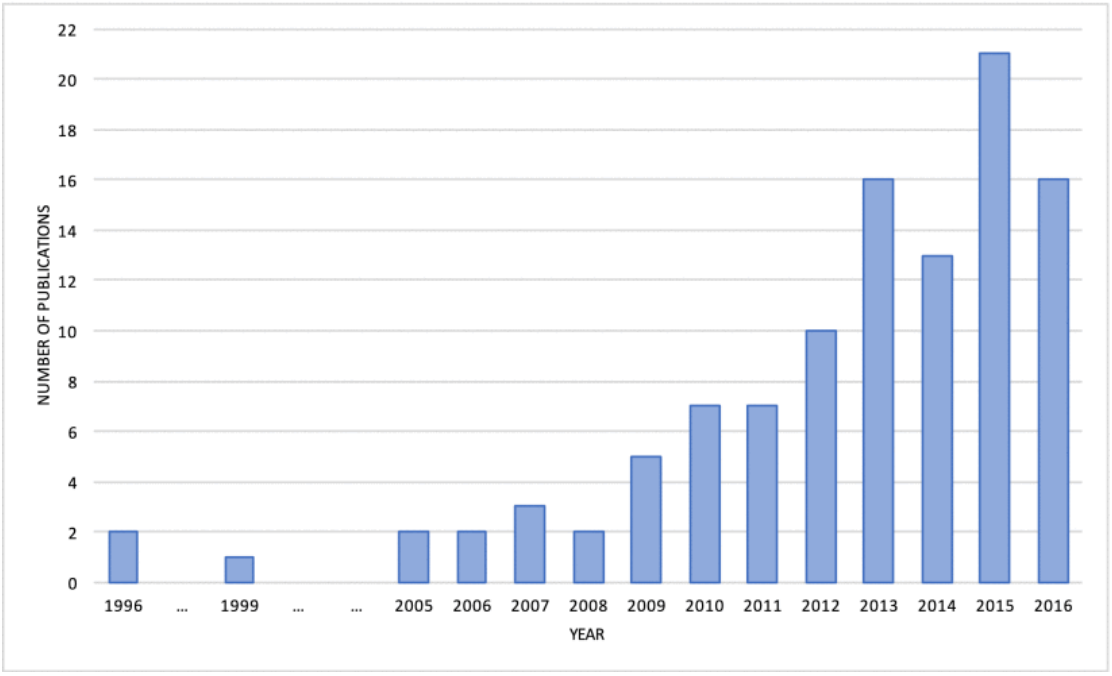


Figure 2.1 Number of papers published on automated program repair from 1996 to 2016. ©2019 IEEE.

Source: [4].

As a part of that survey [4], the authors subdivide the entire field of software repair solutions into two broad categories - software healing and software repair, based on whether the proposed solution detects and mitigates the effects of the failure at runtime on the deployed application, without correcting the fault itself or whether it detects the fault and applies the fix to it at source code level, thus fixing the fault. Software repair approaches are further subdivided into categories based on the following criteria:

1. Localization approach.

   a. Fault localization. Approaches falling under this category are looking for ways to locate the part of the program that needs fixing and use these localization results to drive fix generation. [20, 50-52]

   b. Fix locus localization. Unlike fault localization, this technique is locating all areas of the program, where a fix can be applied (also known as fix loci, hence the name of the category) regardless of where the actual fault is located. It is further subdivided into:

      i. Model-based fix locus localization, which analyzes runtime usage of the program to draw its conclusions about the model of object utilization being applied in the original code and possible incorrect usage of objects and their attributes. [53-55]

      ii. Angelic fix localization, which attempts to identify and fix faulty or missing decision points (such as if/else blocks) by changing execution flow through existing decision points (forcing the execution to follow a different decision branch) or by evaluating code execution, if several instructions are skipped (in order to identify the missing decision point and propose the fix which would effectively gate the skipped instructions.) [14]

2. Fix generation approaches.

   - Based on the scope of repair technique utilized, the approaches are categorized into:

     a. Fault-specific, targeting a specific narrow class of faults by exploiting certain generation techniques that are specific for that fault class or type. [56]

     b. General, without focus on a specific fault class, potentially being applicable to fix any fault encountered in the code. [57, 58]

   - Based on the way the repaired program P_{repair} is defined and addressed the approaches are categorized into:

     a. Generate and validate (see Figure 2.2). Generate and validate approaches are further broken down into:

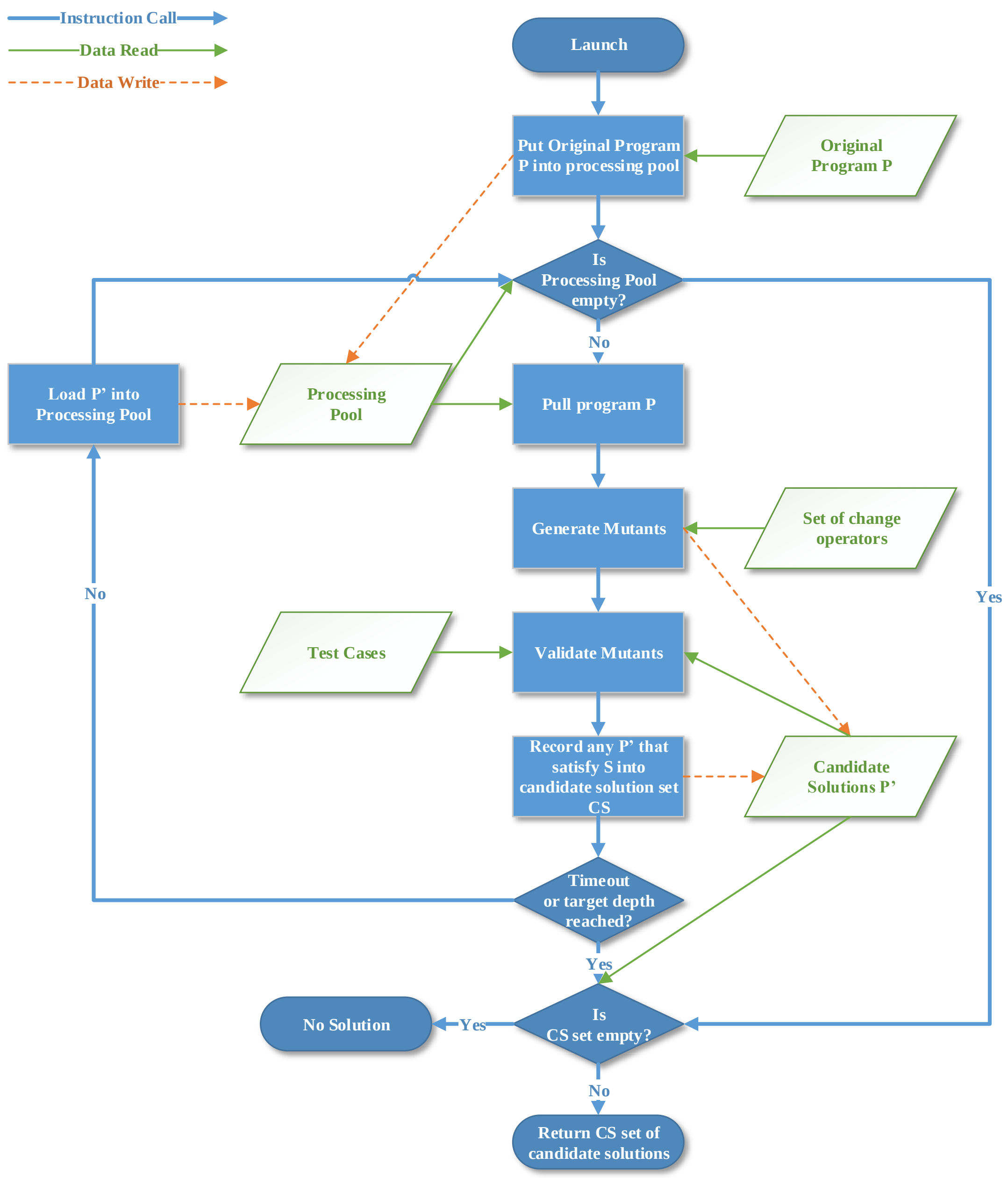
        1. Approaches performing an atomic change in one of the instructions in the code. [5, 23, 59, 60]

        2. Approaches applying pre-defined templates that consist of a set of atomic changes applied together in response to specific faults. [55, 61]

        3. Example-based approaches, which use existing fixes as source of possible change templates. [56-58]

    b. Semantics-driven, also known as correct-by-construction (see Figure 2.3.) Due to their nature these approaches are subdivided based on the class of issues that they attempt to address either being generic or specific to a certain type of faults, with the latter prevailing in the field due to exploiting a certain class of faults being a simpler task as compared to general program repair task formalization [6-8]

3. Fix recommendation approaches. A subset of program repair approaches that follow one of the patterns described above, but are geared toward integration with development environments to provide assistance and hints to developers at design time, as compared to being a standalone product applied during testing and post-production stages [62].



Instruction Call
Data Read
Data Write
Launch
Put Original Program P into processing pool
Original Program P
Is Processing Pool empty?
No
Load P' into Processing Pool
Processing Pool
Pull program P
Generate Mutants
Set of change operators
No
Yes
Test Cases
Validate Mutants
Record any P' that satisfy S into candidate solution set CS
Candidate Solutions P'
Timeout or target depth reached?
Yes
No Solution
Yes
Is CS set empty?
No
Return CS set of candidate solutions

Figure 2.2 An example of Generate and Validate process.

The survey further notes the overall immaturity of the field of software repair with only 46 percent of the approaches surveyed having any corresponding tool built and the majority (62%) of the tools available tending to focus on the same benchmarks, indicating a risk of overfitting a specific benchmark.

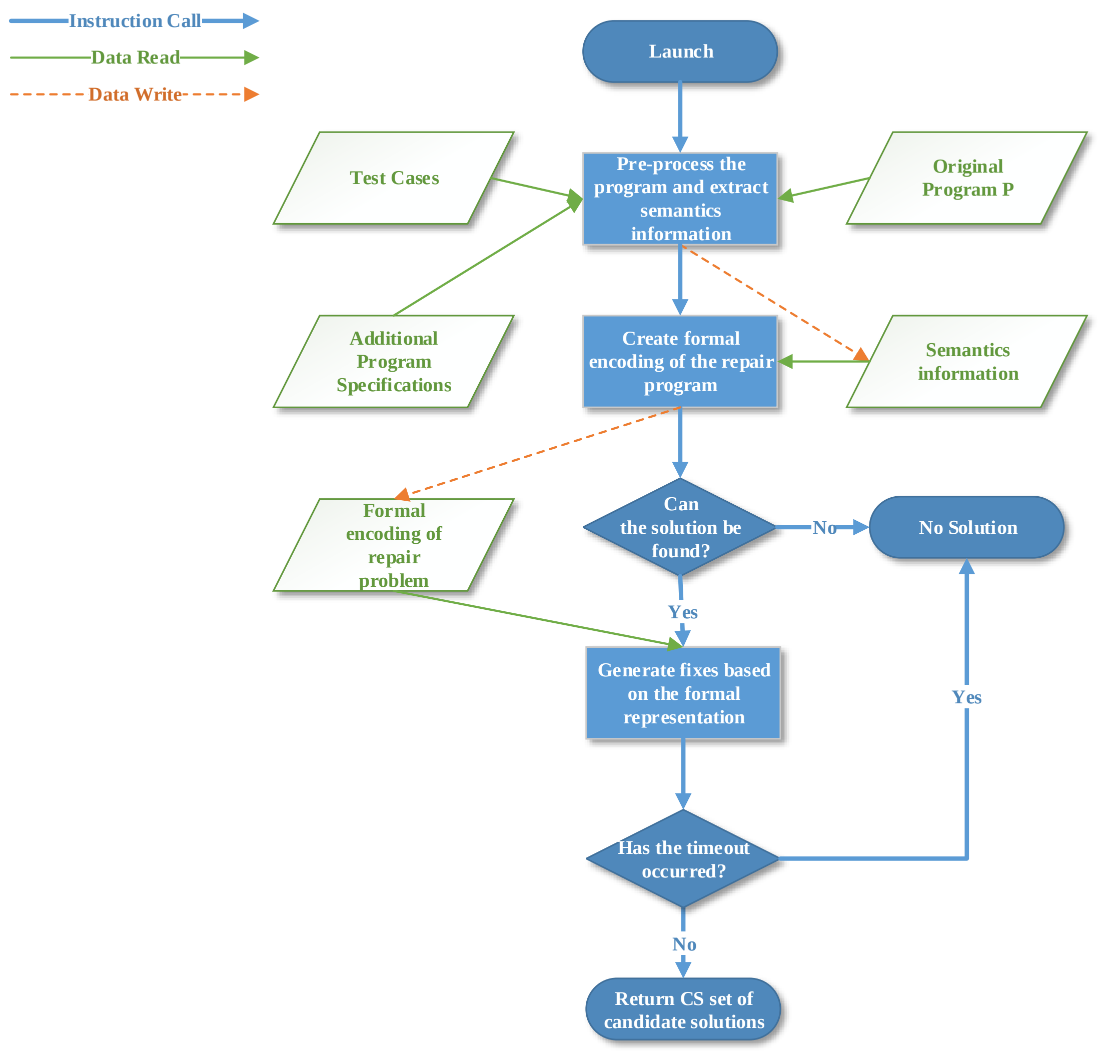

Instruction Call
Data Read
Data Write
Launch
Test Cases
Pre-process the program and extract semantics information
Original Program P
Additional Program Specifications
Create formal encoding of the repair program
Semantics information
Formal encoding of repair problem
Can the solution be found?
No
No Solution
Yes
Generate fixes based on the formal representation
Yes
Has the timeout occurred?
No
Return CS set of candidate solutions

Figure 2.3 An example of Semantic-driven repair process.

2.3 A Focus on Faults

As highlighted by Khaireddine et al. [63], despite focusing on a formal analysis of faults and fault repair, some recent approaches to program repair [64-68] do not fall neatly into the characterization of Gazzola et al. [4]. Here are some notable examples:

Rothenberg and Grumberg [64] introduce the concept of Must Location Set, which is a set of program locations that includes at least one program location from each repair for an observed failure. A fault localization technique is said to be a Must Algorithm if it returns a must location set for each observed program failure. Rothenberg and Grumberg develop a fault localization algorithm and use it in a program repair algorithm to help reduce the search space without loss of recall. The concept of must location is reminiscent of the concept of definite fault introduced by Mili et al. [69]: a definite fault in an incorrect program is a program part that must necessarily be modified if the program is to be corrected.

Lou et al. [70] critique the separation between two lines of research, namely fault localization and fault repair, and the fact that traditionally fault localization has been viewed as a means to achieve fault repair ends. They argue for a unified debugging approach, where fault repair is used to refine fault localization. They implement their approach in a tool, called ProFL, and highlight its performance on test benchmarks and on real software products.

Christakis et al. [71] present a static technique that analyzes an error trace in a program and identifies a small set of statements within the trace that may be modified to satisfy correctness conditions. Suspicious statements are ordered according to their likelihood of being the source of the observed failure.

The research by Li et al. [21], is a machine learning-based approach, which uses information about prior bug fixes to train ML models and use these models for automated code repair. This approach is implemented in a tool called DLFix. Although this tool is not covered by survey [4], it would likely fall under the same category of general brute-force techniques as the other machine learning-based approach R2Fix [58], which is included into the survey. The novelty of DLFix is that by using a two layer tree-based RNN and separating the tasks of learning the code context from learning the transformation the authors are able to mitigate the impact of the noise in the code, significantly improving the results.

Zhu et al. [72] also use a deep learning-based automated program repair approach, achieving improvements in the benchmark results by combining a new approach to the architecture of the encoder/decoder pair to better support small edits in the target code with introduction of placeholder generation to be able to properly support project-specific identifiers as a part of the patch being applied.

Shariffdeen et al. [73] look into the related problem of patch transplantation, automatically identifying fixed version of a common component in a different product and performing a context-aware adjustment of the applied patch, achieving better integration of the applied patch into the application being fixed.

Noda et al. [74] leverage a novel program dependence graphs-based approach to mine and learn systematic edit patterns (SEPs) from information about code changes between different code versions, detect locations, where such SEPs can be appliend in the target code, and apply the same changes that were captured in SEPs to the detected locations, using information about abstract syntax trees to guide the transplantation.

2.4 Bane of Program Repair: Too Much Generation, Too Little Validation

Khaireddine et al. [11] make an argument that repairing a program does not necessarily mean to make it (absolutely) correct, it only means to make it more-correct (in some sense) than it is. It is further claimed that the approximations of absolute correctness that the program repair methods rely on in absence of a clear definition of relative correctness (the property of a program to be more-correct than another with respect to a specification), result in too much generation and too little validation.

The reason for too much generation stems from the need to generate larger search spaces when relying on absolute correctness. Indeed, since relative correctness is expected to culminate in absolute correctness regardless of definition any pool of candidate repairs is more likely to have more candidates that are relatively correct than the ones that are absolutely correct. Conversely, this means that if absolute correctness is the chosen validation criterion, the probability of hitting a match on analyzing each candidate is lower, resulting in a larger search space. This problem is illustrated on Figure 2.4, where the star symbol represents the original (faulty) program, blue dots represent candidate repairs that are relatively correct but not absolutely correct, and red dots represent candidate repairs that are absolutely correct.

Figure 2.5 shows potential flows of step-wise validation for more correct programs under the same conditions.

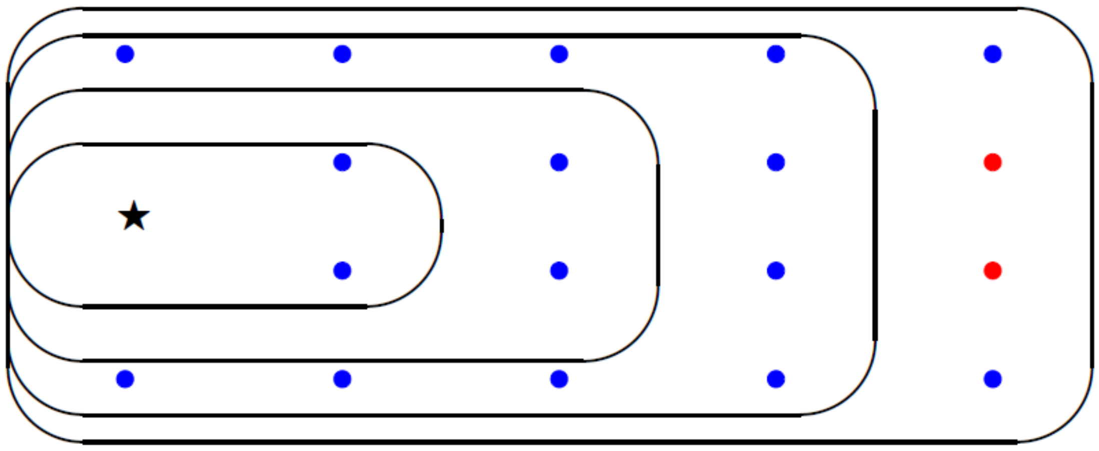


Figure 2.4 Absolute Correctness mandates larger spaces.

Source: [11].

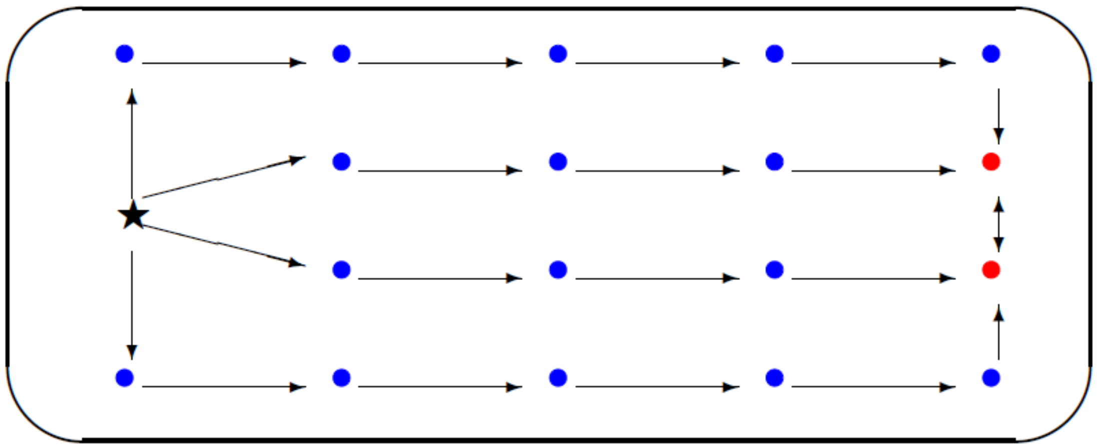


Figure 2.5 Relative Correctness enables step-wise validation.

Source: [11].

The reason for too little validation stems from program repair methods and tools not relying on a sound foundational definition of relative correctness in their validation approaches. In absence of such definition, the combination of criteria for patch validation that gets utilized instead exposes the program repairs methods and tools to a risk of poor efficiency, loss of precision, and loss of recall. Focusing on each of these risks separately:

- Obstacles to Efficient Validation. Defining the concept of a fault requires a concept of relative correctness; in the absence of the latter, it is impossible to define the former. As a result, program repair methods and tools have made failure remediation the focus of program repair, rather than fault repair; in other words, rather than focusing on repairing one fault at a time, they focus on remedying one failure at a time. The trouble with focusing on failure remediation is that the same failure can be the result of several faults, which must all be repaired simultaneously before the failure is addressed.

- Risk of Poor Recall. Several practices in the current methods and tools of program repair are prone to loss of recall. Here are three of them:

    - Testing for Absolute Correctness. In the absence of a concept of relative correctness, traditional methods and tools of program repair validate candidate repairs on the basis of absolute correctness. Absolute correctness is a sufficient but unncessary condition of relative correctness, hence the use of absolute correctness leads to loss of recall.

    - Search Space Pruning. In the face of vast search spaces, many methods and tools resort to a common device in such cases, namely search space pruning; though some program repair techniques take great care to only exclude from consideration candidates that are known to be invalid [64], not all methods are so deliberate. Pruning search spaces carries the risk of loss of recall, as we may be removing from consideration valid repair candidates.

    - The Use of Regression Testing. Most program repair methods and tools perform validation using two sets of test data, both of which have the form of sets of (input, output) pairs: a positive test suite T^+, which reflects correct behavior exhibited by the original program P, which we want candidate programs to preserve; a negative test suite T^-, which reflects behavior that the original program does not exehibit, and we want candidate programs to provide. The condition that a candidate program P' provide the behavior represented by T^- while preserving the behavior represented by T^+ is a sufficient condition of relative correctness of P' over P, but is not a necessary condition (since correct behavior is not unique). As such, this condition leads to a loss of recall.

- Risk of Poor Precision. Not only are some of the common validation methods prone to miss valid repairs, as we discuss above, some are prone to retrieve invalid repairs, as we discuss herein.

    - Fitness Functions. Several methods and tools rely on the use of a fitness function, which is supposed to reflect the validity of each candidate by virtue of some combination of the number of successful tests and unsuccessful tests of the candidate amongst the test suite$(T^+ \cup T^-)$. Regardless of how this function is defined, it creates an artificial total ordering between candidate repairs to represent what is essentially a very partial ordering; because it defines a total ordering, the fitness function ranks any pair of candidate repairs, even when they have no relative correctness relationship. Hence the use of fitness functions is prone to loss of precision.

    - Small Test Suites. The size of search spaces creates a strong incentive to reduce the size of test suites, so as to inspect the largest possible number of candidate repairs per unit of time. Using small test suites causes a loss of precision, since it increases the likelihood that a repair candidate passes the tests without being a valid repair.

2.5 Premises of the Relative Correctness-based Approach

The relative correctness approach is based on the following premises:

- To repair a program does not mean to make it absolutely correct; it only means to make it more correct than it is.

- Any definition of relative correctness ought to satisfy some litmus properties that are introduced and justified in the next chapters.

- Program repair methods ought to be validated by showing that they enhance relative correctness.

- Any program repair method ought to proceed by a variation on the general theme: enhance relative correctness until absolute correctness is achieved.

- For the sake of precision, patch validation ought to use large test suites, including large negative test suites (i.e. data sets where the original program fails).

CHAPTER 3

BACKGROUND FOR A THEORETICAL APPROACH

3.1 A Critique: The Need for the Theory of Relative Correctness

In [4], Gazzola et al. conclude that “it is important to improve the maturity of the field and obtain a better understanding of useful strategies and heuristics”. In line with this conclusion, in [63] Khaireddine et al. the argument is made that one of the most fundamental steps that would help with pushing the industry towards maturity is through development of theoretical foundations, based upon the concept of relative correctness, i.e. "the property of a program to be more-correct or strictly more correct than another with respect to some specification". While the traditional approach is Boolean, defining a program as either correct or incorrect (absolute correctness) the relative correctness introduces a partial ordering among candidate programs with absolutely correct programs being the maximal elements of such ordering, thus allowing to redefine the process of program repair as an iterative process going over a sequence of increasingly more correct states eventually achieving absolute correctness.

In the world of computer science it is sometimes the case that the practical approaches are being created and utilized long before the theory explaining them and structurizing the approaches offered by them is being drawn. For example, the programming approaches utilizing high level programming languages have emerged in the mid to late nineteen fifties with the emergence of such languages as Fortran, Cobol, and Algol (with the first two seeing heavy usage up to this day) [75, 76]; yet, the first theories of program correctness providing theoretical foundation for the programming approaches utilized have only emerged in the late nineteen sixties [77, 78] and it took a decade for such theories to reach maturity and be turned into methodologies for deriving correct-by-design programs [79-81]. The current state of the field of programming repair with successful research producing sophisticated engineering solutions without a formal theory suggests a similar situation and highlights the need to have relative correctness providing a theoretical basis of programming repair in the same way as the traditional (absolute) correctness provides the theoretical basis of program derivation from a specification (programming); the presence of such theory may enhance the state of the art/ practice in the field of program repair, with the theoretical implication being the usage of relative correctness for patch validation and eventually for patch generation.

Performing an abstraction on the methods described in Gazzola et al. [4] the following arguments on the need of the relative correctness theory apply:

1. In absence of a definition of relative correctness, the absolute correctness, by which program repair methods perform patch validation requires transformation to be done in one shot and is therefore useful only within striking distance of absolute correctness. Relative correctness allows to approach the task of transforming a program gradually over several steps of still faulty, but more correct programs, giving an efficient approach to addressing faults at arbitrary depth, repairing a program P to obtain a program P’, where P’ is more-correct than P without being absolutely correct.

2. Same logic applies to the paradigm shift from remedying a failure to removing a fault. Most program repair methods rely on negative test data to drive program modification to remedy the failure represented by the negative test data. If the observed failure is not due to a single-site fault, but rather stems from the combination of several faults, that approach means that in order to make a switch from the program being absolutely incorrect, to program being absolutely correct all the faults responsible for the failure have to be correctly located and remedied, leading to unbounded combinatorial explosion due to imperfect fault generation and fault localization of the tools being utilized. Introducing relative correctness allows to define the concept of elementary fault, which, in turn enables to define program repair as a step-wise repair of elementary faults, rather than

the brute force transformation of an incorrect program into an absolutely correct one. The benefit of such definition change is two-fold:

- The criterion of patch validation can be changed from “Is the program’s failure corrected?” to “Is the new program (relatively) more correct?” making it possible to achieve positive result even if some of the transformations needed to remedy the observed failure are not known to the tool being utilized. An imperfect tool driven by the concepts of relative correctness can still eliminate the faults that it knows, reducing the amount of work that is needed to be done on the remaining ones.

- Enhancing the correctness repeatedly will over a sufficient number of iterations remove enough faults to remedy the observed failure. Combined with running fault localization after each elementary fault removal, it would allow addressing faults as they appear instead of trying to guess the right combination of fixes from the beginning bringing higher level of granularity and precision in targeting the next fault removal.

3. When traversing the field of candidate repairs, the two commonly used approaches are repair methods checking that candidates preserve the correct behavior of the original program (represented by positive test data) or maximizing some user-defined (or system-defined, by default) fitness function. Both approaches have major deficiencies:

    - Preserving correct behavior is unusable in driving patch generation, being unable to generate oracles on the next step and being used only for passive validation. Even for passive validation, with correctness preservation being a sufficient condition of relative correctness, but not a necessary one, it excludes the candidates that preserve correctness without preserving the correct behavior leading to a loss of recall.

    - The fitness function-based approaches that do not account for relative correctness carry the risk of loss of precision as they generate oracles focused on candidates that are more reliable, but not necessarily more correct than the original program. These candidates can be more reliable, not because they are more-correct, but because they succeed for inputs that are more likely to occur.

    Relative correctness provides foundation for a more efficient candidate repair space traversal, with the program being both more reliable being a necessary condition and preserving the correctness as a sufficient condition, thus, essentially, being the next step in evolution of program repair approaches.

3.2 Mathematics for Program Repair

3.2.1. Relational Mathematics

In order to explain the mathematical foundation of the relative correctness theory, the following concepts are briefly introduced, as described in [31]:

Given a program p that operates on some variables x and y, let the *space* of p be the set S of all the values that the aggregate of variables <x, y> may take; elements of S are called *states of the program*, and are usually denoted by lower cases. A *relation on set* S is a subset of S×S; constant relations on a set S include the *empty relation* ($\emptyset$), the *identity relation* (denoted as I and defined as I={(s,s)|s ∈ S}, meaning that each element is related to itself only) and the *universal relation* (denoted as L, defined as L=S×S and meaning that each element of set is related to every element of set); operations on relations include the set theoretic operations of union, intersection, difference and complement; other operations include the *product* of two relations (denoted by R◦R', or RR' for short), the *converse* of a relation (denoted as $\hat{R}$ and defined as $\hat{R}$ = {(s,s')|(s',s) ∈ R}) and the *domain* of a relation (denoted as dom(R) and defined as dom(R) = {s|∃s' : (s,s') ∈ R}). The *pre-restriction* of relation R to set T is denoted by $_T\backslash R$ and defined as $_T\backslash R$ = {(s,s')|s ∈ T ∧ (s,s') ∈ R}. A relation R is said to be reflexive iff I⊆R, symmetric iff R⊆$\hat{R}$, antisymmetric iff R∩$\hat{R}$⊆I, and transitive iff RR⊆R. A relation R is said to be deterministic iff $\hat{R}$R⊆I. A relation R is said to be *deterministic* (or: *a function*) iff R$\hat{R}$ ⊆ I, and *total* iff RL = L. A relation R is said to be *a vector* iff RL = R; vectors have the form R = A × S for some subset A of S and are used here as relational representations of sets. In particular, it should be noted that RL, which is used as a relational representation of the domain of R, can be written as dom(R) × S. For the sake of convenience, symbols representing a set (say T) and the vector (T × S) that represents the same set, in relational form are used interchangeably. Hence, for example, the restriction of relation R to set T can be written as $T \cap R$, where T is interpreted as a vector. Being a well-known property of functions it is admitted without proof that if F and G are functions then $F = G$ iff $F \subseteq G$ and $GL \subseteq FL$.

3.2.2 Program Semantics and Correctness

Adopting the definitions by Khaireddine et al. [31, 63], whereby given two relations R and R', R' refines R ($R' \sqsupseteq R$) if and only if $RL \cap R'L \cap (R \cup R') = R$ and given a program p on space S written in a C-like notation, defining the function of p (denoted by P) as the set of pairs(s, s') such that if program p starts execution in state s it terminates in state s', the program and its function can be referred to by the same name, P, when no ambiguity arises, the following definitions can be given:

Definition 1: Given a specification R on space S, a program p is said to be *correct* on the space S with respect to specification R if and only if its function P refines R.

This definition is equivalent to traditional definitions [79, 83, 84] of total correctness with respect to prespecification $\varphi(s)$ and postspecification $\psi(s)$ for some s_0:

$\varphi(s) \equiv s \in dom(R) \wedge s = s_0$.

$\psi(s) \equiv (s_0,s) \in R$

$\forall s : \varphi(s) \Rightarrow s \in dom(P) \wedge \psi(P(s))$

The following proposition can be made due to Mills et al. [85] and is offered here without proof:

Proposition 1: Program P is correct with respect to specification R on the space S if and only if $(R \cap P)L = RL$.

Definition 2: The set $(R \cap P)$ is called the *competence domain* of P with respect to R and is the set of initial states on which P behaves according to R.

Proposition 2: Given a specification R and a program P on space S, program P is correct with respect to R if and only if the following condition holds:

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

where $\varphi(s) \equiv s \in dom(R) \wedge s = s_0$ and $\psi(s) \equiv (s_0,s) \in R$ for some s_0.

Proof:

Proof of Sufficiency. Replacing φ() and ψ() by their expressions, the condition of the proposition can be simplified into:

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

Since (s,P(s)) is by definition an element of P, this can be written as:

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

By definition of domains it is inferred that:

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

Since $dom(R \cap P) \subseteq dom(P)$ it is inferred that:

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

By set theory, we infer: $RL \subseteq (R \cap P)L$; since the inverse inclusion is a tautology, we infer $(R \cap P)L = RL$.

Proof of Necessity. Since $(R \cap P)L \subseteq RL$ is a tautology, the condition of this proposition is equivalent to $RL \subseteq (R \cap P)L$, which is interpreted as follows:

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

{Interpreting the definition of domain}

$\forall s : s \in dom(R) \Rightarrow \exists s' : (s, s') \in (R \cap P)$

{P is deterministic}

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

{substitution}

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

{Interpreting the definition of domain}

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

{substituting φ() and ψ()}

$\forall s : \varphi(s) \Rightarrow s \in dom(P) \wedge \psi(s) \in R$.

If $s \in dom(R)$ is interpreted as s satisfies the precondition, $s \in dom(P)$ as execution of P on s terminates normally, and $(s, P(s)) \in R$ as the final state (P(S)) satisfies the postcondition, then this formula can be interpreted as: for any initial state that satisfies the precondition, program P terminates normally and returns a final state that satisfies the postcondition: this is the exact definition of total correctness, as given in traditional sources. QED

With the provided definitions and propositions, the following definition of relative correctness can be introduced:

Definition 3: For deterministic programs P and P' a program P' is said to be **more correct** than P with respect to specification R if and only if $(R \cap P')L \supseteq (R \cap P)L$ (and, correspondently **strictly more correct**, iff $(R \cap P')L \supset (R \cap P)L$).

It should be noted that **more correct** is in fact **more-correct-than-or-as-correct-as**, however, for the sake of convenience, a shorter version is utilized, with the stricter clause without the as-correct-as portion labeled as strictly more correct. Khaireddine et al. [63] provide the following proof of this definition:

Assuming the same notation, this definition can be expanded further for non-deterministic programs, as follows [32, 86]:

Definition 4: For non-deterministic programs P and P', P' is more-correct than P with respect to R ($P' \sqsupseteq_R P$) if and only if $(R \cap P)L \subseteq (R \cap P')L \land (R \cap P)L \cap \bar{R} \cap P' \subseteq P$, which can be interpreted as: P' is more-correct than P with respect to R if and only if it has a larger (or equal) competence domain, and for the elements in the competence domain of P program P' has fewer (or the same number of) states that violate R than P does. In other words, a program P' is more-correct than a program P with respect to R if and only if the set of states on which P' violates R is a subset of the set of states on which P violates R.

3.3 Absolute Correctness and Relative Correctness

Validation of the adopted definition of relative correctness requires verification of several relational properties that such definition must satisfy, which, based on Diallo et al. [86] are:

- **Relative correctness is transitive, reflexive, but not antisymmetric.** Indeed, transitivity and reflexivity stem directly from the $(R \cap P')L \supseteq (R \cap P)L$ portion of the definition of relative correctness (where relative correctness is indeed reflexive and transitive due to reflexivity and transitivity of set inclusion [63]), however, the non-antisymmetricity of relative correctness stems from the fact that $(R \cap P)L = (R \cap P')L$ does not necessarily imply P = P'. It can be observed from the following scenario: two functions P and P' may satisfy $(R\cap P)L=(R\cap P')L$ while P and P' are distinct. A combination of $R=\{(0,1),(0,2)\}$, $P=\{(0,1)\}$ and $P'=\{(0,2)\}$ can be considered an example of such scenario. These properties can be expressed [63] as $\sqsupseteq_R \circ \sqsupseteq_R \subseteq \sqsupseteq_R$, $I \subseteq \sqsupseteq_R$, $\sqsupseteq_R \cap \sqsubseteq_R \not\subset I$.

- **Relative correctness culminates in absolute correctness.** Relative correctness culminates in absolute correctness, as, by definition of absolute correctness, an absolutely correct program P satisfies the condition $(R \cap P)L = RL$, hence its competence domain is maximal (hence a superset of the competence domain of any candidate program). The necessity proof looks as follows: given a specification R and a program p’ on space S, p’ is absolutely correct with respect to R, iff p’ is more-correct with respect to R than any candidate program p on S - an absolutely correct program p’ refines the entire specification R and any other program p" on space S built in regards to specification R would either refine it in its entirety or refine only a subset of it, thus allowing program p’ to meet the definition of being more correct than any other program p". The sufficiency proof also holds - if program p’ is more correct than any other program on space S in regards to specification R, it should refine the entire specification R, otherwise, there would exist a program p" refining a larger portion of specification R than p’, which, by definition would mean that p" would be more correct than p’, which contradicts the claim of p’ being more correct than any other p". This property can be recorded [63] as $P \sqsupseteq R \Leftrightarrow (\forall P : P' \sqsupseteq_R P)$.

- **For any specification, refinement is equivalent to relative correctness.** Indeed, program p’ refining p means that p’ can do everything that p does and, for the case of strict relative correctness, can do it better (or that p’ matches every specification $r \in R$ that p matches and in addition there exists specification $r' \in R'$, $r' \notin R$ that p does not meet). In a formal way it can be proven as follows [63]:

    - Proof of Necessity. If $P' \sqsupseteq P$ then (because P and P’ are both functions) $P' \supseteq P$, whence (by monotonicity of intersection and domain) $(R \cap P')L \supseteq (R \cap P)L$.

    - Proof of Sufficiency. From $(\forall R : (R \cap P')L \supseteq (R \cap P)L)$, by letting $R = P$, $(P \cap P')L \supseteq PL$ is inferred. This, in conjunction with the set theoretic identity ($P \cap P' \subseteq P$), yields (because ($P \cap P'$) and P are both functions), $P' \cap P = P$; from which, by set theory $P' \supseteq P$ is inferred; given that P’ and P are both function, this yields $P' \sqsupseteq P$. QED

- **Relative correctness is a sufficient condition of higher reliability, but not a necessary one.** Higher reliability is a stochastic property, and Relative Correctness is a logical/functional one. A program P’ has a higher reliability than program P iff P’ has a higher probability of performing as per specification R than program P does. The fact that P’ meets more of the specification R than P indeed makes it more reliable, but the opposite is not always true, as a more reliable program P’ can fail to meet some of the specifications that P does. This property can be written [63] as: $P' \sqsupseteq_R P \Rightarrow (\forall \theta() : \rho_R^{\theta()}(P') \geq \rho_R^{\theta()}(P))$ and can be given a formal proof looks as follows: Given a specification R and discrete probability distribution $\theta()$ on dom(R), s is a random element of dom(R) selected according to probability distribution $\theta()$. Execution of a program P on s is successful iff s is in the competence domain of P with respect to R. Hence the reliability of P with respect to R and $\theta()$ can be written as: $\rho_R^{\theta()}(P)=\sum_{s \in dom(R \cap P)}\theta(s)$. Clearly, larger competence domains yield greater values for $\sum_{s \in dom(R \cap P)} \theta(s)$, regardless of how θ() is defined. Therefore: $P' \sqsupseteq_R P \Rightarrow (\forall \theta() : \rho_R^{\theta()}(P') \geq \rho_R^{\theta()}(P))$

CHAPTER 4

INGREDIENTS OF A THEORY BASED PROGRAM REPAIR ALGORITHM

4.1. Faults and Elementary Faults

In the work by Avizienis et al. [87] and Laprie [88-90] the fault is defined as adjudged or hypothesized cause of an error. This definition relies on an insufficiently defined concept of error and highly subjective concepts of adjudging and hypothesizing. A more detailed definition, however, should be related to the level of granularity, at which the faults are being isolated. Following Gazzola et al. [4], the following two definitions that determine the scale of faults are adopted:

- A syntactic atom in program P is a fragment of source code of P at the selected level of granularity.
- An atomic change in program P is a pair of source code fragments *(a, a')* such that *a* is a syntactic atom in P and *a'* is a code fragment that can be substituted for *a* without violating the syntactic integrity of P.

Expanding upon the concepts of relative correctness and competence domain, Khaireddine et al. [11, 91-93] introduce the following definitions:

Definition 1: Given a program P and a specification R on the space S, a *software failure* of program P with respect to specification R is an event that occurs if and only if execution of P on some initial state s violates the premise that P is correct with respect to R.

Execution of P on state s violates the assumption that P is correct with respect to R if and only if P either fails to terminate on s, or it does terminates but the final state s' fails to satisfy the condition (s,s') ∈ R.

Definition 2: A *feature* of program P with respect to give level of granularity is any part of the source code, including non-contiguous part that is appropriate to cover all code related to software failure.

Definition 3: Given a specification R and a program P, a *fault* in program P is any feature f that admits a substitute f' such that the program P' obtained from P by replacing f with f' is strictly more correct than P.

Definition 4: A *fault removal* or *fault repair* in P is a pair of features (f, f') such that f is a feature in P and program P' obtained from P by replacing f with f' is strictly more correct than P.

This definition can be also expanded as follows: Let p be a program on space S and R be a specification on S, let f be a fault in p, and let f' be a substitute for f. The pair(f, f') is a (monotonic) fault removal iff the program p' obtained from p by substituting f by f' is strictly more-correct than p.

Definition 5: An *elementary* or *unitary fault* f in program P with respect to specification R is a fault such that no part of it is a fault, in other words, an elementary fault cannot be subdivided into independent faults.

This definition means that all single-site faults (containing just a single atom) are elementary, but in case of multi-site faults they are considered elementary iff no subset of their elements is a fault. The number of atoms in a unitary fault is called the multiplicity of the fault. The concepts of unitary fault and multiplicity can be illustrated with the following example:

Given space S, specification R on space S and program P defined as:

S = {float x; float a[N+1]}

$R = \{(s, s')|x' = \sum^{N}_{i=1} a[i]\}$

P = {int i=0; x=0; while (i<N) {x=x+a[i];i=i+1;}}

Substituting the feature f = (0, <) with the feature f' = (1, ≤) yields a strictly more correct program P':

P' = {int i=1; x=0; while (i<=N) {x=x+a[i]; i=i+1;}}

Hence f is a fault. In order to determine whether f is a unitary fault with multiplicity of 2 or a set of two unitary faults with multiplicity of 1 the competence domains of P_1' and P_2' programs that result from individual application of the constituent atomic faults have to be verified. The programs P_1' and P_2' are therefore

P_1'= {int i=1; x=0; while (i<N) {x=x+a[i]; i=i+1;}}

P_2'= {int i=0; x=0; while (i<=N) {x=x+a[i]; i=i+1;}}

Their competence domains are

CD = {s|a[0] = a[N]}.

CD_1' = {s|a[N] = 0}.

CD_2' = {s|a[0] = 0}.

As no inclusion relation can be established between CD and CD_1' P_1' is not more correct than P. In a similar way, since there is no inclusion relation between CD and CD_2' P_2' is not more correct than P and, therefore, f is the case of a single unitary fault of multiplicity 2 (Figure 4.1).

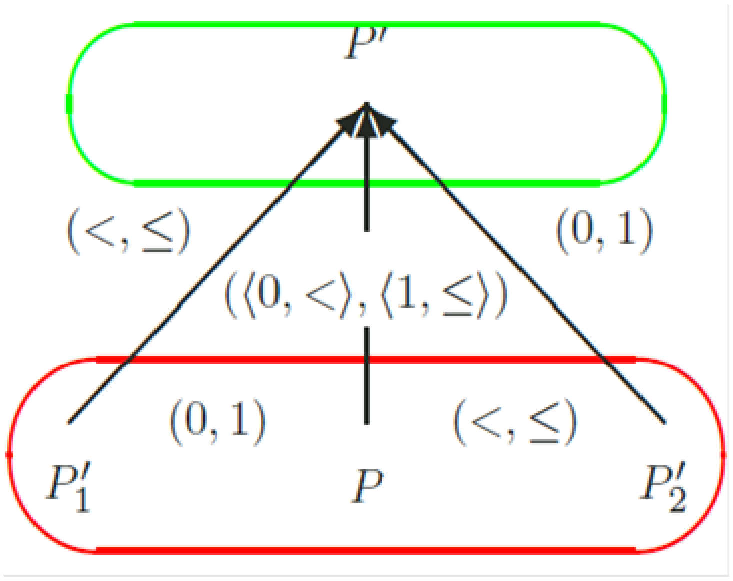


Figure 4.1 An elementary fault of multiplicity 2. Although P_1' and P_2' are both modifications of P, there are no arrows as neither P_1' nor P_2' are more correct than P.

Source: [63].

4.2. Fault Density, Depth and Multiplicity

Definition 6: Given a program P and a specification R on space S, the number of unitary faults in P is the *fault density* of P with respect to R and the minimal number of unitary fault repairs that separate P from a correct program is the *fault depth* of P with respect to R.

A program having N unitary faults does not necessarily need N unitary fault repairs; these two metrics are distinct and whereas with each unitary fault repair the fault depth is decreasing, fault density can vary arbitrarily.

Reusing the same space S, specification R and starting program P from the example that was considered in Definition 5 it can be observed that in addition to the fault $f_1 = (0, <)$ with multiplicity 2 that gets repaired through substitution $f'_1 = (1, \leq)$ generating program P_1':

P_1' = {int i=1; x=0; while (i<=N) {x=x+a[i]; i=i+1;}}

There is also another fault f_2=(i) with multiplicity 1 that gets repaired through substitution with f'$_2$=(i+1) yielding a different program P_2':

P_2' = {int i=0; x=0; while (i<N) {x=x+a[i+1]; i=i+1;}}

That is also absolutely correct to the original specification R. Since there are two faults, the fault density is 2 (Figure 4.2 - two possible ways to get the program corrected), whereas the fault depth is 1, as only one unitary fault correction is needed.

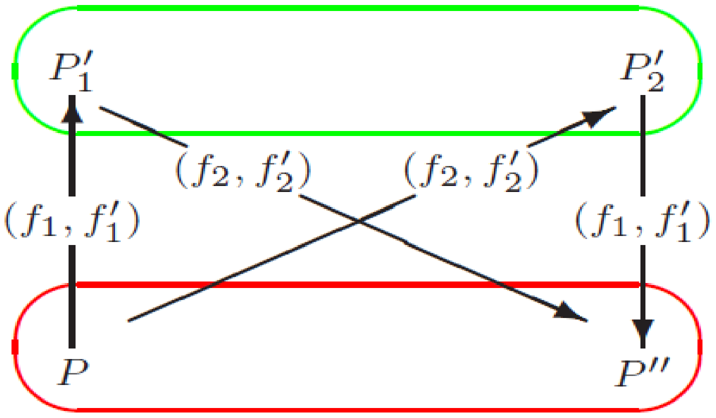


Figure 4.2 Fault Density (=2) vs. Fault Depth (=1).

Source: [63].

4.3. Fault Repair vs. Failure Remediation

As noted before, unlike most of the modern research that uses failure remediation for program repair, this research focuses on fault repair. In order to illustrate the difference, the following definition needs to be introduced.

Definition 7: A *unitary increment of correctness* is a step or a set of steps removing a single fault [94].

This concept is illustrated on Figure 4.3, where going from P0 to P4 requires two steps, but leads to a single unitary increment of correctness, since the intermediary state P2, while required for the next step does not enhance correctness, hence does not qualify as a fault removal. A real life example of the scenario shown on Figure 4.1 would be fixing a program P with two faults with one of them being a simple serialization fault in an object, where one of the attributes A prevents serialization due to being improperly configured and the other being wrong relational operator used somewhere in the code. In such scenario P1 might remove the attribute, P3 attempt to fix the relational operator, while P2, P4 chain would attempt to actually address the root cause of the issue with serialization. As long as the test suite T doesn't specifically check for the attribute A to be present in object - each path would be considered as a valid solution.

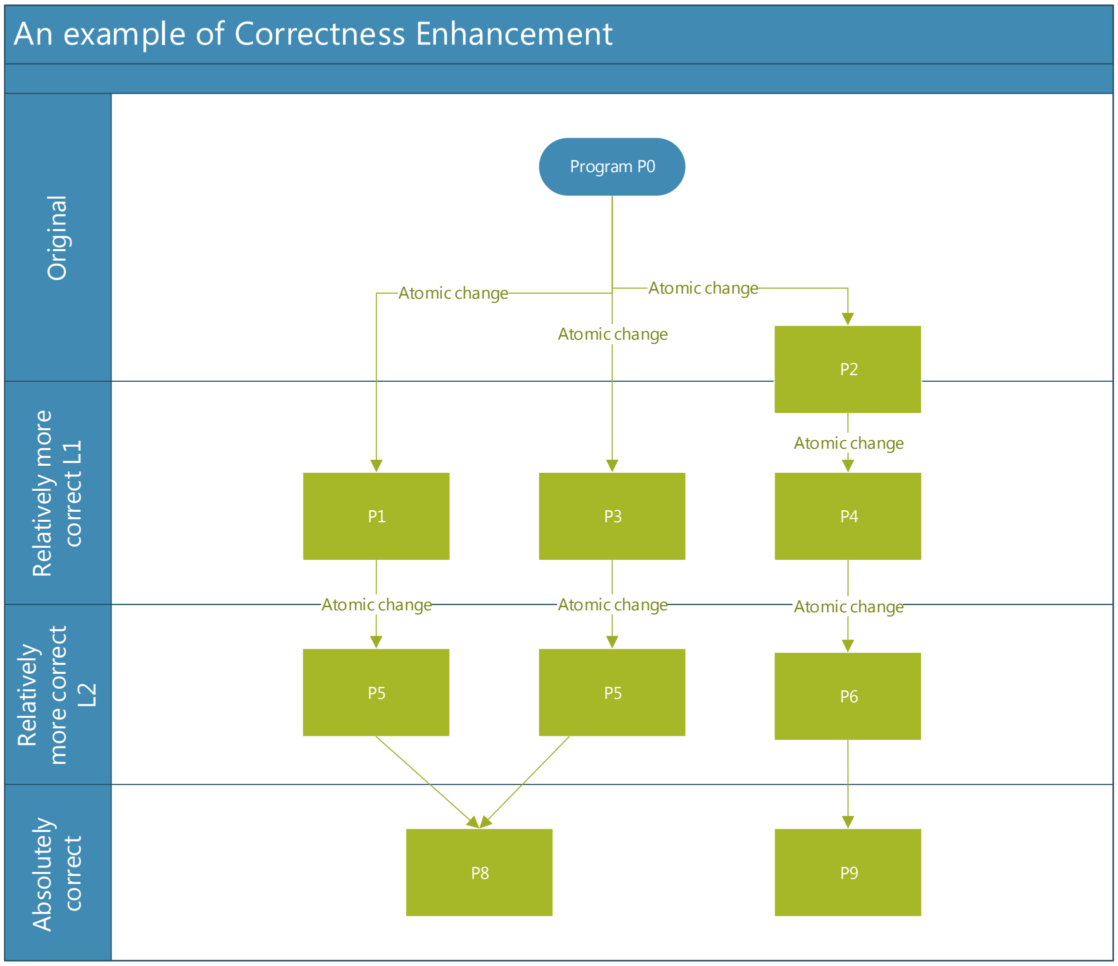


Figure 4.3 A generic example of correctness enhancement.

With incremental enhancement of correctness defined the contrast between failure remediation and program repair can be highlighted as follows: Given program P that fails on input x, a traditional failure remediation approach would simply try to make it correct at x, whereas the incremental correctness enhancing approach would mark P as incorrect and go through a chain of strictly-more-correct programs fixing faults as they appear until the competence domain of program P^n covers x (as shown on Figure 4.4.)

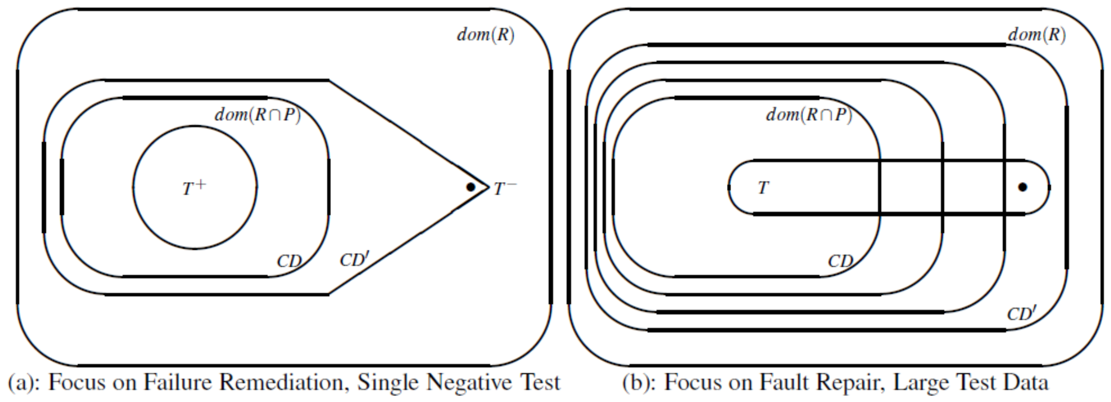

dom(R)
dom(R∩P)
T+
T−
CD
CD′
(a): Focus on Failure Remediation, Single Negative Test
dom(R)
dom(R∩P)
T
CD
CD′
(b): Focus on Fault Repair, Large Test Data

Figure 4.4 Failure Remediation vs. Fault Repair.

Source: [11].

CHAPTER 5

A GENERIC ALGORITHM FOR PROGRAM REPAIR

5.1 General Principle

The process offered to address program repair summarizes what was described in Chapters 3 and 4 so far. It is generic in the sense that it outlines a general process for selecting repair candidates, but does not specify how repair candidates are generated; hence it can be instantiated for any given patch generation method. The algorithm can be succinctly defined as **enhance relative correctness until either the absolute correctness is achieved, the user-set limit is reached or the algorithm determines that it can no longer enhance relative correctness (due to inadequate patch generation).**

5.2. An Infrastructure of Oracles

Given a program P' on space S, with its initial state being s and final state being s', the oracle is a binary predicate in(s, s'), which can take several forms depending on the property being tested about P'. It can be subdivided into following cases:

1) Oracle of absolute correctness with respect to R.

2) Oracle of relative correctness over a program P with respect to a specification R

3) Oracle of strict relative correctness over a program P with respect to a specification R.

5.2.1. Absolute Correctness With Respect to a Specification R.

Definition: Given a specification R on space S, the *oracle for absolute correctness* with respect to R is denoted as $\Omega(s, s')$ and defined by:

$\Omega(s, s') \equiv (s \in dom(R) \Rightarrow (s, s') \in R)$.

Proposition: If a program P satisfies the condition $\Omega(s, P(s))$ for all s in S then it is absolutely correct with respect to R.

In practice, since it is nearly impossible to check $\Omega(s, P(s))$ for all s in S, as even for simplest programs such full testing would take an unacceptable amount of time [45], it is checked for a bounded size test data T. Hence the predicate $\Omega_T(P')$ is defined as:

$\Omega_T(P')\equiv(\forall s\in T: \Omega(s, P'(s)))$

The program P' is absolutely correct with respect to ${}_{T\backslash}R$ if and only if it satisfies this predicate [63].

5.2.2. Relative Correctness Over a Program P With Respect to a Specification R.

Definition: Given a specification R on space S and a program P on S, the *oracle for relative correctness* over program P with respect to R is denoted by $\omega(s, s')$and defined by:

$\omega(s, s')\equiv(\Omega(s, P(s))\Rightarrow\Omega(s, s'))$.

Proposition: A program P' is more-correct than program P with respect to R if and only if $\omega(s, P'(s))$ holds for all s in S.

This formula stems readily from the definition of relative correctness. Again, in practice, only a bounded size data set T is checked since checking $\omega(s, P'(s))$ for all s in S cannot be done. Therefore, the predicate $\omega_T(P')$ is defined as:

$\omega_T(P')\equiv(\forall s\in T:\omega(s, P'(s)))$

The program P' is said to be more correct than P with respect to ${}_{T\backslash}R$ if and only if the execution of P' on every element of T satisfies oracle $\omega(s,s')$ [63].

5.2.3. Strict Relative Correctness Over a Program P With Respect to a Specification R.

Definition: Given a specification R on space S and a program P on S, the oracle of strict relative correctness over program P with respect to R is denoted by $\sigma(s, s')$ and defined as:

$$\sigma(s, s') \equiv (\forall s \in S : \omega(s, P'(s))) \wedge (\exists s \in S : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$$

Proposition: A program P' is strictly more-correct than a program P with respect to R if and only if P' is more-correct than P, and there exists at least one element s in S such that the condition $\Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))$ is satisfied.

Similar to absolute correctness case, $\Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))$ is checked only for a bounded size dataset T, the predicate $\sigma_T(P')$ is defined as:

$$\sigma_T(P') \equiv (\omega T(P') \wedge (\exists s \in T: \Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))))$$

The program P' is strictly more correct than P with respect to $_{T\backslash}R$ if and only if for the program P' the oracle $\sigma_T(P')$ returns true [63].

5.3 A Generic Algorithm

Due to its generic nature, the algorithm applies to programs of arbitrary fault depth, because it does not test for absolute correctness, but rather tests for relative correctness over the base program. It is based on an elementary routine that performs a unitary increment of correctness enhancement; removing one elementary fault at a time. Because elementary faults may be multi-site, it attempts to enhance correctness by single-site features, then double-site features, etc., until it succeeds or reaches a user-imposed threshold of fault multiplicity. The inputs to this algorithm are:

1. The specification R with respect to which correctness is judged in the form of a correctness oracle - a Boolean function between initial states and final states.

2. The faulty program, P.

3. The test data T that would be used to test for absolute correctness and relative correctness.

4. The threshold of multiplicity (M) to be considered for multi-site elementary faults. When restricted to single-site faults, M should be set to 1.

The candidate patches are assumed to be organized as a set of patch streams of increasing multiplicities, which we name, respectively, PS(1), PS(2), ..., PS(M). Each patch stream PS(m) is an ordered sequence, supporting application of sequence operators head() and tail(), referring respectively to the first element, and the remainder of the sequence. It is assumed that patch generator is providing the following functions:

- o MorePatches(P,m), a Boolean function that returns true if and only if there remains more patches of P of multiplicity m.
- o NextPatch(P,m), which returns the next element of PS(m), for $1 \leq m \leq M$.

As shown by Khaireddine et al. [11, 63, 94], the algorithm would look as follows:

```
     void ProgramRepair(program P, specification R, testdata
T, int M) {
          bool incremented=true;
          while (incremented && not abscor(P))
          {
               P = UnitIncCor(P, R, M)
          }
     }

     programtype UnitIncCor (programtype P, specification R,
int M) {
          int mult=1;
          incremented = false;
          while (not incremented && mult <= M)
          {
               programtype Pp=P;
               initPatches(mult);
               while (not somecndtn (Pp, P) && MorePatches(P,
mult))
```

```
                {
                     Pp = NextPatch(P, mult);
                }
                if somecndtn (Pp, P)
                {
                     incremented = true;
                     return Pp;
                }
                else
                {
                     mult = mult+1;
                }
           }
      }
```

5.4 Assessment of Precision and Recall

Since the generic algorithm is merely an iterative application of UnitIncCor(), the focus of propositions will be on UnitIncCor(). According to Khaireddine et al. [63] the following propositions are offered here with proof provided separately in Appendix A:

Proposition 5.4.1: Function UnitIncCor() has perfect recall, in the sense that if the patch stream has a program that is strictly more-correct than P, then UnitIncCor() will return in Pp a program that is strictly more-correct than P.

It should be noted that UnitIncCor(), as proposed does not retrieve all the patches that are strictly more-correct than P; it only retrieves the first patch that it encounters. Hence, the only guarantee that can be provided is that if there exists a patch Q in the patch stream that is strictly more-correct than P, then UnitIncCor() will necessarily return in Pp a program that is strictly more-correct than P (this could be Q or it could be another patch that it encounters before Q).

Proposition 5.4.2: Function UnitIncCor() has perfect precision, in the sense that if incremented is set to true then Pp is strictly more-correct than P.

The perfect precision and perfect recall that the relative correctness algorithm has based on the propositions above make it a better approach to patch validation than the approaches prone to loss of precision, loss of recall or both that are traditionally utilized by most existing program repair tools.

5.5 Introducing Parallelism

The UnitIncCor() and ProgramRepair() functions that were discussed above are aimed at sequential execution. Optimization of the algorithm for parallel machines requires changes to the algorithm to look as follows:

```
     void ParallelProgramRepair(programtype P, specification
R, testdata T, int M, bool stopOnAbsolute) {
          std::list<wrapperprogramtype> controllist;
          controllist.push_back(new
wrapperprogramtype(P,1));
          std::list<wrapperprogramtype>::iterator it;
          it=controllist.begin();
          while (it!=controllist.end())
          {
               int m=*it.getmultiplicitylevel();
               P=*it.getprogram();
               patchLocalization=localize(P,R,T);//fault
localization
               //If   fix   loci   localization   is   used,
patchLocalization=localize(P) is good enough
               programtype[]              Pp              =
parInitPatches(m,patchLocalization);//Apply  all  patches  in
parallel
               int jobNum = 0;
               int                 arrLen                 =
sizeof(Pp)==0?0:sizeof(Pp)/sizeof(Pp[0]);
               std::future<resulttype> resultarr[arrLen];
               while (jobNum<arrLen)
               {
                    //The call to UnitIncCor asynchronous and
non-blocking (future-like datatype).
                    //Next  cycle  will  be  triggered  before
UnitIncCor returns.
```

```
                    resultarr[jobNum]                         =
std::async(std::launch::async, []{ validate(Pp[jobNum], P,
R, T); });
                    jobNum = jobNum+1;
               }
               jobNum=0;
               while (jobNum<arrLen)
               {
                    //Here the call becomes blocking
                    resultarr[jobNum].wait();
                    resulttype
localresult=resultarr[jobNum].get();
                    recordresult(Pp,localresult);
                    if(stopOnAbsolute                         &&
abscor(localresult))
                    {
                         //result is absolutely correct,
abort execution
                         return;
                    }
                    //Use the returned future value to
identify whether the program is strictly more correct
                    //If it is, set multiplicitylevel to 1.
else, it is the current multiplicity level+1
                    //If max multiplicity limit has not been
breached, find place in the list between it
                    //and controllist.end() using binary
search in order to insert the new candidate for repairs
                    int
multiplicitylevel=localresult.issmc()?1:m+1;
                    if (multiplicitylevel<=M)
                    {

     controllist.insert(bsearchInsert(localresult),      new
wrapperprogramtype(Pp[jobNum],multiplicitylevel));
                    }
                    jobNum = jobNum+1;
               }
               it=controllist.erase(it);
          }
     }

     resulttype validate (programtype Pp, programtype P,
specification R, testdata T) {
          //Check whether the candidate is in relative
correctness relation
```

```
          //calculate fitness function (whether more tests
passed), control test count
          return                                          new
resulttype(mc(Pp,P,R,T),ff(Pp,P,R,T),tc(Pp,P,R,T);
     }
```

This solution does away with separation of UnitIncCor from ProgramRepair and slices the process of program repair in a different way sharing localization information for a given level of multiplicity between independent parallel executing nodes and grouping similar complexity operations together to achieve higher level of synchronicity in result generation and minimize the wait that happens on the result aggregation lines:

```
     //Here the call becomes blocking
     resultarr[jobNum].wait();
```

Implementations of this algorithm should also consider passing the results of original test validation down to the parallel nodes in addition to localization information, to minimize redundant calculations.

An example of the high-level algorithm implementation for a loosely-coupled parallel environment like an HPC grid looks as follows:

1) Pre-processing and information extraction

a) Retrieve information about all potential points of mutation application and subdivide it into work buckets.

b) Assign a number to each bucket and store the mapping in a location accessible to cluster control

2) Generate phase /**Parallel**/ on each process:

a) Get the work order number from cluster control.

b) Retrieve work bucket with that number.

c) /**Parallel**/ on each thread:

i) Take next mutation

ii) Apply it to the code based on information in the work bucket

iii) Store mutant candidate P’.

3) Validate phase /**Parallel**/ on each process:

a) Get the work order number from cluster control.

b) Retrieve candidate with that number.

c) Check the shared database for the serialized version of the original execution run with the assigned tests. If missing - perform the run and cache serialized version in the database for other threads to use.

d) /**Parallel**/ on each thread:

i) Take next test

ii) Perform test run execution on the candidate

iii) Compare test execution for P’ to P. Check whether the number of tests passed have increased; whether the tests passing for P’ are a subset of tests passing for P.

The applied approach is illustrated on Figure 5.1 and Figure 5.2.

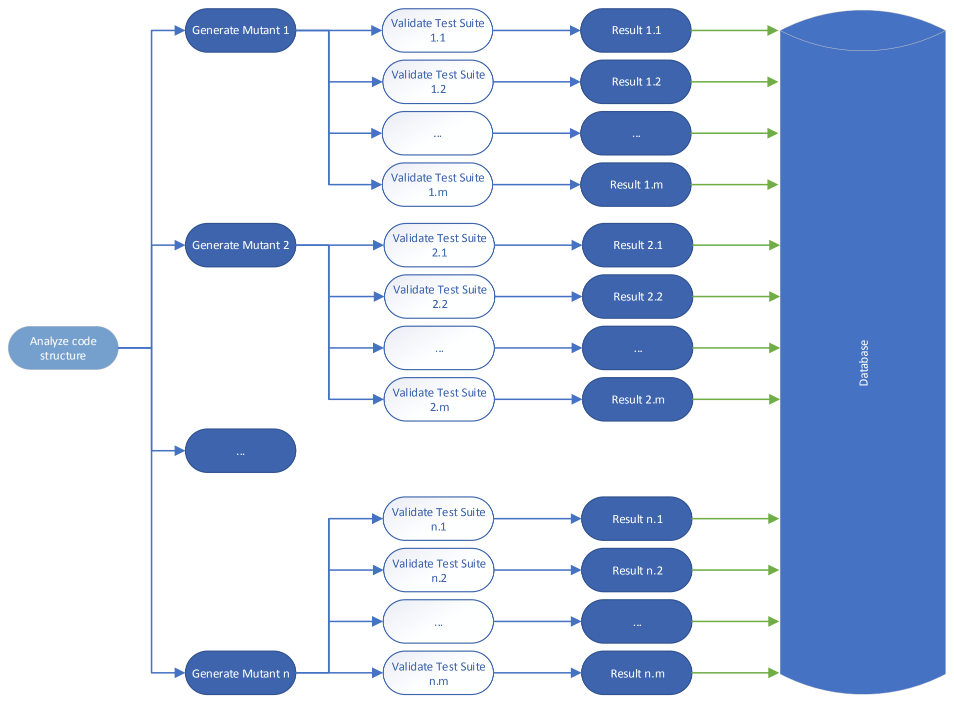


Figure 5.1 Schematic drawing of a possible parallel implementation.

Source: [95].

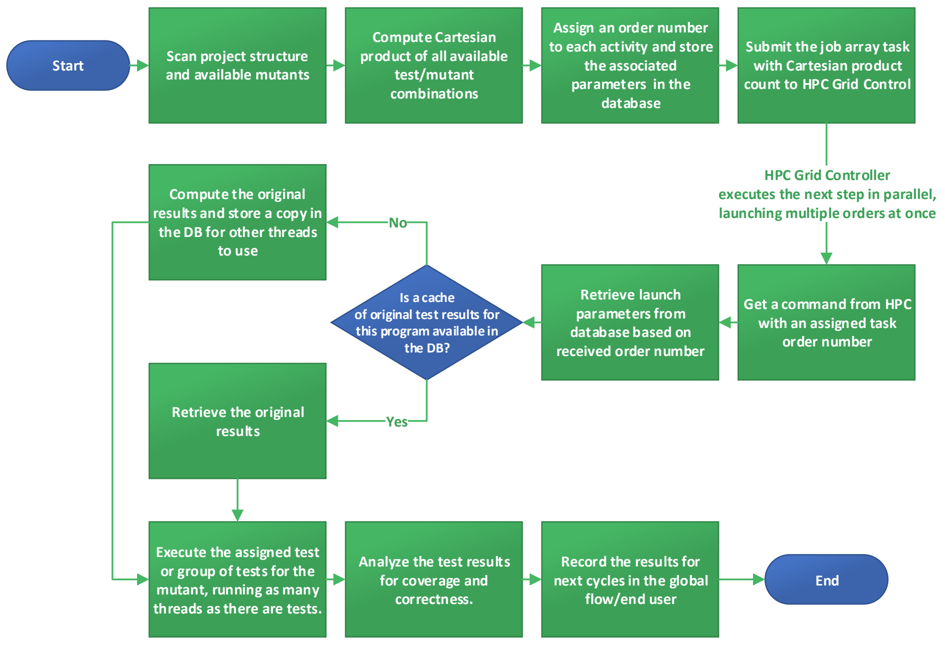

Start
Scan project structure and available mutants
Compute Cartesian product of all available test/mutant combinations
Assign an order number to each activity and store the associated parameters in the database
Submit the job array task with Cartesian product count to HPC Grid Control
HPC Grid Controller executes the next step in parallel, launching multiple orders at once
Get a command from HPC with an assigned task order number
Retrieve launch parameters from database based on received order number
Is a cache of original test results for this program available in the DB?
No
Compute the original results and store a copy in the DB for other threads to use
Yes
Retrieve the original results
Execute the assigned test or group of tests for the mutant, running as many threads as there are tests.
Analyze the test results for coverage and correctness.
Record the results for next cycles in the global flow/end user
End

Figure 5.2 Expanded view of validate stage flow for a single execution path, when launched on HPC Grid.

Source: [95].

CHAPTER 6

AN INSTANCE OF THE GENERIC ALGORITHM: CORRECTNESS ENHANCER

6.1 Specification of Correctness Enhancer

6.1.1 Design Goals

As described in Zakharchenko et al. [95] efficiently addressing the problem of correctness enhancement without artificial limitations requires the tool to follow a set of design goals:

- Reliance on the concepts of relative correctness allowing to gradually approach a solution through a set of relatively more correct solutions, as compared to the all-or-nothing approach that is based on absolute correctness. The need to get an absolutely correct result is understandable from a usage perspective, however, is not really usable for driving the process as it does not provide any feedback to the code on whether the applied change was making the results any better, leading to an essentially stateless trial-and-error, whereas with relative correctness, such feedback is provided.

- Compatibility with existing mainstream commercial software development practices and reliance on commonly available sources of program specifications and ability to integrate into existing systems and pipelines. Keeping the tool compatible with existing mainstream sources of information allows real-life applicability beyond the limits of a single synthetic dataset.

- Modular design and open source nature of the tool. There should be no locking into any black-box or vendor components and individual components of the tools have to be easily replaceable with their analogues, if such change is considered beneficial. This way the risk of dependencies having poor support is mitigated and the functionality of the tool is easy to expand, as needed.

- High level of optimization for massively parallel execution. The task at hand requires significant computing resources, which only a major cluster, HPC grid or cloud can provide and therefore the tool should be optimized to be able to achieve maximum efficiency on the distributed architecture having hundreds and potentially thousands of independent computing nodes with various levels of coupling between them.

6.1.2 Specifications in Practice

In practical software development, the business and technical specifications of a software product are the source of both specification R and space description S in which the program operates. Given through some projection, usually through a set of user stories or JIRA tickets, such specifications are not easy for correct machine-comprehension in their initial form. However, as a part of the normal development process done based on this source of information by human developers the specifications get transcribed into standard unit, integration and regression tests. Such tests, intended mainly for automated verification of key specifications of the software over time, provide an easily usable source of specifications R for the program P. Integrated with automated CI/CD pipelines, synthetic monitoring tools or generally available to the developer for manual execution, these tests are capable of automatically detecting a fault in the system, when it occurs and allow for correctness enhancement procedures to take place as a part of corrective maintenance of program, where a fault is introduced at a later stage through subsequent development of additional features.

The corrective maintenance based on specifications provided through tests is not limited to late-stage development activities, as the continuous rise of popularity of test-driven and behavior-driven development has resulted in machine-readable specifications often being created prior to the program itself thus expanding the potential applicability of correctness-enhancing techniques and allowing for their application as a part of the early stages of the development process, in theory starting with the program P0 being nothing more than just abort().

As a result, language-specific unit-tests are the chosen source of specifications for Correctness Enhancer.

6.2 Design of Correctness Enhancer

6.2.1 A Practical Implementation Analysis - Approaching the Issue

Building up on the theoretical foundation and creating a practical implementation requires answering the following questions:

1. Choice of programming language.
2. Target of application
3. Program repair approach

6.2.2 Programming Language Choice

The question of programming language choice is not critical, when discussing the details of a generic algorithm. It is, however, one of the first questions to come up, when the problem switches from theoretical applications to practical implementation, both in the context of the language choice of the practical implementation as well as in the context of the language choice of the target benchmarks and programs.

Although the algorithm implementation does not necessarily need to be done in the same programming language that is being targeted for repairs, doing so can simplify the deployment and maintenance in production environments, by avoiding dependencies that would not be present otherwise. Even if the implementation language is different from target language, in order to maximize the usability and applicability of any practical framework implementations, they have to be done using one of the mainstream programming languages.

Considering that the choice of the target languages is also tied to usefulness of the resulting implementation it is driven mainly by how mainstream the language is and the availability of high-quality benchmarks in that language. The survey of the current languages [75, 76] highlights C, C++, C#, Java, Python and JavaScript as the potential candidate languages to focus on. In Appendix Gazzola et al. [4] report data on 25 different tools. Out of them, 13 are focused on C-based languages, 9 are focused on Java (with 1 supporting Habanero in addition to Java), 1 on Python, 1 on PHP and 1 on Eiffel. Such breakdown suggests that from the perspective of comparability, any practical research should be focused on C-family or Java, as the remaining languages do not show a sufficient representation in the R&D field of program repair. Upon further analysis, Java is determined to be a better candidate, as, while being a universal language, commonly encountered in all layers of programming from backend to web and being one of the most widely used programming languages for commercial software development, Java also has a significant benefit shared with the scripting languages of being compiled to an intermediate language (bytecode) instead of machine code, this way leaving the technical possibility to perform automated analysis and mutation even if the source code is absent. In addition, its property of relying on garbage collectors for automated memory management makes its structure less complex to analyze and mutate, as compared to C and C++ thus reducing potential issues with mutant generation and increasing the efficiency of the approaches applied. There are however, no technical obstacles, preventing eventual creation of universal program repair tools targeting multiple languages at once.

With Java being selected as the target language, Junit becomes the selected source of specifications.

6.2.3 Program Repair Approach

As discussed in Chapter 2, the program repair approaches currently utilized in program repair industry are subdivided into being generate and validate or semantic driven approaches. Considering that semantic-driven approaches are better geared towards targeting specific patterns in the source code, generate and validate approaches are a much better foundation to test the benefits of practical application of program repair theory. Generate-and-validate approaches are quite popular in the industry, however, the bane of their existing implementations is that, as shown by Khaireddine et al. [11], without applying the concept of relative correctness, program repairs methods and tools expose themselves to a risk of poor efficiency, loss of precision, and loss of recall and using an existing mutator tool as the source of patch generation [10, 59] and rebuilding it based off new theoretical and software architecture approaches, while leveraging its collection of mutators to keep the results comparable is a feasible approach. MuJava (https://cs.gmu.edu/~offutt/mujava/ Retrieved on November 20, 2021) [96] is a potential good candidate of such tool.

6.2.4. MuJava

The intent and purpose of muJava is the opposite of what was being pursued in this research: instead of trying to fix the code that is not operational, muJava is designed to introduce faults into the operational code in order to test the ability of existing test suites to detect changes that could have been accidentally done by the developer. In order to perform this work, out of the box the tool is coming with two modes of operation: a strictly single-threaded mode allowing to apply the entire code base of mutants one by one and a test running mode allowing to take any single test case and execute it against all mutants that have been generated. With most programs in practice having more than one test and a large mutation surface, neither of the two modes are particularly applicable for the task of correctness enhancement, however the set of mutant generators that come with the tool is fully salvageable and easily expandable, determining the next steps of the research being conducted.

6.2.5. The New Patch Validation

When assessing, whether a program is operating in line with its specifications, the concept of absolute correctness is usually applied: either all tests pass and the program meets corresponding specifications or some of the tests fail and the program is considered faulty until all of its issues are resolved. While such definition is good enough for a common, business definition of software meeting or not meeting expectations, the change, necessary to go from the state "the program is faulty" to the state "the program is operational again" is multi-step and often requires days and weeks of qualified work by software development teams in order to make the transition. As a result, the level of granularity that is provided by absolute correctness is insufficient and therefore, in this work the program’s correctness is evaluated through the prism of getting a program to a "strictly more correct" state, once mutation is applied. In practice it translates to evaluating a program P’ against three separate criteria: whether mutated program P’ is more correct with respect to P (meaning that all tests that pass for P pass for P’), a fitness function like approach, which, in its basics, is looking at the percentage of tests that have passed successfully in the test run on program P, comparing it to the percentage of tests that were successful for the mutated program P’ and verifying that there has been no drop in the number of testcases executed between the two program states.

The tool, while being capable of fixing both single-site and multi-site faults, might require traversing the entire search space for a multi-site fault, if the intermediary results required to fix it do not make the program relatively more correct, however, in such scenarios it can often give suggestions on disabling part of the functionality, which can serve as an indicator of potential problematic areas for manual debugging.

6.2.6 Adjusting to Changes in Project Structures With Levenshtein Distance-based Criterion

In practical software development a project structure is typically not standardized and can wary greatly depending on the project itself and the mixture of programming languages, technologies and tools being utilized. For the majority of automated program repair tools, the target software project structure is normally specified through a set of configurations. These configurations, however, in addition to being unique for each project due to lack of uniformity, also require continuous redundant maintenance and adjustment for any kind of continuous deployment alongside the project, as they need to reflect any and all ongoing changes to the project structure, which routinely happens as a part of refactoring, clean-ups, technical debt remediations and changes in the underlying technologies, which reflect in the project structure. Manual maintenance of such configuration-based automation is cumbersome and prone to mistakes. Although automatic rescanning of the project structure can get the adjustments factored in, it can be inconvenient if configuration is used to limit the patch generation with subsequent validation to a selected group of files only.

Researching the problem of merging and automatically auditing hierarchical data structures in medical domain, Zakharchenko et al. [97, 98] have come up with a design of an automated-comparison framework implemented in the form of a software tool, which, through reliance on a combination of partial-string matching and application of

Levenshtein distance-based criterion was successful in detecting similarities and giving suggestions in merging large complex hierarchical data structures. Realizing that the task of automatically adjusting to changes in project structure is dealing with a related problem, this framework has been expanded upon in the Correctness Enhancer tool.

Using the combination of unqualified class name and information about the expected hierarchical placement in the project structure, in case a precise match is not found, Correctness Enhancer is looking for a class that is similarly-named, but has the smallest deviance in the hierarchical path (measured through the smallest Levenshtein distance) from the original described in the configuration. This enhancement allows it to operate from an imprecise or outdated configuration, automatically dealing with package renaming and regrouping of files within projects without a need to rescan the project and rebuild the configurations, dealing with the most common change scenarios. Introducing new files, however, would still require adjusting the configurations or performing a new scan to include them in the scope of the tool.

6.3 Implementation of Correctness Enhancer: Introducing Parallelism

6.3.1. Reasons for Parallelism

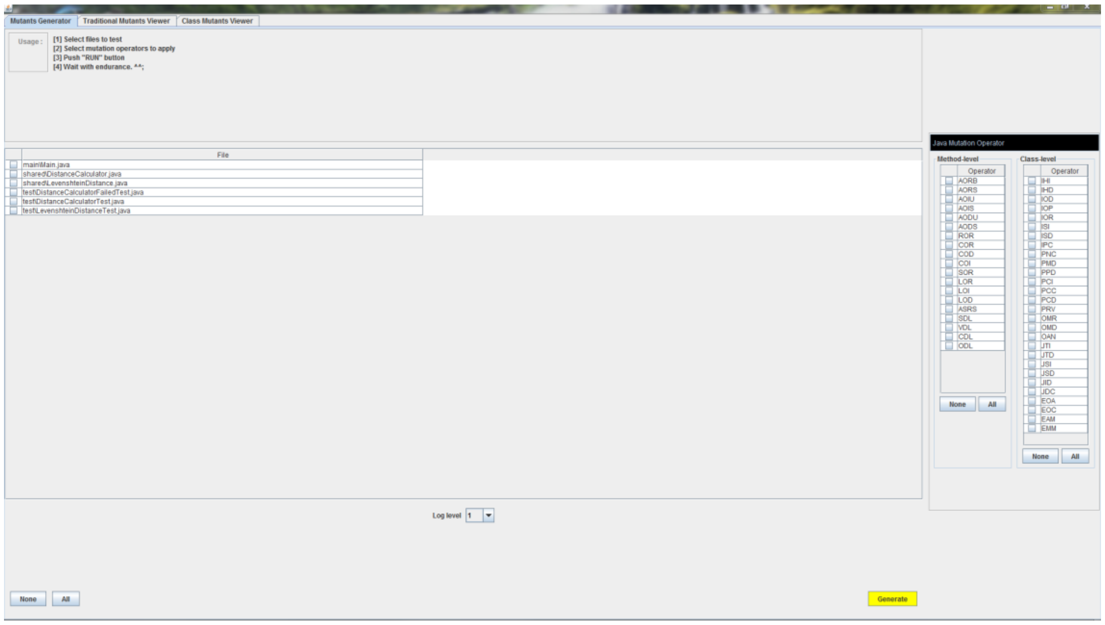
In its general generate and validate form, the task of program repair is traversing the entire field of possible candidate repair programs first generating and then validating each of them. Assuming that the program is prone to generate an average of m mutations that can be applied over n points in code, with the code (and each mutant) covered by t tests, the task of executing a generate and validate program repair becomes O(m*n) for generate and O(m*n*t) for validate portion of task. While m remains relatively constant and is mutation-system defined, for large commercial projects the number t runs in thousands and the number n grows at least proportional to the number of lines of code being written, preparing the ground for combinatorial explosion. While the optimal approach would be reducing the complexity of the algorithm, such optimization is not always possible. A more common practical approach to dealing with similar tasks is transformation of the code architecture and the underlying algorithm to reduce dependencies between different parts of the algorithm and allow their simultaneous parallel execution on a highly parallelized environment, such as GPU or cluster, essentially dividing the total number of sequential steps being needed to compute the problem by the number of threads being utilized, with that number easily reaching thousands on the modern hardware. While the GPU-based approaches are especially common, due to wider hardware availability, the major limitation of such approaches is that they are suitable for tasks with high computational and low data demands as transfer of the data between main memory and GPU memory is extremely slow. The task of generate and validate program repair though is both computation and data-intensive and as a result is a bad candidate for GPU-driven techniques. However, it still remains a perfect candidate for cluster and HPC-based computing [99] as well as APU units, which use regular computer memory for graphic computations, thus making it possible to achieve a significant boost in performance and ability to scale horizontally, if the program repair tool’s architecture is properly designed to support not just multithreaded, but also distributed usage, potentially with elements of service oriented architecture.

In generate and validate process, both the generate part and the validate part of the approach can be built to efficiently handle parallel execution. For generate part parallelism can be injected by either approaching each generate task as an independent and isolated task or by sharing initial code analysis to make the process of generation more efficient. For the validate part, the naive approach of just testing each mutant separately in an isolated environment is often not cost effective and provides insufficient performance, however, as long as isolation is achieved between different runs and utilization of shared resources is limited, each individual run can be further subdivided into runnable sub-tasks that can be executed concurrently.

6.3.2. Implementation of Parallelism

In order to apply that approach in practice, a fork of the original muJava repository has been created with the new tool named "Correctness Enhancer" in order to reflect its intended usage. As per classification from Gazzola et al. [4], this tool is a general purpose tool that utilizes fix locus localization, however, by design, since it relies on static code analysis to identify fix loci it can be used both as a standalone solution and as a source of recommendations deployed alongside the development environment.

The spin-off version was redone to support the parallelism described above both via GUI (Figure 6.1, Figure 6.2) and as a command-line tool. This allowed patch generation and validation to be performed for simplistic faults, however, as the tool was going through the entire search space which offered thousands of mutants running thousands of tests and the space being a Cartesian product between them, generation of all possible mutants for defects4j test suite was taking days and validation of them weeks even on relatively modern personal machines. As a result, the code was further enhanced to support HPC-driven execution, breaking the validate process into subtasks going beyond test suites to the individual test level thus providing a significant increase of horizontal scalability, bringing an almost linear boost in performance directly correlated with the increase in the number of cores allocated for the task.



[1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

	File
	main\Main.java
	shared\DistanceCalculator.java
	shared\LevenshteinDistance.java
	test\DistanceCalculatorFailedTest.java
	test\DistanceCalculatorTest.java
	test\LevenshteinDistanceTest.java

	Operator
	AORB
	AORS
	AOIU
	AOIS
	AODU
	AODS
	ROR
	COR
	COD
	COI
	SOR
	LOR
	LOI
	LOD
	ASRS
	SDL
	VDL
	CDL
	ODL

	Operator
	IHI
	IHD
	IOD
	IOP
	IOR
	ISI
	ISD
	IPC
	PNC
	PMD
	PPD
	PCI
	PCC
	PCD
	PRV
	OMR
	OMD
	OAN
	JTI
	JTD
	JSI
	JSD
	JID
	JDC
	EOA
	EOC
	EAM
	EMM

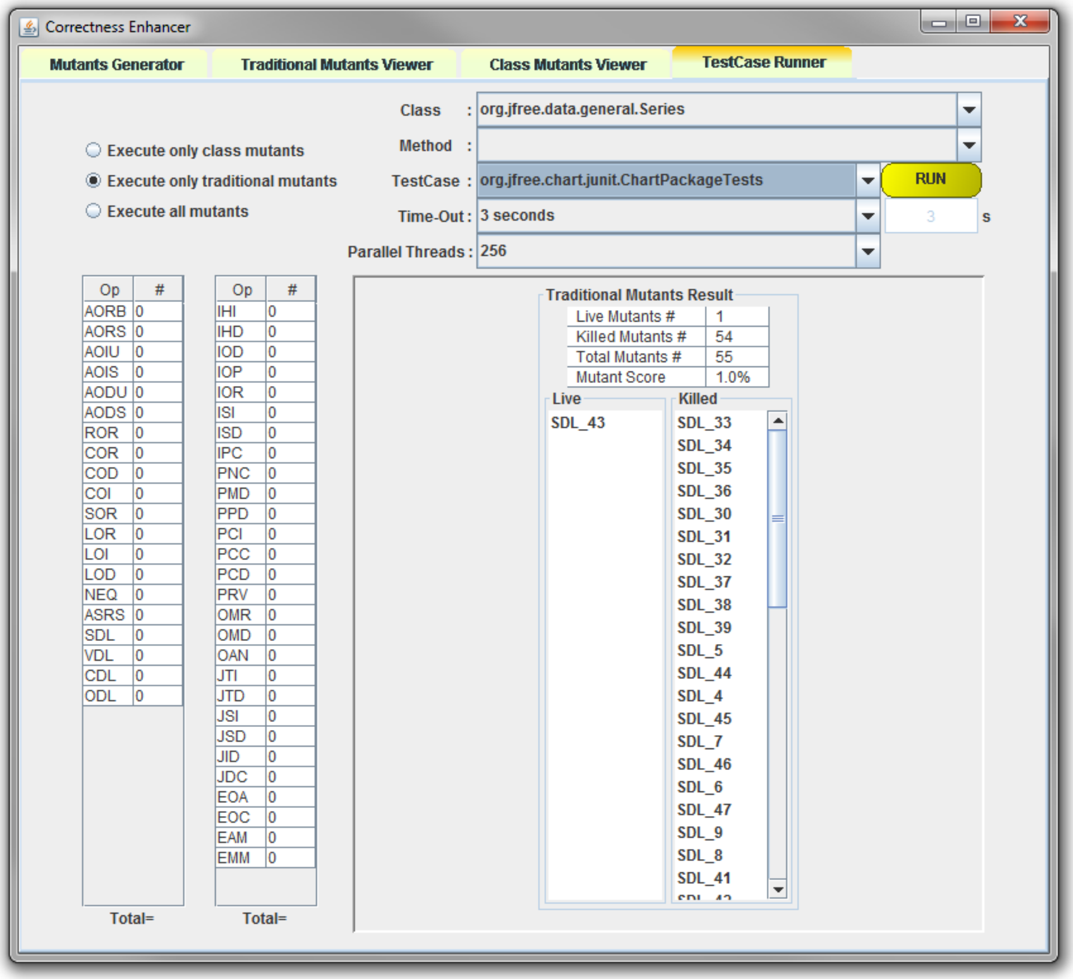
Figure 6.1 Correctness Enhancer has retained MuJava's mutation mode screen in order to allow triggering mutant generation from UI locally, in addition to being able to do it through the console. Similar to MuJava's interface, File column on the left side lists the files of the project where mutation is possible and Method-level and Class-level mutants on the right show available method and class mutations.

Source: [95]

The switch of the tool to massively parallel grid-based execution, deployed on an HPC grid (using NJIT's Kong HPC cluster, which has now been replaced by Lochness [99]) has highlighted two issues:

1. Overloading of the control node feeding work to sub-nodes. By splitting the task into small sub-tasks the issue of control queue overflow has been encountered, where the grid was not able to efficiently operate with millions of subtasks that were assigned to it. This problem has been resolved by adjusting the design of the program for the control node of the grid to pull the work from task-arrays instead of the code pushing it to the node, having a minor penalty on performance, but achieving stable grid operation.

2. The common issue of massively parallel execution is dealing with a bottleneck of combining the results from all processors into a single place. For Correctness Enhancer, the standard approach of distributed applications has been leveraged, overcoming this limitation by using an Apache Derby database (https://db.apache.org/derby Retrieved on November 20, 2021) as a shared collection endpoint.



Execute only class mutants
[✓ Execute only traditional mutants]
Execute all mutants

Class : org.jfree.data.general.Series
Method :
TestCase : org.jfree.chart.junit.ChartPackageTests
Time-Out : 3 seconds 3 s
Parallel Threads : 256

Op	#
AORB	0
AORS	0
AOIU	0
AOIS	0
AODU	0
AODS	0
ROR	0
COR	0
COD	0
COI	0
SOR	0
LOR	0
LOI	0
LOD	0
NEQ	0
ASRS	0
SDL	0
VDL	0
CDL	0
ODL	0

Total=

Op	#
IHI	0
IHD	0
IOD	0
IOP	0
IOR	0
ISI	0
ISD	0
IPC	0
PNC	0
PMD	0
PPD	0
PCI	0
PCC	0
PCD	0
PRV	0
OMR	0
OMD	0
OAN	0
JTI	0
JTD	0
JSI	0
JSD	0
JID	0
JDC	0
EOA	0
EOC	0
EAM	0
EMM	0

Total=

Traditional Mutants Result

Live Mutants #	1
Killed Mutants #	54
Total Mutants #	55
Mutant Score	1.0%

Live
SDL_43

Killed
SDL_33
SDL_34
SDL_35
SDL_36
SDL_30
SDL_31
SDL_32
SDL_37
SDL_38
SDL_39
SDL_5
SDL_44
SDL_4
SDL_45
SDL_7
SDL_46
SDL_6
SDL_47
SDL_9
SDL_8
SDL_41

Figure 6.2 Correctness Enhancer has mostly retained the original UI interface of muJava for local execution on a machine, adding option to control parallelism on validation (up to 256 threads as shown on the screenshot) and using "Live Mutants" to list results that are strictly more correct than the original or absolutely correct, however, main mode of operation is through console.

Source: [95]

The code below shows a bash shell launcher for the SGE-based cluster that illustrates both solutions:

```
#!/bin/bash
for i in "$@"
do
case $i in
    -tf1=*|--testfilter1=*)
    testFilterFile1="${i#*=}"
    shift # past argument=value
    ;;
    -tf2=*|--testfilter2=*)
    testFilterFile2="${i#*=}"
    shift # past argument=value
    ;;

    -rs=*|--rangestart=*)
    rangestart="${i#*=}"
    shift # past argument=value
    ;;
    -re=*|--rangeend=*)
    rangeend="${i#*=}"
    shift # past argument=value
    ;;

    -db2=*|--dbconfig2=*)
    dbconfig2="${i#*=}"
    shift # past argument=value
    ;;
    -db1=*|--dbconfig1=*)
    dbconfig1="${i#*=}"
    shift # past argument=value
    ;;

    -mf1=*|--mutationfilter1=*)
    mutantFilterFile1="${i#*=}"
    shift # past argument=value
    ;;
    -mf2=*|--mutationfilter2=*)
    mutantFilterFile2="${i#*=}"
    shift # past argument=value
    ;;

    -db=*|--dbcontrol=*)
    dbcontrol="${i#*=}"
    shift # past argument=value
    ;;

    -l=*|--launcher=*)
    launcher="${i#*=}"
    shift # past argument=value
    ;;

    -p=*|--program=*)
    program="${i#*=}"
    shift # past argument=value
    ;;
    -m=*|--mode=*)
    mode="${i#*=}"
    shift # past argument=value
    ;;

    -c1=*|--config1=*)
    config1="${i#*=}"
    shift # past argument=value
    ;;
    -c2=*|--config2=*)
    config2="${i#*=}"
    shift # past argument=value
    ;;

    -o=*|--output=*)
    output="${i#*=}"
    shift # past argument=value
    ;;
    -e=*|--error=*)
    error="${i#*=}"
    shift # past argument=value
    ;;


    *)
            # unknown option
    ;;
esac
done

queues=('short' 'medium' 'long' 'short' 'medium')
queuelen=${#queues[@]}

if [[ $mode == *"list"* ]]
then
    for ((i=$rangestart; i<=$rangeend; i++))
    do
        temp="${program} mode=\"${mode}\" configurationmode=\"file\" configurationpath=\"${config1}${i}${config2}\"";
        echo "qsub -l mem_free=1.0G -q ${queues[i%$queuelen]} ${temp}"
        qsub -l mem_free=1.0G -q ${queues[i%$queuelen]}<<MARKER
${temp}
MARKER
done
exit
elif [[ $mode == *"test"* ]]
then
    if [ "${dbcontrol}" ]
    then
        for ((i=$rangestart; i<=$rangeend; i++))
        do
            #echo "Getting ready to cat ${dbconfig1}${i}${dbconfig2}"
            cat ${dbconfig1}${i}${dbconfig2} | while read line
            do
                temp="${launcher} \"${program}\" mode=\"${mode}\" configurationmode=\"file\" configurationpath=\"${config1}${i}${config2}\"";
                echo "qsub -t 1-${line} -l mem_free=1.0G -q ${queues[i%$queuelen]} ${temp}"

                qsub -t 1-${line} -o "~/logs" -e "~/logs" -l mem_free=1.0G -q ${queues[i%$queuelen]}<<MARKER
${temp}
MARKER

            done
        done
    elif [ -z "${mutantFilterFile1}" ]
    then
        for ((i=$rangestart; i<=$rangeend; i++))
        do
            numoflines=wc -l ${testFilterFile1}${i}${testFilterFile2} | awk ‘{print $1;}’
            echo ${numoflines}

            cat ${testFilterFile1}${i}${testFilterFile2} | while read line
            do
```

```
                    temp="${program} mode=\"${mode}\" configurationmode=\"file\" configurationpath=\"${config1}${i}${config2}\" testfilter=\"${line}\"";
                    echo "qsub -l mem_free=1.0G -q ${queues[i%$queuelen]} ${temp}"

                    qsub -l mem_free=1.0G -q ${queues[i%$queuelen]}<<MARKER
${temp}
MARKER

            done
        done
    else
        for ((i=$rangestart; i<=$rangeend; i++))
        do
            numoflines=$(wc -l ${testFilterFile1}${i}${testFilterFile2} | awk '{print $1;}')
            echo ${numoflines}
            numoflines2=$(wc -l ${mutantFilterFile1}${i}${mutantFilterFile2} | awk '{print $1;}')
            echo ${numoflines2}
            echo $((${numoflines}*${numoflines2}))

            cat ${testFilterFile1}${i}${testFilterFile2} | while read line
            do
                cat ${mutantFilterFile1}${i}${mutantFilterFile2} | while read line2
                do
                    temp="-t 1-$((${numoflines}*${numoflines2}))"
                    if  [ -n "${output}" ]
                    then
                        temp+=" -o ${output}"
                    fi

                    if [ -n "${error}" ]
                    then
                        temp+=" -e ${error}"
                    fi
                temp+=" ${program} mode=\"${mode}\" configurationmode=\"file\"
```

```
configurationpath=\"${config1}${i}${config2}\" testfilter=\"${line}\" mutationfilter=\"${line2}\"";
                    echo "qsub -l mem_free=1.0G -q ${queues[i%$queuelen]} ${temp}"

                    qsub -l mem_free=1.0G -q ${queues[i%$queuelen]} <<MARKER
${temp}
MARKER
                done
            done
        done

    fi
elif [[ $mode == *"mutate"* ]]
then
    if [ "${dbcontrol}" ]
    then
        for ((i=$rangestart; i<=$rangeend; i++))
        do
                #echo "Getting ready to cat ${dbconfig1}${i}${dbconfig2}"
            cat ${dbconfig1}${i}${dbconfig2} | while read line
            do
                temp="${launcher} \"${program}\" mode=\"${mode}\" configurationmode=\"file\" configurationpath=\"${config1}${i}${config2}\"";
                echo "qsub -t 1-${line} -l mem_free=1.0G -q ${queues[i%$queuelen]} ${temp}"

                qsub -t 1-${line} -o "~/logs" -e "~/logs" -l mem_free=1.0G -q ${queues[i%$queuelen]}<<MARKER
${temp}
MARKER

            done
        done
    else
        for ((i=$rangestart; i<=$rangeend; i++))
        do
            cat ${mutantFilterFile1}${i}${mutantFilterFile2} | while read line
            do
                temp="${program} mode=\"${mode}\" configurationmode=\"file\"
```

```
configurationpath=\"${config1}${i}${config2}\"
mutationfilter=\"${line}\"";
     193                    echo "qsub -l mem_free=1.0G -q
${queues[i%$queuelen]} ${temp}"
     194
     195                         qsub  -l  mem_free=1.0G  -q
${queues[i%$queuelen]}<<MARKER
     196 ${temp}
     197 MARKER
     198
     199             done
     200         done
     201     fi
     202 else
     203     echo "Mode not recognized"
     204 fi
     205
```

The list of queues on line 85 lists the queues on which the jobs can be executed. Since the cluster being utilized had an uneven distribution of resources between queues, the queues that had more resources available were included more than once with a basic round-robin running against this list on all execution lines (94, 111, 129, 161, 181 and 195) giving a roughly similar computation end time. The execution through this launcher had the modes list, mutate and test with list doing the initial environment preparation, mutate running the patch generation and test running the test validation. Control of what gets submitted for execution is possible either from files generated by list option or by instructing the grid to assign a unique sequence number to each job and correlating it with the database instructions (the portions of code that get triggered if dbcontrol is not null.) While this launcher records logs from each execution for further analysis (the -o and -e options), most of the launchers that were used were setting the target directory as /dev/null, as for heavy runs, due to the number of jobs being executed logs could hit the limit on the maximum number of files in the same directory, destabilizing the process. The launcher above also reflects both modes of operation: the one with task array utilization (seen on lines 111, 181) and the one with direct job triggering (seen on lines 94, 129, 161, 195.)

Inside the code, the Cartesian product used to control all possible tasks is capable of working with arbitrary number of dimensions, allowing further splitting into subtasks, as necessary and it gets computed using Google guava's cartesianProduct method:

```
MutationControl.Inputs[] valuesArray = MutationControl.Inputs.values();
List<Set<String>> cartesianInput=new ArrayList<>();
for(MutationControl.Inputs s:valuesArray)
{
    if(!modeTypes.containsKey(s.getLabel()))
    {
        HashSet<String> fillerList=new HashSet<String>();
        fillerList.add("");
        modeTypes.put(s.getLabel(),fillerList);
        cartesianInput.add(fillerList);
    }
    else
    {
        cartesianInput.add(modeTypes.get(s.getLabel()));
    }

}
Set<List<String>> product=Sets.cartesianProduct(cartesianInput);

for (List<String> entry : product) {
   HashMap<String, String> property=new HashMap<>();
    for(int i=0;i<entry.size();i++) {
        property.put(valuesArray[i].getLabel(),entry.get(i));
    }
    localList.add(new ConfigurationItem(property));
}
if (!localList.isEmpty()) {
    DatabaseCalls.insertConfiguration(localList);
}
```

The description of the tool’s companion files and configurations, as well as a link to the repository containing full source code are provided in Appendices B and C.

CHAPTER 7

EMPIRICAL ASSESSMENT

7.1 Performance on Standard Benchmarks

In order to validate the tool, it is executed in a clustered environment against the Chart program set of Defects4j faults database, which represents the JFreeChart (https://www.jfree.org/jfreechart/ Retrieved on November 20, 2021) library. The faulty set is used automatically, while the repaired set is only relied upon for manual evaluation of the quality of suggestions provided by Correctness Enhancer. Throughout the experiment, the NJIT's Kong HPC cluster has been utilized with resource availability varying from 450 cores to almost 2000 cores at a time in parallel processing. A shared storage with several TBs of free space available has been used to store the intermediate results as well as to host an Apache Derby database, which was running on one of the cluster nodes and was used for job coordination between nodes and the result storage.

7.2 Comparison: Fitness Function vs. Relative Correctness

The quality of the results is controlled by three basic criteria:

1) Whether the tests that pass on a mutant are a superset of the original set of tests (the criteria of relative correctness.)

2) Whether the percentage of tests that have passed on a mutant is greater than on the original program P (a fitness function ensuring the strictness of enhancements.)

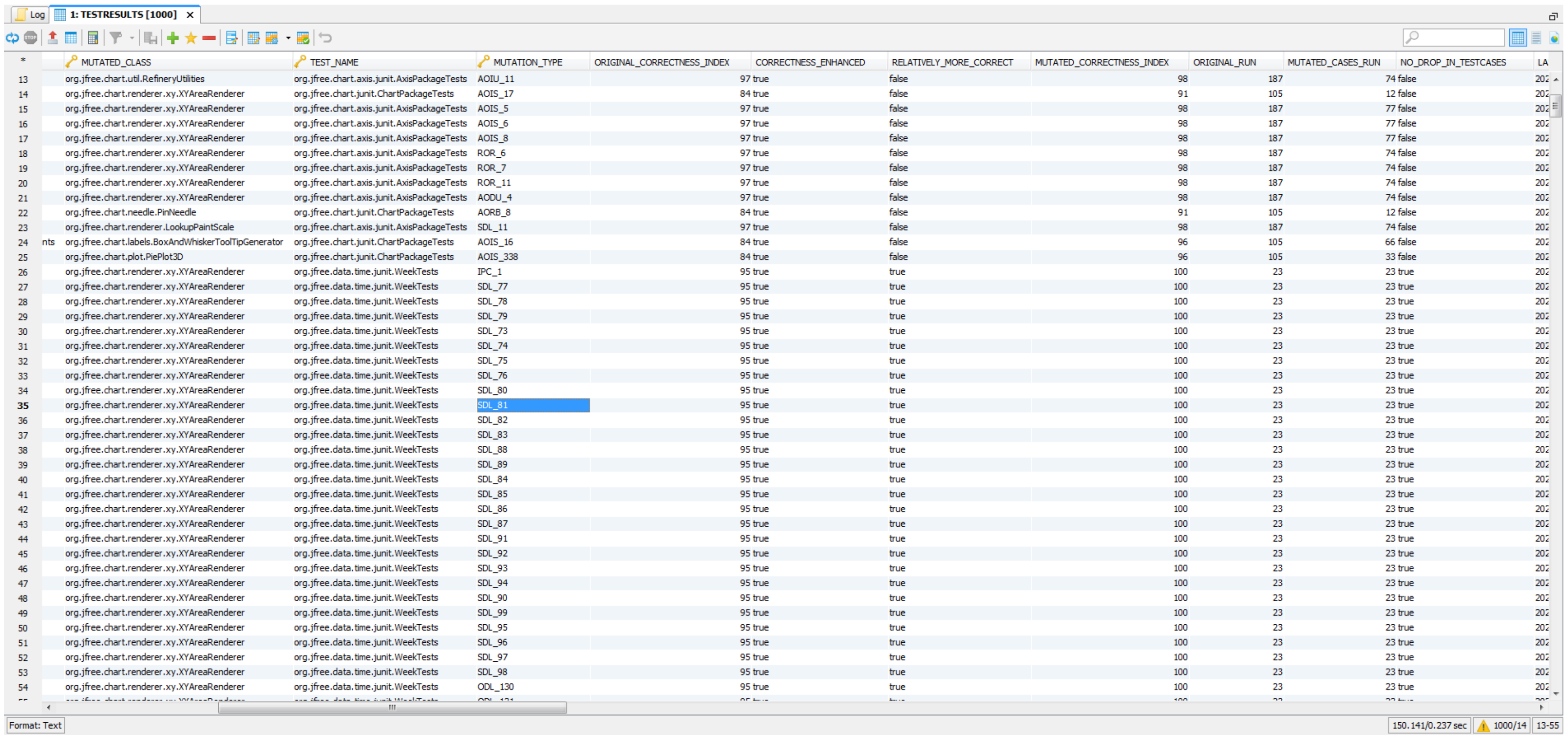
3) Whether the mutant has the same or larger amount of tests executed as compared to the original. This control is used to handle behavior of more complex tests suites, which skip execution of some of the tests if the mutant behavior is identified as a major failure.

The first two are primary driving criteria, whereas the last criterion is an auxiliary one helping remove abnormal edge cases from the result pool. It is worth highlighting that the correctness criterion utilized here is not a strict one on its own, however, it becomes a strict one, when combined with the implemented fitness function, as long as auxiliary test count control does not indicate major issue with the result. The fitness function implementation is also allowing to establish ordering between relatively correct candidates, allowing them to be pursued not in the order they were detected, but in the order better aligned with the impact they make on the results. Figure 7.1 shows an example of such verification applied to Chart group of programs from Defects4j testset.

Considering that an elementary fault with high multiplicity will be ranked low until its multiplicity layer is reached as atomic changes will not be producing more correct results, Correctness Enhancer provides an alternative way to drive the process of mutant validation using simulated annealing on top of the validation criteria described above. When activated, the code will step back from ordering and check random candidates with a given probability, improving the average case of converging on elementary faults of multiplicity m, while reducing the recall for the normal operation.

The input for simulated annealing to walk through the space of P^n is the result of the validation runs for the space of P^{n-1}, in which the degree to which the latest application of mutation has refined the program and the combination of its stop-gap factors (the strictness of refinement and the absence of test suit degradation) is quantified to produce a single number as defined below, with simulated annealing used to find the global maximum.

Higher bits		**Lower bits**
Relative correctness percentage on mutant application	Strictness of correctness enhancement (fitness function)	Lack of drop in testcases



Log | 1: TESTRESULTS [1000]

*		MUTATED_CLASS	TEST_NAME	MUTATION_TYPE	ORIGINAL_CORRECTNESS_INDEX	CORRECTNESS_ENHANCED	RELATIVELY_MORE_CORRECT	MUTATED_CORRECTNESS_INDEX	ORIGINAL_RUN	MUTATED_CASES_RUN	NO_DROP_IN_TESTCASES	LA
13		org.jfree.chart.util.RefineryUtilities	org.jfree.chart.axis.junit.AxisPackageTests	AOIU_11	97	true	false	98	187	74	false	202
14		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.junit.ChartPackageTests	AOIS_17	84	true	false	91	105	12	false	202
15		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	AOIS_5	97	true	false	98	187	77	false	202
16		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	AOIS_6	97	true	false	98	187	77	false	202
17		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	AOIS_8	97	true	false	98	187	77	false	202
18		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	ROR_6	97	true	false	98	187	74	false	202
19		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	ROR_7	97	true	false	98	187	74	false	202
20		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	ROR_11	97	true	false	98	187	74	false	202
21		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.chart.axis.junit.AxisPackageTests	AODU_4	97	true	false	98	187	74	false	202
22		org.jfree.chart.needle.PinNeedle	org.jfree.chart.junit.ChartPackageTests	AORB_8	84	true	false	91	105	12	false	202
23		org.jfree.chart.renderer.LookupPaintScale	org.jfree.chart.axis.junit.AxisPackageTests	SDL_11	97	true	false	98	187	74	false	202
24	nts	org.jfree.chart.labels.BoxAndWhiskerToolTipGenerator	org.jfree.chart.junit.ChartPackageTests	AOIS_16	84	true	false	96	105	66	false	202
25		org.jfree.chart.plot.PiePlot3D	org.jfree.chart.junit.ChartPackageTests	AOIS_338	84	true	false	96	105	33	false	202
26		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	IPC_1	95	true	true	100	23	23	true	202
27		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_77	95	true	true	100	23	23	true	202
28		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_78	95	true	true	100	23	23	true	202
29		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_79	95	true	true	100	23	23	true	202
30		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_73	95	true	true	100	23	23	true	202
31		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_74	95	true	true	100	23	23	true	202
32		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_75	95	true	true	100	23	23	true	202
33		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_76	95	true	true	100	23	23	true	202
34		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_80	95	true	true	100	23	23	true	202
35		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_81	95	true	true	100	23	23	true	202
36		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_82	95	true	true	100	23	23	true	202
37		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_83	95	true	true	100	23	23	true	202
38		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_88	95	true	true	100	23	23	true	202
39		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_89	95	true	true	100	23	23	true	202
40		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_84	95	true	true	100	23	23	true	202
41		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_85	95	true	true	100	23	23	true	202
42		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_86	95	true	true	100	23	23	true	202
43		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_87	95	true	true	100	23	23	true	202
44		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_91	95	true	true	100	23	23	true	202
45		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_92	95	true	true	100	23	23	true	202
46		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_93	95	true	true	100	23	23	true	202
47		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_94	95	true	true	100	23	23	true	202
48		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_90	95	true	true	100	23	23	true	202
49		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_99	95	true	true	100	23	23	true	202
50		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_95	95	true	true	100	23	23	true	202
51		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_96	95	true	true	100	23	23	true	202
52		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_97	95	true	true	100	23	23	true	202
53		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	SDL_98	95	true	true	100	23	23	true	202
54		org.jfree.chart.renderer.xy.XYAreaRenderer	org.jfree.data.time.junit.WeekTests	ODL_130	95	true	true	100	23	23	true	202

Format: Text | 150.141/0.237 sec | 1000/14 | 13-55

Figure 7.1 A screenshot of a portion of the results demonstrating the checks applied. The results that have NO_DROP_IN_TESTCASES returning false are results where test execution after mutation triggered abnormal program termination. While in this scenario relative correctness criterion is also showing that something went wrong, it is theoretically possible to have a situation, where critical failure happens in the part that is failing less critically in the original run and the auxiliary criterion allows to detect such situations. The candidates returning true in CORRECTNESS_ENHANCED, RELATIVELY_MORE_CORRECT and NO_DROP_IN_TESTCASES are strictly more correct with regards to specification provided by the test in TEST_NAME field, but, need to be evaluated in the context of the entire set of specifications.

Source: [95]

7.3 Comparison: Correctness Enhancer vs. Other Tools

The effectiveness of the resulting program has been assessed against the defects4j software faults dataset.

Table 7.1 Defects4j results comparison with other published tools [9, 100].

	Correctness Enhancer	**jGenProg**	**jKali**	**jMutRepair**
Chart1	Fixed	Patched, not fixed	Patched, not fixed	Patched, not fixed
Chart2	Patched - Relative			
Chart3		Patched, not fixed		
Chart4				
Chart5		Patched, not fixed	Patched, not fixed	
Chart6				
Chart7	Patched - Relative	Patched, not fixed		Patched, not fixed
Chart8	Patched - Relative			
Chart9				
Chart10				
Chart11	Patched - Relative			
Chart12	Patched - Relative			
Chart13	Patched - Relative	Patched, not fixed	Patched, not fixed	
Chart14				
Chart15		Patched, not fixed	Patched, not fixed	
Chart16	Patched - Relative			
Chart17	Patched - Relative			
Chart18	Patched - Relative			
Chart19	Patched - Relative			
Chart20	Patched - Relative			
Chart21	Patched - Relative			
Chart22	Patched - Relative			
Chart23	Patched - Relative			
Chart24	Patched - Relative			
Chart25	Patched - Relative	Patched, not fixed	Patched, not fixed	Patched, not fixed
Chart26	Patched - Relative		Patched, not fixed	Patched, not fixed

Note: The results for other tools were differentiated based on the criteria of the resulting solution being similar to one that a human developer would create (Fixed) or just meeting the rules provided (patched, not fixed.) Reliance on relative correctness has created a new category of results (Patched - Relative), where the end result is not necessarily fully conforming to all rules, but it is conforming to all rules that the input does, as well as to some additional rules that the original input did not conform to or is, in other words, strictly more correct.

Source: [95]

This dataset consists of six programs with multiple variations of seeded faults, as well as their corrected versions. The calculations are done on the first program Chart out of 6 available in the database - the JFreeCharts library. This library is offered in 26 faulty variations in the defect4j database, giving a sound sample to validate the tool on. The results compared against other industry-leading tools are provided in the Table 7.1.

It is worth noting that in order to maintain comparability with other tools in the industry, the table reports the quality aspect of the result, not the quantity one, as most of the results in the field are reported in the form of at least a single candidate repair being found for program being repaired, whereas Correctness Enhancer was able to provide multiple repair candidates for each case that was reported as Patched in the Table 7.1.

CHAPTER 8

LESSONS LEARNED

8.1 Theoretical Lessons: The Need for Theoretical Foundations

As can be easily noticed Correctness Enhancer was able to suggest some patches or fixes for almost every task it encountered. The main driver behind it was application of the relative correctness concepts, as it allowed to get a much higher degree of usability of the results than what could have been achieved otherwise. Without the concept of relative correctness, an absolute correctness criteria would have yielded just one fully positive result - Chart1, where the code deficiency was strictly falling under one of the mutations and was correctly patched in a single step, with the rest of the cases making the "issue not resolved" category. Through application of relative correctness the mutation module was able to produce a set of program P' that while still not fully matching the specification (not absolutely correct) have come closer to the specification than the original program P (strictly more correct.) Table 8.1 illustrates that in practice - the SDL_4 mutation was able to increase the percentage of successfully executed test cases under the org.jfree.data.xy.junit.DataXYPackageTests test suite, whereas after mutation, the new results were relatively more correct than the old results, having P' pass more tests than P, but not breaking any of the tests that P was passing. For a different test suite, specifically org.jfree.data.xy.junit.VectorSeriesCollectionTests the same mutation has resulted in an absolutely correct pass with all tests succeeding.

A deeper analysis of the results has demonstrated that some of the mutations that Correctness Enhancer suggests are related not to the original, seeded fault, but rather to the fact that the code had additional faults that were caused by the environment change, specifically, by the test being executed on Java 12 instead of the earlier Java versions that defects4j was designed for. Nevertheless, the code was able to suggest bug-related code adjustments for every combination that was tried.

Table 8.1 Results Variation. A Subset of Results Demonstrating Both Absolute and Relative Correctness Enhancements.

Mutated Class	Test Name	Original Correctness Index	Mutated Correctness Index	Correctness Enhanced	Relatively More Correct	No Drop in Testcases	Mutation Type
org.jfree.data.time.TimeSeriesCollection	org.jfree.chart.junit.ChartPackageTests	99	100	true	true	true	SDL_31
org.jfree.data.time.TimeSeriesCollection	org.jfree.chart.junit.JFreeChartTests	91	100	true	true	true	SDL_31
org.jfree.data.xy.VectorSeriesCollection	org.jfree.data.xy.junit.VectorSeriesCollectionTests	75	100	true	true	true	SDL_4
org.jfree.data.xy.VectorSeriesCollection	org.jfree.data.xy.junit.DataXYPackageTests	96	97	true	true	true	SDL_4
org.jfree.data.gantt.TaskSeriesCollection	org.jfree.data.gantt.junit.TaskSeriesCollectionTests	95	100	true	true	true	SDL_104
org.jfree.data.gantt.TaskSeriesCollection	org.jfree.data.gantt.junit.DataGanttPackageTests	96	100	true	true	true	SDL_104

Source: [95]

The tendency to remove functionality as a way to pass the required tests is not novel for the repair tools, however it allows to highlight the pathways of code execution, which have the impact on the overall code failure - an information, which, in absence of the full-blown solution will allow for simplified manual debugging and operation as a helping tool in the traditional DevOps stack.

8.2 Practical Lessons: The Use / Impact of Computing Power

As any manual process, the process of software development is theoretically automatable with the eventual evolution possibly leading to a situation, where the program development is done not by successive refinements, but by successive correctness enhancements [93] with developers being responsible for maintaining specification R in a machine-readable format and the computers doing the development of program P from the stage of P:{abort()} into a successive stage of relatively more correct P’->P’’->...->P(n) programs with P(n) being the minimal complexity program refining the specification R and being absolutely correct with regards to it. While the computational requirements needed to implement this vision are outside of reach for efficient execution on today’s machines, the program repair, which is a subset and the first milestone of this vision has significantly lower requirements and can be tackled even with the computing power that is provided through HPC grids and cluster computing today.

Given that the algorithm being utilized is computationally intensive and massively parallel systems have to be utilized for execution, Correctness Enhancer is relying on splitting the work into multiple tiny subtasks optimized for loosely coupled massively parallel execution. Integrated into job control on an HPC grid it is able to try all possible combinations with its capacity to try different options only limited by the set of mutations that it is aware of and the time spent to obtain the result.

The tool, however, is just the first milestone in creating a practical parallelized implementation of the relative-correctness-based automated program repair framework. What the program offers right now is transformation from P to P' with a validation of results after the mutation has been applied. The program can easily use the resulting P' to generate P'', P''', etc. however, the major limitation that is currently encountered is the execution time of the validation run. The amount of computation time spent on validating is equivalent to the number of all possible mutations that the program is able to apply multiplied by the computation time of a single end-to-end test run. This task is highly parallelizable and in theory a sufficiently powerful computer would be able to execute all possible combinations in parallel with the worst case scenario having a computation time of a single longest test suite run and a theoretical limit of a single longest test run.

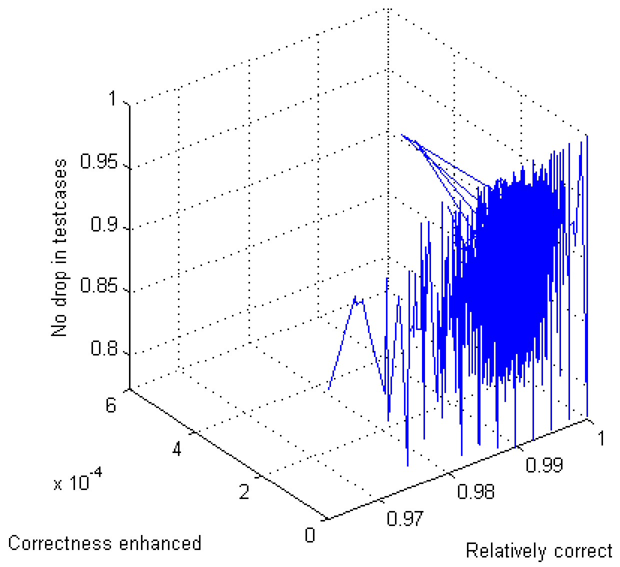


Figure 8.1 P to P' surface of JFreeCharts Charts1b variation.

8.3 Outstanding Research Questions

Based on the review of the results, only a tiny fraction of mutants appears to yield correctness enhancement suggesting that a less resource-intensive approach is possible, It may be worthwhile to investigate ways to identify and prioritize validation of those mutation operators that enhance correctness. It is conceivable that these mutants depend on the specification and on the nature of the failure; subject to future research.

The problem with applying the optimization is that the program does not have any information about the impact of a certain mutation, before it is applied and, while P’’, P’’’, P’’’’, etc. layer runs can rely on the information from P’ run (Figure 8.1 shows the input surface for the P to P’ layer of Chart1 JFreeCharts program,) even the P’ run itself takes a considerable amount of time to complete. A preliminary code flow analysis can be the missing link that would allow to optimize the walk through the mutant space, keeping the code oriented on fix loci localization, but prioritizing the locus based on fault localization.

Another option to make the program more efficient is to provide it with context awareness. One of the key approaches to automating any solution is analyzing the way a human would approach the task and implementing the same approach in an algorithm. With the current approach in the code, the program takes every single mutation operation that it knows and attempts to apply it to the code at hand, until it reaches the result. From a purely theoretical perspective, if the number of transformations that the tool is aware of is increased to cover all possible single step transformations in a software development language and allowed to group them into chains the program would eventually find the solution to any problem at hand. However, this is not the way a human developer would approach the problem, as while the direct application of code patch that would fix the fault is the only way of getting more correct software, a human developer does not go through all possible combinations of the code that could be applied, using a much smaller set of transformations specifically directed at increasing the competence domain of the code. In order to drive this decision-making process, a human developer relies on two main sources of information:

1) The knowledge about specifics of the target programming language. Letting the program "learn" the typical constructs of a programming language can significantly help in increasing efficiency of the patches it suggests.

2) The knowledge about the target competence domain of the program being fixed. The availability of that knowledge to the program is limited and only partially reflected in unit and regression tests, however, newer, more advanced methodics coming with Behavior-Driven Development help to bridge that gap by providing additional insights into what is expected from the program, going beyond specific pairs of input and output values and provide additional information about functions behind them.

CHAPTER 9

CONCLUSION

9.1 Summary

The field of automated program repair is still a developing one. This dissertation has contributed to this field, exploring the following important research topics:

1. Expanding on the framework of relative correctness to offer a new approach to program repair through the framework of relative correctness.
2. Optimizing application of massively parallel approaches in the field of program repair and using the results of the analysis in a practical implementation.
3. Addressing issues of integration and usability of program repair in production environments.
4. Providing tools to support automated program repair research and utility.

Combining the outcomes of the research made it possible to explore and obtain positive answers for questions that would otherwise be beyond reach.

9.2 Assessment

Whereas this research talks about a number of practical approaches and enhancements, the main contribution is considered to be tri-fold: providing a general purpose tool that can serve as a platform for easy expansion, giving a practical implementation of the framework of relative correctness and creating a tool optimized for massively parallel execution. Whilst the latter has been attempted by Matsumoto et al. on GenProg applying parallelization to individual runs and creating KGenProg [60], the approach presented in Correctness Enhancer went deeper into splitting runs into sub-tasks to allow multiple cluster nodes to work on as single run at the same time, achieving a higher level of support for massive parallelism.

The threats to validity of the presented approach include the reliance on a limited set of known mutators to find solution. This issue is partially addressed by keeping the set of available mutants modularized, open-source and easy to expand, allowing users to plug in custom mutators as seen fit. Another constraint is the high resource demand of the solution, however, the computing power necessary for its efficient operation is easily available on HPC-grids and clouds and is becoming available on regular user-level desktops, eroding the concern with ongoing development in the industry.

9.3 Prospects

Further research can achieve additional improvements by switching from the breadth-first search walking through mutations layer by layer to a hybrid one that can interrupt execution of the current layer of mutation to jump to a perspective candidate that is a layer deeper, essentially allowing to explore a chain of k atomic changes to process a highly promising fault with multiplicity n+k before finishing the lookup among faults with multiplicity n, theoretically allowing better convergence and faster best and average cases, subject to further research.

Indicating a potential for expanding the applicability of the tool, some authors [101-104] have highlighted the vector of application security as a potential target of program repair. Whereas addressing some of the security concerns like execution time of different branches of code execution [104], although doable with a general purpose tool like Correctness Enhancer, might require specialized modules for validation and analysis additional security-specific properties recorded alongside general program specifications [103] or a combination thereof to achieve a sufficient level of efficiency, other security flaws like OWASP top 10 vulnerabilities [101, 102] are potentially addressable using tools like Correctness Enhancer without any further changes and modifications to the tool structure, provided that the testing coverage is sufficient to properly define these security flaws as faults under the program specifications and the set of mutators being used is sufficient.

APPENDIX A

PROOF OF PERFECT PRECISION AND PERFECT RECALL

A.1 Proof of Perfect Recall

In order to prove that UnitIncCor() has perfect recall [63], provided below is a proof that the following Hoare formula is valid in Hoare's deductive logic:

v: $\{(\exists m : 1 \leq m \leq M : \exists Q \in P\, S(m) : Q \sqsupset_{R'} P\,)\}$

```
m=1; inc=false; Pp=P;

while (!inc && m<=M)
     {while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

$\{Pp \sqsupset_{R'} P\}$.

Proof: Applying the sequence rule to v, with the following intermediate predicate *int*:

$(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P\,)$
$\wedge m = 1 \wedge \neg inc \wedge Pp = P$

yields the following lemmas:

v0: $\{(\exists m : 1 \leq m \leq M : \exists Q \in P\, S(m) : Q \sqsupset_{R'} P\,)\}$

```
m=1; inc=false; Pp=P;
```

$\{(\exists m : 1 \leq m \leq M : \exists Q \in P\, S(m) : Q \sqsupset_{R'} P\,) \wedge m = 1 \wedge \neg inc \wedge P\, p = P\, \}$.

v1: $\{(\exists m : 1 \leq m \leq M : \exists Q \in P\, S(m) : Q \sqsupset_{R'} P\,) \wedge m = 1 \wedge \neg inc \wedge P\, p = P\, \}$

```
while (!inc && m<=M)
     {while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}
     else {m=m+1;}}//try higher multiplicity
```

$\{Pp \sqsupset_{R'} P\}$.

Applying the (concurrent) assignment rule to *v0* results in:

v00: $(\exists m : 1 \leq m \leq M : \exists Q \in P\,S(m) : Q \sqsupset_{R'} P)$
$\Rightarrow$
$(\exists m : 1 \leq m \leq M : \exists Q \in P\,S(m) : Q \sqsupset_{R'} P) \wedge 1 = 1 \wedge \text{true} \wedge P = P\}$.

This formula is clearly a tautology. The attention is now turned to *v1*. Using *inb(m)* (stands for: in bounds) as shorthand for: $1 \leq m \leq M$ and the while rule is applied to *v1* with the following loop invariant *inv*:

$inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P)$
$\vee(\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P)))$.

This yields three lemmas:

v10: $(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P) \wedge m = 1 \wedge \neg inc \wedge Pp = P$
$\Rightarrow$
$inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))$.

v11: $\{(\neg inc \wedge m \leq M) \wedge inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P)))\}$

```
{while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

$\{inb(m) \wedge ((inc \wedge P\,p \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P)))\}$.

v12: $\neg(\neg inc \wedge m \leq M) \wedge inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))$
$\Rightarrow$
$P\,p \sqsupset_{R'} P$.

To check the validity of *v10*, it is rewritten by distributing *inb(m)* over the disjunction and replacing m by 1 on the right hand side:

v10: $(\exists m : 1 \le m \le M : \exists Q \in P\,S(m) : Q \sqsupset_{R'} P) \land m = 1 \land \neg inc \land P\,p = P$
$\Rightarrow$
$(inb(m) \land inc \land Pp \sqsupset_{R'} P) \lor (inb(m) \land \neg inc \land (\exists h : 1 \le h \le M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P))$.

Now it is clear that *v10* is a tautology, since the left hand side logically implies the second disjunct of the right hand side, assuming, as is done here, that $M \ge 1$. As for *v12*, its left hand side can be simplified into $(inc \land P\,p \sqsupset_{R'} P)$, due to the contradiction between $m > M$ and *inb(m)*, and the contradiction between inc and ¬inc. Hence *v12* is also a tautology. The attention is now turned to v11, which is first simplified as follows:

v11: $\{\neg inc \land inb(m) \land (\exists h : m \le h \le M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)\}$

```
{while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

$\{inb(m) \land ((inc \land P\,p \sqsupset_{R'} P) \lor (\neg inc \land (\exists h : m \le h \le M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P)))\}$.

The sequence rule is now applied to *v11* with the following intermediate predicate *int'*:

$(Pp \sqsupset_{R'} P \lor PS(m) = \epsilon) \land$
$\neg inc \land inb(m) \land$
$(Pp \sqsupset_{R'} P \lor (\exists h : m \le h \le M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))$.

This yields the following two lemmas:

v110: $\{\neg inc \land inb(m) \land (\exists h : m \le h \le M : \exists Q \in PS(h) : Q \sqsupset_{R'} P))\}$

```
{while (!smc(Pp,P) && MorePatches(P,m))
{Pp = NextPatch(P,m);}
```

$\{(Pp \sqsupset_{R'} P \lor PS(m) = \epsilon) \land \neg inc \land inb(m) \land (Pp \sqsupset_{R'} P \lor (\exists h : m \le h \le M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$.

v111: $\{(Pp \sqsupset_{R'} P \lor PS(m) = \epsilon) \land \neg inc \land inb(m) \land (Pp \sqsupset_{R'} P \lor (\exists h : m \le h \le M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P)).\}$

```
        if smc(Pp,P) {inc=true;}
        else {m=m+1;}}//try higher multiplicity
```

{inb(m) ∧ ((inc ∧ Pp ⊐ $_{R'}$ P) ∨ (¬inc ∧ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P)))}.

The while rule is applied to *v110*, with the following loop invariant, *inv'*:

¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P)).

This yields the following three lemmas:

v1100: ¬inc ∧ inb(m) ∧ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P))
⇒
¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P)).

v1101: {¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P)) ∧ ¬(Pp ⊐ $_{R'}$ P ∧ PS(m) ≠ ϵ)}

```
        {Pp = NextPatch(P,m);}
```

{¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P))}

v1102: ¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ P S(h) : Q ⊐ $_{R'}$ P)) ∧ (Pp ⊐ $_{R'}$ P ∨ P S(m) = ϵ)
⇒
(Pp ⊐ $_{R'}$ P ∨ PS(m) = ϵ) ∧ ¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P)).

To see that *v1100* is a tautology, it suffices to distribute the ∧ over the ∨ on the right hand side of the implication, and to notice that the second disjunct on the right hand side is a copy of the left hand side of the implication. As for *v1102*, it is clearly a tautology, since the right hand side of ⇒ is merely a copy of the left hand side. The attention is now turned to *v1101*. Its precondition can be simplified by virtue of Boolean identities:

v1101: {¬inc ∧ inb(m) ∧ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P) ∧ ¬(Pp ⊐ $_{R'}$ P) ∧ PS(m) ≠ ϵ)}

```
        {Pp = NextPatch(P,m);}
```

{¬inc ∧ inb(m) ∧ (Pp ⊐ $_{R'}$ P ∨ (∃h : m ≤ h ≤ M : ∃Q ∈ PS(h) : Q ⊐ $_{R'}$ P))}

In order to apply the assignment statement rule to *v1101*, the semantics of function `NextPatch(P,m)` needs to be analyzed. This function is assumed to perform the following operation:

```
Pp=head(PS(m)); PS(m)=tail(PS(m));
```

Hence application of the assignment rule yields the following formula:

v11010: $\neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)) \wedge (\neg(Pp \sqsupset_{R'} P \wedge P\,S(m) \neq \epsilon)$

$\Rightarrow$

$\neg inc \wedge inb(m) \wedge (head(P\,S(m)) \sqsupset_{R'} P \vee (\exists Q \in tail(PS(m)) : Q \sqsupset_{R'} P) \vee (\exists h : m + 1 \leq h \leq M : \exists Q \in P\,S(h) : Q \sqsupset_{R'} P))$.

The first two disjuncts in the parenthesized expression:

$(head(PS(m)) \sqsupset_{R'} P) \vee (\exists Q \in tail(PS(m)) : Q \sqsupset_{R'} P)$

can be merged into a single expression:

$(\exists Q \in PS(m) : Q \sqsupset_{R'} P)$.

This expression can now be merged with the third disjunct above:

$(\exists Q \in PS(m) : Q \sqsupset_{R'} P) \vee (\exists h : m + 1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$,

to obtain:

$(\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$.

Replacing these in *v11010*, makes it easy to notice that the right hand side is a logical conclusion of the left hand side, hence *v11010* is a tautology.

Switching the attention back to *v111* and applying the if-then-else rule yields two lemmas:

v1110: $\{(Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$

```
inc=true;
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.

v1111: $\{\neg(Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P \vee PS(m) = \epsilon) \wedge \neg inc \wedge inb(m) \wedge (Pp \sqsupset_{R'} P \vee (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)).\}$

```
m=m+1;
```

$\{inb(m) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.

Simplifying v1110 and applying the assignment rule to it yields:

v11100: $(Pp \sqsupset_{R'} P) \wedge \neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$
$\Rightarrow$
$inb(m) \wedge (Pp \sqsupset_{R'} P)$.

This is clearly a tautology.

Simplifying *v1111* and applying the assignment rule to it yields:

v11110: $\neg(Pp \sqsupset_{R'} P) \wedge PS(m) = \epsilon \wedge \neg inc \wedge inb(m) \wedge (\exists h : m \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)$
$\Rightarrow$
$inb(m+1) \wedge ((inc \wedge Pp \sqsupset_{R'} P) \vee (\neg inc \wedge (\exists h : m+1 \leq h \leq M : \exists Q \in PS(h) : Q \sqsupset_{R'} P)))\}$.

If it is known that there exists Q strictly more-correct than P in one of the patch sequences PS(m), PS(m+1), …PS(M) but PS(m) is empty, then it must be in one of the sequence PS(m + 1), PS(m +2), …PS(M). For the same reason, m is necessarily strictly less than M, since Q is somewhere in PS(m + 1), PS(m + 2), …PS(M). Hence inb(m + 1) holds. Therefore, it is concluded that *v11110* is a tautology. Since all the lemmas generated from *v* are valid, so is *v*. Hence `UnitIncCor()` is partially correct with respect to the specification:

– Precondition: $(\exists m : 1 \leq m \leq M : \exists Q \in PS(m) : Q \sqsupset_{R'} P)$.
– Postcocndition: $Pp \sqsupset_{R'} P$.

A.2 Proof of Perfect Precision

In order to prove that `UnitIncCor()` has perfect precision [63], provided below is a proof that the following Hoare formula is valid in Hoare's deductive logic:

v: {true}

```
m=1; inc=false; Pp=P;
while (!inc && m<=M)
     {while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

{inc $\Rightarrow$ Pp $\sqsupset_{R'}$ P}.

Proof: Applying the sequence rule to v with the intermediate predicate int: inc $\Rightarrow$ Pp $\sqsupset_{R'}$ P yields the following formulas:

v0: {true }

```
m=1; inc=false; Pp=P;
```

{inc $\Rightarrow$ Pp $\sqsupset_{R'}$ P}.

v1: {inc $\Rightarrow$ Pp $\sqsupset_{R'}$ P}

```
while (!inc && m<=M)
     {while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

{inc $\Rightarrow$ Pp $\sqsupset_{R'}$ P}.

The (concurrent) assignment rule applied to *v0* yields:

v00: true $\Rightarrow$ (false $\Rightarrow$ P $\sqsupset_{R'}$ P).

This is a tautology.

Applying the while rule to *v1* with the loop invariant *inv*: inc ⇒ Pp $\sqsupset_{R'}$ P yields the following formulas:

v10: (inc ⇒ Pp $\sqsupset_{R'}$ P) ⇒ (inc ⇒ Pp $\sqsupset_{R'}$ P)
v11: {(inc ⇒ Pp $\sqsupset_{R'}$ P) ∧ (¬inc ∧ m ≤ M)}

```
{while (!smc(Pp,P) && MorePatches(P,m))
          {Pp = NextPatch(P,m);}
     if smc(Pp,P) {inc=true;}

     else {m=m+1;}}//try higher multiplicity
```

{inc ⇒ Pp $\sqsupset_{R'}$ P}.

v12: (inc ⇒ Pp $\sqsupset_{R'}$ P) ∧ (inc ∨ m > M) ⇒ (inc ⇒ Pp $\sqsupset_{R'}$ P).

Formulas *v10* and *v12* are clearly tautologies.

Applying the sequence rule to *v11*, with *int*: inc ⇒ Pp $\sqsupset_{R'}$ P yields the following formulas:

v110: {(inc ⇒ Pp $\sqsupset_{R'}$ P) ∧ (¬inc ∧ m ≤ M)}

```
while (!smc(Pp,P) && MorePatches(P,m))
     {Pp = NextPatch(P,m);}
```

{inc ⇒ Pp $\sqsupset_{R'}$ P}

v111: {(inc ⇒ Pp $\sqsupset_{R'}$ P)}

```
if smc(Pp,P) {inc=true;}

else {m=m+1;}//try higher multiplicity
```

{inc ⇒ Pp $\sqsupset_{R'}$ P}.

Applying the while rule to *v110* with the loop invariant *inv*: ¬inc yields the following formulas:

v1100: (inc ⇒ Pp $\sqsupset_{R'}$ P) ∧ (¬inc ∧ m ≤ M) ⇒ ¬inc.
v1101: {¬inc ∧ (¬Pp $\sqsupset_{R'}$ P ∧ MorePatches(P, m))}

```
{Pp = NextPatch(P,m);}
```

{¬inc}.

v1102: $\neg inc \wedge \neg(\neg Pp \sqsupset_{R'} P \wedge MorePatches(P, m)) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$.

Formula *v1100* is clearly a tautology; formula *v1102* is also a tautology because it has the form $((\neg a \wedge b) \Rightarrow (a \Rightarrow c))$, which can be simplified as $(a \vee \neg b) \vee (\neg a \vee c)$.

Applying the assignment statement rule to *v1101* yields:

v11010: $(\neg inc \wedge (\neg Pp \sqsupset R' P \wedge MorePatches(P, m))) \Rightarrow \neg inc$.

This is clearly a tautology.

Switching to *v111* and applying the if-then-else rule yields:

v1110: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P)\}$

```
{inc=true;}
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.

v1111: $\{(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge \neg(Pp \sqsupset_{R'} P)\}$

```
{m=m+1;}
```

$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$.

Application of the assignment statement rule to *v1110* and *v1111* yields, respectively:

v11100: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge (Pp \sqsupset_{R'} P) \Rightarrow (Pp \sqsupset_{R'} P)$.
v11110: $(inc \Rightarrow Pp \sqsupset_{R'} P) \wedge \neg(Pp \sqsupset_{R'} P) \Rightarrow (inc \Rightarrow Pp \sqsupset_{R'} P)$.

Formulas *v11100* and *v11110* are both tautologies. This concludes the proof that

v: {true}
```
UnitIncCor()
```
$\{inc \Rightarrow Pp \sqsupset_{R'} P\}$

is valid in Hoare's logic. Hence `UnitIncCor()` is partially correct with respect to the specification defined by the following pre/post condition pair:

- Precondition: true.
- Postcondition: inc ⇒ Pp $\sqsupset_{R'}$ P.

APPENDIX B

CORRECTNESS ENHANCER USER MANUAL

This tool requires latest version of Java to run. The latest version can be obtained from Oracle's website. The current link is https://www.oracle.com/java/technologies/downloads/ (Retrieved on November 20, 2021.)

As any Java program, this tool can be executed both on Linux-based and Windows-based environments. The tool operation is controlled through a combination of data from the command line parameters, if they are provided during startup, and from mujava.config file, which comes with the tool, and which, although it inherited the naming from the muJava tool, is completely different in terms of the options provided. In the default setting the tool can be launched out of the box as any regular commercial tool with only the setup of the source and target directories required in the configuration. The results can be output to console, file or a database. The configuration below shows example of setup for operation on a Windows-based OS for Chart_b program from defects4j suite that is located in J:\VM-SHARED location with connection to local database:

mujava.config
```
MuJava_HOME=J:\VM-SHARED\Chart_6b
config_mode=true
filter_tests=N
MuJava_src=J:\VM-SHARED\Chart_6b\source
MuJava_class=J:\VM-SHARED\Chart_6b\build
MuJava_mutants=J:\VM-SHARED\Chart_6b\mutants
MuJava_tests=J:\VM-SHARED\Chart_6b\build-tests
MuJava_chain=J:\ VM-SHARED\Chart_6b\mutationchain
number_of_mutation_threads=256
number_of_testing_threads=64
Results_output=J:\VM-SHARED\mutantResults.txt
List_Target_Mutation_Files=J:\VM-SHARED\mujavaMutation.txt
List_Target_Tests=J:\VM-SHARED\mujavaTest.txt
```

```
debug_output_enabled=N
test_results_jdbc=jdbc:derby://localhost:1527/tests.db;create=true
test_results_output_mode=database
soft_class_match_allowed=Y
database_marker=Chart_6b
database_count=J:\VM-SHARED\Chart_6b_dbcount.txt
annealing=0
chain_length=2
stop_on_correct=true
```

For an HPC launch the tool comes equipped with several shell scripts that are easy to operate with to get the user started. Latest instructions and versions of the tool can be found here: https://github.com/zakhalex/correctnessEnhancer (Retrieved on November 20, 2021.)

APPENDIX C

APACHE DERBY LAUNCHER AND DATABASE TABLES

Out of the box, the tool supports three operating modes: console-oriented, file-oriented and database-oriented. If database-oriented mode is utilized, the wrapper around Apache Derby, provided with the tool, can help with spinning up the database environment. Alternatively - any mainstream database can be utilized, however database drivers might need to be replaced with the suitable ones.

DerbyWrapper/Main.java
```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.net.InetAddress;

import org.apache.derby.drda.NetworkServerControl;

public class Main
{

    public static void main(String[] args) throws Exception
    {
        PrintWriter pw=new PrintWriter("database.log");
        String ip="0.0.0.0";
        int
portNumber=NetworkServerControl.DEFAULT_PORTNUMBER;
        if(args.length>2)
        {
            ip=args[1];
            portNumber=Integer.parseInt(args[2]);
            System.out.println("New host is:
"+ip+":"+portNumber);
        }
        InetAddress inetaddr=InetAddress.getByName(ip);
        NetworkServerControl server = new
NetworkServerControl(inetaddr,portNumber);
        server.start (pw);
        System.in.read();
        server.shutdown();
    }
```

}

On startup, the APP.TESTRESULTS, APP.ORIGINALTESTRESULTS, APP.CHAINCONTROL and APP.CONFIGURATIONS tables will be checked for on the provided database connection and generated if they are not detected. If database other than Derby is utilized or the default implementation is not considered optimal for the usage, they might need to be created manually, using the SQL syntax conforming to the environment being utilized. The default implementations are provided below:

APP.TESTRESULTS

```
CREATE TABLE
   TESTRESULTS
   (
      BASE_DIR VARCHAR(1024),
      PROGRAM_LOCATION VARCHAR(1024) NOT NULL,
      MUTATED_CLASS VARCHAR(1024) NOT NULL,
      TEST_NAME VARCHAR(1024) NOT NULL,
      MUTATION_TYPE VARCHAR(64) NOT NULL,
      ORIGINAL_CORRECTNESS_INDEX INTEGER,
      CORRECTNESS_ENHANCED BOOLEAN,
      RELATIVELY_MORE_CORRECT BOOLEAN,
      MUTATED_CORRECTNESS_INDEX INTEGER,
      ORIGINAL_RUN INTEGER,
      MUTATED_CASES_RUN INTEGER,
      NO_DROP_IN_TESTCASES BOOLEAN,
      LAST_UPDATED TIMESTAMP,
      COMMENT VARCHAR(1024),
      PRIMARY KEY (PROGRAM_LOCATION, MUTATED_CLASS, TEST_NAME,
MUTATION_TYPE)
   );
```

APP.ORIGINALTESTRESULTS

```
CREATE TABLE
   ORIGINALTESTRESULTS
   (
      BASE_DIR VARCHAR(1024) NOT NULL,
      TEST_NAME VARCHAR(1024) NOT NULL,
      ORIGINAL_CORRECTNESS_INDEX INTEGER,
      SERIALIZED_DATA BLOB(2147483647),
      PRIMARY KEY (BASE_DIR, TEST_NAME)
   );
```

APP.CONFIGURATIONS

```
CREATE TABLE
   CONFIGURATIONS
   (
      ID INTEGER NOT NULL,
      FILE_NAME VARCHAR(1024) NOT NULL,
      CLASS_NAME VARCHAR(4096) NOT NULL,
      METHOD_NAME VARCHAR(4096) DEFAULT ‘‘ NOT NULL,
      TEST_NAME VARCHAR(4096) DEFAULT ‘‘ NOT NULL,
      LAST_UPDATED TIMESTAMP,
      PRIMARY KEY (FILE_NAME, CLASS_NAME, METHOD_NAME,
TEST_NAME)
   );
```

APP.CHAINCONTROL

```
CREATE TABLE
   CHAINCONTROL
   (
      BASE_DIR VARCHAR(1024) NOT NULL,
      MUTATION_CHAIN VARCHAR(1024) NOT NULL,
      SERIALIZED_MUTATION_CHAIN BLOB,
      OVERALL_INDEX INTEGER DEFAULT 0,
      LAST_UPDATED TIMESTAMP,
      PRIMARY KEY (BASE_DIR, MUTATION_CHAIN)
   );
```

REFERENCES

[1] Serguei Khramtchenko. 2004. Comparing eXtreme Programming and Feature Driven Development in Academic and Regulated Environments. *Feature Driven Development (2004)*. Final paper for CSCIE-275: Software Architecture and Engineering, Harvard University May 17, 2004, Retrieved November 12, 2021 from http://www.featuredrivendevelopment.com/files/FDD_vs_XP.pdf

[2] Shigeru Igarashi, Ralph L. London, and David C. Luckham. 1973. *Automatic Program Verification I: A Logical Basis and Its Implementation.* May 1973, Stanford University, Department of Computer Science, Technical Reports, Stanford University, Palo Alto, CA

[3] Jack R. Buchanan. 1974. *A Study in Automatic Programming.* May 1974, Memo (Stanford Artificial Intelligence Laboratory). Stanford University, Palo Alto, CA

[4] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. In *IEEE Transactions on Software Engineering*. vol. 45, no. 1, 34-67, DOI: https://doi.org/10.1109/TSE.2017.2755013

[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. In *IEEE Transactions on Software Engineering*. vol. 38, no. 1. 54-72. DOI: https://doi.org/10.1109/TSE.2011.104

[6] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, USA. IEEE Press. 772–781.

[7] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. Florence, Italy. IEEE Press. 448–458.

[8] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. DOI: https://doi.org/10.1145/2884781.2884807

[9] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. DOI: https://doi.org/10.1145/2931037.2948705

[10] Vidroha Debroy and W. Eric Wong. 2014. Combining Mutation and Fault Localization for Automated Program Debugging. *Journal of Systems and Software*. 90, C (April 2014), 45–60. DOI: https://doi.org/10.1016/j.jss.2013.10.042

[11] Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2021. The Bane of Generate-and-Validate Program Repair: Too Much Generation, Too Little Validation. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*. IOS Press. 113-126. DOI: https://doi.org/10.3233/FAIA210013

[12] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering. 20, 1*. 176–205. DOI: https://doi.org/10.1007/s10664-013-9282-8

[13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned From Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, USA. IEEE Press, 802–811. DOI: https://doi.org/10.1109/ICSE.2013.6606626

[14] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions With SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*. Association for Computing Machinery, New York, NY, USA, 30–39. DOI: https://doi.org/10.1145/2593735.2593740

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Communications of the ACM* 62, 12. 56–65. DOI: https://doi.org/10.1145/3318162

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. San Francisco, CA, USA. AAAI Press, 1345–1351.

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space With Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018).* Association for Computing Machinery, New York, NY, USA, 298–309. DOI: https://doi.org/10.1145/3213846.3213871

[18] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* Association for Computing Machinery, New York, NY, USA, 593–604. DOI: https://doi.org/10.1145/3106237.3106309

[19] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal*, 21(3). 421–443.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland. 3-13. DOI: https://doi.org/10.1109/ICSE.2012.6227211

[21] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20).* Association for Computing Machinery, New York, NY, USA, 602–614. DOI: https://doi.org/10.1145/3377811.3380345

[22] Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned From Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 234–242. DOI: https://doi.org/10.1145/2568225.2568324

[23] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. DOI: https://doi.org/10.1145/2771783.2771791

[24] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering (ICSE ‘19).* Montreal, Quebec, Canada. IEEE Press, 13–24. DOI: https://doi.org/10.1109/ICSE.2019.00020

[25] Mauricio Soto and Claire Le Goues. 2018 Using a probabilistic model to predict bug fixes. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy. 221–231. DOI: https://doi.org/10.1109/SANER.2018.8330211

[26] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE ‘18)*. Association for Computing Machinery, New York, NY, USA, 1–11. DOI: https://doi.org/10.1145/3180155.3180233

[27] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. Urbana-Champaign, IL, USA. IEEE Press. 660–670.

[28] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE ‘17).* Buenos Aires, Argentina. IEEE Press. 416–426. DOI: https://doi.org/10.1109/ICSE.2017.45

[29] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* Association for Computing Machinery, New York, NY, USA, 52–63. DOI: https://doi.org/10.1145/2635868.2635906

[30] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 187–198. DOI: https://doi.org/10.1145/3180155.3180243

[31] Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2017. A Generic Algorithm for Program Repair. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE ’17)*. Buenos Aires, Argentina. IEEE Press. 65-71

[32] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo F. Frias, Ali Jaoua, and Ali Mili. 2015. Relational Mathematics for Relative Correctness. In *RAMICS, 2015*, volume 9348 of LNCS, 191-208, Braga, Portugal. Springer Verlag.

[33] Robert R. Schaller. 1997. Moore's Law: Past, Present, and Future. *IEEE Spectrum 34*, 6. 52–59. DOI: https://doi.org/10.1109/6.591665

[34] Volodymyr Kindratenko and Pedro Trancoso. 2011. Trends in High-Performance Computing. In *Computing in Science and Engineering*. 13, 3. 92–95. DOI: https://doi.org/10.1109/MCSE.2011.52

[35] Mayank Daga, Ashwin M. Aji and Wu-chun Feng. 2011. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. *2011 Symposium on Application Accelerators in High-Performance Computing*. Knoxville, TN, USA. 141-149. DOI: https://doi.org/10.1109/SAAHPC.2011.29

[36] H. D. Benington. 1983. Production of Large Computer Programs. In *Annals of the History of Computing*, vol. 5, no. 4. IEEE. 350-361. DOI: https://doi.org/10.1109/MAHC.1983.10102

[37] Barry W. Boehm, 1988. A Spiral Model of Software Development and Enhancement. In *Computer*. 21, 05. IEEE. 61-72, 1988. DOI: https://doi.org/10.1109/2.59

[38] Kent Beck. 1999. Embracing Change with Extreme Programming. In *Computer*. 32, 10. IEEE. 70–77. DOI: https://doi.org/10.1109/2.796139

[39] Kent. Beck. 2000. *Extreme Programming Explained: Embrace Change*. Reading, MA, USA: Addison-Wesley, ISBN: 0201616416

[40] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. 2001. *Manifesto for Agile Software Development*. Retrieved on November 20, 2021 from https://agilemanifesto.org/.

[41] Ken Schwaber and Mike Beedle, 2001, *Agile Software Development with Scrum (1st ed.)* Upper Saddle River, NJ, USA: Prentice Hall PTR. 2001, ISBN: 0130676349

[42] Corey Ladas. 2008. *SCRUMBAN, And Other Essays on Kanban Systems for Lean Software Development*. Seattle, USA: A Division of Modus Cooperandi, Inc.

[43] Jaap-Henk Hoepman, Bart Jacobs. 2008. Increased Security Through Open Source. arXiv:0801.3924v1. Retrieved on November 20, 2021 from https://arxiv.org/abs/0801.3924v1

[44] Dirk Riehle. 2010. The Economic Case for Open Source Foundations. Computer 43, 1, 86–90. DOI: https://doi.org/10.1109/MC.2010.24

[45] Larry J. Morell. 1989. Unit Testing and Analysis, Software Engineering Institute, April 1989. CMU/SEI Report Number: CMU/SEI-89-CM-009. Carnegie Mellon University Software Engineering Institute. Pittsburgh, PA, USA. Retrieved on April 23, 2019 from https://apps.dtic.mil/dtic/tr/fulltext/u2/a236119.pdf

[46] Kent Beck, 2003. *Test-driven Development: By Example*. Reading, MA, USA: Addison-Wesley Professional.

[47] R. Owen Rogers. 2004. Scaling Continuous Integration. In *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering. XP 2004.* Garmisch-Partenkirchen, Germany. Springer. 68-76.

[48] Dan North. 2006. Introducing Behaviour Driven Development. *Better Software Magazine*. Retrieved online on November 20, 2021 from https://dannorth.net/introducing-bdd/

[49] Chadarat Phipathananunth and Panuchart Bunyakiati, 2018. Synthetic Runtime Monitoring of Microservices Software Architecture, In *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC '18).* Tokyo, Japan. IEEE Press. 448-453. DOI: https://doi.org/10.1109/COMPSAC.2018.10274

[50] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. DOI: https://doi.org/10.1145/1101908.1101949

[51] Rui Abreu, Peter Zoeteweij and Arjan JC Van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization, *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89-98, DOI: https://doi.org/10.1109/TAIC.PART.2007.13

[52] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox and Eric Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services, In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington D.C., USA. 595-604. DOI: https://doi.org/10.1109/DSN.2002.1029005

[53] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, Auckland, New Zealand. IEEE. 550-554, DOI: https://doi.org/10.1109/ASE.2009.15

[54] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated Fixing of Programs With Contracts. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 61–72. DOI: https://doi.org/10.1145/1831708.1831716

[55] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2011. Code-based Automated Program Fixing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. Lawrence, KS, USA IEEE. 392–395. DOI: https://doi.org/10.1109/ASE.2011.6100080

[56] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 43–54. DOI: https://doi.org/10.1145/2737924.2737988

[57] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. Florence, Italy. IEEE Press. 471–482. https://doi.org/10.1109/ICSE.2015.65

[58] Chen Liu, Jinqiu Yang, Lin Tan and Munawar Hafiz. 2013. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Luxembourg, Luxembourg. IEEE. 282-291. DOI: https://doi.org/10.1109/ICST.2013.24

[59] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. Paris, France. IEEE. 65-74. DOI: https://doi.org/10.1109/ICST.2010.66

[60] Junnosuke Matsumoto, Yoshiki Higo, Hiroyuki Matsuo, Ryo Arima, Shinsukue Matsumoto and Shinji Kusumoto. 2019. GenProg Meets Cluster Computing. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. Tokyo, Japan. 337-375. DOI: https://doi.org/10.1109/IWESEP49350.2019.00015

[61] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. DOI: https://doi.org/10.1145/2837614.2837617

[62] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 133–146. DOI: https://doi.org/10.1145/2384616.2384626

[63] Besma Khaireddine, Aleksandr Zakharchenko, Matias Martinez, Ali Mili. 2021. Toward a Theory of Program Repair, *Acta Informatica*. 2021. Revision submitted for review.

[64] Bat-Chen Rothenberg and Orna Grumberg. 2020. Must Fault Localization for Program Repair. In *Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV '20).* Virtual Event. 658–680.

[65] Jan A. Bergstra. 2020. Instruction Sequence Faults With Formal Change Justification. *Scientific Annals of Computer Science,* 30(2):105–166.

[66] Evren Ermis,Martin Schaef, and ThomasWies. 2012. Error Invariants. *In Proceedings of the 18th International Symposium on Formal Methods (FM '12)*. Paris, France. Springer. 187–201. DOI: https://doi.org/10.1007/978-3-642-32759-9_17

[67] Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* Association for Computing Machinery, New York, NY, USA, 437–446. DOI: https://doi.org/10.1145/1993498.1993550

[68] W Ric Wong, Ruizhi Gao, Yi Hao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey of Software Fault Localization. *IEEE Transactions on Software Engineering, 42*. 707–740.

[69] Ali Mili, Marcelo Frias, and Ali Jaoua. 2014. On Faults and Faulty Programs. In P. Hoefner, P. Jipsen, W. Kahl, and M. E. Mueller, editors, Proceedings, *RAMICS 2014*, volume 8428 of LNCS, pages 191–207.

[70] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020).* Association for Computing Machinery, New York, NY, USA, 75–87. DOI: https://doi.org/10.1145/3395363.3397351

[71] Maria Christakis, Matthias Heizmann, Muhammad Numair Mansur, Christian Schilling, and Valentin Wuestholz. 2019. Semantic Fault Localization and Suspiciousness Ranking. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '19)*. Prague, Czech Republic. Springer. 226–243.

[72] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, Association for Computing Machinery, New York, NY, USA, 341-353. DOI: https://doi.org/10.1145/3468264.3468544

[73] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Transactions on Software Engineering and Methodology* 30, 1, Article 6 (January 2021), 36 pages. DOI: https://doi.org/10.1145/3412376

[74] Kunihiro Noda, Haruki Yokoyama, and Shinji Kikuchi. 2021. Sirius: Static Program Repair with Dependence Graph-Based Systematic Edit Patterns. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME, '21)*. IEEE. 437-447. DOI: https://doi.org/10.1109/ICSME52107.2021.00045

[75] Stephen Cass. 2018. The 2018 Top Programming Languages. *IEEE Spectrum*. Retrieved online on November 20, 2021 from https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

[76] Stephen Cass. 2020. Top Programming Languages 2020 Python Rules the Roost, but Cobol Gets a Pandemic Bump. *IEEE Spectrum*. Retrieved online on November 20, 2021 from https://spectrum.ieee.org/top-programming-language-2020

[77] Robert W. Floyd. 1967. Assigning Meaning to Programs. In *Proceedings of the American Mathematical Society Symposium in Applied mathematics*, 19, New York, NY, USA. American Mathematical Society. 19–31, 1967. DOI: https://doi.org/10.1090/psapm/019/0235771

[78] C.A.R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Communications of the Association for Computing Machinery*, 12(10):576–583, October 1969.

[79] David Gries. 1981. *The Science of Programming*. Heidelberg, Germany: Springer Verlag.

[80] Eric C.R. Hehner. 1992. *A Practical Theory of Programming*. Englewood Cliffs, NJ, USA: Prentice Hall.

[81] Ralph-Johan J. Back, Abo Akademi, J. Von Wright, F. B. Schneider, and David Gries. 1998. *Refinement Calculus: A Systematic Introduction (1st. ed.).* Berlin, Heidelberg, Germany: Springer-Verlag.

[82] Carroll Morgan. 1998. *Programming From Specifications (2nd ed.)* Great Britain: Prentice Hall International (UK) Ltd.

[83] Zohar Manna. 1974. *A Mathematical Theory of Computation*. New York, NY: McGraw-Hill.

[84] Edsger Wybe Dijkstra. 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall.

[85] Harlan D. Mills, Victor R. Basili, John D. Gannon, and Dick R. Hamlet. 1986. Structured Programming: A Mathematical Approach. Boston, MA, USA: Allyn and Bacon.

[86] Nafi Diallo, Wided Ghardallou, Jules Desharnais, Marcelo Frias, Ali Jaoua, and Ali Mili. 2017. What Is a Fault? And Why Does It Matter? *Innovations in Systems and Software Engineering.* 13, 2–3. 219–239. DOI: https://doi.org/10.1007/s11334-017-0300-7

[87] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*. 1, 1. 11–33. DOI: https://doi.org/10.1109/TDSC.2004.2

[88] Jean-Claude Laprie. 1991. *Dependability: Basic Concepts and Terminology. In English, French, German, Italian and Japanese*. Wien, Austria: Springer Verlag.

[89] Jean-Claude Laprie. 1995. Dependability — Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems*, Berlin, Heidelberg, Germany. Springer. 3-18.

[90] Jean-Claude Laprie. 2004. Dependable Computing: Concepts, Challenges, Directions. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04).* 1. Hong Kong, China. IEEE. 242.

[91] Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2020. A Semantic Definition of Faults and Its Implications. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Macau, China. IEEE. 14-21. DOI: https://doi.org/10.1109/QRS51102.2020.00015

[92] Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. 2019. Fault Density, Fault Depth and Fault Multiplicity: The Reward of Discernment. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C.)* Sofia, Bulgaria. IEEE. 532-533.

[93] Besma Khaireddine, Marwa Ben AbdelAli, Lamia Labed Jilani, Aleksandr Zakharchenko, and Ali Mili. 2020. Correctness Enhancement: a Pervasive Software Engineering Paradigm. In *International Journal of Critical Computer-Based Systems*. 10(1), 37-73.

[94] Besma Khaireddine, Matias Martinez, and Ali Mili. 2019. Program Repair at Arbitrary Fault Depth. In *International Conference on Software Testing (ICST '19.)* Xian, China. IEEE. April 2019. 465-472.

[95] Aleksandr Zakharchenko, Besma Khaireddine, and Ali Mili. 2021. A Massively Parallel Approach to Automated Software Correctness Enhancement in Java. In *New Trends in Intelligent Software Methodologies, Tools and Techniques*, IOS Press. 141-154. DOI: https://doi.org/10.3233/FAIA210015

[96] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2005. MuJava : An Automated Class Mutation System. *Journal of Software Testing, Verification and Reliability*, 15(2):97-133, June 2005. Retrieved on November 20, 2021 from https://cs.gmu.edu/~offutt/rsrch/papers/mujava.pdf

[97] Aleksandr Zakharchenko and James Geller. 2015. Auditing of SNOMED CT's Hierarchical Structure using the National Drug File - Reference Terminology. *Studies in Health Technology and Informatics*. 210. IOS Press. 130-134. DOI: https://doi.org/10.3233/978-1-61499-512-8-130

[98] Aleksandr Zakharchenko and James Geller. 2016. Expansion of the Hierarchical Terminology Auditing Framework Through Usage of Levenshtein Distance-Based Criterion. *Studies in Health Technology and Informatics*. 228. IOS Press. 491-495. DOI: https://doi.org/10.3233/978-1-61499-678-1-491

[99] NJIT High Performance Computing Machine Specifications. Retrieved on October 10, 2021 from https://ist.njit.edu/high-performance-computing-machine-specifications/

[100] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: a Large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 302–313. DOI: https://doi.org/10.1145/3338906.3338911

[101] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. 2019. Automated Repair of Cross-Site Scripting Vulnerabilities through Unit Testing. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '19.)* IEEE. 370-377.

[102] Alexander Marchand-Melsom and Duong Bao Nguyen Mai. 2020. Automatic repair of OWASP Top 10 security vulnerabilities: A survey. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW '20)*. Association for Computing Machinery, New York, NY, USA, 23–30. DOI: https://doi.org/10.1145/3387940.3392200

[103] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP '19.)* IEEE. 539-554.

[104] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 15–26. DOI: https://doi.org/10.1145/3213846.3213851