Automated Program Repair: A Perspective from Metamorphic Testing

Mingyue Jiang

Faculty of Science, Engineering and Technology
Swinburne University of Technology
Australia

This thesis is submitted for the degree of
Doctor of Philosophy

June 2017

Abstract

Test suite based automated program repair (which is abbreviated as test suite based APR, or simply APR) refers to a broad category of techniques that use input test suites to automatically fix faulty programs. Test suite based APR techniques have been receiving increasing attention in recent years, and several novel repair methodologies have been developed that have yielded very promising results. However, APR still faces a number of challenges, and there is a pressing need to both extend its scope of applicability, and enhance its repair effectiveness.

This thesis reports on the application of *metamorphic testing* (MT) concepts to address some challenging APR problems. A series of formalisations of different aspects of test suite based APR are introduced, and later used to support the systematic measurements and evaluations in the thesis. In order to extend the scope of applicability of test suite based APR techniques, an integration of APR and MT, called APR-MT, is proposed. A key advantage of APR-MT compared with conventional APR techniques is that it does not rely on the availability and feasible use of a test oracle, and thus can be applied to repair a broader range of programs. The problem of how to improve APR repair effectiveness is also addressed, from two perspectives. Firstly, motivated by the observation that a better input test suite helps APR techniques to deliver a higher repair effectiveness, a novel input test suite generation approach is proposed. This approach uses specific information related to program properties and failures, generating test suites that contain richer information than those constructed by other approaches, such as random and coverage based. Secondly, an innovative APR approach drawing on the strengths of MT, MTRepair, is proposed. MTRepair has several novel aspects, including the use of a metamorphic relation instead of a test suite as input, a quality based program validation procedure, and an incremental repair process.

The proposed approaches were implemented into a set of prototype tools, and have been evaluated using several subject programs from benchmark suites commonly used by the APR community. These approaches are demonstrated to be not only feasible, but also

effective, and have revealed some important and promising findings: (1) in the application of APR-MT techniques, instead of complete test oracles, the use of metamorphic relations (which can be regarded as partial test oracles) does not significantly deteriorate the repair effectiveness; (2) compared with random and coverage based test suite generation approaches, the proposed input test suite generation approach is more effective for APR techniques that semantically synthesise a repair; and (3) the characteristics of MTRepair do contribute to a higher repair effectiveness.

Acknowledgements

First and foremost, I sincerely express my deepest gratitude to my supervisor Prof. Tsong Yueh Chen for his invaluable guidance and continuous support during my PhD study. He has introduced me into the research field of program repair, and has devoted numerous time and patience discussing my ideas and giving me countless comments and suggestions. His experienced knowledge and insights have taught me the right way to do research, and his wisdom and encouragement have made me both more confident to overcome difficulties and more mature in my way of academic. I am grateful for all the help and guidance he has provided me in both my PhD study and my life.

I would like to thank Dr. Fei-Ching Kuo for supporting me so much in both my studies and my life. I also would like to specifically thank Zhi Quan Zhou and Dave Towey for their critical comments and helps in improving my approaches and my writing. Thanks to staff members, research assistants and research students at FICT, in particular, Prof. Yun Yang, Dr. Qiang He, Lu Chen, Ru Jia, Yanchun Wang. Many thanks to my previous supervisor (for Master degree) Zuohua Ding, who opened the door for my academic study and has kept supporting me all the time. I am also thankful to my friends for their friendships and encouragements.

Finally, special thanks to my family. I am deeply grateful to my parent, my father Xianwen Jiang and my mother Guineng Wang, for their endless love and continuous support. A heartfelt thank goes out to my dear husband, Chao Liu, for his love, understanding, support and sacrifice. Their understanding and help are the most reliable foundation supporting me to pursue my dream.

Declaration

I hereby declare that the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university and that to the best of my knowledge this thesis contains no material previously published or written by another person except where due reference is made in the text of this thesis.

Mingyue Jiang
June 2017

Table of contents

List of figures **x**

List of tables **xii**

1 Introduction **1**

1.1 Automated program repair 1

1.2 Motivation 2

1.3 Contributions 4

1.3.1 Formalisation of APR concepts 4

1.3.2 Program repair without the need for a test oracle 5

1.3.3 A novel approach for constructing effective APR input test suites 6

1.3.4 MTRepair: An MT based APR approach 7

1.4 Organisation 7

2 Background **9**

2.1 Test suite based automated program repair 9

2.2 Metamorphic testing 11

2.3 Metamorphic failure-causing condition 13

2.3.1 Semi-proving 14

2.3.2 Illustration of the MFCC generation 15

3 Formalisations 16

3.1 Formalisation of repair quality 16

3.2 Formalisation of test suite based APR 19

3.3 Formalisation of the input test suite 20

3.4 Related work 23

3.4.1 Repair quality 23

3.4.2 The APR input test suite 24

4 Program repair without the need for a test oracle 26

4.1 Preliminary 26

4.2 Test suite based APR in the absence of a test oracle 27

4.2.1 Framework 28

4.2.2 Implementations 29

4.3 Experimental setup 33

4.3.1 Subject programs 34

4.3.2 APR tools setup 38

4.3.3 Measurements 40

4.4 Experimental results 42

4.4.1 Experimental results for GenProg and GenProg-MT 42

4.4.2 Experimental results for CETI and CETI-MT 54

4.4.3 Summary 58

4.5 Discussion 58

4.5.1 Impact of MRs on the effectiveness of APR-MT techniques . . . 58

4.5.2 Impact of the source test cases on the effectiveness of APR-MT techniques 59

4.6 Conclusion 59

5 A novel approach for constructing effective APR input test suites 61

5.1 Preliminary 61

5.2 A novel approach for APR test suite generation 63

5.2.1 Intuition and motivation 63

5.2.2 Generating test cases from an MFCC 64

5.2.3 Generating a test suite from a set of MFCCs 66

5.2.4 MFCC based input test suites 67

5.3 Experimental design 68

5.3.1 Subject programs and MRs 68

5.3.2 Test suite based APR tools used in the experiments 75

5.3.3 Test suite generation approaches 77

5.4 Experimental results 78

5.4.1 Independent and dependent variables 78

5.4.2 Analysis of the usefulness of input test suite generation approaches 79

5.4.3 Interplay between the test suite generation approaches and the APR techniques 86

5.5 Related work 88

5.6 Discussion 88

5.6.1 Input test suites for APR 89

5.6.2 Limitations 92

5.6.3 Threats to validity 93

5.7 Conclusion 93

6 MTRepair: An MT based APR approach 95

6.1 Motivation 95

6.2 MTRepair overview 97
6.3 MTRepair technique 100
6.3.1 Candidate program validation in MTRepair 101
6.3.2 Candidate program construction in MTRepair 103
6.3.3 Repairing a PUR 106
6.4 Implementation 107
6.5 Evaluation 109
6.6 Related work 112
6.7 Conclusion 114

7 Conclusion **116**
7.1 Summary 116
7.2 Future work 118

References **120**

List of the author's publications **128**

List of figures

2.1 Program *Max* (with fault) 10

2.2 Illustrative example of test suite based APR 10

2.3 A shortest path from vertex V_1 to vertex V_7 12

3.1 Motivating example 17

3.2 The failure regions of two repairs differ dramatically in size 18

4.1 The GenProg-MT approach 30

4.2 A reachability instance program constructed by CETI with the application of the constant template 32

4.3 A reachability instance program constructed by CETI-MT applying the constant template 33

4.4 Distribution of repair qualities 46

4.4 Distribution of repair qualities (*Continued*) 47

4.4 Distribution of repair qualities (*Continued*) 48

4.5 Comparisons of *groupB* repair quality 56

4.6 Comparisons of *groupW* repair quality 57

5.1 Quality of repairs produced by Angelix, with the application of different test suite generation approaches 83

5.2 Quality of repairs produced by CETI, with the application of different test suite generation approaches 84

5.3 Quality of repairs produced by GenProg, with the application of different test suite generation approaches 85

5.4 Coverage of test suites generated by different approaches (for ease of presentation, a limit was set on the lower bound of the y-axis) 91

6.1 MTRepair repairing a PUR containing one fault 99

6.2 MTRepair repairing a PUR containing two faults 100

6.3 The architecture of MTRepair tool. 108

6.4 Repair quality analysis for program *median* 110

6.5 Repair quality analysis for program *smallest* 111

List of tables

2.1 The source and follow-up symbolic evaluation results of *Max* 15

4.1 Correspondence between conventional APR and APR-MT techniques . . 28

4.2 Subject programs 35

4.3 Sizes of test suites and MTG sets 38

4.4 *Success rates* for GenProg and GenProg-MT. SR_v = number of versions repaired / number of versions to which the scenario is applied, and SR_p = number of successful repair processes/ number of repair processes conducted. 44

4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality 49

4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality (*Continued*) 50

4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality (*Continued*) 51

4.6 Summary of comparison results 52

4.7 Repair time for different scenarios of GenProg and GenProg-MT (in seconds) 53

4.8 *Success rates* for CETI and CETI-MT 55

4.9 Repair time for different scenarios of CETI and CETI-MT (in seconds) . . 58

5.1 Repairing the *isUpward* program 62

5.2 Subject packages 69

5.3 Test suite information 80

5.4 The numbers of repaired programs and plausible repairs ("-" means that the APR tool is not applicable) 81

5.5 Applicability of the test suites generated by different approaches 82

5.6 Usefulness of the test suites generated by different approaches 86

5.7 Mean repair costs, measured in execution time (seconds) per repair, including the test suite generation time and the APR tool execution time ("-" means that no repair was generated and thus no time cost collected) . . . 89

Chapter 1

Introduction

1.1 Automated program repair

The writing of correct programs has always been a great challenge, and even mature commercial software systems are frequently shipped containing both known and unknown bugs [Jalbert and Weimer, 2008][1] . Bugs can impede a program from delivering expected functionality, thereby decreasing the quality of the overall software system. It is therefore essential that program bugs be fixed in order to ensure the software quality.

Bug fixing is the activity aiming at transforming a faulty program into a correct one. It is a crucial part of program debugging and program maintenance, and has historically been considered the responsibility of programmers [Arcuri and Yao, 2008; Fry et al., 2012]. Unfortunately, the manual bug fixing process is not only time-consuming and complicated, but also can itself be error-prone — faults may propagate from one place to others within a program [Malik et al., 2009], and inappropriate modifications of a program may introduce new faults [Stutzke and Smidts, 2001]. Moreover, increases in the complexity and the number of faults directly lead to increases in the demand for resources, including financial costs, for fixing them [Britton et al., 2013]. There is, therefore, a great demand for automated techniques for fixing bugs.

In the past decade, automated program repair techniques have emerged to automatically repair faulty programs, without any human intervention [Arcuri, 2011; Arcuri and Yao, 2008; Gopinath et al., 2011; Jobstmann et al., 2005; Le Goues et al., 2013, 2015, 2012b;

[1] In this thesis, the terms *bug*, *fault* and *defect* are used interchangeably.

Monperrus, 2014; Pei et al., 2014, 2011; Wei et al., 2010]. This automatic repair process can significantly reduce costs, especially compared with manual bug fixing, and also targets to improve the quality of the *program under repair* (PUR).

A broad category of automated program repair is *test suite based automated program repair* [DeMarco et al., 2014; Jeffrey et al., 2009; Kaleeswaran et al., 2014; Kim et al., 2013; Le Goues et al., 2015, 2012b; Mechtaev et al., 2015, 2016; Nguyen et al., 2013; Nguyen, 2014; Qi et al., 2014; Weimer et al., 2013, 2009], which takes as input a PUR and a test suite, and attempts to produce a repair[2]. A repair is a program variant of the PUR (candidate program) that can pass all the test cases in the provided input test suite. Notably, the input test suite acts as a kind of specification for repairing the PUR — it must contain at least one *failing test case*, for which the PUR's output is incorrect; it may also contain some *passing test cases*, for which the PUR's outputs are correct. The failing test cases provide information about faults in the PUR, and the passing test cases describe the intended behaviour of the PUR. This thesis focuses only on test suite based automated program repair, *which will be abbreviated as test suite based APR, or simply APR*.

Due to the popularity of test suites in both academia and industry, test suite based APR is attractive and applicable from the practical point of view. Substantial progress has been made in this area, with various novel approaches having been developed, demonstrating the potential of APR techniques for repairing programs. ClearView [Perkins et al., 2009], for example, has been shown to generate repairs that successfully eliminated security vulnerabilities in seven out of ten defects. GenProg, a prominent test suite based APR technique, has also successfully repaired 55 out of 105 defects in eight open-source C programs, with an average cost per repair of only US$7.32 [Le Goues et al., 2012a]. Par [Kim et al., 2013] has generated repairs for 27 out of 119 real bugs in open-source Java projects. A random search based APR technique, RSRepair, has been reported to be more efficient than GenProg in repairing 24 versions of seven real-life programs [Qi et al., 2014]; and SPR, another APR technique, has been demonstrated to be able to generate more repairs than GenProg [Long and Rinard, 2015].

1.2 Motivation

In spite of the advances in test suite based APR, concerns remain about its *scope of applicability* (how many application domains can the APR technique be applied in), and the *repair effectiveness* (how effective is the APR technique at repairing programs)

[2] In this thesis, the terms *repair* and *patch* are used interchangeably.

[Le Goues et al., 2013, 2012b; Qi et al., 2015; Smith et al., 2015]. Both of these concerns are critical for the practical application of APR techniques, but, due to the nature of test suite based APR, both face several limitations [Le Goues et al., 2013; Monperrus, 2014]. There is, therefore, a pressing need to extend the scope of applicability for APR techniques and to improve its repair effectiveness, and thereby increase the practical benefits of test suite based APR techniques.

A key factor impacting the scope of applicability of test suite based APR techniques is the *test oracle* [Le Goues et al., 2013] — a mechanism to verify the correctness of any test case's execution result [Barr et al., 2015; Liu et al., 2014]. Test suite based APR requires a test oracle: the APR technique has to know the correctness of every execution of the individual test cases of the input test suite, and thus these APR techniques may not be applicable in the absence of a test oracle. In other words, the application of test suite based APR techniques is restricted by the feasibility and availability of a test oracle. Current test suite based APR techniques assume that there is a set of complete test cases (both test inputs and test oracles). GenProg [Forrest et al., 2009], for example, applies an oracle comparator function to check the output of each test input, and JAFF [Arcuri, 2011] checks the execution result of individual test cases against assertions. The majority of other techniques (such as DirectFix [Mechtaev et al., 2015], NoPOL [DeMarco et al., 2014], and SemFix [Nguyen et al., 2013]), simply use a set of test cases for which the outputs are known.

Nevertheless, in many applications, a test oracle may not be available, or may be available, but too expensive to be applied — a situation known as the test oracle problem [Barr et al., 2015]. Similar to other software engineering activities such as testing and fault localization, test suite based APR faces the test oracle problem. *Any approach that can alleviate the test oracle problem in APR will extend the scope of applicability for APR techniques*. This thesis represents the first study to address the test oracle problem in test suite based APR.

Improving repair effectiveness has been a basic motivation in program repair research, inspiring the development of innovative repair methods and strategies that support or improve existing methods [Le Goues et al., 2013; Qi et al., 2015]. Since 2009, test suite based APR has generated a growing body of research [DeMarco et al., 2014; Kim et al., 2013; Le Goues et al., 2012b; Mechtaev et al., 2015, 2016; Nguyen et al., 2013; Nguyen, 2014; Qi et al., 2014; Weimer et al., 2013, 2009]. The majority of these studies focus on repair methodologies, proposing innovative APR methods for repairing different types of faults, programs implemented in different languages, and large and complex programs. Some other studies have proposed strategies to improve existing APR methods,

or investigated factors affecting the repair effectiveness of existing APR methods: two strategies have been proposed to improve APR techniques using genetic searches, one for designing better fitness functions and the other for selecting appropriate representations of program variants and operators [Fast et al., 2010; Le Goues et al., 2012c]. To enhance the capability of APR techniques that use a stochastic search based repair process, a strategy was proposed to leverage historical bug fixes from open source projects to assess the fitness of candidate programs [Le et al., 2016b]. A controlled experiment was conducted to study the impact of different fault localization techniques on APR [Assiri and Bieman, 2016]. Another recent empirical study investigated the effectiveness of different synthesis engines for APR [Le et al., 2016a].

Most importantly, in-depth investigations into the repair results of some existing APR techniques have revealed that their repair effectiveness is not encouraging because most of the generated repairs are not correct [Qi et al., 2015], and repairs produced by some APR techniques tend to overfit to the provided input test suites [Smith et al., 2015]. Therefore, in spite of the effort already devoted to test suite based APR, *more work is still needed to further advance this field*.

1.3 Contributions

The goal of this thesis is to report on extending the scope of applicability as well as enhancing the repair effectiveness of test suite based APR. Because an APR technique takes a test suite as input (in addition to the PUR), and its repair process consists of several modules (such as the module locating faulty statements and the module constructing repairs), both the characteristics of the input test suite and the performance of these modules may impact on the scope of applicability and repair effectiveness of the APR technique. Intuitively, therefore, it should be possible to extend the scope of applicability and achieve a higher repair effectiveness by improving the input test suite or the methodologies implementing the repair process.

1.3.1 Formalisation of APR concepts

The repair effectiveness of an APR technique is reflected in *its ability to generate repairs* (how likely a PUR can be repaired to pass the entire input test suite), and the *quality of its resulting repairs* (how correct the constructed repair is with respect to the entire input

domain). Although test suite based APR has been intensively studied, there has been little exploration of systematic approaches for repair effectiveness evaluation, especially with respect to the quality of the repairs. Furthermore, no study has yet been conducted to quantify the effectiveness of the input test suite from the perspective of program repair.

In order to develop systematic approaches to evaluate APR techniques and input test suites, a series of concepts for test suite based APR are formalised. The formalisation provides systematic approaches for measuring the quality of a repair, and also characterises the impact of input test suites on APR. Based on these concepts, some evaluation metrics for APR techniques, and for measuring the effectiveness of input test suites on APR, are proposed. These formalisations and evaluation metrics form the basis of the investigations reported in this thesis.

1.3.2 Program repair without the need for a test oracle

To extend the scope of applicability of APR techniques, an approach to alleviate the test oracle problem of test suite based APR is proposed. The basic idea of the approach involves integration of *metamorphic testing* (MT) [Chen et al., 2003, 2011], a testing strategy that is effective in alleviating the test oracle problem, with test suite based APR. The approach makes changes to both the input data and some modules of the conventional APR process, and the resulting integrated technique, called APR-MT, can be applied without reference to a test oracle. This study makes the following key contributions:

- **Framework.** A general framework is developed to support the approach, which describes in detail how to integrate MT with test suite based APR, including how to handle the input data and how to adjust some of the APR process modules. The framework can be adapted to a wide range of APR techniques.
- **Implementation.** The proposed framework is applied to two different APR techniques, GenProg [Le Goues et al., 2012b] and CETI [Nguyen, 2014], resulting in two APR-MT techniques, GenProg-MT and GenProg-CETI. Because both GenProg-MT and CETI-MT no longer require a test oracle to repair a program, they can be applied regardless of whether or not one exists. These two implementations demonstrate the feasibility of the proposed framework.
- **Evaluation.** An empirical study is conducted to investigate the effectiveness of the APR-MT techniques. The experiments compare the repair effectiveness of GenProg-MT and CETI-MT with that of GenProg and CETI, in terms of the ability

to repair programs from the IntroClass benchmark suite [Le Goues et al., 2015]. The empirical results show that the proposed integration is both practically feasible and effective, and thus successfully extends test suite based APR techniques to a broader application domain.

1.3.3 A novel approach for constructing effective APR input test suites

APR techniques repair the PUR by referring to the input test suite. Moreover, it is a common practice for the APR technique to accept a repair if it passes all test cases of the given input test suite. Obviously, the input test suite therefore has a direct impact on the repair effectiveness. Intuitively, a *good* test suite should provide useful information for the repair of the PUR, hence giving a higher repair effectiveness. Many studies have expressed a strong desire for better test suites in order to achieve higher repair effectiveness [Monperrus, 2014; Qi et al., 2015; Smith et al., 2015]; however, to date, there have been few attempts to design new input test suite generation approaches specifically for APR.

With the observation that *the enhancement of the input test suite will in turn improve the repair effectiveness of the APR techniques*, a novel input test suite generation approach for APR is proposed. The key contributions are as follows:

- **A novel approach for APR test suite generation.** A novel input test suite generation approach for APR, using the concepts of *metamorphic relation* and *metamorphic failure-causing condition* [Chen et al., 2003, 2011], is proposed. In addition to the information associated with individual test cases of the test suite, the resulting input test suite contains information related to the satisfaction and violation of the relevant metamorphic relations. The rationale of this approach is that violated metamorphic relations provide additional useful information for APR. This is the first instance of a test suite generation approach purposely designed for APR.

- **Empirical evaluation.** A series of experiments are conducted to compare the proposed approach against random and code coverage based test suite generation approaches. In the experiments, three APR tools (Angelix [Mechtaev et al., 2016], CETI [Nguyen, 2014], and GenProg [Le Goues et al., 2012b]) are applied to both the Siemens programs [SIR, 2005] and large scale programs taken from the ManyBugs benchmark suite [Le Goues et al., 2015]. The empirical results show that the approach is promising, especially for APR techniques that utilise the input test suite

for semantically constructing repairs or candidate programs, such as Angelix and CETI.

- **New insights.** The experimental results lead to an in-depth analysis of the interplay between the input test suites and the relevant APR techniques. This provides new insights for future APR research.

1.3.4 MTRepair: An MT based APR approach

Inspired by the characteristics of MT and its related concepts, a new APR approach is developed. The key contributions are as follows:

- **The design of a novel APR approach.** MTRepair, an APR approach built on a series of concepts related to MT, is proposed. MTRepair accepts a metamorphic relation and a PUR as input — an input test suite is no longer needed — and its goal is to generate a repair that can satisfy the given metamorphic relation. To construct a repair, MTRepair adopts a specific measurement for validating candidate programs and conducts the repair tasks in an incremental manner.
- **Implementation and evaluation.** The MTRepair approach is implemented into a prototype tool, and evaluated using some subject programs from the IntroClass benchmark suite [Le Goues et al., 2015]. The comparison between MTRepair and GenProg shows that MTRepair is more effective for repairing the selected subject programs.

1.4 Organisation

The rest of this thesis is structured as follows.

Chapter 2 introduces some background to the thesis. It first presents the details of test suite based APR, and then introduces MT, explaining how it is typically conducted. Finally, the chapter explains the concept of metamorphic failure-causing conditions, and further illustrates how they can be generated.

Chapter 3 focuses on the formalisation of APR concepts. It first provides formalisations for repair quality, followed by a systematic approach to measuring and comparing repair

quality. Based on this, a series of concepts related to APR are formalised. Finally, the chapter provides a series of formalisations to characterise the APR input test suite, leading to a systematic approach for measuring the effectiveness of input test suites.

Chapter 4 examines the problem of extending the scope of applicability of APR techniques by alleviating the test oracle problem. It first presents the framework for integrating MT with test suite based APR, explaining how to alleviate the test oracle problem of test suite based APR. Then, it gives the implementation details of two APR-MT techniques, CETI-MT and GenProg-MT. This chapter presents the empirical studies into these techniques, and analyses the experimental results. Finally, some important issues related to the proposed approach are discussed.

Chapter 5 presents a novel approach for constructing effective input test suites specifically for APR. It first gives a motivating study to reveal the impact of input test suites on repair effectiveness. The chapter then presents a novel test suite generation approach, and the experimental results demonstrating the effectiveness of input test suites constructed by the approach. This is followed by a discussion of the relationship between input test suites and APR techniques. Finally, this chapter discusses properties of APR input test suites, and limitations of the proposed test suite generation approach.

Chapter 6 discusses the APR approach MTRepair. It first explains the motivations behind the design of MTRepair, then presents the MTRepair methodology, including an overview of the entire repair process and a description of core strategies of MTRepair. This chapter finally provides implementation details of MTRepair, and reports on its experimental evaluation.

Chapter 7 summarises the thesis, and discusses possible research directions and topics for future work.

Chapter 2

Background

This chapter provides the background for the research presented in this thesis. First, an overview of test suite based automated program repair is presented in Section 2.1, then, metamorphic testing and metamorphic failure-causing conditions are introduced in Sections 2.2 and 2.3, respectively.

2.1 Test suite based automated program repair

Test suite based automated program repair (which, in this thesis, will be referred to as test suite based APR or simply APR) uses the information provided by an input test suite to repair a program. APR accepts as input a faulty program (which is known as the *program under repair*, PUR), and a test suite containing both *passing* and *failing* test cases — the PUR produces correct outputs for individual passing test cases, but incorrect outputs for failing test cases. APR attempts to fix the PUR by producing a *repair* — a program variant of the PUR (candidate program) that can pass all test cases in the input test suite. If such a program variant can be generated, a repair is said to have been completed with respect to the input test suite; otherwise, the APR technique is said to have failed to repair the PUR.

To clearly explain the test suite based APR procedure, the program *Max* (Figure 2.1) is used as an illustrative example. *Max* attempts to find the maximum of two input integers, but it contains a fault at line 1. The procedure for repairing *Max* is shown in Figure 2.2. For the given test cases, *Max* computes correct outputs for test cases t_1 and t_2, but gives incorrect output for t_3. In this situation, the two passing test cases (t_1 and t_2) and the failing

```
  int Max(int x,int y)
  {
1:    if(x>=y+10) /* Should be: if(x>=y). */
2:       return x;
3:    else
4:       return y;
  }
```

Fig. 2.1 Program *Max* (with fault)

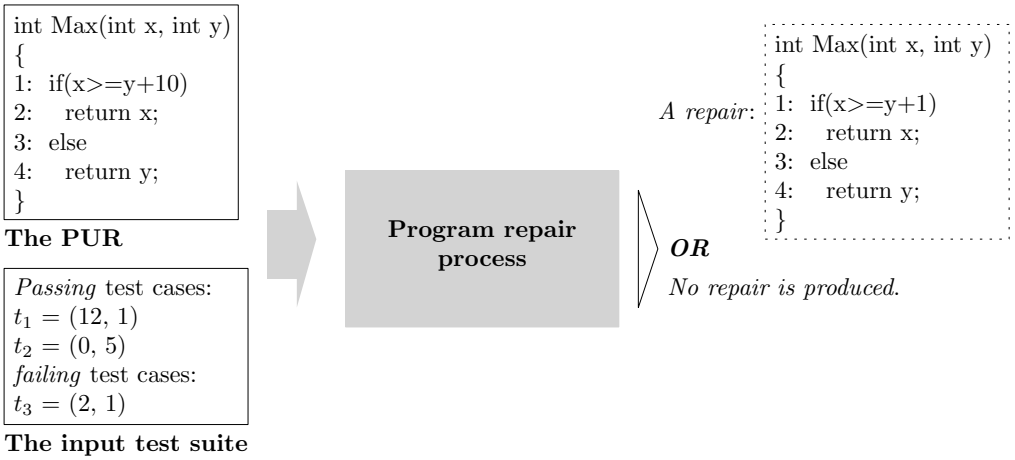


Fig. 2.2 Illustrative example of test suite based APR

test case (t_3) constitute an input test suite for repairing *Max*. Given the PUR *Max* and this input test suite, an APR technique uses its specific repair algorithm to explore a repair: for example, a possible repair with respect to this input test suite is given in Figure 2.2.

Although different APR techniques may produce different repairs (even if using the same PUR and input test suite), all repairs pass the entire input test suite. On the other hand, it is also very likely that a repair may not be delivered. The reason for these various outcomes is that repairs are determined by several factors, including: the repair algorithm; the class of faults in the PUR; and the information provided by the input test suite. Most importantly, (i) *test suite based APR requires a test oracle* to determine whether an individual test case of the input test suite passes or fails, and (ii) *a repair constructed by APR may still fail on test cases not included in the input test suite*.

Typically, test suite based APR techniques are classified as either *generate-and-validate* or *semantics-based* [Le Goues et al., 2015; Mechtaev et al., 2016]. Generate-and-validate techniques construct a set of candidate programs, based on which a validation process determines whether or not a repair exists (e.g., AE [Weimer et al., 2013], ClearView [Perkins et al., 2009], GenProg [Forrest et al., 2009; Le Goues et al., 2012a,b], JAFF [Arcuri, 2011], Kali [Qi et al., 2015], PAR [Kim et al., 2013], PACHIKA [Dallmeier et al., 2009], RSRepair [Qi et al., 2014], and TrpAutoRepair [Qi et al., 2013]). Semantics-based techniques, on the other hand, encode the input test suite into a formula or constraint that

the expected repair should satisfy, based on which some technique is applied to derive the repair (e.g., Angelix [Mechtaev et al., 2016], CETI [Nguyen, 2014], DirectFix [Mechtaev et al., 2015], NoPOL [DeMarco et al., 2014], and SemFix [Nguyen et al., 2013]). These two classes of APR techniques use very different repair processes, with both strengths and weaknesses.

2.2 Metamorphic testing

Metamorphic testing (MT) [Chen et al., 1998, 2001, 2003; Liu et al., 2014; Segura et al., 2016; Zhou et al., 2016] is a property-based software testing approach, the most basic and important concept of which is the *metamorphic relation* (MR), which specifies program properties through a relation among *multiple* test cases and their outputs. In MT, MRs are used for generating test cases and for checking test results. MT differs from conventional testing methods in that instead of focusing on the correctness of each individual output, it checks the relationships among the inputs and outputs of *multiple* executions of the program under test. This means that MT does not require a test oracle, and therefore *can alleviate the test oracle problem* [Chen, 2015].

A growing body of research has reported that MT has excellent fault-detection capability [Chen et al., 2016, 2005; Kuo et al., 2010; Lindvall et al., 2015; Murphy et al., 2009; Núñez and Hierons, 2015; Segura et al., 2015; Xie et al., 2011; Zhou et al., 2012], including successfully detecting real-life faults in the Siemens suite [Xie et al., 2013] and two popular compilers [Le et al., 2014]. Moreover, MT has been applied to test various applications facing the test oracle problem, including scientific programs [Kanewala and Bieman, 2013], machine learning algorithms [Xie et al., 2011], heuristic algorithms [Barus et al., 2011], web services [Chen et al., 2012], model transformations [Jiang et al., 2014], bioinformatics programs [Chen et al., 2009], and data access systems [Lindvall et al., 2015].

The *ShortestPath* (*SP*) program can be used to illustrate how MT works, and also to demonstrate how MT alleviates the test oracle problem: *SP* accepts an undirected graph and two of its vertices as inputs, and then searches through the graph to find the shortest path from one vertex to another vertex. For example, consider the graph shown in Figure 2.3. Suppose all edges in this graph have the same length. Although there are several paths from vertex V_1 to V_7 (such as '$V_1-V_2-V_4-V_5-V_6-V_7$' and '$V_1-V_2-V_3-V_4-V_5-V_6-V_7$'), *SP* will report the shortest path from V_1 to V_7 as '$V_1-V_2-V_4-V_5-V_7$' (the highlighted path shown in Figure 2.3).

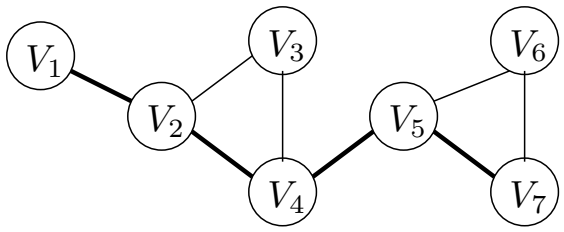


Fig. 2.3 A shortest path from vertex V_1 to vertex V_7

Testing *SP* may face the test oracle problem, because it can be difficult and expensive to verify whether the shortest path between two selected vertices of a large graph is correct or not. Nevertheless, *SP* can be tested using MT by considering a property related to the *SP* algorithm: that the length of the shorted path between two vertices remains unchanged regardless of which vertex is selected as the starting point, that is: $|SP(G,\ a,\ b)| = |SP(G,\ b,\ a)|$ (where G is an undirected graph, and a and b are two of its vertices). This is an MR, and it can be used to construct test cases as well as to check whether or not the relevant property is satisfied by the target program.

Given an undirected graph G_0, consisting of the n vertices v_1, ..., v_n ($n > 1$), and assume the existence of a test case $t_1 = (G_0, v_i, v_j)$, where $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$. Based on t_1, another test case can be constructed according to the MR above: $t_2 = (G_0, v_j, v_i)$. In this case, t_1 is referred to as the *source test case*, and t_2 is the *follow-up test case*. In MT, source test cases may be constructed by applying some existing test case generation strategy, but follow-up test cases must be constructed from source test cases according to the relevant MR. A source test case and its follow-up test case form a *metamorphic test group* (MTG) of the relevant MR. For example, $mtg_1 = (t_1, t_2)$ is an MTG for the MR above. Given a set of source test cases and an MR, a set of MTGs can be constructed and then used to test a program.

To test program *SP* using mtg_1, MT executes *SP* with t_1 and t_2, obtaining their outputs $SP(t_1)$ and $SP(t_2)$ (referred to as the source and follow-up outputs, respectively). Then, instead of verifying the correctness of the individual outputs, MT checks the source and follow-up test cases and their outputs against the relevant MR — MT checks whether or not $|SP(t_1)| = |SP(t_2)|$ holds. If this relationship does not hold, it indicates that *SP* violates the above MR, and, therefore, is faulty. In this case, mtg_1 is a *violating* MTG revealing violation of the MR; otherwise, mtg_1 is a *non-violating* MTG. In general, although it is possible to confirm that at least one test case of a violating MTG fails, it is not possible to know exactly which one.

It is important to emphasise that *MT can be applied regardless of whether or not a test oracle exists*. In particular, the synergy between MT and a test oracle in some applications

can deliver specific advantages: previous studies have demonstrated MT's application beyond testing, in activities such as proving [Chen et al., 2011], fault localization [Xie et al., 2011], fault tolerance [Liu et al., 2014], and validation [Zhou et al., 2016]. In this thesis, MT is applied to test suite based APR, representing the first attempt to explore the strength of MT in program repair. In the study investigating alleviation of the test oracle problem of APR, MT is applied without the use of a test oracle (Chapter 4). However, in the study of the input test suite generation approach (Chapter 5) and the study of repair methodology (Chapter 6), MT is used with a test oracle.

As already observed, a core task in MT is the identification of MRs, which, when done, means that the entire MT procedure can be conducted automatically. Unsurprisingly, the effectiveness of MT is highly associated with the effectiveness of the MRs used, and an important observation is that the application of diverse MRs can enhance the fault detection capability of MT [Cao et al., 2013; Liu et al., 2014]. To date, several strategies for selecting and generating MRs have been proposed [Chen, 2015; Liu et al., 2012; Mayer and Guderlei, 2006; Zhang et al., 2014]. When applying MT, it is common that several MRs are identified, and that MTGs from multiple MRs are used. Furthermore, although a program can be determined to be faulty by any violating MTG, different violating MTGs (either from the same MR or different MRs) may reveal different failures of a program, and may thus also provide different information to assist in repairing the program.

2.3 Metamorphic failure-causing condition

The basic motivation of software testing is to reveal program failures. In conventional testing, program failures are detected with concrete failing test cases. Other than these concrete failing test cases, it may sometimes be possible to obtain a condition on program input parameters under which the execution of the program leads to a failure. Similarly, in MT, in addition to concrete violating MTGs, it may be possible to find characterisations of MTGs that lead to violations of a specific MR. Such a condition can characterise violating MTGs and is referred to as a *metamorphic failure-causing condition* (MFCC).

The concept of MFCCs was first proposed in a study of MR proving [Chen et al., 2011]. An MFCC carries specific information about how any MTG satisfying the MFCC will violate the relevant MR. In other words, an MFCC characterises a set (finite or infinite) of MTGs for which the relevant MR is violated. Consequently, MFCCs contain information about a group of concrete metamorphic test cases, and thus have great potential for program debugging. MFCCs can be constructed in a number of ways, but in this thesis, the technique

of semi-proving [Chen et al., 2011] was used. Next, semi-proving is briefly introduced, after which, the procedure for constructing MFCCs is illustrated.

2.3.1 Semi-proving

Semi-proving [Chen et al., 2011] is an integrated method of program proving, testing and debugging, based on MRs. It attempts to prove MRs by means of symbolic execution and constraint solving. While conventional software testing techniques (including MT) generate concrete failing test cases (such as "$a = 2$, $b = 5$" to detect failures in a program that accepts two input parameters a and b), semi-proving can generate MFCCs that describe the condition(s) under which an MR will be violated (such as "$b = 2a + 1$").

For verification, semi-proving takes a program P and a metamorphic relation *MR* as input, and verifies P against *MR*. Although an MR may involve two or more executions of the program under test, for ease of presentation, assume that *MR* involves only two executions of P. The core procedure for verification can thus be outlined as follows:

First, take a source symbolic input vector I_s and perform symbolic evaluation of P using I_s: let $o_1^s, o_2^s, \ldots, o_n^s$ ($n > 0$) be the symbolic outputs, and let $c_1^s, c_2^s, \ldots, c_n^s$ be their respective path conditions. Then, by referring to *MR*, a follow-up symbolic input vector I_f is generated to conduct another symbolic evaluation: let $o_1^f, o_2^f, \ldots, o_m^f$ ($m > 0$) be the relevant symbolic outputs, and $c_1^f, c_2^f, \ldots, c_m^f$ be their respective path conditions — m may or may not be equal to n. For each c_i^s ($i = 1, 2, \ldots, n$) and each c_j^f ($j = 1, 2, \ldots, m$), the conjunction of c_i^s and c_j^f is evaluated: if there is no contradiction, then there must exist a source execution path under condition c_i^s, *followed by* a follow-up execution path under condition c_j^f. Semi-proving will then check whether the *MR* is satisfied under this combination (that is, under the condition given by $c_i^s \wedge c_j^f$). If a violation is detected, semi-proving will report MFCCs under which the violation will occur. If all possible combinations of paths are verified and no violation is detected, then the relevant MR is proven. As a reminder, in situations where the underlying symbolic analysis tool cannot analyse or generalise all possible paths, semi-proving can still be used as a symbolic testing technique to verify MRs on a finite number of paths [Chen et al., 2011].

Path	$Max(a,\ b)$		$Max(b,\ a)$	
	Path Condition	Output	Path Condition	Output
P_1: (1, 2)	c_1^s: $(a \geq b+10)$	a	c_1^f: $(b \geq a+10)$	b
P_2: (3, 4)	c_2^s: $(a < b+10)$	b	c_2^f: $(b < a+10)$	a

Table 2.1 The source and follow-up symbolic evaluation results of *Max*

2.3.2 Illustration of the MFCC generation

Using the example program *Max* (Figure 2.1), how semi-proving generates MFCCs will next be illustrated. If *maximum* denotes the intended program's functionality, then one possible MR is as follows:

MR1: $maximum(a,\ b) = maximum(b,\ a)$, where a and b are two valid integers.

To verify *Max* against *MR1*, semi-proving uses a source symbolic input $I_s = (a,\ b)$ and a follow-up symbolic input $I_f = (b,\ a)$ to conduct the source and follow-up symbolic evaluations, the results of which are shown in Table 2.1.

Next, consider the following four conjunctions: (i) $c_1^s \wedge c_1^f$; (ii) $c_1^s \wedge c_2^f$; (iii) $c_2^s \wedge c_1^f$; and (iv) $c_2^s \wedge c_2^f$. It is found that (i) is a contradiction and, therefore, will not be considered. Furthermore, (ii) will not be considered because the outputs for c_1^s and c_2^f are both a, which will not violate *MR1*. For the same reason, (iii) will not be considered either. Conjunction (iv) is equivalent to $b-10 < a < b+10$, which is not a contradiction: semi-proving will therefore check whether or not *MR1* can be violated under this condition. Because the outputs for c_2^s and c_2^f are b and a, respectively, semi-proving will identify the conjunction "$b-10 < a < b+10 \wedge b \neq a$" (which is equivalent to "$b-10 < a < b$ *OR* $b < a < b+10$") as an MFCC. Any MTG $((a,\ b),(b,\ a))$ satisfying this MFCC (such as $a = 1,\ b = 2$) will cause a violation of *MR1*, that is, cause $Max(a,\ b) \neq Max(b,\ a)$, and hence reveal a fault.

It should be noted that semi-proving may construct multiple MFCCs for a given program and MR. These MFCCs, although related to the same MR, involve different source and follow-up executions, and thus may capture different information about the violation of the MR. Furthermore, if more than one MR is used, MFCCs from different MRs can express information related to the violations of different MRs.

Chapter 3

Formalisations

This chapter formalises a series of APR concepts which will be used to enable systematic evaluation of APR techniques and input test suites. First, some fundamental repair quality concepts are formalised, followed by some key concepts in test suite based APR. This leads to the creation of evaluation metrics for measuring the repair effectiveness of APR techniques. Next, formalisations characterising APR input test suites are provided, giving some evaluation metrics for measuring the effectiveness of input test suites. Finally, the chapter reviews related work and discusses the advantages of the proposed evaluation metrics.

3.1 Formalisation of repair quality

When repairing a PUR with respect to a given test suite, a program variant of the PUR is accepted as a repair if it passes all test cases in the given test suite. Such a program variant is formally referred to as a *plausible repair* [Qi et al., 2015]. However, although plausible repairs pass all test cases in the given test suite, they may still fail on inputs outside of the given test suite.

Definition 1 (Plausible Repair) *Let P be a PUR and T be a test suite. A program variant of P is a* plausible repair for P with respect to T *iff it passes all test cases of T.*

Obviously, many different plausible repairs may exist for the same P and T. Ideally, a plausible repair should completely rectify the PUR — it should produce correct outputs for

```
  int isUpward(int in, int up, int down)
  {
1:    int bias,r;
2:    if (in)
3:      bias = down; /* Should be: bias = up +100. */
4:    else
5:      bias = up;
6:    if (bias>down)
7:      r = 1;
8:    else
9:      r = 0;
10:   return r;
  }
```

Fig. 3.1 Motivating example

all possible inputs to the PUR (the entire PUR *input domain*) — in which case it is referred to as a *correct* repair. In reality, however, such an ideal situation may not always occur, which raises concerns about the *quality of a repair*: how correct is a repair with respect to the entire input domain.

Consider the program *isUpward* (Figure 3.1), which is taken from a traffic control avoidance system [Do et al., 2005]. The intended functionality of this program is to check whether $(100 \times in + up) > down$ (where *in*, *up*, and *down* are three input parameters, and *in* can have a value of either 1 or 0).

Suppose there is a test suite T_1 that contains two test cases: one passing $(in = 0, up = 300, down = 200)$, and one failing $(in = 1, up = 150, down = 200)$. With respect to T_1, a plausible repair of program *isUpward* can be constructed by replacing the statement on line 3 with "bias = up + 100", resulting in a correct repair. Other plausible repairs also exist, such as by replacing line 3 with "bias = 201" (denoted R_1), or with "bias = up + 87" (denoted R_2). Unlike the correct repair, however, neither R_1 nor R_2 produces correct outputs for all elements of the *isUpward* input domain.

R_1 fails when the input satisfies the condition $(in = 1) \wedge ((201 \leq down < up + 100) \vee (up + 100 \leq down < 201))$, and R_2 fails when $(in = 1) \wedge (up + 87 \leq down < up + 100)$. To reveal the differences between these two repairs in terms of their quality, their failure regions [Ammann and Knight, 1988], which are the regions formed by their failing inputs, are analysed. Let $Region_1$ and $Region_2$ denote the failure regions of R_1 and R_2, respectively (under the assumption of $in = 1$). Then $Region_1$ is enclosed by the lines L and L_1, and $Region_2$ is enclosed by the two parallel lines L and L_2 (Figure 3.2). Although neither R_1 nor R_2 is correct, the sizes of their failure regions differ significantly: R_2 has far more correct outputs than R_1, and therefore, intuitively can be considered a higher quality repair than R_1.

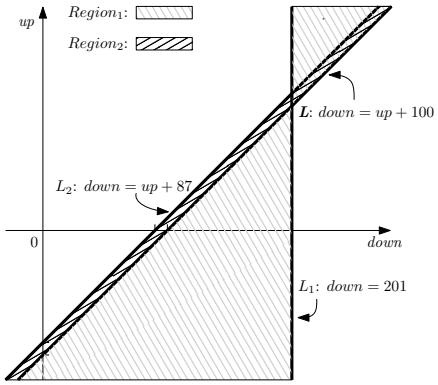


Fig. 3.2 The failure regions of two repairs differ dramatically in size

Clearly, the ability to *evaluate repair quality* in terms of how close a repair is to the correct program is very desirable, but, to date, little work has been done examining systematic approaches for such evaluations [Diallo et al., 2015]. In the following, some concepts related to the quality of a repair are defined. $\mathscr{D}_P$ denotes the entire input domain of program P.

Definition 2 (Repair Quality) *Let R be a plausible repair of program P. The* quality *of R, denoted* $\theta_R^{\mathscr{D}_P}$*, is the ratio of the number of inputs for which R gives correct outputs to the number of all inputs in* $\mathscr{D}_P$ *(i.e., the pass rate of R for* $\mathscr{D}_P$*).*

According to Definition 2, a correct repair should pass all possible inputs of a program, and therefore have a quality score of 1. However, although Definition 2 is intuitively appealing, the pass rate of an entire input domain may not be easily obtained in practice. Therefore, instead of attempting to measure the pass rate for the entire input domain, a more practical approach is to measure the pass rate for a given benchmark test suite, referred to as an *evaluation test suite*.

Definition 3 (Repair Quality with Respect to an Evaluation Test Suite) *Let R be a plausible repair of program P. For a given evaluation test suite* $\mathscr{T}_E$*, the* quality of R with respect to $\mathscr{T}_E$*, denoted* $\theta_R^{\mathscr{T}_E}$*, is the ratio of the number of test cases in* $\mathscr{T}_E$ *for which R gives correct outputs to the number of all test cases in* $\mathscr{T}_E$ *(i.e., the pass rate of R for* $\mathscr{T}_E$*).*

Although Definition 3 is less appealing than Definition 2, it is much more applicable in practice, and is therefore used when Definition 2 cannot be applied. As observed from Figure 3.2, according to Definition 2, the repair R_2 is of higher quality than the repair R_1, because it fails for far fewer inputs of $\mathscr{D}_{isUpward}$ than R_1. A problem arises with

Definition 3, however, when R_1 may appear to be of higher quality than R_2 with respect to an arbitrary $\mathscr{T}_E$. Consider a $\mathscr{T}_E$ that consists of inputs in which the value range of variable *up* is $[73, 125]$: a simple analysis shows that R_1 fails on fewer $\mathscr{T}_E$ inputs than R_2, and therefore, following Definition 3, would be considered to be of higher quality than R_2. Thus, selection of a suitable evaluation test suite in the application of Definition 3 is critical: if it is not possible to measure $\theta_R^{\mathscr{D}_P}$, then an appropriate $\mathscr{T}_E$ (such that the relevant $\theta_R^{\mathscr{T}_E}$ will be a good estimator for $\theta_R^{\mathscr{D}_P}$) must be identified.

With the above definitions, it is now possible to compare the quality of two repairs.

Definition 4 *Let R_1 and R_2 be two plausible repairs of program P.*
(i) R_1 and R_2 are of equal quality *iff* $\theta_{R_1}^{\mathscr{D}_P} = \theta_{R_2}^{\mathscr{D}_P}$.

(ii) R_1 is of higher quality than R_2 *iff* $\theta_{R_1}^{\mathscr{D}_P} > \theta_{R_2}^{\mathscr{D}_P}$.

Definition 5 *Let R_1 and R_2 be two plausible repairs of program P, and $\mathscr{T}_E$ be an evaluation test suite.*
(i) R_1 and R_2 are of equal quality with respect to $\mathscr{T}_E$ *iff* $\theta_{R_1}^{\mathscr{T}_E} = \theta_{R_2}^{\mathscr{T}_E}$.

(ii) R_1 is of higher quality than R_2 with respect to $\mathscr{T}_E$ *iff* $\theta_{R_1}^{\mathscr{T}_E} > \theta_{R_2}^{\mathscr{T}_E}$.

For brevity, θ_R will be used to refer to $\theta_R^{\mathscr{D}_P}$ or $\theta_R^{\mathscr{T}_E}$ when there is no ambiguity.

3.2 Formalisation of test suite based APR

Test suite based APR repairs a program with respect to an input test suite, producing either a plausible repair (that passes all test cases of the input test suite) or no repair at all. The *repair effectiveness* of an APR technique, therefore, is twofold: the capability of generating repairs (how likely a PUR can be repaired to pass the entire input test suite), and the quality of the repairs, which has been described in Definitions 2 and 3.

Definition 6 (Repair Context) *A* repair context *is an ordered pair $\Delta = <P, \mathscr{A}>$, where P is a PUR, and $\mathscr{A}$ is an APR technique to be applied to repair P.*

Definition 7 (Repair Process) *Let T be an input test suite for a repair context $\Delta = <P, \mathscr{A}>$. A* repair process $\mathscr{RP} : (\Delta, T) \mapsto (R \text{ or } null)$ *is a process where $\mathscr{A}$ is applied together with T to repair P, producing a repair result, which is either a plausible repair R or null (no plausible repair).*

A repair process can be conducted only if T can detect failures for P. A *successful* repair process is one yielding a plausible repair; otherwise, it is a *failed* repair process. Following the practice of most APR tools (which terminate when a plausible repair is produced [Le Goues et al., 2012b; Mechtaev et al., 2016; Qi et al., 2014]), in this thesis, a successful repair process yields *one and only one* plausible repair. Furthermore, where there is no ambiguity, the word *repair* is used to refer to a *plausible repair*.

Obviously, an APR technique's ability to generate repairs can be measured by collecting all its produced plausible repairs. Following other APR studies [Le Goues et al., 2012b; Nguyen et al., 2013; Qi et al., 2014], the term *success rate* is used to denote this metric:

Definition 8 (Success Rate) *Given an APR technique $\mathscr{A}$ and a series of repair contexts $\mathbb{D}=\{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ $(n \geq 1)$, where $\Delta_i = <P_i, \mathscr{A}>$ $(1 \leq i \leq n$, and P_i denotes the i^{th} PUR). Let $\mathbb{T} = \{T_1, T_2, \ldots, T_n\}$ be a series of input test suites where the application of T_i to Δ_i corresponds to a repair process $\mathscr{RP}_i$ $(1 \leq i \leq n)$. The* success rate *of $\mathscr{A}$* with respect to $\mathbb{D}$*, denoted $SR_{\mathscr{A}}$, is defined as $\frac{x}{m}$, where x is the number of resulting plausible repairs, which is also the number of successful repair processes among $\{\mathscr{RP}_1, \mathscr{RP}_2, \ldots, \mathscr{RP}_n\}$, and m $(m \leq n)$ is the number of repair processes conducted.*

$SR_{\mathscr{A}}$ is in the range $[0,\ 1]$, and it is possible that the same input test suite may be applied to repair different PURs (i.e., T_i may be the same as T_j; $1 \leq i, j \leq n$ and $i \neq j$). Higher *SR* values indicate better plausible repair generation capabilities. In addition to the success rate, an APR technique can also be evaluated according to the quality of all of its resulting repairs. A preferred APR technique is expected to be effective in terms of both success rate and repair quality.

3.3 Formalisation of the input test suite

The input test suite plays an important role in test suite based APR: it is the only source of information (apart from the PUR) to assist the repair process, and it also provides the only

criterion for determining a plausible repair. Different test suites may contain different test cases, and thus express different information about the PUR: therefore, the use of different input test suites may yield different repair results for a given APR technique and PUR. This means that the input test suite has a direct impact on the repair effectiveness of an APR technique [Le Goues et al., 2012b; Monperrus, 2014; Nguyen et al., 2013]. Thus, it is necessary to evaluate the effectiveness of the input test suite from the perspective of APR.

Although many studies have reported the impact of input test suites on repair effectiveness, to date, none have quantified this impact, nor has any study attempted to measure the effectiveness of the input test suites. In the following, a series of concepts for characterising APR input test suites are introduced.

Definition 9 (Applicable Input Test Suite) *An input test suite T is* applicable *to a repair context $\Delta = <P, \mathscr{A}>$ iff $\mathscr{RP} : (\Delta, T) \mapsto R$.*

According to Definition 7 and Definition 9, application of an applicable input test suite to a repair context yields exactly one plausible repair.

Definition 10 (Applicability of Input Test Suites) *Given a series of repair contexts $\mathbb{D} = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ $(n \geq 1)$, let $\mathbb{T} = \{T_1, T_2, \ldots, T_n\}$ be a series of input test suites where the application of T_i to Δ_i corresponds to a repair process $\mathscr{RP}_i$ $(1 \leq i \leq n)$. The* applicability *of $\mathbb{T}$* with respect to *$\mathbb{D}$, denoted $a^{\mathbb{D}}_{\mathbb{T}}$, is defined as $\frac{x}{n}$, where x is the number of resulting plausible repairs, which is also the number of successful repair processes among $\{\mathscr{RP}_1, \mathscr{RP}_2, \ldots, \mathscr{RP}_n\}$.*

According to this definition, $a^{\mathbb{D}}_{\mathbb{T}}$ is in the range of $[0,\ 1]$, and, as with the success rate (Definition 8), it is possible that T_i may be the same as T_j $(1 \leq i, j \leq n$ and $i \neq j)$. The applicability of a series of input test suites indicates its probability of yielding a plausible repair, and hence is a fundamental concept for the evaluation of input test suites for APR. Note that this concept is closely related to that of the success rate (Definition 8) used in previous studies [Le Goues et al., 2012b; Nguyen et al., 2013; Qi et al., 2014].

Definition 11 *Let $\mathbb{D}$ be a series of repair contexts, and $\mathbb{T}_1$ and $\mathbb{T}_2$ be series of input test suites with applicability $a^{\mathbb{D}}_{\mathbb{T}_1}$ and $a^{\mathbb{D}}_{\mathbb{T}_2}$, respectively.*
(i) $\mathbb{T}_1$ *and* $\mathbb{T}_2$ are of equal applicability with respect to $\mathbb{D}$ *iff* $a^{\mathbb{D}}_{\mathbb{T}_1} = a^{\mathbb{D}}_{\mathbb{T}_2}$.

(ii) $\mathbb{T}_1$ is of higher applicability than $\mathbb{T}_2$ with respect to $\mathbb{D}$ *iff* $a^{\mathbb{D}}_{\mathbb{T}_1} > a^{\mathbb{D}}_{\mathbb{T}_2}$.

For a given series of repair contexts involving the same APR technique, a series of input test suites with higher applicability is better in terms of assisting the APR technique to produce plausible repairs. A high applicability score is only one expected characteristic of good input test suites: it is obvious also necessary to examine the quality of the resulting repairs.

Definition 12 *Given a repair context* $\Delta = <P, \mathscr{A}>$*, let* T_1 *and* T_2 *be two input test suites that are both applicable to* Δ*.*
(i) T_1 *and* T_2 are of equal effectiveness with respect to Δ *iff the resulting plausible repairs of* T_1 *and* T_2 *are of equal quality.*

(ii) T_1 is of higher effectiveness than T_2 with respect to Δ *iff the resulting plausible repair of* T_1 *is of higher quality than that of* T_2*.*

To measure the overall effectiveness of input test suites — considering both applicability and repair quality — a new concept of *usefulness* is proposed. A repair process, involving application of an input test suite to a repair context, produces either a plausible repair or no repair: a plausible repair result affects both the applicability and the repair quality; but a result with no repair affects only the applicability. Intuitively, "no repair" is the worst result, and can be interpreted as a *dummy repair* with a quality score of 0. This interpretation enables calculation of the average quality score of *all* results (both plausible repairs and dummy repairs).

Definition 13 (Usefulness of Input Test Suites) *Let* $\mathbb{D} = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ $(n \geq 1)$ *be a series of repair contexts, and* $\mathbb{T} = \{T_1, T_2, \ldots, T_n\}$ *be a series of input test suites, where the application of* T_i *to* Δ_i *yields a repair result* R_i $(1 \leq i \leq n)$ *(which is either a plausible repair or a dummy repair, null). Let* θ_{R_i} *be the quality score of* R_i*,* $1 \leq i \leq n$ ($\theta_{R_i} = 0$ *if* R_i *is null*)*. The* usefulness *of* $\mathbb{T}$ with respect to $\mathbb{D}$ *is defined as* $\left(\sum_{i=1}^{n} \theta_{R_i}\right) \div n$*, which is also equal to* $a_{\mathbb{T}}^{\mathbb{D}} \times q$ *when* $a_{\mathbb{T}} > 0$*, where* $a_{\mathbb{T}}^{\mathbb{D}}$ *is the applicability score of* $\mathbb{T}$ *with respect to* $\mathbb{D}$*, and* q *is the mean quality score of all plausible repairs in* $\{R_1, R_2, \ldots, R_n\}$*.*

For the purpose of illustration, suppose the application of 10 input test suites to 10 repair contexts yields 10 repair processes, among which only 6 successful repair processes respectively produce 6 repairs, with the quality scores of $\{0.80, 0.87, 0.90, 0.92, 0.93, 0.90\}$. According to Definition 10, the applicability of the set of input test suites (10 input test suites) is $\frac{6}{10} = 0.6$. Moreover, the average quality of all 6 repairs is 0.89. Therefore, according to Definition 13, the usefulness of the set of input test suites is $0.6 * 0.89 = 0.53$.

3.4 Related work

3.4.1 Repair quality

As explained by Le Goues et al. [2013, 2012b], repair quality is critical to the practical usefulness of APR techniques, whereas its evaluation has not yet been fully addressed.

While most previous APR studies report on the success rates (the percentage of successful repair processes) of the APR tools for the investigated subject programs, many other metrics have also been used to evaluate the repair quality. Perkins et al. [2009], for example, tested repairs against designed external hostile attacks. Le Goues et al. [2012b] studied programs with security defects and evaluated the repair quality with fuzz testing. Assiri and Bieman [2014] measured the repair quality by calculating the percentage of failed repairs and the average percentage of failed tests in regression test suites — thus focusing on the overall quality of a set of repairs. When evaluating the performance of different APR strategies, Arcuri [2011] used an additional set of 1,000 test cases, independent of the input test suite used to construct the repair. This evaluation, however, only counted repairs that had a 100% pass rate — it did not distinguish between repairs with a 99% pass rate and those with a 1% pass rate. Pei et al. [2014] manually inspected repairs to determine how many could fix the fault without introducing new ones. Smith et al. [2015] emphasised that the evaluation of repair quality should be done independently from the repair construction, and used both black-box and white-box test suites such that when one was used to generate a repair, the other was used to evaluate the repair quality. Their experimental results show that multiple factors (including the nature of the input test suite, and characteristics of the faulty program) affect the repair quality. Quality aspects other than functionality have also been studied, with Kim et al. [2013], for example, conducting a user study to measure the acceptability of repairs, and emphasising the importance of developer acceptance. Fry et al. [2012] also conducted studies examining the maintainability of repairs.

The concept of repair quality in this thesis differs from previous studies in several aspects. In this chapter, the pass rate for an entire input domain was proposed as a fundamental metric for repair quality. However, because the pass rate for the entire input domain may not be obtainable, the pass rate of an evaluation test suite was proposed as a possible alternative. In such a case, such an evaluation test suite would need to be carefully designed so that the evaluation result (pass rate) could be representative and statistically meaningful.

Diallo et al. [2015] proposed concepts of *absolute* and *relative correctness*, where absolute correctness refers to a program's correctness with respect to its specification, and relative correctness involves comparisons amongst multiple programs, and identification of the one that is "more" correct (with respect to the specification). Their concept of absolute correctness corresponds to a quality score of 1, according to Definition 2. Checking the relative correctness between two programs involves comparing two sets of initial states (which can be represented by the inputs to the program) for each of which the corresponding program has correct behaviour: if the set for one program is a superset of that for the other, then the first program is the more correct of the two. Obviously, if one program is regarded as relatively more correct than another, then it should also have a higher pass rate, with respect to Definition 3. However, according to the relative correctness definition in Diallo et al. [2015], one program may not necessarily be more correct than another, even if it has a higher pass rate according to Definition 2 — this will happen when the set of passing test cases for the program with the higher pass rate does not include those for the other program. Because of this, the relative correctness in Diallo et al. [2015] is a special instance of the definitions of repair quality in this thesis (Definitions 2 and 3). Furthermore, their concept of relative correctness requires the comparison of at least two programs, but the repair quality concept in this thesis does not have such a constraint.

3.4.2 The APR input test suite

The importance of the input test suite for APR has been well recognised by researchers. Le Goues et al. [2012b] have pointed out that the input test suite's size and scope directly impact on the repair quality, and that too many test cases may impede running time as most of the repair cost is related to the validation of the candidate programs using the test suite. Nguyen et al. [2013] reported that the success rate of APR tools decreases when the number of test cases of the input test suite increases, and that repairs generated with a small number of test cases may not be valid for other test cases. Furthermore, Monperrus [2014] emphasised the need for assessing the test suite effectiveness, and argued that: "if the research community is able to characterise what a good test suite is, we can simply clarify the [APR] problem statement as follows, 'given a good and trustable test suite, generate a patch that makes the test suite passing (on page 238).' "

Further analyses and experimental studies have revealed some useful characteristics of good input test suites. Assiri and Bieman [2014] analysed the effectiveness of three different types of test suites for operator repair, and found that the branch coverage criterion

had a positive impact on the repair quality. Smith et al. [2015] experimented with two APR tools and found that the repair quality was proportional to the coverage of the test suite used for the repair. They also pointed out that more characteristics of input test suites for APR should be further investigated.

This chapter contributes to a better understanding of the nature of good test suites for APR, and also provides a systematic approach to measure the effectiveness of APR input test suites.

Chapter 4

Program repair without the need for a test oracle

This chapter presents an approach to extending the scope of test suite based APR. As explained in Section 1.2 of Chapter 1, current APR techniques assume the existence of a test oracle, and thus may not be applicable in situations where a test oracle is not available. To enable application of APR in the absence of a test oracle, this chapter explains how to integrate MT with conventional APR techniques. The chapter first proposes a general framework for supporting the integration of MT and APR, and then demonstrates the feasibility of the integration through two implementations involving two different APR techniques. Finally, the effectiveness of the integrated techniques is empirically investigated, with detailed experimental results and analyses presented.

4.1 Preliminary

A *test oracle* is a mechanism that can verify the correctness of any test case's execution result [Barr et al., 2015; Liu et al., 2014]. In practice, when a test oracle is not available, or is available, but is too expensive to be applied, the situation is known as the test oracle problem [Barr et al., 2015]. The test oracle problem is generally regarded as a fundamental challenge in software testing, and has restricted the degree of automation and applicability of many software testing methods.

As with software testing, test suite based APR faces the test oracle problem due to its reliance on the input test suite [Arcuri, 2011; DeMarco et al., 2014; Forrest et al., 2009; Le Goues et al., 2013; Mechtaev et al., 2015; Nguyen et al., 2013]: during a repair process, APR needs to know the test result of individual test cases — whether a test case *passes* or *fails*, yielding a correct or incorrect output, respectively. Obviously, the application of test suite based APR techniques is restricted by the availability and the feasible application of the test oracle. In other words, alleviating the test oracle problem of APR will extend the scope of applicability of APR techniques.

MT is a testing method that is effective in alleviating the test oracle problem, and has been successfully applied to various application domains [Chan et al., 2005; Chen et al., 2009; Xie et al., 2011]. Integration of MT with test suite based APR should support the application of APR techniques in the presence of the test oracle problem, and thereby extend their scope of applicability. This chapter focuses on the following research questions.

RQ1 : How can MT be integrated with APR?

RQ2 : How effectively can MT-based APR techniques repair programs?

The rest of this chapter presents an integration framework and two implementations (Section 4.2) to answer *RQ1*, followed by an empirical study (Section 4.3) and experimental analysis (Section 4.4) to answer *RQ2*.

4.2 Test suite based APR in the absence of a test oracle

This section addresses the first research question, describing the proposal for alleviating the test oracle problem of test suite based APR. First, a framework to support the integration of MT and test suite based APR, is proposed, then two implementations of the framework are presented by incorporating MT into two APR techniques — the *generate-and-validate* technique GenProg and the *semantics-based* technique CETI.

	Conventional APR techniques	**APR-MT techniques**
Inputs	(1) A faulty program; (2) A test suite with at least one failing test case.	(1) A faulty program; (2) A set of MTGs with at least one violating MTG.
Repair process	Utilising information provided by the input test suite.	Utilising information provided by the set of MTGs.
Output	(1) A repair passing the input test suite or (2) null.	(1) A repair satisfying the set of MTGs or (2) null.

Table 4.1 Correspondence between conventional APR and APR-MT techniques

4.2.1 Framework

In test suite based APR, the input test suite acts as a kind of specification for repairing a PUR, and has two essential characteristics:

- The test outcome of each individual test cases is known.
- There is at least one failing test case in the test suite.

In the integration of MT and APR, the key step is to use a set of MTGs (the detail of which is introduced in Section 2.2 of Chapter 2) as a substitute for a set of test cases. The term APR-MT is used to refer to the integrated APR technique, and the integration is based on the following correspondences between conventional APR and APR-MT.

- A test case in the conventional APR corresponds to an MTG in APR-MT.
- A test outcome of pass or fail in the conventional APR corresponds to an MTG's test outcome of satisfaction or violation.

In order to apply MT to test suite based APR, conventional APR techniques need to be adjusted accordingly. Table 4.1 summarises the correspondences between conventional APR and APR-MT, based on which, APR-MT can follow the same repair procedures as conventional APR. The two groups of APR techniques use different information for guiding their repair processes, and different criteria for selecting the resulting repair. Most importantly, because APR-MT techniques verify the relationship among multiple test cases rather than the correctness of individual outputs, a test oracle is no longer required: APR-MT can be applied to repair a program facing the test oracle problem, and thus can be applied to a broader range of programs.

4.2.2 Implementations

To demonstrate the feasibility of the proposed integration framework, it was applied to some APR techniques. As explained in Section 2.1 of Chapter 2, there are basically two categories of APR techniques: *generate-and-validate* and *semantics-based*. Based on these two categories, two APR-MT techniques, *GenProg-MT* and *CETI-MT*, were developed (by integrating MT with the generate-and-validate technique GenProg and the semantics-based technique CETI, respectively). This section presents these two APR-MT techniques, first introducing the original APR technique, and then describing the implementation details of the resulting APR-MT technique.

GenProg-MT

GenProg [Forrest et al., 2009; Le Goues et al., 2012a,b] is a typical *generate-and-validate* APR technique that has shown promising repair results, and been used as a comparison baseline in many APR studies [Kim et al., 2013; Nguyen et al., 2013; Qi et al., 2014; Smith et al., 2015; Tan and Roychoudhury, 2015]. GenProg uses a genetic algorithm to repair a program. The algorithm first creates an initial set of candidate programs (program variants of the PUR), then, in each subsequent generation, the candidate programs are generated using the crossover operator and mutation operator. Using the input test suite, GenProg evaluates the fitness of each candidate program such that those with higher fitness are more likely to proceed to the next generation. GenProg iterates this procedure, terminating when either a candidate program that passes the entire input test suite is obtained (i.e., a plausible repair is obtained), or a predefined maximum number of generations have been searched (in which case no repair is reported).

During a GenProg repair process, the following two activities typically require a test oracle:

(1) *Localisation of faulty statements.* When generating candidate programs, GenProg operates on likely faulty statements. In order to select which statement to modify, GenProg constructs a weighted path, which is a sequence of statements and their associated *weight* — a statement's weight is measured based on its coverage information of passing and failing test cases from the input test suite. Obviously, during this procedure, identifying a test case as passing or failing requires a test oracle.

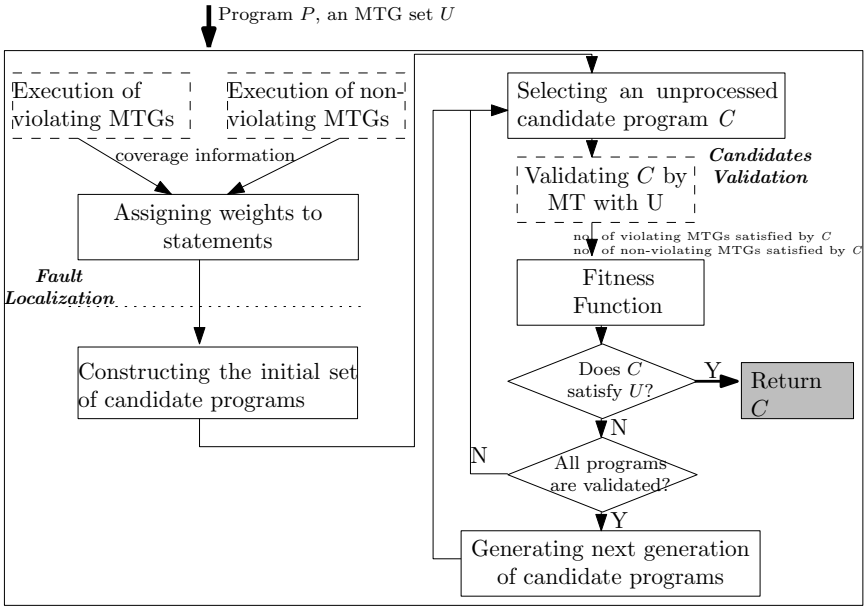


Fig. 4.1 The GenProg-MT approach

(2) *Validation of candidate programs.* Each generated candidate program is validated for two purposes: to examine whether or not it is a plausible repair of the program under repair; and, if not, to determine if it can be passed to the next generation of the repair process. The first step in the GenProg validation procedure consists of testing candidate programs using the input test suite and collecting the number of passing and failing test cases for each candidate. Next, a fitness function, making use of the collected information, calculates a fitness value for each candidate — candidate programs with higher fitness values are more likely to be passed to the next generation. At this point, if any candidate program meets the termination criterion (passing the entire input test suite), then it is reported as a plausible repair. Clearly, again, evaluation of the fitness requires a test oracle.

GenProg-MT is an implementation of the integration framework applying MT to GenProg, aiming to automatically repair programs without the need for a test oracle. It basically imitates GenProg, but with slight adjustments to eliminate reference to the test oracle.

The GenProg-MT workflow is shown in Figure 4.1, where differences between GenProg-MT and GenProg — in the fault localisation and the candidate program validation — are highlighted using dashed rectangles. GenProg-MT collects statement coverage information from the execution of both violating and non-violating MTGs in a manner similar to how GenProg gathers coverage information from passing and failing test cases. Since each MTG contains multiple test cases, statements covered by an MTG are those visited by at least one of its test cases. This coverage information is then used to assign weights to individual statements. When validating, GenProg-MT applies MT to assess each candidate

Algorithm 1: The CETI algorithm

Input: the faulty program P, the test suite T.
Output: a repair R passing T or a *null* indicating the fail of repairing P.

```
1  ps = ∅;
2  sstmts = FaultLocalization(P, T);
3  for i = 1 to length(sstmts) do
4      s = sstmts[i];
5      for tpl ∈ all_tpls do
6          p_stmt = GenParameterizedStmt(s, tpl);
7          r_prog = GenReachabilityInstance(P, p_stmt, T);
8          ps = ps.add(r_prog);
9      end
10 end
11 for c in ps do
12     d = GenTestInputs(c);
13     if d ≠ null then
14         R = GenARepair(c, d);
15         return R;
16     end
17 end
18 return null
```

program by testing it against the input set of MTGs, calculating the number of violating and non-violating MTGs. This information is then fed into the fitness function to calculate a fitness value for the candidate program. Accordingly, the GenProg-MT procedure no longer requires a test oracle, and can thus be applied in situations with the test oracle problem.

CETI-MT

CETI [Nguyen, 2014] is a *semantics-based* APR technique that uses knowledge of program reachability and repair templates to create statements to repair C programs: the process is outlined in Algorithm 1. Given a faulty program P and a test suite T (containing both test inputs *and* test oracles), CETI first applies statistical fault localisation [Jones and Harrold, 2005] to identify a ranked list of statements in descending order of their likelihood of being faulty (line 2), then iteratively processes the ranked list from top to bottom (lines 3 to 10).

With a suspicious statement s and a selected repair template (e.g., constant template, operator template, etc.), CETI creates a parameterised statement *p_stmt* (line 6), where some parameters without specific values are used. A reachability instance program *r_prog* is then constructed from P by replacing s with *p_stmt*, and by encoding the requirement

```
int Max(int x, int y, int uk_0){
    if(x>=y + uk_0)
      return x;
    else
      return y;
}
void main() {
    int uk_0;
    if(Max(12, 1, uk_0) == 12 &&
      Max(0, 5, uk_0) == 5 &&
      Max(2, 1, uk_0) == 2 ) {
      Location L;
    } }
```

Fig. 4.2 A reachability instance program constructed by CETI with the application of the constant template

of passing all test cases of T through a precondition for reaching a specific location L in the program (line 7). Consider, for example, application of the constant template to the faulty statement of program *Max* (Figure 2.1 in Section 2.1 of Chapter 2): CETI creates a parameterised statement 'if ($x >= y + uk\_0$)', where the constant 10 is replaced by a parameter *uk_0* (which has an unknown value). Based on this statement and the given input test suite (Figure 2.2 in Section 2.1 of Chapter 2), CETI constructs the reachability instance program as shown in Figure 4.2: a function *main* is created, with the location L defined to only be reachable when the function *Max* produces correct outputs for all test cases of the input test suite with a specified concrete value for *uk_0*.

CETI further uses the test input generation technique to find values for parameters used by *p_stmt* in order to make the location L reachable (line 12). If such parameter values are available, the application of these values concretises *p_stmt* and then yields a repair (line 14).

CETI-MT was developed by applying the integration framework to CETI, and accepts as input a set of MTGs (rather than a set of test cases), whose information is then used in the repair process. The CETI-MT repair process differs that of CETI in the following ways:

(1) *Using the MTGs to support the fault localisation*. CETI-MT uses a non-violating (or violating) MTG to locate likely faulty statements in the same way that CETI uses a passing (or failing) test case. Moreover, the MTG coverage information is collected in the same way as with GenProg-MT: the accumulated coverage information of test cases for an individual MTG is collected.

```
int Max(int x, int y, int uk_0){
    if(x>=y + uk_0)
      return x;
    else
      return y;
}
int MR1Checker(int a, int b) {
    if(a == b)
      return 1;
    else
      return 0;
}
int main(int argc, char ** argv) {
    int uk_0;
    if(MR1Checker(Max(12, 1, uk_0), Max(1, 12, uk_0)) == 1 &&
      MR1Checker(Max(0, 5, uk_0), Max(5, 0, uk_0)) == 1 &&
      MR1Checker(Max(2, 1, uk_0), Max(1, 2, uk_0)) == 1 ) {
      Location L;
    } }
```

Fig. 4.3 A reachability instance program constructed by CETI-MT applying the constant template

(2) *Using the set of MTGs and relevant MR to construct the reachability instance program*. To produce a repair satisfying the input set of MTGs, CETI-MT constructs the reachability instance program using the input MTGs and the relevant MR: to achieve this, the location *L* in a reachability instance program is defined to only be reachable when the MR is satisfied by all input MTGs.

To illustrate the differences between reachability instance programs created by CETI and by CETI-MT, the program created by CETI-MT for program *Max* is presented in Figure 4.3, where the same template is applied as was in Figure 4.2, and the metamorphic relation *MR1* (as defined in Section 2.3.2 of Chapter 2) is used. Comparing the two programs, the first difference is that in addition to the function *main*, another function *MR1Checker* is also created by CETI-MT: *MR1Checker* verifies the satisfaction of *MR1* against every element of the input MTG set. Another difference between the two programs lies in the *main* function itself: in Figure 4.3, the location *L* is defined to be reachable only when all input MTGs satisfy *MR1*. With these modifications, CETI-MT can be applied without any reference to a test oracle.

4.3 Experimental setup

Having demonstrated the feasibility of integrating MT with APR in the previous section, the next issue is the repair effectiveness of the proposed integration (related to the second

research question). An empirical analysis was used to answer the second research question, that is, how effectively the proposed APR-MT technique can repair programs. This section explains the design of the experiments, including: detailed information about the subject programs and identified MRs; the construction of the test suites and MTG sets; the configuration of the APR tools; and the measurements used to evaluate the repair effectiveness.

4.3.1 Subject programs

A total of $1,143$ versions of C programs from the IntroClass benchmark suite [Le Goues et al., 2015] were used in the empirical study. These program versions were written by students enrolled in an introductory C programming class. The IntroClass programs were designed for automated program repair research, and although they are small in size, this benchmark suite contains many incorrect programs involving various types of faults. Furthermore, each of the IntroClass programs has accompanying black box and white box test suites, both purposely designed for describing the behaviour of the target program. It is therefore appropriate to use them to evaluate factors affecting the performance of APR techniques, and they have actually been used for performance evaluation in many studies [Ke et al., 2015; Le et al., 2016a; Oliveira et al., 2016; Smith et al., 2015]. A summary of the subject programs is given in Table 4.2. For each subject program, two MRs were identified. The programs and MRs are discussed in detail in the following.

Program *checksum*

The program *checksum* takes as input a line of string, namely, "$c_1 ... c_i ... c_n$" ($n \geq 1$), where c_i ($1 \leq i \leq n$) represents a character. It first calculates the sum of all input characters, that is, *sum* = ((int)c_1 + ... + (int)c_i + ... + (int)c_n), and then outputs a string "Check sum is X", where the ASCII value of X is equal to (*sum* % $64 + 32$) and has a value in the range $[32, 95]$. For example, if the input string is "AB", then its output is "Check sum is #", because (int)A is 65, (int)B is 66, and (int)# is 35.

Using t_s and t_f to denote the source and follow-up test inputs, and x_s and x_f to denote the ASCII values of character X in the source and follow-up outputs, i.e., (int)X, the following MRs for *checksum* were identified:

Name	# versions	LOC	Description
checksum	69	*max*: 45 *min*: 15 *avg*: 27	Computing the sum of a string
digits	236	*max*: 190 *min*: 15 *avg*: 40	Listing all digits of an integer
grade	268	*max*: 53 *min*: 18 *avg.*: 29	Determining the grade of a score
median	232	*max*: 62 *min*: 13 *avg*: 24	Computing the median of three integers
smallest	177	*max*: 51 *min*: 17 *avg*: 25	Computing the smallest of four integers
syllables	161	*max*: 58 *min*: 12 *avg*: 29	Counting vowel characters of a string

Table 4.2 Subject programs

MR1: If t_f is constructed by inserting a character 'A' (with ASCII value 65) at the end of t_s, then $x_f < x_s$ if x_s is 95 ((the *sum* of t_s) % 64 = 63); or $x_f = x_s + 1$, otherwise.

MR2: Suppose that t_s contains n characters ($n > 0$). If t_f is constructed by replacing each character of t_s with its successor in the ASCII table, then $x_f = x_s + n$ % 64 if $(x_s - 32) + n$ % $64 < 64$; or $x_f = x_s + n$ % $64 - 64$, otherwise.

Program *digits*

The program *digits* accepts an integer *N*, and outputs every digit in *N* from the least significant to the most significant. For example, if N is 1234, *digits* outputs the list "4, 3, 2, 1".

Using N_s and N_f to denote the source and follow-up test inputs, and arrays L_s and L_f to denote the source and follow-up outputs: for an array *L*, *L.size* denotes its number of elements; *L*[0] is the first element (storing the least significant digit); and *L*[*L.size* - 1] is the last element (storing the most significant digit). The following MRs were identified for *digits*:

MR1: Suppose that N_s contains n digits ($n \geq 2$). N_f is constructed from N_s such that $N_f = N_s$ / 10^{n-1}, that is, N_f becomes of the most significant digit of N_s. Then, $L_f.size = L_s.size$-(n-1), and $L_f[0] = L_s[L_s.size$ -1].

MR2: Suppose that N_s contains n digits ($n \geq 2$). N_f is constructed from N_s such that $N_f = |N_s|$ % 10 (where $|.|$ denotes the absolute value). In this way, N_f becomes the least significant digit of N_s. Then, $L_f.size < L_s.size$, and $L_f[0] = L_s$ [0].

Program *grade*

The inputs of *grade* are five floating point values, A_b, B_b, C_b, D_b, and G, where the first four represent thresholds for the four different passing academic grade levels 'A', 'B', 'C', and 'D'; and G denotes the score of a student. By comparing G with these thresholds, program *grade* finally outputs "Student has a X grade" (where X is the symbol of the grade level that is closest to G among all grade levels that are not greater than G) or "Student has *failed* the course" when $G < D_b$. For example, if $A_b = 80.0$, $B_b = 70.0$, $C_b = 60.0$, $D_b = 50.0$, and $G = 73.0$, then *grade* outputs "Student has a B grade".

Using G_s and G_f to denote the score values in the source and follow-up test inputs, and X_s and X_f to denote the output grades for the source and follow-up inputs, respectively, then, using the same threshold values for both source and follow-up test inputs, the following MRs were identified for *grade*:

MR1: Suppose that G_s takes a randomly selected value. If G_s is equal to the value of one of the given thresholds, then G_f is set to $G_s - 1$, which results in X_f being exactly one level lower than X_s. Otherwise, G_f is set to $G_s - y$, where y is a randomly selected positive value, resulting in X_f not being higher than X_s.

MR2: Let G_s have a value smaller than A_b. Two follow-up test inputs are constructed, G_f^1 and G_f^2, where G_f^1 takes the value of the grade level that is closest to G_s among all grade levels that are higher than G_s, and G_f^2 takes a value from the range $[G_s, G_f^1]$. As a result, either: (1) X_f^2 and X_s are of the same grade, while X_f^2 is lower than X_f^1; or (2) X_f^2 and X_f^1 are of the same grade, but X_f^2 is higher than X_s.

Program *median*

The program *median* accepts three integers as inputs, and then outputs their median. Using t_s and t_f to denote the source and follow-up test inputs, with t_{s_i} (or t_{f_i}) representing the i^{th} ($1 \leq i \leq 3$) integer of the input, and m_s and m_f to denote the source and follow-up outputs, the following MRs were identified for *median*:

MR1: If t_f is constructed from t_s, such that $t_{f_i} = t_{s_i} + |t_{s_i}|$, then $m_f = m_s + |m_s|$.

MR2: If t_f is constructed from t_s, such that $t_{f_i} = -t_{s_i}$, then $m_f = -m_s$.

Program *smallest*

The program *smallest* takes four integers as inputs, and outputs the smallest amongst them. Using t_s and t_f to denote the source and follow-up test inputs, where t_{s_i} (t_{f_i}) denotes the i^{th} ($1 \leq i \leq 4$) integer of the input, and x_s and x_f to denote the source and follow-up outputs, the following MRs were identified for *smallest*:

MR1: If a is a randomly selected positive integer, and t_f is constructed by adding a to every integer of t_s, then $x_f = x_s + a$.

MR2: If t_f is constructed from t_s, such that $t_{f_i} = -t_{s_i}$, then this results in the following cases:

(1) If all elements of t_s are identical, then $x_f + x_s = 0$.

(2) If every element of t_s is less than or equal to 0, then $x_f > x_s$.

(3) In other cases, if $x_s \geq 0$, then $x_f < x_s$; otherwise $x_s + x_f < 0$.

Program *syllables*

The input of *syllables* is a string (of maximum length 20), and the output is a string "The number of syllables is *N*", where *N* is the number of vowel characters ('a', 'e', 'i', 'o', 'u', and 'y') found in the input. For example, if the input string is "abiad", then the output is "The number of syllables is 3". Using t_s and t_f to denote the source and follow-up test

Program	T_b	T_w	M_b^1	M_w^1	M_b^2	M_w^2
checksum	6	10	6	10	6	10
digits	6	10	5	9	5	9
grade	9	9	9	9	7	7
median	7	6	7	6	7	6
smallest	8	8	8	8	8	8
syllables	6	10	6	10	6	8

Table 4.3 Sizes of test suites and MTG sets

inputs, and n_s and n_f to denote the values of N in the source and follow-up outputs, the following MRs were identified for *syllables*:

MR1: If t_s is split into two sub-strings, and each of them becomes a follow-up test case (t_f^1 and t_f^2), then $n_f^1 + n_f^2 = n_s$.

MR2: If t_s contains less than 20 characters, and t_f is constructed by randomly inserting a vowel character into t_s, then $n_f = n_s + 1$.

4.3.2 APR tools setup

GenProg and GenProg-MT Configurations

Both GenProg and GenProg-MT were used to repair the subject programs. For each subject program, GenProg used the accompanying black and white box test suites (denoted T_b and T_w, respectively); and MTG sets for GenProg-MT were prepared as follows:

(1) M_b^i ($1 \leq i \leq 2$) is a set of MTGs for *MRi* of the target program, constructed using the black box test cases as source test cases. Because some MRs have restrictions on their source test cases, only those satisfying the relevant constraints were used, which meant that it was possible for M_b^i to be smaller than T_b. However, as shown in Table 4.3, the differences were very small, and thus M_b^i and T_b are said to be of *similar* sizes.

(2) M_w^i ($1 \leq i \leq 2$) is a set of MTGs for *MRi* of the target program, constructed using the white box test cases as source test cases. As can be observed from Table 4.3, M_w^i and T_w are also of *similar* sizes.

The experiments on GenProg and GenProg-MT involved the following pairs of APR tools and input test suites or MTG sets — with each pairing being referred to as a *scenario*.

- GP-BTS: GenProg using input test suite T_b.
- GP-WTS: GenProg using input test suite T_w.
- MGP-BTG1: GenProg-MT using input MTG set M_b^1.
- MGP-BTG2: GenProg-MT using input MTG set M_b^2.
- MGP-WTG1: GenProg-MT using input MTG set M_w^1.
- MGP-WTG2: GenProg-MT using input MTG set M_w^2.

The first two scenarios (the GenProg scenarios) need a test oracle, but the other four (the GenProg-MT scenarios) do not.

All of the scenarios were applied to repair faulty versions of the subject programs. Since the repair processes of both GenProg and GenProg-MT are randomised, it was necessary to run each tool multiple times for the same faulty program. For these two tools, the randomisation of their repair processes relates to the value of a setting parameter, a seed, such that the repair result of a repair process is reproducible when the same seed is used. Following the approach taken in previous GenProg studies [Le Goues et al., 2015, 2012b; Qi et al., 2014], ten seed values were used in these experiments: for a given faulty program, each scenario was applied to repair it ten times using these different seeds. The values for the other parameters of GenProg and GenProg-MT were set to be the same as those in the study by Le Goues et al. [2015]. All experiments were conducted on a 32-bit Ubuntu 10.04 machine.

CETI and CETI-MT Configurations

CETI and CETI-MT were configured in a similar way to GenProg and GenProg-MT: T_b and T_w were passed to CETI but the corresponding MTG sets were used by CETI-MT. Thus, the experiments on CETI and CETI-MT involved the following scenarios.

- CE-BTS: CETI using input test suite T_b.
- CE-WTS: CETI using input test suite T_w.

- MCE-BTG1: CETI-MT using input MTG set M_b^1.
- MCE-BTG2: CETI-MT using input MTG set M_b^2.
- MCE-WTG1: CETI-MT using input MTG set M_w^1.
- MCE-WTG2: CETI-MT using input MTG set M_w^2.

Because both CETI and CETI-MT use deterministic repair processes, application of any of the above scenarios to a program involves only one execution of the relevant tool. Moreover, parameters of CETI and CETI-MT were configured the same for repairing each individual subject programs.

4.3.3 Measurements

According to the examination of evaluation metrics for APR techniques (Section 3.2 of Chapter 3), the effectiveness of an APR tool should be measured by the *success rate* and *repair quality*.

- ***Success rate***: The success rate measures the APR technique's ability to generate repairs, as described in Definition 8 (Section 3.2 of Chapter 3), and has been applied in many other APR studies [Le Goues et al., 2012b; Mechtaev et al., 2016; Nguyen et al., 2013; Qi et al., 2014]. Obviously, higher success rates indicate better repair generation ability.
- ***Repair quality***. Definition 3 (Section 3.1 of Chapter 3) was used to measure the repair quality in the experiments: that is, the repair quality was evaluated against evaluation test data. As with Smith et al. [2015], independent test data (rather than the repair test data) were used in the repair quality evaluation. Specifically, each repair was evaluated by three sets of test data, including both a test suite and two MTG sets. For example, in the experiments on GenProg and Genprog-MT, a repair produced by GP-BTS was evaluated by T_w and M_w^i ($1 \leq i \leq 2$); and a repair produced by GP-WTS was evaluated by T_b and M_b^i ($1 \leq i \leq 2$). However, repairs produced by MGP-BTGi were evaluated by T_w and M_w^i, while repairs from MGP-WTGi were evaluated by T_b and M_b^i. Similarly, repairs from CE-BTS and CE-WTS were evaluated in the same way as those from GP-BTS and GP-WTS, respectively; and repairs from MCE-BTGi and MCE-WTGi were evaluated in the same way as those from MGP-BTGi and MGP-WTGi, respectively.

In all these evaluations, the passing rate (or non-violating rate) of the evaluation test suite (or MTG set) was used as the repair quality measurement — a repair was tested against T_b or T_w when using a test oracle; and against an MTG set when not using an oracle. Generally, a higher passing rate (or non-violating rate) indicates that the repair is of a higher quality, with respect to the evaluation data.

To further compare the repair effectiveness between GenProg-MT and GenProg (and between CETI-MT and CETI), the scenarios above were classified into two groups. The six GenProg and GenProg-MT scenarios were classified into *groupB* and *groupW*, as follows:

- *groupB* consisting of GP-BTS, MGP-BTG1 and MGP-BTG2.
- *groupW* consisting of GP-WTS, MGP-WTG1 and MGP-WTG2.

The six CETI and CETI-MT scenarios were also classified into *groupB* and *groupW*, as follows:

- *groupB* consisting of CE-BTS, MCE-BTG1 and MCE-BTG2.
- *groupW* consisting of CE-WTS, MCE-WTG1 and MCE-WTG2.

These classifications are based on the correspondences between the test suites and MTG sets. Obviously, all *groupB scenarios use information from the black box test suite* — for example, GP-BTS uses T_b as the input test suite; and the two GenProg-MT scenarios (MGP-BTG1 and MGP-BTG2) use M_b^1 and M_b^2, whose source test cases are from T_b. Similarly, all *groupW scenarios relate to the white box test suite*. Therefore, when applying a GenProg (CETI) scenario and a GenProg-MT (CETI-MT) scenario of the same group to repair a target program, the input test cases have the same origin. This eliminates the impacts of different input data on repair results of scenarios of the same group, and thus discrepancies between the repair results can be attributed to the APR technique used. Because of this, the comparisons of GenProg-MT (CETI-MT) and GenProg (CETI) were conducted in individual groups, within which the effectiveness of the GenProg (CETI) scenario was compared with its corresponding two GenProg-MT (CETI-MT) scenarios. That is, GP-BTS (CE-BTS) was compared with MGP-BTG1 and MGP-BTG2 (MCE-BTG1 and MCE-BTG2); and GP-WTS (CE-WTS) was compared with MGP-WTG1 and MGP-WTG2 (MCE-WTG1 and MCE-WTG2).

According to the above classification, each group consists of one scenario with an APR technique that needs a test oracle, and two scenarios for the corresponding APR-MT technique that do not need a test oracle. Intuitively speaking, because an oracle can clearly distinguish correct from incorrect execution results, and because an MR violation is less specific (only indicating that at least one of the test inputs produces incorrect output), it is natural to expect the APR technique to deliver a better repair effectiveness than its corresponding APR-MT technique. The question, therefore, is whether or not the APR-MT and APR techniques have comparable effectiveness, that is, whether or not GenProg-MT is comparable to GenProg (or CETI-MT to CETI), in terms of the repair effectiveness.

4.4 Experimental results

The results from the experiments to answer the second research question are presented in this section. The experimental design of GenProg and GenProg-MT involved ten repair processes (with ten individual seeds per scenario) for each faulty version of the subject programs. Consequently, a total of 20 (2×10) repair processes for GenProg and 40 (4×10) for GenProg-MT were conducted to repair each faulty version. CETI and CETI-MT, on the other hand, only used one repair process per scenario for each faulty version, giving a total of two (2×1) processes for CETI, and four (4×1) for CETI-MT, to repair each faulty version. Data were collected from all conducted repair processes, and the success rates and repair quality for all tools were analysed with respect to the individual scenarios. Since a faulty version failing all test cases of the input test suite (or violating all MTGs of the input data) is considered to be extremely malformed, following the practice of Smith et al. [2015], the repair results for such repair processes were excluded. The experimental analysis compared the repair effectiveness of APR-MT tools (GenProg-MT and CETI-MT) with that of the corresponding APR tools (GenProg and CETI).

4.4.1 Experimental results for GenProg and GenProg-MT

This section presents a comparison of the success rates of GenProg and GenProg-MT, followed by an investigation of their resulting repair qualities, and finally a report on the time taken by individual tools.

GenProg and GenProg-MT success rates

This analysis investigates whether or not GenProg-MT has a comparable success rate to GenProg. Due to the configuration of GenProg and GenProg-MT, ten repair processes for each tool were conducted to repair individual program versions, with a faulty program being regarded as repaired by a tool if at least one of the repair processes successfully produced a repair. The success rates were therefore calculated on two different levels: (i) SR_p stands for the success rate at the repair process level (as proposed by Definition 8 in Section 3.2 of Chapter 3), which is the ratio of the number of successful repair processes to the total number of repair processes conducted; (ii) SR_v stands for the success rate at the faulty program level, which is the ratio of the number of programs successfully repaired to the total number of programs to which the scenario was applied. The two success rates for both tools for each program are summarised in Table 4.4, with the accumulated success rates of individual scenarios summarised in the *Total* row: the accumulated SR_p denotes the ratio of successful repair processes to the number of conducted repair processes for all subject programs; and the accumulated SR_v denotes the ratio of the overall number of successfully repaired faulty versions to the total number of faulty programs. Obviously, an accumulated success rate reveals the overall ability of the relevant tool to produce repairs for all the subject programs.

As explained in Section 4.3.3, the success rate comparisons were conducted within *groupB* and *groupW* scenarios. According to Table 4.4(a), in the *groupB* comparisons, MGP-BTG1 has higher success rates than GP-BTS for five of the six subject programs, and lower rates for one (with respect to both SR_v and SR_p). However, MGP-BTG2 has higher success rates than GP-BTS for three of the six programs, has an equal success rate for one, and lower success rates for two (again, with respect to both SR_v and SR_p). On the other hand, as shown in Table 4.4(b), in the *groupW* comparisons, MGP-WTG1 and MGP-WTG2 both have higher success rates than GP-WTS for four of the six subject programs, and similar or lower success rates for two (with respect to both SR_v and SR_p). Overall, based on Table 4.4, it can be concluded that GenProg-MT is comparable to GenProg in terms of success rates. It can also be observed from the last rows of Tables 4.4(a) and 4.4(b) that GenProg-MT slightly outperformed GenProg in terms of the accumulated success rates — because the GenProg-MT scenarios exhibited higher accumulated success rates than those of GenProg. Therefore, the integration of MT and GenProg is effective in terms of the success rate.

Program		**GP-BTS**	**MGP-BTG1**	**MGP-BTG2**
checksum	SR_v	$\frac{2}{30} = 0.067$	$\frac{12}{30} = \mathbf{0.400}$	$\frac{0}{30} = 0.000$
	SR_p	$\frac{2}{300} = 0.007$	$\frac{50}{300} = \mathbf{0.167}$	$\frac{0}{300} = 0.000$
digits	SR_v	$\frac{18}{99} = 0.182$	$\frac{27}{99} = \mathbf{0.273}$	$\frac{10}{88} = 0.114$
	SR_p	$\frac{136}{990} = 0.137$	$\frac{239}{990} = \mathbf{0.241}$	$\frac{67}{880} = 0.076$
grade	SR_v	$\frac{0}{190} = 0.000$	$\frac{4}{188} = \mathbf{0.021}$	$\frac{0}{188} = 0.000$
	SR_p	$\frac{0}{1900} = 0.000$	$\frac{10}{1880} = \mathbf{0.005}$	$\frac{0}{1880} = 0.000$
median	SR_v	$\frac{65}{166} = 0.392$	$\frac{113}{120} = \mathbf{0.942}$	$\frac{144}{153} = 0.941$
	SR_p	$\frac{288}{1660} = 0.173$	$\frac{1129}{1200} = \mathbf{0.941}$	$\frac{1412}{1530} = 0.923$
smallest	SR_v	$\frac{120}{154} = 0.779$	$\frac{91}{102} = \mathbf{0.892}$	$\frac{90}{109} = 0.826$
	SR_p	$\frac{466}{1540} = 0.303$	$\frac{899}{1020} = \mathbf{0.881}$	$\frac{731}{1090} = 0.671$
syllables	SR_v	$\frac{5}{108} = 0.046$	$\frac{0}{21} = 0.000$	$\frac{19}{108} = \mathbf{0.176}$
	SR_p	$\frac{25}{1080} = 0.023$	$\frac{0}{210} = 0.000$	$\frac{48}{1080} = \mathbf{0.044}$
Total	SR_v	$\frac{210}{747} = 0.281$	$\frac{247}{560} = \mathbf{0.441}$	$\frac{263}{676} = 0.389$
	SR_p	$\frac{917}{7470} = 0.123$	$\frac{2327}{5600} = \mathbf{0.416}$	$\frac{2258}{6760} = 0.334$

(a) Success rates of scenarios of *groupB*

Program		**GP-WTS**	**MGP-WTG1**	**MGP-WTG2**
checksum	SR_v	$\frac{2}{52} = 0.039$	$\frac{10}{29} = 0.345$	$\frac{22}{52} = \mathbf{0.423}$
	SR_p	$\frac{13}{520} = 0.025$	$\frac{14}{290} = 0.048$	$\frac{195}{520} = \mathbf{0.375}$
digits	SR_v	$\frac{65}{173} = 0.376$	$\frac{20}{89} = 0.225$	$\frac{103}{168} = \mathbf{0.613}$
	SR_p	$\frac{401}{1730} = 0.232$	$\frac{154}{890} = 0.173$	$\frac{676}{1680} = \mathbf{0.402}$
grade	SR_v	$\frac{0}{186} = 0.000$	$\frac{3}{184} = \mathbf{0.016}$	$\frac{0}{183} = 0.000$
	SR_p	$\frac{0}{1860} = 0.000$	$\frac{7}{1840} = \mathbf{0.004}$	$\frac{0}{1830} = 0.000$
median	SR_v	$\frac{32}{152} = 0.211$	$\frac{96}{103} = 0.932$	$\frac{127}{136} = \mathbf{0.934}$
	SR_p	$\frac{140}{1520} = 0.092$	$\frac{957}{1030} = \mathbf{0.929}$	$\frac{1238}{1360} = 0.910$
smallest	SR_v	$\frac{144}{149} = \mathbf{0.966}$	$\frac{89}{94} = 0.947$	$\frac{81}{152} = 0.533$
	SR_p	$\frac{1370}{1490} = 0.919$	$\frac{868}{940} = \mathbf{0.923}$	$\frac{771}{1520} = 0.507$
syllables	SR_v	$\frac{0}{116} = 0.000$	$\frac{67}{110} = \mathbf{0.609}$	$\frac{37}{123} = 0.301$
	SR_p	$\frac{0}{1160} = 0.000$	$\frac{472}{1100} = \mathbf{0.429}$	$\frac{166}{1230} = 0.135$
Total	SR_v	$\frac{243}{828} = 0.293$	$\frac{285}{609} = \mathbf{0.468}$	$\frac{370}{814} = 0.455$
	SR_p	$\frac{1924}{8280} = 0.232$	$\frac{2472}{6090} = \mathbf{0.406}$	$\frac{3046}{8140} = 0.374$

(b) Success rates of scenarios of *groupW*

Table 4.4 *Success rates* for GenProg and GenProg-MT. SR_v = number of versions repaired / number of versions to which the scenario is applied, and SR_p = number of successful repair processes/ number of repair processes conducted.

GenProg and GenProg-MT repair quality

This section reports on the quality of repairs produced by GenProg and GenProg-MT. The aim of this investigation was to determine whether or not GenProg-MT is as effective

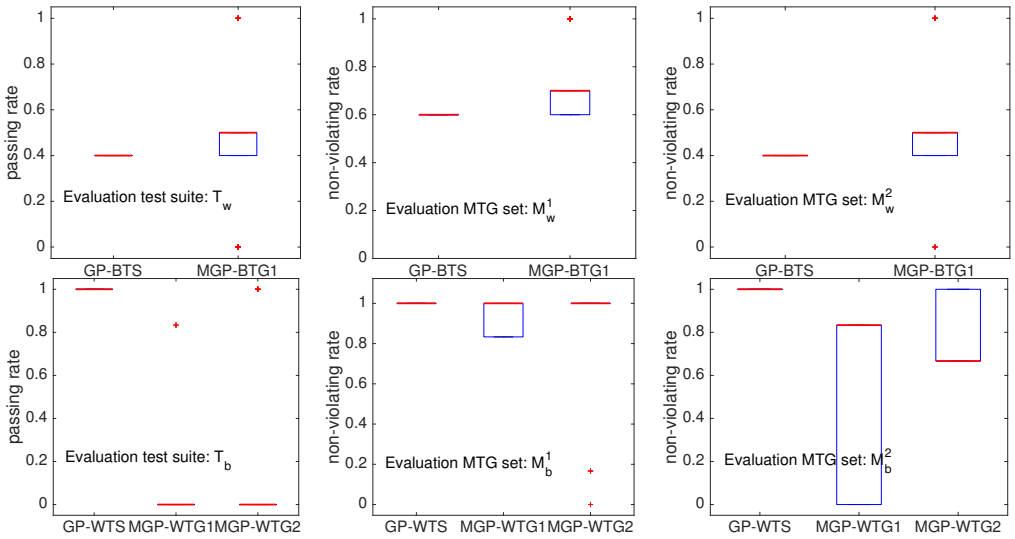
as GenProg in terms of repair quality, that is, whether GenProg-MT can produce repairs of similar quality to those produced by GenProg (the measurement of repair quality was explained in Section 4.3.3). A series of comparisons were conducted on different sets of repairs produced by both *groupB* and *groupW* scenarios. In each comparison, different sets of repairs were evaluated using the same three sets of evaluation data, with the set of repairs produced by GenProg compared with those produced by GenProg-MT. The distributions of quality of repairs are next presented, followed by statistical analyses of both *groupB* and *groupW*, the results of which are further explained to reveal the effectiveness of GenProg-MT in terms of the repair quality.

Distribution of repair quality

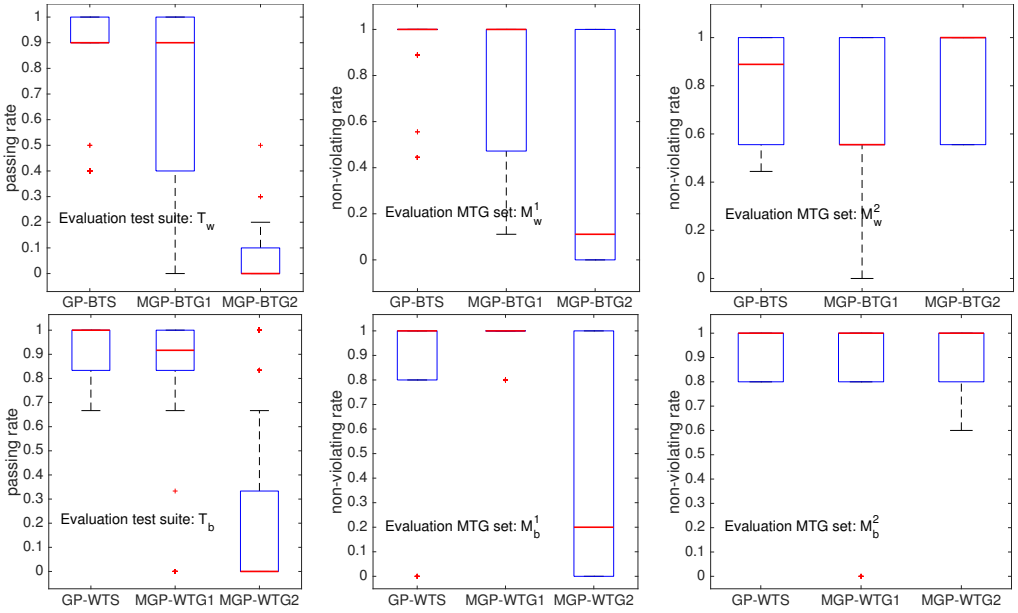
A series of box plot graphs were used to graphically display the distribution of quality in groups of repairs. In each box plot, a box describes the distribution of the quality of a set of repairs with respect to an evaluation test suite or MTG set. Each box also displays some important statistics of the collected data, including the median value (denoted by a bar inside the box), the maximum and minimum values (denoted by bars outside the box), and the 25^{th} and 75^{th} percentiles (denoted by the upper and lower edges of the box, respectively). The results are presented in Figure 4.4, in which there are six sub-figures, referring to the six subject programs. A sub-figure consists of six box plots, each of which shows the distribution of the quality of three sets of repairs produced by *groupB* or *groupW* scenarios, with respect to the corresponding evaluation data. Note that some box plots have less than three sets of data presented — this is because the application of some scenarios to the faulty versions of the relevant subject program failed to produce any repair, and thus no information related to repair quality was collected. Naturally, scenarios producing no repair were regarded to be less effective than scenarios that produced repairs.

It can be observed from Figure 4.4 that the repair quality for GenProg and GenProg-MT fluctuates strongly amongst the different subject programs. Moreover, even for the same subject program, the repair quality related to one scenario of a given APR tool also varies for different evaluation test suites or MTG sets. However, it can also be observed that, except for box plots containing data for only one APR tool (related to program *grade* and the *groupW* comparison of program *syllables*), about one third of the box plots related to these two tools show only minimal variations: the difference between the median values of the groups of data is always less than 0.1.

Statistical analyses



(a) Subject program: *checksum*

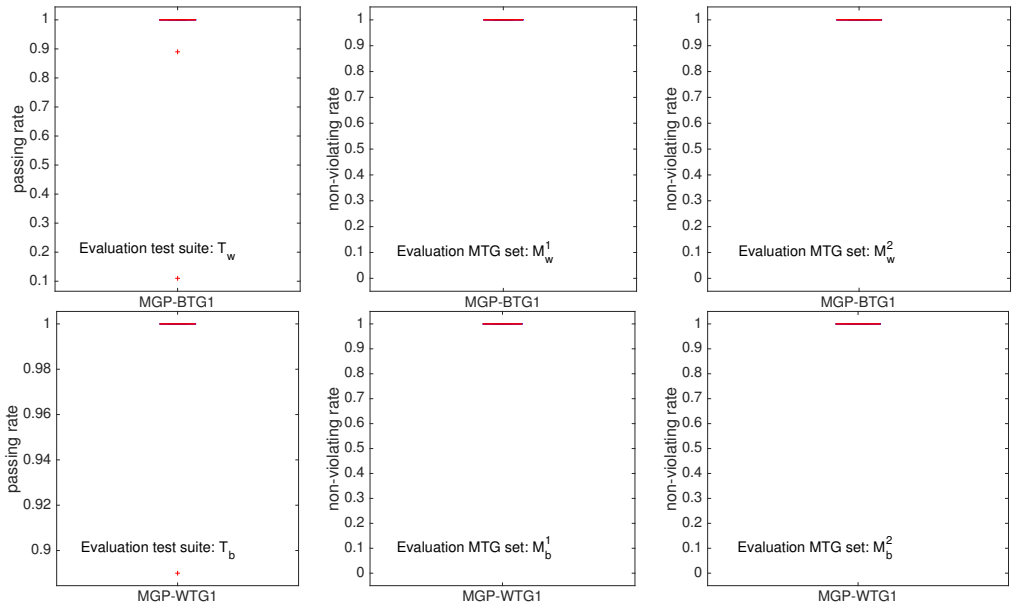


(b) Subject program: *digits*

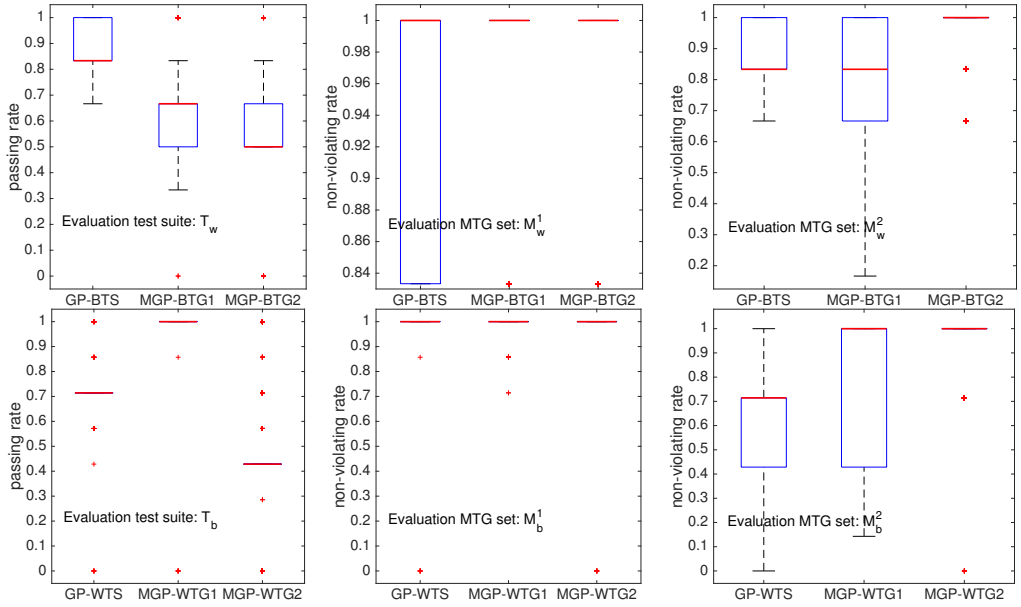
Fig. 4.4 Distribution of repair qualities

Although Figure 4.4 visually presents the differences between the repair qualities of GenProg and GenProg-MT, in order to more rigorously compare their repair quality effectiveness, statistical techniques were applied to conduct pairwise comparisons on *groupB* and on *groupW*.

When comparing a pair of GenProg and GenProg-MT scenarios, the hypothesis test was first applied to verify whether or not there was a significant difference between the quality of the two groups of repairs. Due to the varying success rates of the different scenarios (Table 4.4), two groups of data under comparison may have had different sizes. Furthermore, the data referring to the quality of a set of repairs may not have been normally



(c) Subject program: *grade*

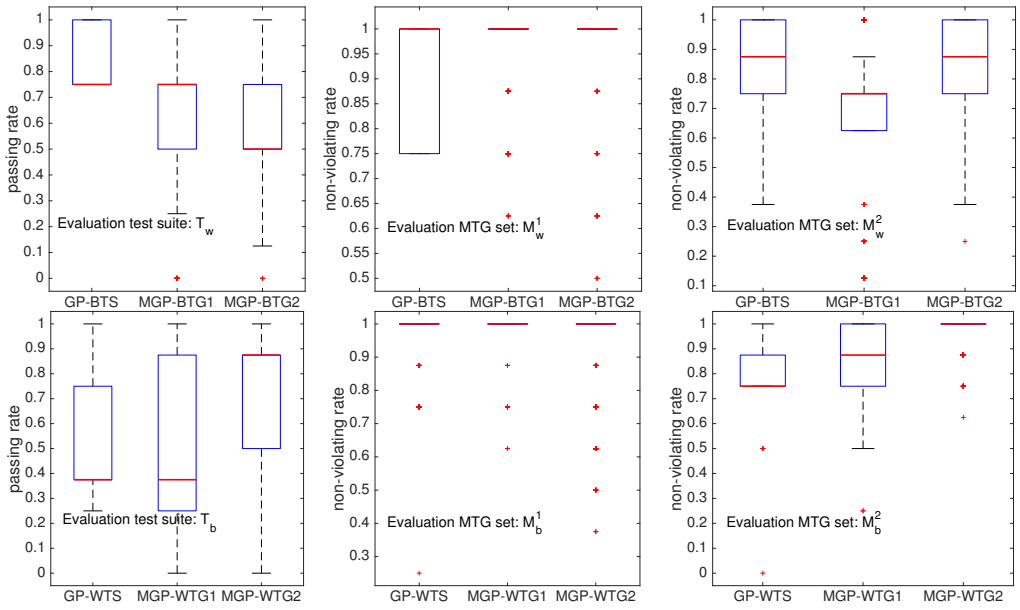


(d) Subject program: *median*

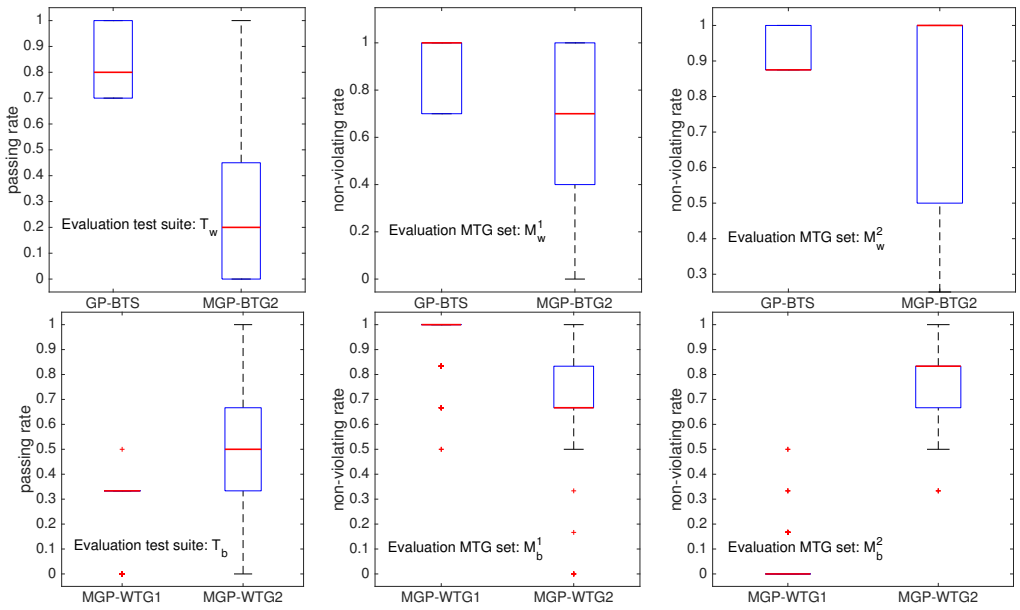
Fig. 4.4 Distribution of repair qualities (*Continued*)

distributed. Therefore, the Wilcoxon rank-sum test [Wilcoxon, 1945] was applied: it is a non-parametric test to verify the null hypothesis that two groups of data come from the same population, with a *p-value* less than 0.05 indicating rejection of the null hypothesis at the 5% significance level.

The Wilcoxon rank-sum test was supplemented by measuring the magnitude of the difference between the two groups of repairs — the *effect size* — by calculating the $\hat{A}_{12}$ statistic [Arcuri and Briand, 2011; Vargha and Delaney, 2000]. The $\hat{A}_{12}$ statistic is a non-parametric effect size measure that assesses the probability that one technique



(e) Subject program: *smallest*



(f) Subject program: *syllables*

Fig. 4.4 Distribution of repair qualities (*Continued*)

outperforms another one — based on two groups of data representing the capabilities of the two techniques under comparison. Setting the data related to GenProg as the first group, and that related to GenProg-MT as the second group, the following interpretations were used: $\hat{A}_{12} < 0.44$ suggests that repairs produced by GenProg-MT are of higher quality than those produced by GenProg; $\hat{A}_{12} > 0.56$ suggests that those by GenProg are of higher quality; and an $\hat{A}_{12}$ value between 0.44 and 0.56 indicates that repairs are of similar quality. Furthermore, $\hat{A}_{12} < 0.29$ or $\hat{A}_{12} > 0.71$ indicates a large effect size; $0.29 \leq \hat{A}_{12} < 0.36$ or $0.71 \geq \hat{A}_{12} > 0.64$ indicates a medium effect size; and $0.36 \leq \hat{A}_{12} < 0.44$ or $0.64 \geq \hat{A}_{12} > 0.56$ indicates a small effect size.

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	0.40	0.00	-	0.60	0.60	-	0.40	0.00	-
mean	0.40	0.47	-	0.60	0.72	-	0.40	0.53	-
median	0.40	0.50	-	0.60	0.70	-	0.40	0.50	-
max	0.40	1.00	-	0.60	1.00	-	0.40	1.00	-
GP-BTS vs. MGP-BTG1	p= 0.386 $\hat{A}_{12}$ = 0.320 **MGP-BTG1** is *better*.			p= 0.106 $\hat{A}_{12}$ = 0.180 **MGP-BTG1** is *better*.			p= 0.165 $\hat{A}_{12}$ = 0.220 **MGP-BTG1** is *better*.		
GP-BTS vs. MGP-BTG2	- **GP-BTS** is *better*.			- **GP-BTS** is *better*.			- **GP-BTS** is *better*.		

Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	1.00	0.00	0.00	1.00	0.83	0.00	1.00	0.00	0.67
mean	1.00	0.18	0.56	1.00	0.95	0.98	1.00	0.60	0.77
median	1.00	0.00	0.00	1.00	1.00	1.00	1.00	0.83	0.67
max	1.00	0.83	1.00	1.00	1.00	1.00	1.00	0.83	1.00
GP-WTS vs. MGP-WTG1	$p < 0.05$ $\hat{A}_{12}$ = 1.000 **GP-WTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 0.643 **GP-WTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 1.000 **GP-WTS** is *better*.		
GP-WTS vs. MGP-WTG2	$p < 0.05$ $\hat{A}_{12}$ = 0.972 **GP-WTS** is *better*.			$p = 0.61$ $\hat{A}_{12}$ = 0.510 *Similar*.			$p < 0.05$ $\hat{A}_{12}$ = 0.867 **GP-WTS** is *better*.		

(a) Subject program:*checksum*

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	0.40	0.00	0.00	0.44	0.11	0.00	0.44	0.00	0.56
mean	0.87	0.69	0.71	0.90	0.82	0.38	0.78	0.72	0.86
median	0.90	0.90	0.00	1.00	1.00	0.11	0.89	0.56	1.00
max	1.00	1.00	0.50	1.00	1.00	1.00	1.00	1.00	1.00
GP-BTS vs. MGP-BTG1	$p < 0.05$ $\hat{A}_{12}$ = 0.640 **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 0.580 **GP-BTS** is *better*.			$p = 0.05$ $\hat{A}_{12}$ = 0.550 *Similar*.		
GP:BTS vs. MGP-BTG2	$p < 0.05$ $\hat{A}_{12}$ = 0.990 **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 0.790 **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 0.401 **MGP-BTG2** is *better*.		

Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	0.67	0.00	0.00	0.00	0.80	0.00	0.80	0.00	0.60
mean	0.93	0.76	0.25	0.94	0.96	0.47	0.94	0.89	0.93
median	1.00	0.92	0.00	1.00	1.00	0.20	1.00	1.00	1.00
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GP-WTS vs. MGP-WTG1	$p < 0.05$ $\hat{A}_{12}$ = 0.601 **GP-WTS** is *better*.			$p = 0.15$ $\hat{A}_{12}$ = 0.470 *Similar*.			$p = 0.02$ $\hat{A}_{12}$ = 0.551 *Similar*.		
GP-WTS MGP:WTG2	$p < 0.05$ $\hat{A}_{12}$ = 0.852 **GP-WTS** is *better*.			$p < 0.05$ $\hat{A}_{12}$ = 0.768 **GP-WTS** is *better*.			$p = 0.60$ $\hat{A}_{12}$ = 0.508 *Similar*.		

(b) Subject program:*digits*

Table 4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality

Statistical analyses were conducted on both *groupB* and *groupW* scenarios. For each subject program, the investigation of *groupB* consisted of six comparisons — comparisons of the set of repairs produced by GP-BTS with those produced by MGP-BTG1 (or MGP-BTG2), using three different sets of evaluation data. Totally, 36 comparisons were conducted on *groupB* (18 of which related to the comparison of GP-BTS with MGP-BTG1, and 18 related to the comparison of GP-BTS with MGP-BTG2). Similarly, for *groupW*, 36

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	-	0.11	-	-	1.00	-	-	1.00	-
mean	-	0.90	-	-	1.00	-	-	1.00	-
median	-	1.00	-	-	1.00	-	-	1.00	-
max	-	1.00	-	-	1.00	-	-	1.00	-
GP-BTS vs. MGP-BTG1	- **MGP-BTG1** is *better*.			- **MGP-BTG1** is *better*.			- **MGP-BTG1** is *better*.		
GP-BTS vs. MGP-BTG2	- *Similar*.			- *Similar*.			- *Similar*.		
Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	-	0.89	-	-	1.00	-	-	1.00	-
mean	-	0.98	-	-	1.00	-	-	1.00	-
median	-	1.00	-	-	1.00	-	-	1.00	-
max	-	1.00	-	-	1.00	-	-	1.00	-
GP-WTS vs. MGP-WTG1	- **MGP-WTG1** is *better*.			- **MGP-WTG1** is *better*.			- **MGP-WTG1** is *better*.		
GP-WTS vs. MGP-WTG2	- *Similar*.			- *Similar*.			- *Similar*.		

(c) Subject program:*grade*

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	0.67	0.00	0.00	0.83	0.83	0.83	0.67	0.17	0.67
mean	0.89	0.63	0.57	0.95	0.98	0.99	0.87	0.78	0.98
median	0.83	0.67	0.50	1.00	1.00	1.00	0.83	0.83	1.00
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GP-BTS vs. MGP-BTG1	$p < 0.05$ $\hat{A}_{12} = 0.899$ **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.436$ **MGP-BTG1** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.564$ **GP-BTS** is *better*.		
GP-BTS vs. MGP-BTG2	$p < 0.05$ $\hat{A}_{12} = 0.925$ **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.390$ **MGP-BTG2** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.249$ **MGP-BTG2** is *better*.		
Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	0.00	0.00	0.00	0.00	0.71	0.00	0.00	0.14	0.00
mean	0.71	0.96	0.44	0.96	1.00	1.00	0.63	0.74	0.93
median	0.71	1.00	0.43	1.00	1.00	1.00	0.71	1.00	1.00
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GP-WTS vs. MGP-WTG1	$p < 0.05$ $\hat{A}_{12} = 0.112$ **MGP-WTG1** is *better*.			$p = 0.06$ $\hat{A}_{12} = 0.488$ *Similar*.			$p < 0.05$ $\hat{A}_{12} = 0.388$ **MGP-WTG1** is *better*.		
GP-WTS vs. MGP-WTG2	$p < 0.05$ $\hat{A}_{12} = 0.899$ **GP-WTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.481$ *Similar*.			$p < 0.05$ $\hat{A}_{12} = 0.170$ **MGP-WTG2** is *better*.		

(d) Subject program:*median*

Table 4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality (*Continued*)

comparisons were conducted to study repairs produced by GP-WTS and by MGP-WTG1 (or MGP-WTG2).

Table 4.5 reports the results of the analyses, with each sub-table presenting the detailed analysis for a subject program. In a sub-table, the p and the $\hat{A}_{12}$ values for all twelve

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	0.75	0.00	0.00	0.75	0.63	0.50	0.38	0.13	0.25
mean	0.83	0.68	0.57	0.93	0.97	0.99	0.84	0.66	0.89
median	0.75	0.75	0.50	1.00	1.00	1.00	0.88	0.75	0.89
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GP-BTS vs. MGP-BTG1	$p < 0.05$ $\hat{A}_{12} = 0.698$ **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.427$ **MGP-BTG1** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.754$ **GP-BTS** is *better*.		
GP-BTS vs. MGP-BTG2	$p < 0.05$ $\hat{A}_{12} = 0.801$ **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.388$ **MGP-BTG2** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.421$ **MGP-BTG2** is *better*.		
Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	0.25	0.00	0.00	0.25	0.63	0.38	0.00	0.25	0.63
mean	0.56	0.54	0.72	0.99	0.99	0.94	0.82	0.86	0.98
median	0.38	0.38	0.88	1.00	1.00	1.00	0.75	0.88	1.00
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GP-WTS vs. MGP-WTG1	$p < 0.05$ $\hat{A}_{12} = 0.562$ **GP-WTS** is *better*.			$p = 0.03$ $\hat{A}_{12} = 0.508$ *Similar*.			$p < 0.05$ $\hat{A}_{12} = 0.426$ **MGP-WTG1** is *better*.		
GP-WTS vs. MGP-WTG2	$p < 0.05$ $\hat{A}_{12} = 0.300$ **MGP-WTG2** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.575$ **GP-WTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.133$ **MGP-WTG2** is *better*.		

(e) Subject program:*smallest*

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2	GP-BTS	MGP-BTG1	MGP-BTG2
min	0.70	-	0.00	0.70	-	0.00	0.88	-	0.25
mean	0.85	-	0.31	0.89	-	0.69	0.93	-	0.80
median	0.80	-	0.20	1.00	-	0.70	0.88	-	1.00
max	1.00	-	1.00	1.00	-	1.00	1.00	-	1.00
GP-BTS vs. MGP-BTG1	- **GP-BTS** is *better*.			- **GP-BTS** is *better*.			- **GP-BTS** is *better*.		
GP-BTS vs. MGP-BTG2	$p < 0.05$ $\hat{A}_{12} = 0.895$ **GP-BTS** is *better*.			$p < 0.05$ $\hat{A}_{12} = 0.658$ **GP-BTS** is *better*.			$p = 0.66$ $\hat{A}_{12} = 0.528$ *Similar*.		
Evaluation data / **groupW**	T_b			M_b^1			M_b^2		
Typical statistics	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2	GP-WTS	MGP-WTG1	MGP-WTG2
min	-	0.00	0.00	-	0.50	0.00	-	0.00	0.33
mean	-	0.31	0.47	-	0.98	0.71	-	0.02	0.80
median	-	0.33	0.50	-	1.00	0.67	-	0.00	0.83
max	-	0.50	1.00	-	1.00	1.00	-	0.50	1.00
GP-WTS vs. MGP-WTG1	- **MGP-WTG1** is *better*.			- **MGP-WTG1** is *better*.			- **MGP-WTG1** is *better*.		
GP-WTS vs. MGP-WTG2	- **MGP-WTG2** is *better*.			- **MGP-WTG2** is *better*.			- **MGP-WTG2** is *better*.		

(f) Subject program:*syllables*

Table 4.5 Statistical analysis of GenProg and GenProg-MT in terms of repair quality (*Continued*)

pairwise comparisons (six for *groupB* and six for *groupW*) are also presented. Based on the analysis, each comparison of two scenarios X and Y, leads to one of three conclusions, either: (1) X is *better* (which indicates that repairs produced by X are of higher quality than those produced by Y); or (2) *Similar* (which indicates that repairs produced by the two scenarios are of similar quality); or (3) Y is *better* (which indicates that repairs produced

Comparison		***Better***	***Similar***	***Worse***
GP-BTS vs. MGP-BTG1	T_w	2	0	4
	M_w^1	4	0	2
	M_w^2	2	1	3
GP-BTS vs. MGP-BTG2	T_w	0	1	5
	M_w^1	2	1	3
	M_w^2	3	2	1
GP-WTS vs. MGP-WTG1	T_b	3	0	3
	M_b^1	2	3	1
	M_b^2	4	1	1
GP-WTS vs. MGP-WTG2	T_b	2	1	3
	M_b^1	1	3	2
	M_b^2	3	2	1

Table 4.6 Summary of comparison results

by *Y* are of higher quality than those produced by *X*). The overall comparison results for individual pairs of scenarios are summarized in Table 4.6. For each pair of scenarios, Table 4.6 reports the following:

- the number of comparisons in which the GenProg-MT scenario produces repairs of higher quality than those produced by the GenProg scenario (reported in the *Better* column).
- the number of comparisons in which the GenProg-MT scenario produces repairs of similar quality to those produced by the GenProg scenario (reported in the *Similar* column).
- the number of comparisons in which the GenProg-MT scenario produces repairs of lower quality than those produced by the GenProg scenario (reported in the *Worse* column).

Tables 4.5 and 4.6 show that GenProg and GenProg-MT have varying performance with respect to different subject programs and different evaluation test suites (or MTG sets). However, in the 36 *groupB* comparisons, MGP-BTG1 is similar or better than GP-BTS for nine of the 18 cases; likewise, MGP-BTG2 is also similar or better for nine of the 18 cases (both MGP-BTG1 and MGP-BTG2 are worse for the remaining *groupB* comparisons). Moreover, for the 36 *groupW* comparisons, 13 show that MGP-WTG1 is similar or better than GP-WTS, and another 12 show that MGP-WTG2 is similar or better than GP-WTS, while the remaining 11 cases show that GP-WTS is better.

Program	**GP-BTS**	**MGP-BTG1**	**MGP-BTG2**	**GP-WTS**	**MGP-WTG1**	**MGP-WTG2**
checksum	25.325	103.602	-	7.115	353.876	353.876
digits	14.049	16.025	51.096	35.517	76.584	66.031
grade	-	224.906	-	-	295.716	-
median	35.562	8.760	34.293	46.216	19.771	40.609
smallest	38.606	14.353	40.047	35.803	34.154	39.342
syllables	109.587	-	117.334	-	11.422	104.673
Average	35.915	14.634	38.420	36.308	29.440	51.680

Table 4.7 Repair time for different scenarios of GenProg and GenProg-MT (in seconds)

Overall, among the 72 results, GenProg-MT has 28 better results and 15 *similar* results (and GenProg has 29 *better* results). This means that repairs produced by GenProg-MT are of comparable quality to those produced by GenProg. In other words, GenProg-MT is of similar effectiveness to GenProg, in terms of repair quality.

In summary, based on the experimental analysis of both the success rates and repair quality, it can be concluded that GenProg-MT is of comparable repair effectiveness to GenProg, and the integration of MT with GenProg is, therefore, not only feasible but also effective. This further indicates that in the application of MT to GenProg, the use of the less precise MRs (rather than the more precise test oracles) may not significantly impair the repair effectiveness.

Repair time

In addition to the repair effectiveness, the time required for each successful repair process was recorded: Table 4.7 summarises the average times required to generate a repair for each subject program and each scenario over all faulty versions of the subject program; and the overall average time spent per repair. Note that, when a scenario fails to generate any repair for a subject program, no time information is collected and thus a '-' is allocated to the relevant cell of Table 4.7 (e.g., the cell corresponds to GP-BTS and program *grade*). As can be seen from the table, GenProg and GenProg-MT have different repair times for the different programs. Furthermore, the repair time (for both tools) for each subject program varies with different input test suites (or MTG sets). In general, GenProg and GenProg-MT had similar repair times. As shown in the last row of Table 4.7, for the *groupB* scenarios, MGP-BTG1 required much less time than GP-BTS to produce a repair; and MGP-BTG2 required a little more time than GP-BTS. Similarly, for the *groupW* scenarios, MGP-WTG1 used less time than GP-WTS to produce a repair, while MGP-WTG2 used more. It can also be observed that the discrepancies between the average repair time required by these two tools are quite small.

4.4.2 Experimental results for CETI and CETI-MT

This section reports on the experimental results for CETI and CETI-MT, which were compared in a similar manner to the GenProg-MT and GenProg comparisons. However, due to the differences between the configurations of these two groups of techniques (as explained in Section 4.3.2), the analysis of CETI and CETI-MT differs in the following ways: 1) the success rate was only calculated at the repair process level because only one scenario repair process was conducted per faulty program; and 2) all repairs from different programs (but from the same scenario) were put into one group in the statistical analysis of the repair quality — the reason for this was that the number of repairs yielded from scenarios of CETI and CETI-MT was relatively small, and thus classifying repairs with respect to subject programs (as was done for the GenProg and GenProg-MT comparison) is not appropriate for statistical analysis.

CETI and CETI-MT success rates

A comparison of the success rates for CETI-MT and CETI is presented in Table 4.8[3], which shows the success rates for individual subject programs, and the overall accumulated success rates (the ratio of the number of successfully repaired programs one scenario has to the total number of programs that it has been applied to).

As can be observed from Table 4.8, scenarios for both CETI-MT and CETI have varying success rates for different subject programs, but generally show similar effectiveness in terms of the accumulated success rate. Consider, for example, the *groupB* scenarios MCE-BTG1 and CE-BTS: MCE-BTG1 has higher success rates for three programs, equal rates for one, and lower rates for two, resulting in similar accumulated success rates for both scenarios (0.657 and 0.636, respectively). For the corresponding pair of *groupW* scenarios (MCE-WTG1 and CE-WTS): MCE-WTG1 has higher success rates than CE-WTS for three programs, lower rates for the other three, and these two have very similar accumulated rates (0.628 and 0.626, respectively). Similar results can also be observed from the other

[3] In order to apply CETI (or CETI-MT) to IntroClass programs, these programs were slightly modified to change how inputs were accepted: the original programs accepted inputs through *scanf* or *gets*, but the modified programs only accept inputs through *command line arguments*. Other programs were also modified to ensure that the *printf* statements appear only in the *main* function, and that all other functions only use the *return* statement to handle outputs. Therefore, some faults caused through improper reading of input data or printing of output data were eliminated, which meant that the number of faulty versions detected by individual test suites (or MTG sets) is slightly different from that reported in the experiments with GenProg and GenProg-MT.

Program	**CE-BTS**	**MCE-BTG1**	**MCE-BTG2**
checksum	$\frac{5}{16}$ = **0.313**	$\frac{0}{9}$ = 0.000	$\frac{1}{10}$ = 0.100
digits	$\frac{12}{98}$ = 0.122	$\frac{42}{92}$ = **0.457**	$\frac{11}{83}$ = 0.133
grade	$\frac{86}{165}$ = 0.521	$\frac{86}{165}$ = 0.521	$\frac{96}{165}$ = **0.582**
median	$\frac{159}{166}$ = 0.958	$\frac{118}{118}$ = **1.000**	$\frac{152}{152}$ = **1.000**
smallest	$\frac{119}{143}$ = **0.832**	$\frac{66}{90}$ = 0.733	$\frac{72}{96}$ = 0.750
syllables	$\frac{7}{22}$ = 0.318	$\frac{2}{4}$ = 0.500	$\frac{16}{22}$ = **0.727**
Total	$\frac{388}{610}$ = 0.636	$\frac{314}{478}$ = 0.657	$\frac{348}{528}$ = **0.659**

(a) Success rates of scenarios of *groupB*

Program	**CE-WTS**	**MCE-WTG1**	**MCE-WTG2**
checksum	$\frac{5}{38}$ = **0.132**	$\frac{0}{8}$ = 0.000	$\frac{1}{32}$ = 0.031
digits	$\frac{36}{166}$ = 0.217	$\frac{19}{71}$ = 0.268	$\frac{42}{151}$ = **0.278**
grade	$\frac{86}{161}$ = 0.534	$\frac{85}{161}$ = 0.528	$\frac{96}{161}$ = **0.596**
median	$\frac{151}{152}$ = 0.993	$\frac{101}{101}$ = **1.000**	$\frac{135}{135}$ = **1.000**
smallest	$\frac{144}{144}$ = **1.000**	$\frac{67}{91}$ = 0.736	$\frac{117}{141}$ = 0.830
syllables	$\frac{1}{15}$ = 0.067	$\frac{2}{4}$ = 0.500	$\frac{17}{22}$ = **0.773**
Total	$\frac{423}{676}$ = 0.626	$\frac{274}{436}$ = **0.628**	$\frac{408}{642}$ = 0.636

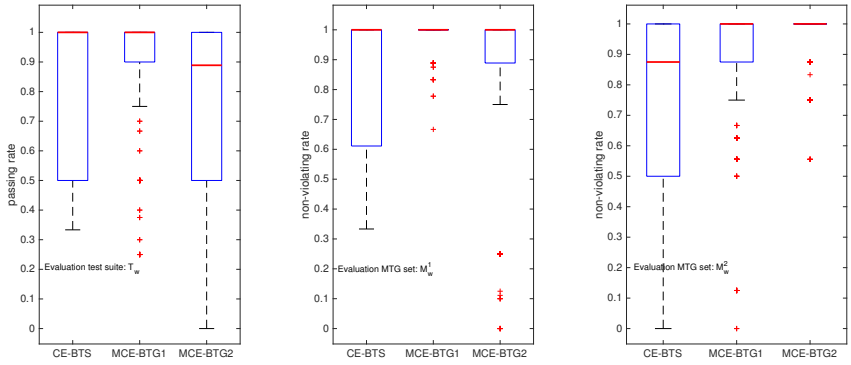
(b) Success rates of scenarios of *groupW*

Table 4.8 *Success rates* for CETI and CETI-MT

pairs of CETI-MT and CETI scenarios. In summary, it can be concluded that CETI-MT can deliver comparable success rates to CETI.

CETI and CETI-MT repair quality

To further elaborate on the the repair effectiveness of CETI and CETI-MT, this section focuses on the quality of repairs produced by the two techniques. As discussed in Section 4.3.3, the quality of each repair was evaluated using a test suite and two MTG sets (all of which were independent of the input test data), with comparisons conducted on both *groupB* and *groupW* scenarios. The results are grouped according to scenario (rather than subject program), giving six *groupB* comparisons and six *groupW* comparisons. The distributions of repair quality are also presented, using box plot graphs. Moreover, the



(a) Distributions of the repair quality

Evaluation data / **groupB**	T_w			M_w^1			M_w^2		
Typical statistics	CE-BTS	MCE-BTG1	MCE-BTG2	CE-BTS	MCE-BTG1	MCE-BTG2	CE-BTS	MCE-BTG1	MCE-BTG2
min	0.33	0.25	0.00	0.33	0.67	0.00	0.00	0.00	0.56
mean	0.82	0.90	0.81	0.86	0.99	0.83	0.80	0.89	0.94
median	1.00	1.00	0.89	1.00	1.00	1.00	0.88	1.00	1.00
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CE-BTS vs MCE-BTG1	p <0.05 $\hat{A}_{12}$ = 0.436 **MCE-BTG1** is *better*.			p <0.05 $\hat{A}_{12}$ = 0.380 **MCE-BTG1** is *better*.			p <0.05 $\hat{A}_{12}$ = 0.377 **MCE-BTG1** is *better*.		
CE-BTS vs MCE-BTG2	p <0.05 $\hat{A}_{12}$ = 0.544 *Similar*.			p <0.05 $\hat{A}_{12}$ = 0.566 **CE-BTS** is *better*.			p <0.05 $\hat{A}_{12}$ = 0.344 **MCE-BTG2** is *better*.		

(b) Statistical analysis results

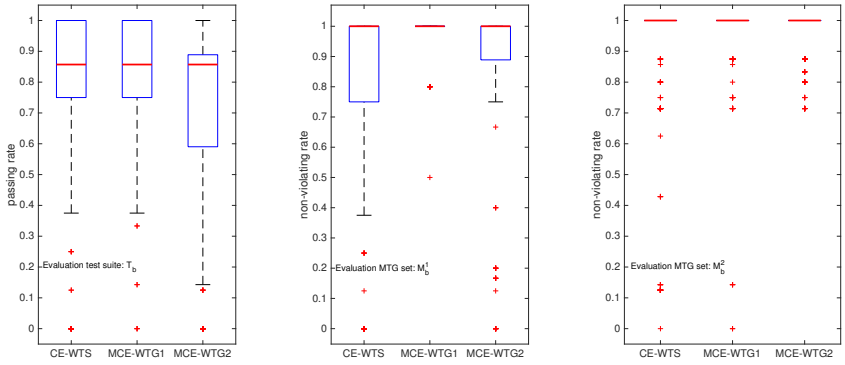
Fig. 4.5 Comparisons of *groupB* repair quality

same statistical analysis techniques as in the experiments of GenProg and GenProg-MT (Section 4.4.1) were used here.

The results of the *groupB* comparisons are summarised in Figure 4.5. In the six *groupB* comparisons — three between CE-BTS and MCE-BTG1 and three between CE-BTS and MCE-BTG2 — CETI-MT is shown to be *better* than CETI for four comparisons, is worse for only one, and is *similar* to CETI for one. Specifically, MCE-BTG1 is shown to be *better* than CE-BTS for all three cases; but MCE-BTG2 has one *better* case than CE-BTS, one *worse*, and one *similar*.

A similar analysis conducted on the *groupW* scenarios (Figure 4.6) reveals CETI-MT to be *better* than CETI for one of the six comparisons, *worse* for one, and *similar* for four: MCE-WTG1 has one *better* case than CE-WTS, and two *similar* cases; and MCE-WTG2 has one worse case than CE-WTS, and two *similar* cases.

These results suggest that repairs constructed by CETI-MT are of comparable quality to those constructed by CETI — CETI-MT is of comparable effectiveness to CETI, in terms of the repair quality. Based on both the success rate and repair quality, it can be concluded



(a) Distributions of the repair quality

groupW \ Evaluation data	T_b			M_b^1			M_b^2		
Typical statistics	CE-WTS	MCE-WTG1	MCE-WTG2	CE-WTS	MCE-WTG1	MCE-WTG2	CE-WTS	MCE-WTG1	MCE-WTG2
min	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.71
mean	0.82	0.83	0.74	0.87	0.99	0.92	0.89	0.94	0.97
median	0.86	0.86	0.86	1.00	1.00	1.00	1.00	1.00	1.00
max	1.00	0.83	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CE-WTS vs MCE-WTG1	p =0.4305 $\hat{A}_{12}$ = 0.483 *Similar.*			p <0.05 $\hat{A}_{12}$ = 0.370 **MCE-WTG1** is *better.*			p =0.648 $\hat{A}_{12}$ = 0.492 *Similar.*		
CE-WTS vs MCE-WTG2	p <0.05 $\hat{A}_{12}$ = 0.565 **CE-WTS** is *better.*			p =0.090 $\hat{A}_{12}$ = 0.473 *Similar.*			p <0.05 $\hat{A}_{12}$ = 0.462 *Similar.*		

(b) Statistical analysis results

Fig. 4.6 Comparisons of *groupW* repair quality

that CETI-MT can achieve comparable repair effectiveness to CETI. This further indicates that the use of MRs by CETI-MT may not necessarily deteriorate the repair effectiveness as compared with CETI.

Repair time

The time taken by CETI and CETI-MT to construct the repairs was also recorded: Table 4.9 presents the average time taken by a scenario to generate a repair for the relevant subject program. As a reminder, when a scenario fails to generate any repair for a subject program, no time information is collected and thus a '-' is allocated to the relevant cell of Table 4.9 (e.g., the cell corresponds to MCE-BTG1 and program *checksum*). In addition, the *Average* row gives the overall average time taken for a repair for each scenario. It can again be observed that each scenario took different amounts of time to repair the different subject programs, sometimes with CETI outperforming CETI-MT, and sometimes with CETI-MT outperforming CETI. Although CETI generally used less time than CETI-MT, the differences between their repair times are not significant.

Program	CE-BTS	MCE-BTG1	MCE-BTG2	CE-WTS	MCE-WTG1	MCE-WTG2
checksum	84.435	-	33.979	15.166	-	36.482
digits	171.836	150.623	227.809	107.762	132.638	146.583
grade	0.565	1.395	1.065	0.170	1.406	1.112
median	2.408	5.410	4.891	3.716	3.704	4.339
smallest	1.288	2.432	2.555	0.294	2.350	2.394
syllables	108.453	1.843	324.082	72.497	2.715	143.299
Average	9.866	21.021	23.986	10.983	11.593	23.533

Table 4.9 Repair time for different scenarios of CETI and CETI-MT (in seconds)

4.4.3 Summary

In summary, the experimental results were very positive, showing that, even without the use of a test oracle, GenProg-MT and CETI-MT were able to demonstrate comparable effectiveness to GenProg and CETI. These results confirm the feasibility of applying MT to APR, and also emphasise the practical effectiveness of APR-MT techniques. Furthermore, it can be concluded that in the application of MT to test suite based APR, although the less precise MRs (instead of the more precise test oracles) are used, the resulting repair effectiveness may not significantly differ from that of the relevant APR technique.

4.5 Discussion

This section examines the implications of the empirical results, and discusses several important factors impacting on the effectiveness of APR and APR-MT techniques.

4.5.1 Impact of MRs on the effectiveness of APR-MT techniques

It is generally recognised that different MRs have different impacts on the fault detection capability of MT [Liu et al., 2014]. A new, but similar, observation from this study is that MRs are crucial to the effectiveness of the APR-MT techniques. In the experiments, although only two MRs (*MR1* and *MR2*) were identified for each subject program, the effectiveness with the *MR1* MTG sets was quite different from that with the *MR2* MTG sets. Taking the program *grade* as an example: when using *MR1*, both MGP-BTG1 and MGP-WTG1 yield sets of repairs, but GenProg-MT does not produce any repair when *MR2* is used. Moreover, all repairs produced by MGP-BTG1 satisfy the *MR2* MTG evaluation

set, M_w^2; and all repairs produced by MTG-WTG1 satisfy M_b^2, as shown in Table 4.5(c). This suggests that the effectiveness of the used MR is a factor to the effectiveness of APR-MT technique. Furthermore, it will be worthwhile to investigate the effectiveness of APR-MT techniques that use a set of MTGs based on multiple MRs. It will also be important to study whether *diversity* in MRs is helpful to the APR-MT techniques, as diversity has been observed to play an important role on the effectiveness of MT [Liu et al., 2014].

4.5.2 Impact of the source test cases on the effectiveness of APR-MT techniques

Intuitively, for the same MR, different source test cases should have different impact on MT's fault detection ability, as has previously been investigated by Barus [2010]. The empirical results presented here also show the impact of source test cases on the effectiveness of the APR-MT techniques. For each MR used in the study, two sets of MTGs were constructed (using T_b and T_w as source test suites). Both GenProg-MT and CETI-MT produce different repair results with the use of these two MTG sets. For example, consider the two MTG sets, M_b^2 and M_w^2, which were constructed by using *MR2* for program *checksum*, GenProg-MT only successfully produced repairs when using M_w^2, but not when using M_b^2. Similarly, when using *MR2* for the program *syllables*, GenProg-MT produces many more repairs with M_w^2 than with M_b^2. These results imply that the effectiveness of APR-MT techniques can be affected by the choice of source test cases.

4.6 Conclusion

Test suite based APR techniques have been widely studied in recent years, and many APR techniques have shown great potential for repairing large scale and real-life programs. However, current APR techniques usually assume the existence of a test oracle, a constraint which severely restricts the scope of applicability of these APR techniques, particularly where there is no available test oracle. Therefore, research into the alleviation of the test oracle problem in APR is of significant and practical importance.

Motivated by the proven effectiveness of MT in alleviating the test oracle problem, this chapter has proposed to integrate MT with test suite based APR to enable application of APR techniques without the need for a test oracle. The correspondence between the

conventional and the integrated APR techniques (APR-MT techniques) was explained, and a framework to facilitate the integration has been presented. Two APR-MT techniques, GenProg-MT and CETI-MT, were developed based on the GenProg and CETI APR techniques, and a series of experiments were conducted to compare the repair effectiveness between GenProg-MT and GenProg (and between CETI-MT and CETI) by using the IntroClass benchmark programs. The experimental results were very positive, showing that, even without the use of a test oracle, the APR-MT techniques (GenProg-MT and CETI-MT) were able to demonstrate comparable effectiveness to the original APR techniques (GenProg and CETI). These results not only confirm the feasibility of applying MT to test suite based APR, but also demonstrate the effectiveness of APR-MT techniques.

Chapter 5

A novel approach for constructing effective APR input test suites

This chapter focuses on the development of an effective input test suite generation approach for APR, designed with the goal of improving repair effectiveness. First, the impact of input test suites on APR repair effectiveness is analysed and summarised. Then, a novel input test suite generation approach for APR that leverages information derived from violated MRs is proposed. The effectiveness of the proposed test suite generation approach is empirically investigated, which is followed by a discussion of the interplay between input test suites and the APR techniques used.

5.1 Preliminary

Test suite based APR attempts to automatically generate a repair that can fix a faulty PUR by making use of the information associated with a given input test suite (which contains at least one failing test case and also some passing test cases), and only considering a PUR program variant as a repair *if it passes all test cases in the given input test suite*. A failing test case in the input test suite captures some information associated with the PUR's fault that can be used by the APR technique to understand and correct the fault. Even if all failures are caused by the same fault, different failing test cases may reveal the nature of the fault in different ways and to different degrees. Consequently, the use of different failing test cases may provide different kinds of help to repair the fault. Similarly, an APR technique also relies on the passing test cases to understand the PUR's intended

	Input			Output	P/F
	in	*up*	*down*		
t_1	0	300	200	1	P
t_2	1	98	200	0	P
t_3	1	101	200	0	F
t_4	1	150	200	0	F
Test suite $T_1 = \{t_1, t_3\}$					
Test suite $T_2 = \{t_2, t_3\}$					
Test suite $T_3 = \{t_2, t_4\}$					

(a) Test cases and test suites for the program *isUpward*. The "P/F" column indicates whether the test case is passing or failing.

Input test suite / APR tool	T_1	T_2	T_3
Angelix	−3 bias = down; +3 **bias = 201**;	−3 bias = down; +3 **bias = up+100**;	−3 bias = down; +3 **bias = up+87**;
CETI	−9 r =0; +9 **r =1**;	−3 bias = down; +3 **bias = up+100**;	−3 bias = down; +3 **bias= up+66**;
GenProg	−9 r = 0; +9 **r = 1**;	null	null

(b) Repair results from three APR tools under different test suites ("$-i$" and "$+i$" indicates the removal and insertion of the i^{th} statement, respectively, and "null" indicates no repair).

Table 5.1 Repairing the *isUpward* program

behaviour and functionality — obviously, different passing test cases express the intended functionality in different ways. Therefore, *for a given APR technique, the use of different input test suites may yield different repair results*, including, whether a repair can be constructed to pass the input test suite, and the degree to which the generated repair is a correct program (that is, *the quality of the repair*)[4]. In other words, for a given APR technique, a *good* test suite should provide useful information to guide repair of the PUR, and provide a higher repair effectiveness.

The impact of input test suites on the repair effectiveness of APR techniques can be illustrated by taking the program *isUpward* (Figure 3.1 in Section 3.1 of Chapter 3) as an example. Three test suites (denoted T_1, T_2, and T_3) and three APR tools (Angelix [Mechtaev et al., 2016], CETI [Nguyen, 2014], and GenProg [Le Goues et al., 2012b]) were applied to repair this program. Each application of an APR tool took a different input test suite but kept the other configurations unchanged. Table 5.1 gives the details of the test suites and the repair results for the APR tools for each of the three test suites.

[4] Furthermore, in APR, the input test suite also affects fault localisation, which in turn affects the final repair results.

Consider first the two input test suites T_1 and T_2, which both contain the same failing test case t_3, but different passing test cases t_1 and t_2. Every APR tool produced different repair results for these two test suites, as shown in the T_1 and T_2 columns of Table 5.1(b). Notably, GenProg generated a repair when T_1 was applied but failed to produce any repair for T_2. Angelix and CETI produced a repair for both test suites, but the quality of the resulting repairs were different: in both cases, the resulting repair for T_2 was correct but that for T_1 was not. Similarly, consider the input test suites T_2 and T_3, which both contain the same passing test case t_2, but different failing test cases t_3 and t_4. Angelix and CETI produced correct repairs using T_2 but incorrect ones using T_3, as shown in the last two columns of Table 5.1(b). Finally, for the input test suites T_1 and T_3, which contain completely different test cases, Angelix and CETI produced different repairs (neither of which was correct) using these two test suites, but GenProg generated a repair with T_1 but none with T_3.

From the above analysis, it can be observed that: *for a given APR technique, the use of different input test suites can yield different repair results*; and *for a given input test suite, its application to different APR techniques can yield different repair results*. In other words, *the repair results depend on both the APR technique and the input test suite*. These observations motivate the search for *better* input test suites for APR, inspiring the study of APR test suite generation. This chapter presents an input test suite generation approach specifically for APR.

5.2 A novel approach for APR test suite generation

This section describes the input test suite generation approach for APR. The approach is based on MT and MFCCs (the details for which were introduced in Chapter 2, Sections 2.2 and 2.3, respectively). The intuition behind the approach is introduced in Section 5.2.1, and the technical details are presented in Sections 5.2.2 and 5.2.3. Finally, characteristics of the generated test suites are summarised in Section 5.2.4.

5.2.1 Intuition and motivation

MT uses MRs to construct a set of follow-up test cases from a given set of source test cases, and, in this way, can be applied as a new test case generation strategy. The success in detecting unknown faults in some Siemens suite programs [SIR, 2005] and hundreds of

real-life faults in two popular C compilers suggest that MTGs are effective for detecting faults [Le et al., 2014; Xie et al., 2013]. Moreover, an MFCC carries core information about the violation of MRs that should be very useful for APR. It is also likely that MTGs that do not violate the MRs will also be useful for APR — because they show the expected behaviour with respect to the MRs.

The above intuitions motivated an investigation into APR input test suite construction that would include test cases from MTGs satisfying the identified MFCCs, and from MTGs *not* satisfying them. An APR technique using such a test suite[5] would not only obtain information about the individual test cases, but also acquire information about the relevant MRs. Moreover, a resulting plausible repair (if successfully generated) would not only pass each individual test case, but also satisfy the MRs for the relevant MTGs in the test suite. Such a test suite is, therefore, expected to be very useful for delivering better APR repair effectiveness.

5.2.2 Generating test cases from an MFCC

Here, test case generation based on a single MFCC is explained. Recall that an MFCC is expressed as a constraint on the input parameters of a program, and it describes the condition under which the relevant MR is violated. Therefore, both the MFCC constraints and the MR are used to construct test cases.

Given a PUR, an MR[6], and an MFCC, the first step is to solve the MFCC to generate a concrete source test case. Next, the corresponding follow-up test case is generated by referring to the given MR. In this way, an MTG (denoted g) for which the PUR violates the given MR is obtained. It should be noted that the output of the PUR must be incorrect for the source or the follow-up test case (or both): at least one test case in g fails.

A set of MTGs satisfying the negation of an MFCC is also constructed — such MTGs may or may not violate the given MR (because there could be multiple MFCCs for the same MR), but, compared with g, they should carry different kinds of information: they therefore complement g for APR. A straightforward approach for constructing such a set of MTGs is to negate every clause in the conjunctive normal form of the MFCC. This strategy, however, may yield a large number of expressions to be solved, resulting in a large number

[5] Here, it is assumed that, in addition to the metamorphic relations being used, there is an oracle for every test case in the test suite.

[6] For ease of presentation, the MR in this chapter is assumed to involve only one source and one follow-up execution. Treatment of other cases is similar.

of test cases (especially when the MFCC is a long expression). Because a large number of test cases can incur high repair costs in APR, this approach is not desirable. However, another characteristic of MRs can be used: an MR specifies how the follow-up input parameters are related to the source input parameters. Source and follow-up execution results can differ because some parameters can be assigned different values in the two executions. These parameters, referred to as *dominating input parameters* of the MR, are expected to have a higher impact on the satisfaction/violation of the MR than other parameters. By definition, at least one dominating input parameter exists. Yet, an MFCC may not necessarily contain clauses involving dominating input parameters. By only negating clauses involving dominating parameters, the number of constraint expressions to be solved can be reduced, thereby giving a smaller number of test cases for APR.

To formally explain the above strategy, consider an illustrative MFCC structured in the following form: $s_1 \wedge s_2 \wedge \ldots \wedge s_u \wedge s_{u+1} \wedge s_{u+2} \ldots \wedge s_{u+k}$, where $s_1, s_2, \ldots, s_u$ $(u > 0)$ are clauses involving at least one dominating input parameter, and $s_{u+1}, s_{u+2}, \ldots, s_{u+k}$ $(k \geq 0)$ are clauses without any. For this MFCC, the following $u+1$ constraints will be constructed[7]:

$$
\begin{aligned}
c_1 &\equiv \neg s_1 \wedge s_2 \wedge s_3 \wedge \ldots \wedge s_u \wedge s_{u+1} \wedge s_{u+2} \ldots \wedge s_{u+k}, \\
c_2 &\equiv s_1 \wedge \neg s_2 \wedge s_3 \wedge \ldots \wedge s_u \wedge s_{u+1} \wedge s_{u+2} \ldots \wedge s_{u+k}, \\
&\ldots \\
c_u &\equiv s_1 \wedge s_2 \wedge s_3 \wedge \ldots \wedge \neg s_u \wedge s_{u+1} \wedge s_{u+2} \ldots \wedge s_{u+k}, \\
c_{u+1} &\equiv \neg\big(s_1 \wedge s_2 \wedge s_3 \wedge \ldots \wedge s_u\big) \wedge s_{u+1} \wedge s_{u+2} \ldots \wedge s_{u+k}.
\end{aligned}
$$

Each of the above constraints represents a negation of the MFCC. Moreover, solving the above $u+1$ constraints will give $u+1$ concrete source test cases, based on which the respective follow-up test cases can then be generated. These test cases, in addition to the MTG test cases that satisfy the MFCC, will form a test suite.

[7] This treatment is for the situation where $u > 0$. When $u = 0$ (that is, when the MFCC does not involve any dominating input parameter), every clause of the MFCC is negated, one by one.

To illustrate this, consider a program P that accepts three input parameters a, b, and c. Suppose an MR is specified as $P(a,b,c) = P(b,a,c)$, and an MFCC for P under this MR is $(a < b) \wedge (b > 4) \wedge (c < 10)$. One solution to this constraint is $t_0 : (a = 0, b = 20, c = 1)$, for which the follow-up input is $t_0' : (a = 20, b = 0, c = 1)$. Since a and b are the dominating input parameters, and they are only involved in the $a < b$ and $b > 4$ clauses in the MFCC, the following constraints are constructed:

$$
\begin{aligned}
c_1 &\equiv \neg(a < b) \wedge (b > 4) \wedge (c < 10)), \\
c_2 &\equiv (a < b) \wedge \neg(b > 4) \wedge (c < 10)), \quad \text{and} \\
c_3 &\equiv \neg((a < b) \wedge (b > 4)) \wedge (c < 10)).
\end{aligned}
$$

Solutions to c_1, c_2, and c_3 can be, for instance, $t_1 : (a = 6, b = 5, c = 9)$, $t_2 : (a = 2, b = 3, c = 8)$, and $t_3 : (a = 4, b = 3, c = 7)$, respectively, for which the follow-up inputs are $t_1' : (a = 5, b = 6, c = 9)$, $t_2' : (a = 3, b = 2, c = 8)$, and $t_3' : (a = 3, b = 4, c = 7)$. An APR test suite for the given program under the given MR and MFCC could therefore be: $\{t_0, t_1, t_2, t_3, t_0', t_1', t_2', t_3'\}$.

5.2.3 Generating a test suite from a set of MFCCs

When an MR has multiple MFCCs, test suites can be constructed from individual MFCCs, and then combined to form a final test suite. The entire test suite generation procedure is outlined in Algorithm 2, the input to which is a metamorphic relation *MR* and a set of relevant MFCCs of the PUR, denoted *M* — *M* was constructed by conducting semi-proving [Chen et al., 2011] on the PUR, using *MR*. The output is a test suite T for repairing the PUR. The algorithm first identifies a set of dominating input parameters of *MR*, denoted *DP* (line 2). Then, for each MFCC, the procedure described in Section 5.2.2 is repeated (lines 3 to 11). The subroutine GenMTGFromAConstraint (line 4) first solves the current MFCC (m_i) to get a source test case, and then generates the corresponding follow-up test case by referring to the metamorphic relation *MR*. Both the source and follow-up test cases are included in T (line 5). Line 6 generates a set of negations $\{n_1, n_2, \ldots, n_k\}$ based on m_i and *DP*, using the method presented in Section 5.2.2. For each generated negation n_j, the subroutine GenMTGFromAConstraint is called to construct an MTG that satisfies n_j

Algorithm 2: Generating a test suite for a PUR from a set of MFCCs of an MR

Input: *MR*: a metamorphic relation,
$M = \{m_1, m_2, \ldots, m_s\}$ $(s > 0)$: a set of MFCCs of *MR* for the PUR.
Output: A test suite T for the PUR.

```
1  T = ∅;
2  DP = GetDominatingParameters(MR);
3  for each m_i in M do
4      mtg = GenMTGFromAConstraint(m_i, MR);
5      T = T ∪ {t : t ∈ mtg};
6      negations = GenerateNegations(m_i, DP);
       /* Let negations be {n_1, n_2, ..., n_k}, (k > 0).                    */
7      for each n_j in negations do
8          mtg = GenMTGFromAConstraint(n_j, MR);
9          T = T ∪ {t : t ∈ mtg};
10     end
11 end
```

(line 8). Both the source and follow-up test cases of the MTG are added to T (line 9). It should be noted that Algorithm 2 can be applied only when *MR* detects failures for the PUR, in which case M is non-empty. Moreover, once Algorithm 2 is successfully applied, the resulting test suite T must contain at least one failing test case, and thus can be used as an input test suite for repairing the PUR.

Algorithm 2 can also be adapted for multiple MRs, each of which has at least one MFCC for the PUR. In this situation, the algorithm is repeated on each MR, and the union of the test suites generated will be the final test suite for repairing the PUR.

5.2.4 MFCC based input test suites

The proposed input test suite generation approach makes use of information about the violated MRs and MFCCs. The constructed test suite, therefore, has the following characteristics:

(1) The input test suite captures information related to program properties. A test suite consists of MTGs, each of which is a group of source and follow-up test cases for the relevant MR. This means that, in addition to the information captured by individual test cases, test cases belonging to the same individual MTGs also encode the relevant MR to a certain extent. Specifically, test cases from non-violating MTGs specify the relevant MR that should be satisfied, while those from violating MTGs relate to the faults that led

to the relevant MR violation. As a result, *when the input test suite is used for repairing a program, the resulting repair (if successfully constructed) not only passes all the input test suite test cases, but also satisfies the relevant MRs on all the input test suite MTGs*.

(2) The input test suite contains passing test cases useful for fixing the fault. A test suite may contain two categories of passing test cases. The first category comes from MTGs constructed by solving MFCCs. Each such passing test case has a corresponding failing test case of the same MTG — this group of passing and failing test cases is attached to an MR. The passing test case, therefore, provides information that can be used to repair the program to satisfy the relevant MR on the MTG. The second category of passing test cases come from MTGs constructed by solving negations of MFCCs, and are constructed by negating some clauses of the relevant MFCC — thus they have similar execution paths to the failing test cases derived from the MFCC. This second category of test cases provides useful information about how to repair the program to make the failing test cases pass. Regardless of the category of passing test cases involved in the test suite, *they all assist the APR technique to construct a plausible repair*.

5.3 Experimental design

This section presents the design of a series of experiments conducted to examine the effectiveness of the proposed test suite generation approach. The experiments involved the use of different APR tools, and test suites constructed by different approaches.

5.3.1 Subject programs and MRs

Since the ultimate goal of APR is to provide solutions to industrial bug fixing practice, the experimental evaluation used both seeded faults (from small and medium-sized programs) and real-world faults (from large programs). The seeded faults were in subject programs from the well known Siemens suite [SIR, 2005], downloaded from the Software-artifact Infrastructure Repository (SIR)[8]. The real-worlds faults were in two subject programs from the ManyBugs benchmark suite [Le Goues et al., 2015][9]. Table 5.2 summarises the subject packages used in the experiments, listing the name, number of faulty versions, and number of source lines of code (SLOC).

[8] http://sir.unl.edu
[9] http://repairbenchmarks.cs.umass.edu

Name	Versions	SLOC
tcas	41	173
print_tokens2	10	570
schedule	9	412
schedule2	10	374
replace	32	564
gmp	2	145,000
gzip	5	491,000

Table 5.2 Subject packages

MRs for *tcas*

The program *tcas* is an aircraft collision avoidance application. It accepts 12 input parameters of the signed integer type, based on which an operation (*up*, *down*, or *unresolved*) is recommended for the aircraft to take in order to avoid collisions. Using arg_i $(1 \leq i \leq 12)$ to denote the i^{th} input parameter, the expected functionality of *tcas* is summarised as follows: (1) let $cond_1$ be $c_1 \wedge c_2 \wedge c_3$ and, if $cond_1$ is *true*, the output is *up*; (2) let $cond_2$ be $c_1 \wedge \neg c_2 \wedge c_4$ and, if $cond_2$ is *true*, the output is *down*; and (3) in other situations, the output is *unresolved*, where

$$
\begin{aligned}
c_1 \quad &\equiv \quad (arg_2 = 1) \wedge (arg_1 > 600) \wedge (arg_5 \leq 600) \wedge \\
&\qquad (((arg_{11} = 1) \wedge (arg_3 = 1) \wedge (arg_{10} = 0)) \vee \\
&\qquad (arg_{11} \neq 1)), \\
c_2 \quad &\equiv \quad ((arg_{12} = 1) \wedge (arg_8 + 100 > arg_9)) \vee \\
&\qquad ((arg_{12} = 0) \wedge (arg_8 > arg_9)), \\
c_3 \quad &\equiv \quad (arg_4 < arg_6) \wedge (arg_9 < Thresh[arg_7]), \quad \textit{and} \\
c_4 \quad &\equiv \quad (arg_6 < arg_4) \wedge (arg_8 \geq Thresh[arg_7]),
\end{aligned}
$$

where *Thresh* is a predefined array to be explained shortly.

Based on the above functionality, the following MRs were identified (where t_s, t_f, O_s, and O_f denote the source test case, follow-up test case, source output, and follow-up output, respectively):

MR1:

(1) When O_s is *up*, t_f is constructed to make c_3 *false*, then O_f and O_s must be different. In this case, t_f can be constructed as follows: let $t_f.arg_4$ be $t_s.arg_6$, $t_f.arg_6$ be $t_s.arg_4$, and $t_f.arg_8$ be $t_s.arg_9$.
(2) When O_s is *down*, t_f is constructed to make c_1 *false*, then O_f and O_s must be different. In this case, t_f can be constructed as follows: if $t_s.arg_{11}$ is 1, then let $t_f.arg_3$ be $t_s.arg_3 - 1$; otherwise, let $t_f.arg_1$ and $t_f.arg_5$ be $(t_s.arg_1 + t_s.arg_5)/2$, where "/" denotes integer division.
(3) When O_s is *unresolved*, t_f is constructed to make both c_3 and c_4 *false*, then O_f and O_s must be identical. In this case, t_f can be constructed as follows: let $t_f.arg_4$ and $t_f.arg_6$ be $(t_s.arg_4 + t_s.arg_6)$ / 2.

MR2:

The *tcas* program uses an array to store a list of threshold levels, *Thresh*[4] = {400, 500, 640, 740}. These threshold values are used to decide whether or not there is an adequate separation between two aircrafts' routes. arg_7 can take a value of 0, 1, 2, or 3, which specifies the currently used threshold level, namely, $Thresh[arg_7]$. Changes to the threshold level should not affect the output if certain input conditions hold true. Therefore:

(1) If $t_s.arg_7$ is 2 or 0, then let $t_f.arg_7$ be $t_s.arg_7 + 1$, $t_f.arg_8$ be $t_s.arg_8 + 100$, and $t_f.arg_9$ be $t_s.arg_9 + 100$;
(2) If $t_s.arg_7$ is 1, then let $t_f.arg_7$ be $t_s.arg_7 + 1$, $t_f.arg_8$ be $t_s.arg_8 + 140$, and $t_f.arg_9$ be $t_s.arg_9 + 140$.

Then O_f and O_s must be identical.

MR3:

(1) When O_s is *up*, t_f is constructed to make c_2 *false* by adjusting the values of arg_8 and arg_9.
(2) When O_s is *down*, t_f is constructed to make c_2 *true* by adjusting the values of arg_8 and arg_9.
(3) When O_s is *unresolved*, t_f is constructed to satisfy either $cond_1$ or $cond_2$ by adjusting the values of arg_1, arg_2, arg_5, arg_6, arg_8, arg_9, and arg_{11}.

Then O_f and O_s must be different.

MRs for *print_tokens2*

The program *print_tokens2* is a lexical analyser, accepting a text file as input, and outputting all tokens and their corresponding categories.

MR1: Deleting comments.

The source test case contains some comments, and the follow-up test case is constructed by deleting these comments. Since *print_tokens2* does not parse comments, the source and follow-up outputs should be identical.

MR2: Transformation into *identifier* tokens.

An *identifier* token starts with an alphabetic character and consists of alphanumeric characters. Some *error* tokens and *keyword* tokens can be transformed into *identifier* tokens by removing or adding some characters. For example, an *error* token “1a" can be transformed into an *identifier* token “a" by deleting the “1,” and a *keyword* token “and" can be transformed into an *identifier* token “andx" by adding an “x”.

MR2 assumes that the source test case only contains the above categories of *error* and *keyword* tokens, and that the follow-up test case is constructed by adding or removing some characters based on the source test case so that the follow-up test case will be parsed as a list of *identifier* tokens. Thus, the total number of *error* and *keyword* tokens in the source output is equal to the number of *identifier* tokens in the follow-up output.

MR3: Transformation into *digital* tokens.

A *digital* token consists of numeric characters. A *digital* token can be used to construct another *digital* token by adding some digits, and an *error* token starting with a digit can also be turned into a *digital* token by removing its non-digital characters.

MR3 assumes that the source test case only contains the above categories of *digital* and *error* tokens, and that the follow-up test case is constructed by adding or removing some characters based on the source test case so that the follow-up test case will be parsed as

a list of *digital* tokens. Hence, the total number of *digital* and *error* tokens in the source output is equal to the number of *digital* tokens in the follow-up output.

MRs for *schedule* and *schedule2*

Both *schedule* and *schedule2* conduct priority based scheduling on a list of jobs, using non-preemptive and preemptive scheduling, respectively. Both programs use three internal job queues to maintain jobs with different priorities, Q_1, Q_2, and Q_3 — where Q_3 has the highest priority and Q_1 has the lowest. The input to both programs includes three integers (s_1, s_2, s_3), representing the initial sizes of the three queues, and a list of commands representing the operations to be performed. Two MRs were identified for both programs, in each of which the source and follow-up test cases take the same values for s_1, s_2, and s_3, but different lists of commands.

MR1: The source test case for *MR1* includes a sequence of commands that create a new job for Q_2 and request to schedule it before all other jobs in Q_2. The follow-up test case includes a sequence of commands that create a new job for Q_1 and then upgrade it to be the last job in Q_3. The source and follow-up test cases are equivalent in the sense that their outputs should give the same scheduling results.

MR2: *MR2* observes that invalid commands (such as creating an invalid job or upgrading a non-existing job) should have no effect on the scheduling output because these commands should be discarded. *MR2*, therefore, constructs the follow-up test case by inserting invalid commands into the command list of the source test case. The source and follow-up outputs should be identical.

MRs for *replace*

The program *replace* is the most complex one in the Siemens suite, and covers the most varieties of logical errors [Liu et al., 2006]. The program is a text parser that uses pattern matching and text substitution. It accepts three input parameters: an expression *pStr*, which describes a pattern to be matched using a regular-expression-like syntax, a string *sStr*, and a string *aStr*. The program *replace* substitutes *sStr* for every substring of *aStr* that matches the pattern specified by *pStr*. For the MRs identified below, ($pStr_1$, $sStr_1$, $aStr_1$) and ($pStr_2$, $sStr_2$, $aStr_2$) denote the source and follow-up test cases, respectively.

MR1: In this MR, $sStr_1 = sStr_2$, $aStr_1 = aStr_2$, and $pStr_2$ uses square brackets to re-express $pStr_1$ in the context of *sStr* and *aStr*. The two outputs should be identical. For example: both '[b]' and '@b' mean 'b', and both '?b' and '[^]b' mean a string starting with an arbitrary character but ending with character 'b'. Therefore, *replace* 'ab' 'r' 'abc' and *replace* '[a]b' 'r' 'abc' should both return 'rc'; *replace* '@b' 'r' 'ab' and *replace* '[b]' 'r' 'ab' should both return 'ar'; and *replace* '?b' 'r' 'ab' and *replace* '[^]b' 'r' 'ab' should both return 'r'.

MR2: In this MR, $sStr_1 = sStr_2$, $aStr_1 = aStr_2$, and $pStr_2$ is equivalent to $pStr_1$ (in the context of *sStr* and *aStr*), using the range operator '-' to re-express $pStr_1$. The two outputs should be identical. For example: '[bcdef]' and '[b-f]' both mean a character from 'b' to 'f', and '[a-g]' means a character from 'a' to 'g'. Therefore, *replace* '[bcdef]' 'r' 'c,' *replace* '[b-f]' 'r' 'c,' and *replace* '[a-g]' 'r' 'c' should all return 'r.'

MR3: In this MR, the follow-up test case is constructed based on both the source test case and the source output, according to the following three cases:

(1) The entire string of $aStr_1$ completely matches $pStr_1$, in which case $pStr_2$ is constructed by appending $aStr_1$ to $pStr_1$, and $aStr_2$ is constructed by concatenating two copies of $aStr_1$. The two outputs should be identical. For example: *replace* 'a' 'r' 'a' and *replace* 'aa' 'r' 'aa' should both return 'r'.
(2) Only a part of $aStr_1$ matches $pStr_1$, in which case $pStr_2$ is constructed using the negation of the other part of $aStr_1$. The two outputs should be identical. For example: *replace* 'a' 'r' 'ax' and *replace* '[^x]' 'r' 'ax' (where '[^x]' means a non-'x' character) should both return 'rx'.
(3) The source execution does not find any match, in which case $sStr_2$ and $aStr_2$ are set to be the same as $sStr_1$ and $aStr_1$, respectively, and $pStr_2$ becomes a re-expression of $pStr_1$ using square brackets. The two outputs should be identical.

MRs for *gmp*

gmp is a library for arbitrary precision arithmetic containing several functions implementing different functionalities. Two MRs were identified for *gmp*, each for a different functionality.

MR1: *MR1* focuses on the *gmp* exponentiation function *mpz_powm*[10], which accepts three multiple precision integers b, x, and y, as input parameters, and calculates the value of $b^x \% y$ (where % denotes the remainder operator).

In *MR1*, the source and follow-up test cases take the same values for b and x, but different values for y. Using y_1 and y_2 to denote the values of y in the source and follow-up test cases respectively, *MR1* assumes that y_1 is non-zero and odd, such that $b^x < y_1$, and constructs y_2 as $y_1 * y_1$. The source and follow-up outputs should be identical.

MR2: *MR2* focuses on another function, *mpz_gcdext*[11], which calculates the greatest common divisor g of two multiple precision integers a and b, and also returns the coefficients s and t such that $s * a + t * b = g$. One specific property of *mpz_gcdext* is that when the absolute values of a and b are equal, then t is calculated according to b while keeping the value of s a constant.

Using a_1 and b_1, and a_2 and b_2 to denote the values of a and b in the source and follow-up test cases, respectively, *MR2* assumes that a_1 and b_1 have the same absolute value, and constructs the follow-up test case such that $a_2 = b_2 = x$, where the absolute value of x is less than that of a_1, and x is randomly selected. As a result, the values of s in the source and follow-up outputs are identical.

MRs for *gzip*

gzip is a data compression utility that accepts a series of options for encoding a compression or decompression operation, and the files on which the operation will be conducted. If *gzip* successfully accomplishes the operation, then the corresponding outputs are available (e.g., the relevant compressed or decompressed files), otherwise an error message explaining the cause of the failed operation is output. The following two MRs were identified for *gzip*:

MR1: Reading files from *stdin*.

gzip treats the symbol '–' as an indicator of an input from the *stdin*. Therefore, when multiple files need to be handled simultaneously, *gzip* can read them from the *stdin*.

MR1 makes use of the above property of *gzip*. It assumes that both source and follow-up test cases involve decompression operations on two files, one of which is directly provided

[10] https://gmplib.org/manual/Integer-Exponentiation.html
[11] https://gmplib.org/manual/Number-Theoretic-Functions.html

to *gzip* while the other one is provided through *stdin*. *MR1* requires that the source and follow-up test cases read different files from *stdin*: if the source test case has *gzip* read the first file from *stdin*, then the follow-up test case requires *gzip* to read the second file from *stdin*. As a result, *gzip* should produce the same outcome for both test cases (i.e., success or fail), for example: '*gzip* -df a.gz – < b.gz' and '*gzip* -df – b.gz < a.gz' should either both succeed or both fail.

MR2: Using the same suffix for compression and decompression.

By default, *gzip* considers files with the suffix '.gz' as compressed. To specify a different suffix, the option '-S' can be used with the other compression or decompression options. Although some user-specified suffixes may be regarded as invalid for *gzip*, it is natural to assume that *gzip* should accept (or refuse) the same set of suffixes for different operations. For example: if a suffix is considered valid for a compression operation, then it should also be valid for a decompression operation.

MR2 assumes that the source test case involves a compression operation with a given suffix, and that the follow-up test case involves a decompression operation with the same suffix, in which case *gzip* should produce the same outcome for both test cases. For example: if '*gzip* -c -S '.a' 1.txt' succeeds, then '*gzip* -d -S '.a' 1.txt.a' should also succeed.

5.3.2 Test suite based APR tools used in the experiments

Three APR tools were used in the experimental evaluation, each adopting very different methodologies to repair a program.

The first tool was GenProg[12] [Le Goues et al., 2012a,b; Weimer et al., 2013, 2009], which has been used as a baseline APR tool in many studies [Kim et al., 2013; Nguyen et al., 2013; Qi et al., 2014; Smith et al., 2015; Tan and Roychoudhury, 2015]. GenProg constructs candidate programs using genetic algorithm. In each generation of the genetic algorithm, a set of candidate programs are created using crossover and mutation operators. The fitness value of each candidate program is calculated using the input test suite, and candidates with higher fitness values are more likely to progress to the next generation. This process is repeated until a candidate program that passes all test cases of the input test suite is obtained, or the maximum number of generations has been reached. In the experiments, the latest version of GenProg (version 3.0) was configured in the same way

[12] http://dijkstra.cs.virginia.edu/genprog/

as in the studies by Le Goues et al. [2012b], including that the search algorithm was the genetic algorithm, each population was of size 40, there was a maximum of ten generations, etc. Because the GenProg repair process involves a randomised procedure, to obtain a reliable result, GenProg was run ten times (using ten different random seeds) for each PUR and each test suite.

The second tool used was CETI[13]. Unlike GenProg, which uses only statements from the PUR to fix bugs without inventing new code, CETI produces a repair by replacing a PUR statement with a newly created one [Nguyen, 2014]. Given a PUR and an input test suite, CETI first locates suspicious statements by applying statistical fault localisation techniques [Jones and Harrold, 2005]. For each suspicious statement, CETI constructs a list of reachability instance programs using different repair templates. The reachability instance programs are then used to synthesise the repairs. CETI terminates the repair process when either a repair is produced, or all of the generated reachability instance programs have been processed. Because CETI has a deterministic repair process, it was only applied to each PUR and input test suite once, and was configured in a similar way to in the study by Nguyen [2014]. As a reminder, more details about GenProg and CETI can also be found in Section 4.2.2 of Chapter 4.

The third tool was Angelix[14], which conducts a semantics-based analysis to synthesise a repair [Mechtaev et al., 2016]. Given a PUR and an input test suite, Angelix first transforms the PUR into a semantically equivalent program, and then attempts to identify faulty statements by applying the Jaccard formula (one of the commonly used formulas in statistical fault localisation [Jones and Harrold, 2005]) at the expression level. With a ranked list of suspicious expressions, Angelix modifies the PUR by replacing the suspicious expressions with symbolic values. It then extracts semantic information for repairing the PUR by conducting symbolic executions using the input test suite test cases. Angelix then performs component-based synthesis to construct a repair. In the experiments, Angelix was only applied to each PUR and each input test suite once, and was configured in the same way as in the study by Nguyen [2014] for the Siemens programs, and in the study by Mechtaev et al. [2016] for the ManyBugs programs

For each of the above APR techniques, a 12-hour time limit was set for each repair trial, regardless of the test suite type — if a repair process could not finish within 12 hours, it was terminated.

[13] https://bitbucket.org/nguyenthanhvuh/ceti
[14] http://angelix.io

Algorithm 3: The generation of T_r and $\mathbb{T}_r$

Input: P: a PUR,
n: the size of each test suite T_r that is to be generated.
Output: $\mathbb{T}_r$: a collection of T_r's, where each T_r contains at least one failing test case for P.

```
1 T_r = ∅;
2 numberOfTrials = 0;
3 while sizeof(T_r) < 10 and numberOfTrials < 2000 do
4     T_r = GenerateARandomTestSuite(P, n);
      /* Generate a test suite T_r containing n random test cases.  */
5     numberOfTrials = numberOfTrials + 1;
6     if T_r contains at least one failing test case for P then
7         T_r = T_r ∪ T_r;
8     end
9 end
```

5.3.3 Test suite generation approaches

In the experiments, three test suite generation approaches were compared, as outlined in the following:

(1) G_m: the proposed approach. In this study, semi-proving, using the symbolic engine KLEE [Cadar et al., 2008], was used to construct MFCCs for the faulty programs. Furthermore, Algorithm 2 was implemented to generate test cases from a set of MFCCs using the constraint solver STP [Ganesh and Dill, 2007]. The resulting test suite was denoted by T_m.

(2) G_r: enhanced random approach. A procedure to generate a set (T_r) of random test cases (such as random integers, strings, and expressions) for the PURs was implemented, with the resulting test suite set to be the same size as T_m. However, it was found that such a purely random T_r often did not contain any failing test case and hence could not be used for APR. Therefore, another procedure was used to (i) enforce the inclusion of failing test cases in T_r, and (ii) generate a set ($\mathbb{T}_r$) of such test suites. The procedure is described by Algorithm 3: each T_r contains n test cases (which is the size of T_m), and $\mathbb{T}_r$ contains no more than ten test suites (T_r's). The maximum number of trials was set to 2000, after which, if none of the test suites had caused a failure in the given program, then the algorithm terminated and $\mathbb{T}_r$ was an empty set.

(3) G_c: white-box approach based on code coverage. The test case generation tool KLEE [Cadar et al., 2008], which is known for generating high coverage test cases, was used to generate the test suite denoted T_c.

In both the G_m and G_c experiments, KLEE was configured to use the default search heuristics: "random-path interleaved with nurs:covnew" — where "nurs:covnew" means "use Non Uniform Random Search (NURS) with Coverage-New heuristic"[15]. Furthermore, the execution time of KLEE was restricted to a maximum of 120 seconds for the Siemens suite programs and 480 seconds for the ManyBugs suite programs, as was done by other researchers [Mechtaev et al., 2016; Nguyen et al., 2013]. In most cases, KLEE could successfully finish within this time limit.

5.4 Experimental results

This section presents the results of the experiments conducted to examine the effectiveness of the proposed input test suite generation approach. It should be noted that only input test suites containing at least one failing and one passing test case were used to repair the programs [Smith et al., 2015].

5.4.1 Independent and dependent variables

In the Angelix experiment, the repair context and test suite generation approach were independent variables. There were a total of 109 repair contexts (109 faulty programs), for each of which, three different test suite generation approaches were used to generate the input test suite. The dependent variable was the repair result, which was either null (corresponding to an *inapplicable* input test suite and *failed* repair process), or a plausible repair (corresponding to an *applicable* input test suite and *successful* repair process). The independent and dependent variables in the CETI experiment were the same as those for Angelix.

In contrast, the independent variables in the GenProg experiment were the repair context, the test suite generation approach, and the random seed. The application of an input test suite to a repair context involved ten runs of GenProg, each of which was based on a different random seed. As with the other experiments, the dependent variable in the GenProg experiment was the repair result (either null or a plausible repair). Therefore, the GenProg experiments on a repair context and a test suite generation approach involved ten different repair processes, each of which was based on a different random seed but was applied to the same PUR with the same input test suite — these ten repair processes yielded

[15] https://klee.github.io/docs/options/

repair results of null or a plausible repair. In this study, the application of an input test suite to GenProg was regarded to yield a repair for a PUR if at least one of the corresponding repair processes successfully produced a repair.

5.4.2 Analysis of the usefulness of input test suite generation approaches

The experimental evaluation involved comparisons of the proposed approach with other test suite generation approaches (random and code coverage based), according to their effectiveness for APR.

Based on the evaluation metrics proposed in Section 3.3 (Chapter 3), different input test suites can be compared according to their *applicability*, *repair quality* and *usefulness* — *applicability* and *repair quality* each describe a single aspect of the input test suite effectiveness, and *usefulness* describes the overall effectiveness. The effectiveness of a test suite generation approach can be measured by the effectiveness of its test suites, thus a comparison of two approaches can be achieved by comparing the effectiveness of their resulting test suites.

The following sections report on the performance of the three approaches under study, with respect to their applicability and the quality of their resulting repairs. Using the applicability and repair quality, an overall usefulness score for each approach is then calculated (according to Definition 13, in Section 3.3 of Chapter 3), and used to give a final assessment of the effectiveness of the proposed test suite generation approach.

Applicability

The three test suite generation approaches were used to generate test suites for each of the 109 faulty programs, as summarised in Table 5.3. Because the faulty programs had different source code, the test suites generated had differences, including containing different test cases, and being of different size. As explained in Algorithms 2 and 3, it was possible for G_m and G_r to not generate tests suites, in which case the test suite size was reported as 0.

G_m and G_c were applied independently, with the average, maximum, and minimum sizes of their test suites summarised in the $|T_m|$ and $|T_c|$ columns of Table 5.3, respectively. The Linux utility *gcov* was used to collect the T_c coverage data, with S_{cov} and B_{cov} denoting

Program	$\|T_c\|$	$\|T_m\|$	$\|\mathbb{T}_r\|$
tcas	*Avg*: 26 *Max*: 55 *Min*: 20 B_{cov}: 87.4% S_{cov} : 98.5%	*Avg*: 50 *Max*: 142 *Min*: 0	*Avg*: 8 test suites *Max*: 10 test suites *Min*: 0 test suites
print_tokens2	*Avg*: 30 *Max*: 33 *Min*: 29 B_{cov} : 72.1% S_{cov} : 81.7%	*Avg*: 29 *Max*: 60 *Min*: 0	*Avg*: 5 test suites *Max*: 10 test suites *Min*: 0 test suites
schedule	*Avg*: 40 *Max*: 44 *Min*: 35 B_{cov} : 78.2% S_{cov} : 91.5%	*Avg*: 11 *Max*: 35 *Min*: 0	*Avg*: 4 test suites *Max*: 10 test suites *Min*: 0 test suites
schedule2	*Avg*: 43 *Max*: 51 *Min*: 34 B_{cov} : 81.3% S_{cov} : 97.5%	*Avg*: 3 *Max*: 6 *Min*: 0	*Avg*: 4 test suites *Max*: 10 test suites *Min*: 0 test suites
replace	*Avg*: 50 *Max*: 57 *Min*: 45 B_{cov}: 83.4% S_{cov} : 92.0%	*Avg*: 27 *Max*: 93 *Min*: 0	*Avg*: 2 test suites *Max*: 10 test suites *Min*: 0 test suites
gmp	*Avg*: 103 *Max*: 210 *Min*: 5 B_{cov}: 22.4% S_{cov} : 45.4%	*Avg*: 4 *Max*: 4 *Min*: 4	*Avg*: 5 test suites *Max*: 10 test suites *Min*: 0 test suites
gzip	*Avg*: 5 *Max*: 6 *Min*: 4 B_{cov}: 20.0% S_{cov} : 31.4%	*Avg*: 2 *Max*: 5 *Min*: 0	*Avg*: 0 test suites *Max*: 0 test suites *Min*: 0 test suites

Table 5.3 Test suite information

the average statement and branch coverage data of T_c, respectively (also reported in the $|T_c|$ column of Table 5.3).

As explained in Section 5.3.3, the size of T_r should be determined before applying G_r: G_r is applied by referring to T_m. For each faulty program, there was one T_m and one $\mathbb{T}_r$ (where a $\mathbb{T}_r$ is a collection of all T_r's having the same size as T_m). When $|T_m| = 0$ (i.e.,

Program	Angelix				CETI			
	G_c	G_m	$G_r(\#P)$	$G_r(\#R)$	G_c	G_m	$G_r(\#P)$	$G_r(\#R)$
tcas	16	26	32	274	16	30	35	320
print_tokens2	1	5	5	36	1	2	3	30
schedule	0	3	4	36	0	1	0	0
schedule2	0	3	1	10	0	0	0	0
replace	3	8	5	21	5	16	9	55
gmp	1	2	1	10	-	-	-	-
gzip	0	1	0	0	-	-	-	-
Total	21	48	48	387	22	49	47	405

(a) The Angelix and CETI results: the G_c and G_m columns report the number of repaired programs (which is also the number of plausible repairs); the $G_r(\#P)$ column reports the number of repaired programs; and the $G_r(\#R)$ column reports the number of plausible repairs.

Program	GenProg					
	G_c		G_m		G_r	
	#P	*#R*	*#P*	*#R*	*#P*	*#R*
tcas	12	54	16	88	28	1841
print_tokens2	2	17	2	20	5	189
schedule	0	0	2	9	3	92
schedule2	0	0	3	21	4	223
replace	5	25	13	83	7	433
gmp	1	10	1	10	0	0
gzip	0	0	0	0	0	0
Total	20	106	37	231	47	2778

(b) The GenProg results: the *#P* column reports the number of repaired programs, and the *#R* column reports the number of plausible repairs.

Table 5.4 The numbers of repaired programs and plausible repairs ("-" means that the APR tool is not applicable)

when G_m could not generate a failing test case within the time limit), then $\mathbb{T}_r$ was also set to be empty. The $\mathbb{T}_r$ sizes are summarised in the $|\mathbb{T}_r|$ column of Table 5.3.

	Angelix	**CETI**	**GenProg**
G_c	$\frac{21}{109} = 0.1927$	$\frac{22}{109} = 0.2018$	$\frac{106}{1090} = 0.0972$
G_m	$\frac{48}{109} =$ **0.4404**	$\frac{49}{109} =$**0.4495**	$\frac{231}{1090} = 0.2119$
G_r	$\frac{387}{1090} = 0.3550$	$\frac{405}{1090} = 0.3716$	$\frac{2778}{10900} =$ **0.2549**

Table 5.5 Applicability of the test suites generated by different approaches

In the experiments, all test suites were applied to all three APR tools (Angelix, CETI and GenProg), to repair all subject programs. As explained (Section 5.4.1), the G_r performance is the collective performance of all its test suites: a program is repaired by G_r if at least one T_r has produced a repair. The number of plausible repairs from G_r is the total number of plausible repairs from all individual T_r's.

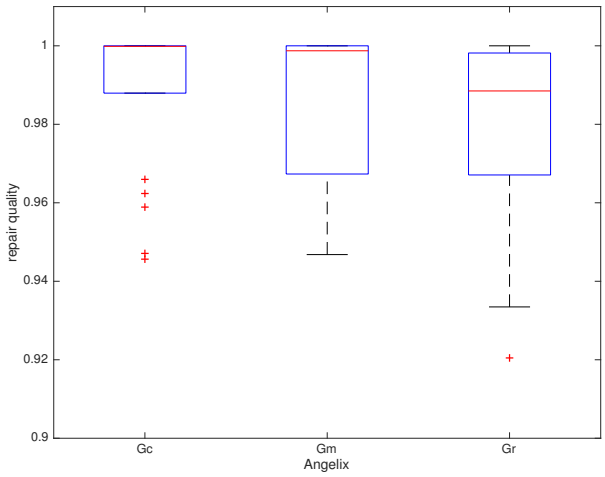
For Angelix and CETI, a program repaired by G_c or G_m corresponds to exactly one resulting plausible repair, but a repair by G_r corresponds to between one and ten plausible repairs (because of the existence of up to ten T_r's): Table 5.4 (a) reports the number of repaired programs for G_c and G_m, and the numbers of repaired programs and plausible repairs for G_r.

For GenProg, a program repaired by G_c or G_m implies one to ten plausible repairs, but a repair by G_r corresponds to up to 100 plausible repairs (because of the use of up to ten T_r's, each with ten different seeds). Table 5.4 (b) gives the details of repaired programs and plausible repairs.

Table 5.5 presents the applicability scores of the test suites generated by three APR tools. Table 5.5 shows that, with respect to the applicability scores for all three APR tools, G_m outperforms G_c in all three comparisons, but only outperforms G_r for two (in the context of Angelix and CETI). In summary, G_m displayed better applicability than the other approaches in five out of six cases.

Repair quality

The examination of repair quality made use of an evaluation test suite formed by randomly sampled test cases from the input domain. It also made use of Definition 3 rather than Definition 2 (Section 3.1 of Chapter 3), which was not feasible for this study. The quality of a repair was measured according to its pass rate for the evaluation test suite, which,



(a) Distributions of the repair quality (for ease of presentation, the lower bound of the y-axis is set to 0.9).

	Min	*Mean*	*Median*	*Max*
G_c	0.9456	0.9889	0.9999	1.0000
G_m	0.7938	0.9731	0.9987	1.0000
G_r	0.7938	0.9685	0.9885	1.0000

G_m vs G_c : p >0.05 $\hat{A}_{12}$=0.4484
No significant difference between G_m and G_c

G_m vs G_r : p <0.05 $\hat{A}_{12}$=0.6367
G_m outperformed G_r with a *small* effect size

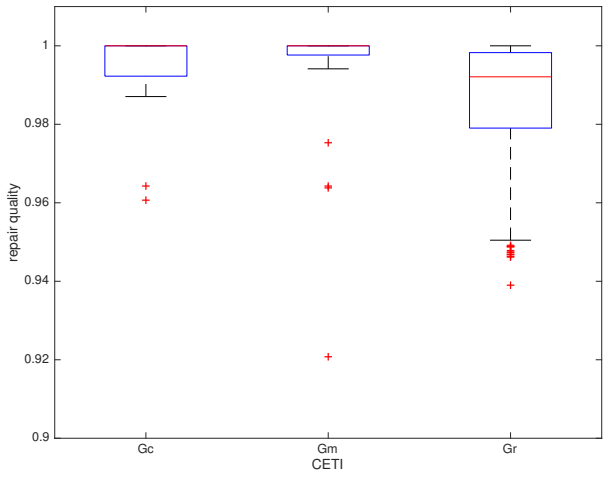
(b) Statistics

Fig. 5.1 Quality of repairs produced by Angelix, with the application of different test suite generation approaches

for each subject program, contained 30,000 test cases. A sample of 30,000 randomly generated test cases should be sufficiently large to make the measurement close to the repair quality if based on the entire input domain.

Figure 5.1(a) presents the repair quality details for Angelix, with three box plots showing the distribution of quality for the three test suite generation approaches. In each box plot, the horizontal line inside the box denotes the median; the top and bottom of the box denote the 25th and 75th percentiles, respectively; and the top and bottom bars outside the box represent the maximum and minimum values, respectively (excluding outliers, which are shown as points outside the top or bottom bars). Figure 5.1(a) indicates that, for Angelix, application of G_r yields repairs of lower quality than the other two approaches.

A statistical analysis was conducted to compare all G_m repairs with those for G_c and G_r. In this comparison, the samples were independent (repairs resulted from the use of test suites from different approaches), and the sample sizes varied (the different



(a) Distributions of the repair quality

	Min	*Mean*	*Median*	*Max*
G_c	0.6274	0.9775	1.0000	1.0000
G_m	0.9208	0.9956	1.0000	1.0000
G_r	0.8590	0.9861	0.9921	1.0000

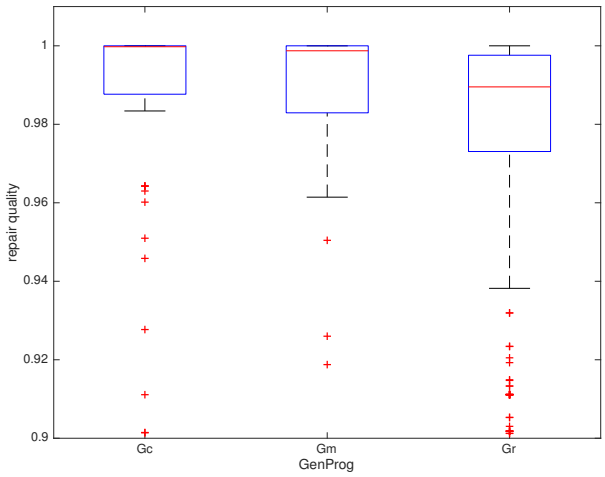
G_m vs G_c : p >0.05 $\hat{A}_{12}$=0.5195
No significant difference between G_m and G_c
G_m vs G_r : p <0.05 $\hat{A}_{12}$=0.7836
G_m **outperformed G_r with a *large* effect size**

(b) Statistics

Fig. 5.2 Quality of repairs produced by CETI, with the application of different test suite generation approaches

numbers of repairs for individual approaches are shown in Table 5.4). Therefore, the non-parametric Wilcoxon rank-sum test [Wilcoxon, 1945] was used to check whether there was a statistically significant difference between two samples. In addition to the *statistical significance*, the *practical significance* (the *effect size*) was examined with the $\hat{A}_{12}$ statistic [Arcuri and Briand, 2011; Vargha and Delaney, 2000], which measures the probability that the first method outperforms the second method. G_m was set to be the first method, therefore $\hat{A}_{12}$ gives the probability that application of G_m yields higher quality repairs than the other method under comparison. G_m is considered comparable to the other method if $\hat{A}_{12}$ is between 0.44 and 0.56; it outperforms the other if $\hat{A}_{12} > 0.56$; and it underperforms the other if $\hat{A}_{12} < 0.44$. Moreover, the effect size can be considered as: "*large*" if $\hat{A}_{12} > 0.71$ or $\hat{A}_{12} < 0.29$; "*medium*" if $\hat{A}_{12} > 0.64$ or $\hat{A}_{12} < 0.36$; and "*small*" if $\hat{A}_{12} > 0.56$ or $\hat{A}_{12} < 0.44$.

The calculaed statistics are shown in Figure 5.1(b). The Wilcoxon rank-sum test suggests that there is no statistically significant difference between the repairs yielded from



(a) Distributions of the repair quality

	Min	*Mean*	*Median*	*Max*
G_c	0.3543	0.9602	0.9998	1.0000
G_m	0.6780	0.9767	0.9987	1.0000
G_r	0.4825	0.9821	0.9895	1.0000

G_m vs G_c : $p > 0.05$ $\hat{A}_{12}$=0.4418
No significant difference between G_m and G_c
G_m vs G_r : $p < 0.05$ $\hat{A}_{12}$=0.6443
G_m outperformed G_r with a *medium* effect size

(b) Statistics

Fig. 5.3 Quality of repairs produced by GenProg, with the application of different test suite generation approaches

G_m and G_c (because $p > 0.05$), but that there is a statistically significant difference between the repairs from G_m and G_r ($p < 0.05$). The $\hat{A}_{12}$ statistic indicates that for Angelix, G_m outperformed G_r with a *small* effect size.

A similar analysis was performed for repairs constructed by CETI and GenProg (Figures 5.2 and 5.3), showing that G_m and G_c yielded repairs of similar quality, but G_m outperformed G_r with a *large* effect size for CETI and a *medium* effect size for GenProg.

In summary, in terms of the repair quality, G_m was comparable to G_c and outperformed G_r, for all three APR tools. Furthermore, the magnitude of the outperformance of G_m over G_r varied for different APR tools.

	Angelix	CETI	GenProg
G_c	0.1906	0.1973	0.0933
G_m	0.4286	0.4475	0.2070
G_r	0.3438	0.3664	0.2503

Table 5.6 Usefulness of the test suites generated by different approaches

Usefulness

The overall effectiveness of individual test suite generation approaches was investigated using the usefulness score, which was calculated based on the applicability and repair quality data. Table 5.6 presents the usefulness scores, and shows that G_m performs better than G_c for all three tools (Angelix, CETI and GenProg), and better than G_r for two (Angelix and CETI). G_r had the best score in one case: GenProg. Overall, G_m was the best performer for five out of six cases, which means that application of G_m can yield more effective repair results most of the time.

5.4.3 Interplay between the test suite generation approaches and the APR techniques

As explained in Section 5.2.4, a key characteristic of the test suite constructed by the proposed approach (T_m) is that it can provide information about some expected program properties in addition to the information carried by the individual test cases. Intuitively, because T_m contains richer information, it should be more useful than the randomly constructed test suite (T_r). However, an apparently contradictory observation was obtained from Section 5.4.2, where, for GenProg, T_r was more useful (a higher usefulness score) than T_m. This particular observation triggered an in-depth analysis of the interplay between the input test suites and the APR techniques.

Different APR techniques have different repair methodologies, and thus use the input test suite in different ways. Because of this, an input test suite may show varying effectiveness for different APR techniques, so does an input test suite generation approach. In the study in this chapter, each test suite generation approach had different usefulness scores for the three APR techniques, as shown in Table 5.6. The Angelix and CETI techniques use the input test suite to derive a kind of specification for synthesising a repair, but GenProg,

in contrast, only uses the test suite to validate candidate programs, not construct them. In this sense, Angelix and CETI rely on the input test suite to a larger extent than GenProg. Moreover, they use the input test suite in a similar way, which is quite different to how GenProg uses it. Probably because of this, every test suite generation approach shown in Table 5.6 had similar usefulness scores for Angelix and CETI, which were very different from those for GenProg.

This analysis reveals an interplay between the test suite generation approaches and the APR techniques. A good input test suite generation approach suitable for one APR technique may not necessarily be good or suitable for another. Furthermore, it is much more likely for an approach to retain similar effectiveness across APR techniques that use the input test suite in a similar fashion than across those using it in very different ways.

Given the experimental results, and the nature of the proposed test suite generation approach, it is possible to hypothesise that the more an APR technique makes uses of the input test suite information, the better the repair effectiveness the approach will achieve. As observed, the proposed approach was more useful than the other two approaches for Angelix and CETI, and was better than the coverage based approach (but worse than random) for GenProg. Consequently, it can be postulated that *the proposed approach is more appropriate for APR techniques that use the input test suite for the construction of a repair or repair candidate — techniques such as Angelix and CETI.*

Generally speaking, a systematic test suite generation approach is based on certain intuitions and, therefore, its test suites will have specific characteristics pertaining to those intuitions. Such test suites should be effective for the APR techniques whose intuitions are supported by the test suites' characteristics, but they should be less effective for others. It is therefore inappropriate to argue whether an input test suite is effective, or useful, without reference to any type of APR technique. Given the large variety of APR techniques, it can be difficult to identify a test suite generation approach that is "the best" for all situations. When studying APR test suite generation, therefore, it is important to understand the types of APR techniques that can be best supported by the relevant test suite generation approach under study. Furthermore, given an APR technique, it is essential to investigate what input test suite characteristics support its effective use.

5.5 Related work

Test case generation is a key step in software testing, which itself is an essential part of software development. Automatically generated test cases have been found to be as useful as manually generated ones for program debugging [Ceccato et al., 2015]. Some test case generation approaches have been proposed specifically for program debugging activities. Artzi et al. [2010] proposed a directed test case generation approach for fault localisation that aims to generate a test suite with high branch coverage. Basically, given a test case execution outcome, their approach constructs new test cases by negating some clauses of the path constraint of this execution in a systematic way. The resulting test suite, while smaller than other studied test suites, has been reported to be equally effective. EntBug [Campos et al., 2013] is a search-based generation approach designed to improve fault localisation effectiveness by using the diagnostic accuracy to guide the generation. The study has shown the resulting test suite to successfully reduce uncertainty in ranking potential faults. The BugEx approach [Röβler et al., 2012] uses a test case generation approach to identify failure causes: a search-based testing technique that constructs passing test cases from a given failing test case such that comparisons between the executions of the passing and failing test cases can help to reveal information about the root cause of the failure.

This chapter has described a test suite generation approach designed to improve the repair effectiveness of APR techniques. The approach differs from others in two ways: firstly, it makes use of both MRs and MFCCs; and secondly, it applies negation operations on MFCCs with respect to the relevant MR. Because of this, the resulting test suites capture essential information related to the violation and satisfaction of the specific MRs.

5.6 Discussion

This section discusses several issues related to input test suites, and presents an analysis of some limitations of the study.

Program	Angelix			CETI			GenProg		
	G_c	G_m	G_r	G_c	G_m	G_r	G_c	G_m	G_r
tcas	44	164	124	3	58	8	42	40	21
print_tokens2	66	317	47	71	168	54	380	116	146
schedule	-	38	90	-	141	-	-	45	106
schedule2	-	90	16	-	-	-	-	25	28
replace	141	339	26	652	334	473	169	263	289
gmp	916	1396	141	-	-	-	387	28	-
gzip	-	158	-	-	-	-	-	-	-
Average	96	248	105	154	154	74	159	126	75

Table 5.7 Mean repair costs, measured in execution time (seconds) per repair, including the test suite generation time and the APR tool execution time ("-" means that no repair was generated and thus no time cost collected)

5.6.1 Input test suites for APR

Time costs

Producing a plausible repair involves time for both (1) the test suite generation and (2) the APR tool execution.

Table 5.7 summarises the mean repair costs (time) associated with the three test suite generation approaches. The time for G_m includes both the MFCC generation time and the time taken to construct test cases from the MFCCs. For G_c and G_r, because test suites exist prior to repairing the program, the repair costs only include the APR tool execution time.

Table 5.7 shows that not only do the time costs vary with different approaches, but even the times for each individual approach vary for different ARP tools or subject programs. Because of the variation in performances, no conclusions can be drawn from the comparisons between G_m and G_c (or G_r) with respect to individual pairs of APR tools and programs. When considering the average time cost per repair (as shown in the Average row of Table 5.7), G_m performed comparably to G_c (taking more time for Angelix, less for GenProg, and the same time for CETI), but required more time than G_r for all three APR tools.

However, it may not be meaningful to consider the average time cost per repair for an APR test suite generation approach without also considering the quality of the repair. Construction of a high quality repair is likely to require more time than construction of a

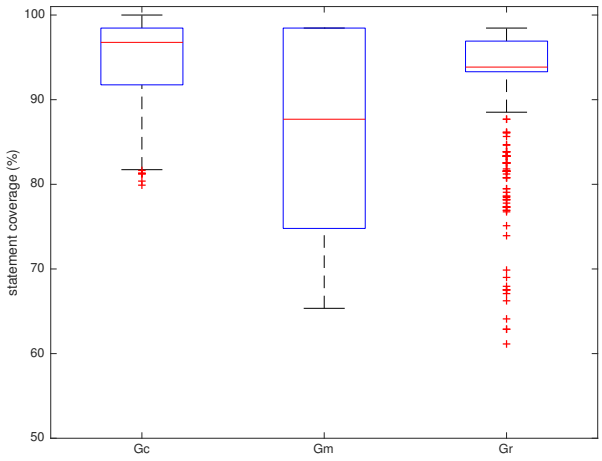
lower quality one. As observed from the experiments on the motivating example (Section 5.1), the application of the input test suite T_2 to Angelix took ten seconds to generate a high quality repair, but the application of the input test suite T_1 only took three seconds to get a lower quality repair; likewise, with CETI, the use of T_2 generated a high quality repair in 2.068 seconds, but the use of T_1 got a lower quality one in only 0.960 seconds. Clearly, therefore, the extra time taken by G_m cannot be simply interpreted as an indication of ineffectiveness: G_m always yields higher quality repairs than G_r, for all three APR tools (as reported in detail in Section 5.4.2).

Unlike G_c and G_r, application of G_m incurs the additional costs of identifying MRs. In this chapter, although the MRs were manually identified, the process was not expensive. Previous studies involving MT and the Siemens programs had already revealed some essential MRs [Xie et al., 2013], which provided very useful guidance for the identification of new MRs. Because the *gmp* and *gzip* programs are commonly used (and hence their functionality is well understood), MR identification for them was also not difficult: as explained in Section 5.3.1, the *gmp* MRs use only some basic mathematics, and the *gzip* MRs only use some common features.

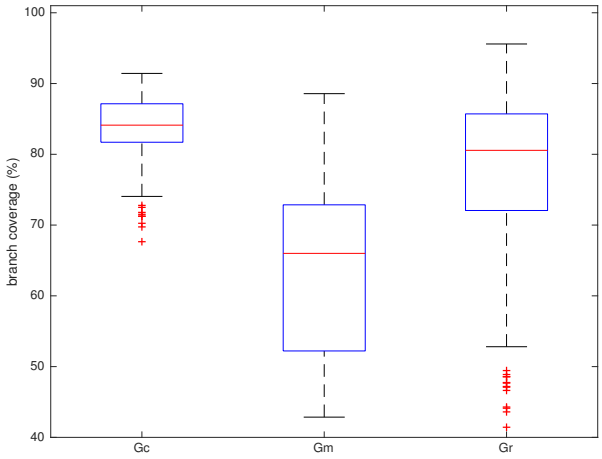
Nevertheless, it should be noted that MR identification should have taken place during the software testing phase, when MT was first applied, not in the APR phase, after the failure had been detected. Obviously, the choice of testing method or test case generation approach should be made before testing can begin. The proposed test suite generation approach G_m should be applied when MT is the chosen testing method [Segura et al., 2016], and hence the cost of identifying MRs can be excluded in this study. For the same reason, the test suite generation time for G_r and G_c have also been excluded.

Characteristics of input test suites

Size of the input test suite. Previous studies have reported that the use of large input test suites can increase the repair time [Le Goues et al., 2012b; Nguyen et al., 2013]. The size of the input test suite has also been reported to affect the repair quality and the number of repaired programs [Le Goues et al., 2012b; Nguyen et al., 2013]. Although a similar observation has been made in the present study — T_c and T_r were of different sizes, and they resulted in different numbers of repaired programs, repair quality, and execution times — it has also been shown that test suite size is not the only factor affecting the repair results. In the comparison between G_m and G_r, for example, the constructed test suites were designed to be of equal sizes, but they yielded very different repair results in terms of the repair costs, quality, and number of programs repaired.



(a) Statement coverage.



(b) Branch coverage.

Fig. 5.4 Coverage of test suites generated by different approaches (for ease of presentation, a limit was set on the lower bound of the y-axis)

Input test suite coverage. Previous studies have also reported that high-coverage test suites have a positive impact on the repair quality [Assiri and Bieman, 2014; Smith et al., 2015]. To investigate this, the coverage information of the test suites used in the experiments was examined: Figure 5.4 shows that the G_c test suites achieved the highest statement coverage and branch coverage. The Wilcoxon rank-sum test and $\hat{A}_{12}$ confirmed that, among the three approaches, G_c test suites had the best coverage, and G_m had the worst.

Although the intuition that a test suite with high coverage should yield high quality repairs is partially supported by the study in this chapter — the coverage based test suites did indeed result in higher quality repairs than the random test suites — the study also showed that, in spite of the difference in coverage, G_c and G_m yielded repairs of similar

quality, and, more importantly, G_c had much poorer applicability. It can be concluded, therefore, that although coverage is a factor, it appears not to be the most important one influencing the APR input test suite effectiveness.

5.6.2 Limitations

Symbolic executions

The proposed test suite generation approach makes use of MFCCs, which in this study were generated using semi-proving. The practical applicability of this implementation, therefore, is determined by the capability of the underlying symbolic execution engine. While there are well known open challenges for symbolic execution techniques, such as the complexity of the path conditions and the path explosion problem, innovative algorithms and optimisation strategies have been developed to alleviate these problems [Boonstoppel et al., 2008; Cadar et al., 2008; Cadar and Sen, 2013; Godefroid et al., 2005; Păsăreanu and Visser, 2009; Sen et al., 2005]. For example, the symbolic execution engine KLEE, which has been successfully applied to various software systems [Cadar and Sen, 2013], uses a mixture of concrete and symbolic executions, and applies various constraint solving optimisations to cope with complex path conditions [Cadar et al., 2008].

Identification of metamorphic relations

As explained in Section 5.6.1, the MRs used in the study were identified manually, without referring to the faults embedded in the subject programs. These MRs turned out to be effective for the detection of failures and the construction of good APR test suites, but further assessment of the APR effectiveness based on different MRs is beyond the scope of this dissertation. While it is widely accepted that different MRs have different fault detection capabilities, the identification of effective MRs is an open and increasingly active research area [Cao et al., 2013]. Liu et al. [2014] showed that identification of suitable MRs is possible, even for inexperienced testers. Zhou et al. [2016] extended MT into a user-oriented testing framework, where MRs can be identified without knowledge of the PUT system specifications or design. Chen [2015] developed a systematic methodology for MR identification, and implement the related tool support. Future progress in automating MR identification will enhance the efficiency of the proposed test suite generation method.

5.6.3 Threats to validity

In terms of the internal validity, it is critical to ensure that the test suite generation methods have been correctly implemented and the experimental data correctly collected: every step of the implementation was carefully verified, and both the intermediate and final experimental data have been thoroughly inspected.

A potential threat to the external validity of the experimental results may be that the study only used three APR tools and a small set of subject programs. Nevertheless, the results reported are still representative because the APR tools implement different APR methods, and the subject programs have also been widely used in the literature.

There should little threat to the construct validity of the experimental results. All of the metrics and statistical methods have been properly justified, and the related concepts have also been formally defined.

5.7 Conclusion

The systematic construction of good input test suites for test suite based APR is a challenging task. This chapter has proposed a novel test suite generation approach for APR, the basic rationale behind which is that MRs are necessary properties of the intended program's functionality and, hence, violated MRs carry rich semantic information that can be used to fix faults in the PUR. The approach was applied with three APR tools (Angelix, CETI, and GenProg) to repair some small and medium-sized programs in the Siemens suite, and some very large programs from the ManyBugs benchmark suite. In the empirical study, the proposed approach was compared with two other test suite generation approaches — an enhanced random approach, and a code coverage based approach. The empirical results clearly show that the proposed approach had best performance for Angelix and CETI, and was also better than the coverage based approach for GenProg.

Further analysis of the experimental results revealed why the proposed test suite generation approach was more effective for Angelix and CETI (which use the input test suite to construct repairs) than for GenProg (which does not use the input test suite to construct repair candidates). The basic observation was that the more an APR technique makes use of the input test suite information, the better will be the effectiveness of the proposed approach. An important conclusion from this is that because of the strong

interplay between the APR technique and the test suite generation approach, analysis of the usefulness of a test suite generation approach should take into account the APR techniques used.

In conventional APR test suite generation approaches, failing and passing test cases are used to identify faults and intended functionality, respectively. The proposed test suite generation approach not only includes the individual failing and passing test cases, but also incorporates the violating and non-violating MTGs. The special construction of these MTGs, which exploit the cause leading to an MR violation, means that the resulting test suite contains specific information pertaining to the nature of the fault and the expected functionality of the intended program. Such test suites have been shown to be very useful for some APR techniques.

Chapter 6

MTRepair: An MT based APR approach

This chapter introduces a new APR approach called MTRepair, which is based on the concepts of MT and MFCC, and aims to generate a plausible repair that satisfies a given MR. In this chapter, the motivation behind the design of MTRepair is first explained, and then a brief overview and description of the MTRepair technical details are given. A prototype tool was also implemented to support the MTRepair approach, and an experimental analysis was conducted to examine its repair effectiveness.

6.1 Motivation

One of the major challenges in test suite based APR relates to the *description of the intended PUR functionality* [Monperrus, 2014; Qi et al., 2015; Tan et al., 2016]. In most APR studies, the input test suite has served as a kind of specification, both revealing failures and encoding intended functionality. Intuitively, however, the PUR's functionality specified in such a way may not be complete, potentially resulting in plausible repairs that pass all input test suite test cases, but fail other test cases (not in the input test suite) [Qi et al., 2015; Smith et al., 2015]. In other words, specifying the PUR functionality (for APR) with test cases (especially those are not comprehensive) may yield low quality repairs, thus reducing the effectiveness of APR techniques. To address this problem, an effective input test suite generation approach for APR was developed, as described in Chapter 5. The study of the input test suite generation approach not only demonstrated promising results,

but also highlighted the interplay between input test suites and APR techniques, revealed that it is very difficult, if not impossible, to develop an input test suite generation approach effective for all APR techniques. It is therefore worthwhile to examine and design new repair methodologies that make use of other information to encode the PUR functionality.

On the other hand, the majority of APR techniques *validate candidate programs against the given input test suite, and do not compare the quality with that of the PUR*. Consequently, there is no guarantee that a plausible repair is of higher quality than the PUR, with respect to the entire input domain. If the input test suite is not sufficiently comprehensive, then a resulting plausible repair, even if it passes more input test suite test cases than the PUR, may not necessarily be of higher quality over the entire input domain. It is therefore important that an APR technique conducts more reliable validation of candidate programs, and constructs repairs of higher quality than the PUR.

Finally, with existing APR techniques, *a candidate program is either accepted as a plausible repair or ignored (without being considered as a potential alternative to the PUR for constructing a plausible repair)*. Intuitively, various ways should exist to rectify a PUR, with each possibly requiring different degrees of technical detail. Clearly, it is preferable that a repair be constructed through a single repair action — for example, GenProg uses a genetic operator to construct a candidate program [Le Goues et al., 2012b], while SemFix constructs repairs by replacing PUR statements with a newly generated ones [Nguyen et al., 2013]. Nevertheless, if such direct repair construction is not possible, then an alternative may be to construct a *successful candidate program* — a candidate program that is of higher quality than the PUR. Such a successful candidate program should be much closer to the final repair than the PUR, and thus should be used as a substitute for the original PUR to construct a plausible repair.

Motivated by these observations, a novel APR approach, referred to as MTRepair, has been developed. MTRepair is a semantics-based APR technique, and makes use of different strategies to address each of the above issues, as discussed in the following.

- **Use MRs instead of input test suites**. MTRepair uses MRs to describe the intended functionality of the PUR, and thus accepts *an MR and a PUR* as input. Because an MR relates to program properties, it should capture more information about the program's functionality than some test cases. Accordingly, MTRepair treats a candidate program that can satisfy the given MR as a plausible repair.
- **Validate a candidate program to check whether it is of higher quality than the PUR with respect to the given MR**. Based on the evaluation metrics proposed in

Chapter 3, MTRepair measures the quality of a program with respect to the entire input domain and the relevant MR. Moreover, an MFCC based measurement is used to compare the quality of a candidate program with that of the PUR.

- **Use an incremental approach to conduct the repair task**. Unlike all existing APR techniques, MTRepair starts with the given PUR, but may operate on a sequence of different programs during a repair process. That is, once a successful candidate program is constructed, MTRepair stops repairing the current PUR and initiates a new repair procedure using the candidate program instead. In this way, a sequence of repair actions is conducted to incrementally repair the faults of the original PUR.

6.2 MTRepair overview

The MTRepair repair process is presented in Algorithm 4. MTRepair accepts as input, a PUR (referred to as the original PUR to avoid confusion) and an MR, and attempts to produce a plausible repair that satisfies the MR. MTRepair first checks if the MR can detect the original PUR failures, which is done by conducting semi-proving on the original PUR and the MR (line 1) — the repair process only continues if the MR can detect them. Next, MTRepair attempts to iteratively repair a sequence of different PURs (line 6 to line 21), including the original PUR and some successful candidate programs constructed in the repair process. A repair result is finally reported as either: (1) *null*, if no plausible repair and no successful candidate programs are constructed (line 10); or (2) a plausible repair, if one is constructed (line 15); or (3) the last successful candidate program (line 12). Similar to other APR techniques, outcomes (1) and (2) indicate the failure and success of a repair process, respectively. However, MTRepair may also have outcome (3), where a successful candidate program is reported as a repair if no plausible repair is constructed. The reason for accepting such a successful candidate program is that, although it does not completely satisfy the given MR, it is of higher quality than the PUR with respect to the MR.

MTRepair attempts to repair a current PUR (denoted *PUR*) using the information about *PUR*'s MFCC constraints and the given MR. The function RepairAPUR (which will be explained in Section 6.3.3) constructs a set of candidate programs that are then validated to identify a successful candidate program. Basically, a candidate program is constructed by replacing one statement in *PUR* with a newly synthesised one. RepairAPUR returns either a successful candidate program P_c; or a *null*, indicating that the repair of *PUR* failed, and thus MTRepair could not construct a plausible repair for the original PUR (line 7). In the case of a successful candidate program, if P_c is also a plausible repair, then MTRepair

Algorithm 4: MTRepair conducts an incremental repair process.

Input: the faulty program P, and the metamorphic relation MR.
Output: a repair R satisfying MR or a *null* indicating the fail of repairing P.

```
1  M_p = SemiProving(P, MR);
2  if M_p is not null then
       /* MR detects failures for P.                                          */
3      PUR = P;
4      M_pur = M_p;
5      P_s = null;
       /* P_s records the last successful candidate program, and is null if no successful candidate
          program has been constructed.                                       */
6      while True do
7          P_c = RepairAPUR(PUR, M_pur, MR);
           /* P_c is the repair result of PUR, which is either a successful candidate program of PUR, or a
              null if MTRepair fails to construct any successful candidate program for PUR.   */
8          if P_c is null then
9              if P_s is null then
10                 return null;
                   /* Neither a plausible repair nor a successful candidate program is available, thus
                      MTRepair fails to repair P.                             */
11             else
12                 return P_s;
                   /* When a plausible repair is not available but a successful candidate program is
                      available, return the last successful candidate program P_s as a repair.   */
13             end
14         else if P_c satisfies MR then
15             return P_c;
               /* P_c is reported as a plausible repair.                      */
16         else
17             P_s = P_c;
               /* P_s records the last successful candidate program.          */
18             PUR = P_c;
19             M_pur = CollectMFCCs(P_c);
               /* Hereafter, MTRepair starts to repair P_c.                   */
20         end
21     end
22 end
```

terminates the repair process (line 15); otherwise P_c is treated as a new PUR to be further repaired (line 18). In this way, the entire repair process iteratively repairs a sequence of different PURs until a plausible repair is obtained or all the repair resources are exhausted.

Obviously, the entire MTRepair repair process of repairing the original PUR consists of a series of procedures repairing individual PURs. All these PURs form a sequence, the first of which is the original PUR. This means that every time a new PUR is used, it is one

```
int median(int a,int b,int c)
1: { int m=c;
2:   if(b < c) {
3:     if(a < b − 5) //should be: a < b
4:       m=b;
5:     else {
6:       if(a < c)
7:         m=a;}}
8:   else {
9:     if(a > b)
10:      m=b;
11:    else {
12:      if(a > c)
13:        m=a; }}
14:  return m; }
```

(a) The original program.

```
int median(int a,int b,int c)
1: { int m=c;
2:   if(b < c) {
3:     if( a <= b-5 )
4:       m=b;
5:     else {
6:       if(a < c)
7:         m=a;}}
8:   else {
9:     if(a > b)
10:      m=b;
11:    else {
12:      if(a > c)
13:        m=a; }}
14:  return m; }
```

(b) The candidate program P_{c_1}.

```
int median(int a,int b,int c)
1: { int m=c;
2:   if(b < c) {
3:     if( a <= b-2 )
4:       m=b;
5:     else {
6:       if(a < c)
7:         m=a;}}
8:   else {
9:     if(a > b)
10:      m=b;
11:    else {
12:      if(a > c)
13:        m=a; }}
14:  return m; }
```

(c) The candidate program P_{c_2}.

```
int median(int a,int b,int c)
1: { int m=c;
2:   if(b < c) {
3:     if( a <= b-1 )
4:       m=b;
5:     else {
6:       if(a < c)
7:         m=a;}}
8:   else {
9:     if(a > b)
10:      m=b;
11:    else {
12:      if(a > c)
13:        m=a; }}
14:  return m; }
```

(d) The candidate program P_{c_3}.

Fig. 6.1 MTRepair repairing a PUR containing one fault

step closer to the construction of a plausible repair. As a result, all PURs used in the repair process together imply a chain of repair actions that gradually fix the original PUR, and in turn incrementally improve its quality. This differs from existing APR techniques (whose repair actions are all performed on the original PUR, but for MTRepair, repair actions may be performed on different PURs), and can deliver additional benefits and effectiveness.

The following two examples illustrate the MTRepair repair process. The first example shows the effect of applying an incremental repair process to gradually fix a single fault, and the second demonstrates the benefit of the incremental repair process for repairing a program with multiple faults.

MTRepair fixes a single-fault program. Figure 6.1 shows MTRepair repairing a faulty version of the program *median*, which aims to output the median of three input integers. It starts with the original PUR and *MR2* for program *median* (as defined in Section 4.3.1 of Chapter 4), which produces a successful candidate program P_{c_1} (Figure 6.1(b)). MTRepair therefore next attempts to repair P_{c_1} (rather than the original PUR), using a similar procedure, which in turn yields a successful candidate program P_{c_2} (Figure 6.1(c)). MTRepair next attempts to repair P_{c_2}, which results in a successful candidate program P_{c_3} (Figure 6.1(d)). Note that P_{c_3} satisfies *MR2* and thus is reported as a plausible repair. It

```
int median(int a, int b,int c)
1: { int m=c;
2:   if(c <= a <= b||b <= a < −c)
3:     m=a;
4:   if(c <= b <= a||a <= b <= c)
5:     m=b;
6:   if((c <= b&&a >= c) || (c >= b&&a <= c) //multiple faults.
7:     m=c;
8:   return m;
9: }
```

(a) The original program.

```
int median(int a, int b,int c)
1: { int m=c;
2:   if(c <= a <= b||b <= a < −c)
3:     m=a;
4:   if(c <= b <= a||a <= b <= c)
5:     m=b;
6:   if(( c > b &&a >= c) || (c >= b&&a <= c)
7:     m=c;
8:   return m;
9: }
```

(b) The candiate program P_{c_1}.

```
int median(int a, int b,int c)
1: { int m=c;
2:   if(c <= a <= b||b <= a < −c)
3:     m=a;
4:   if(c <= b <= a||a <= b <= c)
5:     m=b;
6:   if((c > b&&a >= c) || ( c<b &&a <= c)
7:     m=c;
8:   return m;
9: }
```

(c) The candiate program P_{c_2}.

Fig. 6.2 MTRepair repairing a PUR containing two faults

can be observed that MTRepair processes successive programs in the order "the original PUR" $\rightarrow P_{c_1} \rightarrow P_{c_2} \rightarrow P_{c_3}$ (changes between these programs are highlighted in rectangles in Figure 6.1). In this way, the quality of the original PUR is improved in an incremental manner, and eventually the fault is fixed.

MTRepair fixes a program with multiple faults. Figure 6.2 shows MTRepair using *MR1* for program *median* (also defined in Section 4.3.1 of Chapter 4) to repair a faulty version containing two faults (with changes made to individual PURs again highlighted in rectangles). In this example, a plausible repair satisfying *MR1* is obtained after repairing two PURs: the original PUR (Figure 6.2(a)) $\rightarrow P_{c_1}$ (Figure 6.2(b)), with P_{c_2} (Figure 6.2(c)), the successful candidate program of P_{c_1} that satisfies *MR1* reported as a repair. Importantly, the two generated candidate programs (P_{c_1} and P_{c_2}) each individually fix one fault, which shows that although the repair process only modifies one statement at one time, MTRepair can effectively repair programs containing multiple faults.

6.3 MTRepair technique

MTRepair is a semantics-based APR technique. In addition to the incremental repair process explained in the previous section, another innovation lies in how candidate programs are generated and validated — two activities guided by MFCCs as well as the given MR.

6.3.1 Candidate program validation in MTRepair

Because both repairs and the candidate programs constructed during a repair process are program variants of the PUR, their quality can be evaluated using the same measurement used for repair quality. Hereafter, where there is no ambiguity, the expression "*quality of a program*" will refer to the quality of a repair, a candidate program, or a PUR.

According to the earlier formalisations of repair quality (Section 3.1 of Chapter 3), the quality of a program should be measured with respect to the input domain (denoted $\mathscr{D}_P$). This should require an analysis of the program's failure-causing condition (a constraint describing all failing inputs), but because of the difficulties associated with obtaining the failure-causing condition, MTRepair instead approximates it with the MFCC (Section 2.3 of Chapter 2): instead of measuring the quality of a program with respect to its input domain, the quality is measured in terms of *both the input domain and the relevant MR* by making use of MFCCs. This measurement not only enables characterisation of a plausible repair that satisfies the given MR, but also supports comparison of a candidate program's quality with that of its PUR. The MFCC based measurement and MTRepair's candidate program validation strategy are explained in the following.

MFCC based quality measurement

Given a program P and a metamorphic relation MR, $\forall t : t \in \mathscr{D}_P$, an MTG of MR, $(t,\ t')$, can be constructed if MR can be applied to t — where t is used as a source test case and t' is the follow-up test case of t according to MR. As a result, all such MTGs constitute the MTG domain of P for MR, which is denoted $\mathscr{G}_P^{MR}$. If P satisfies MR, then $\mathscr{G}_P^{MR}$ should contain only non-violating MTGs, otherwise, it contains at least one violating MTG. Intuitively, the more non-violating MTGs that $\mathscr{G}_P^{MR}$ has, the higher the quality P should have with respect to MR and $\mathscr{D}_P$. Accordingly, the quality of a program can be measured with respect to an MR, and thus the quality of any two programs implementing the same algorithm can be compared with respect to an MR.

Definition 14 (Repair Quality with Respect to an MR) *If P is a program implementing a given algorithm and MR is a metamorphic relation of that algorithm, then the quality of P, with respect to MR, denoted θ_P^{MR}, is the ratio of the number of non-violating MTGs to the total number of MTGs in $\mathscr{G}_P^{MR}$ (i.e., the non-violating rate of P for $\mathscr{G}_P^{\mathrm{MR}}$).*

Definition 15 *If P_1 and P_2 are two programs implementing the same algorithm, and MR is a metamorphic relation of the algorithm, then:*
(i) P_1 and P_2 are of equal quality, with respect to MR iff $\theta_{P_1}^{MR} = \theta_{P_2}^{MR}$.

(ii) P_1 is of higher quality than P_2, with respect to MR iff $\theta_{P_1}^{MR} > \theta_{P_2}^{MR}$.

As explained in Section 3.1 of Chapter 3, although the pass rate of the entire input domain is the ultimate measurement, this measurement faces the difficult requirement of needing to explore the entire input domain. The same problem arises when applying Definitions 14 and 15. However, MFCCs can alleviate this problem, and hence provide a feasible way for comparing two programs.

As explained in Section 2.3 of Chapter 2, an MFCC characterises a (possibly infinite) set of MTGs for which the relevant MR is violated. When a program violates an MR, multiple MFCCs may exist. The final *MFCC constraint*, which is the disjunction of all MFCCs for the MR, describes all possible violating MTGs for the MR.

If M_p is an MFCC constraint of P and MR, then it corresponds to a set of violating MTGs: $G_{M_p} = \{g | g \text{ satisfies } M_p\}$. $G_{M_p} = \emptyset$ indicates that there are no violating MTGs satisfying M_p, and thus P satisfies MR — in this case, no MFCC is generated for MR and M_p is said to be *null*. If $G_{M_p} \neq \emptyset$, then a smaller G_{M_p} indicates a smaller number of violating MTGs of $\mathscr{G}_P^{MR}$, and thus a higher quality of P, with respect to MR. Based on this, the MFCC based quality measurement can be defined as follows.

Definition 16 *If M_1 and M_2 are two MFCC constraints derived from two different programs using the same MR (and G_{M_2} and G_{M_2} are the corresponding violating MTG sets), then M_2 is said to refine M_1, denoted $M_2 \sqsubset M_1$, iff*

(1) M_1 is not null and M_2 is null; or

(2) Both M_1 and M_2 are not null, and $G_{M_2} \subset G_{M_1}$.

Obviously, *if a candidate program yields no MFCC for a given MR, then it satisfies the MR and thus should be returned as a plausible repair.*

Proposition 1 *Let P_1 and P_2 be two implementations of an algorithm, and MR be a metamorphic relation of that algorithm. Let M_{p_1} and M_{p_2} be MFCC constraints on P_1 and P_2 for MR. If $M_{P_2} \sqsubset M_{P_1}$, then P_2 is of higher quality than P_1, with respect to MR.*

Algorithm 5: MTRepair validates a candidate program against its PUR.

```
1 Function ValidateACandidate(M_p, P_c, MR)
2     M_c = SemiProving(P_c, MR);
3     if M_c = null then
4         return 2;
          /* P_c satisfies MR.                                          */
5     else if M_c ⊏ M_p then
6         return 1;
          /* P_c is of higher quality than its PUR with respect to MR.  */
7     else
8         return 0;
          /* P_c is not of higher quality than its PUR with respect to MR. */
9     end
```

Proof. According to Definition 16, the relationship of $M_{p_2} \sqsubset M_{p_1}$ implies that P_2 has a smaller set of violating MTGs than P_1. Because of this, P_2 should have a higher non-violating rate than P_1 for $\mathscr{G}_P^{MR}$. Therefore, it follows from Definition 15 that P_2 is of higher quality than P_1, with respect to MR. □

Validating a candidate program

MTRepair validates a candidate program to check whether it is of higher quality than its PUR. The validation procedure is implemented by function ValidateACandidate (Algorithm 5), the input to which is the MFCC constraint of the current PUR (M_p), the candidate program (P_c), and the given metamorphic relation (MR). To compare P_c with its PUR, the MFCC constraint for P_c (denoted M_c) is first constructed by applying semi-proving on P_c using MR (line 2). Then, M_p and M_c are compared according to Proposition 1, resulting in one of three possible outcomes: (1) M_c is *null*, suggesting that P_c satisfies MR (line 4); or (2) M_c is not *null* but $M_c \sqsubset M_p$, indicating that P_c does not satisfy MR, but is of higher quality than its PUR, with respect to MR (line 6); or (3) M_c is not of higher quality than its PUR (line 8).

6.3.2 Candidate program construction in MTRepair

MTRepair basically operates on only one statement at one time, and follows the practice of other APR techniques [Nguyen et al., 2013; Nguyen, 2014] of designing repair actions to modify the right-hand side of an assignment statement or the conditional expression

of a predicate statement. This results in every candidate program being constructed by replacing one of the statements of its PUR with a newly created one.

Given a program P, $P_c \leftarrow P[s'/s]$ denotes a candidate program constructed from P by replacing the statement s with the statement s'. The core task of constructing P_c, therefore, is to create a statement s' from s. MTRepair uses information from repair templates, the given MR, and the MFCC constraint of P to synthesise s'. The first step involves creation of a parameterised statement s_t by applying a repair template to s, based on which a program P_t can be constructed: $P_t \leftarrow P[s_t/s]$. The statement s_t has some parameters without specific values, which are declared symbolic in P_t. MTRepair then extracts a *repair constraint*[16] that encodes the requirement on an expected repair according to the given MR, from a series of concolic executions on P_t. The repair constraint describes a condition on the parameters of s_t, the satisfaction of which suggests a way to enable P_t to satisfy the given MR to a certain extent. In other words, if the repair constraint can be successfully constructed and solved, then the result can be used to set the s_t parameters, and thus yield s'. MTRepair currently uses a similar set of templates to those used by Nguyen [2014]: for example, applying the constant template to an assignment statement 'a=b+2' yields a parameterised statement 'a=b+x', where the constant '2' of the original statement is replaced with parameter 'x'.

Next, the construction of a repair constraint is explained. Let M_p denote the MFCC constraint of P for a metamorphic relation *MR*, then, as explained in Chapter 5, a test suite T can be constructed from M_p. A subset set of T (which consists of a set of MTGs) is used to derive a repair constraint. That is, $G = \{g_1, \ ..., \ g_n\}(n \geq 1)$, where each g_i $(1 \leq i \leq n)$ is either a violating MTG (which reveals *MR* violations, and thus suggests functionalities to be rectified), or a non-violating MTG whose execution on P covers s (which relates to the successful executions involving the current s, and thus indicates functionalities that should be retained).

MTRepair aims to extract a repair constraint that describes *the condition under which P_t can satisfy MR on all MTGs of G*. To this end, it concolically executes P_t using MTGs of G. For an MTG g_i, the concolic executions on P_t involve the use of both the source test input t_{s_i} and the follow-up input t_{f_i} (again, it is assumed that *MR* involves two executions) — denoted $P_t(t_{s_i})$ and $P_t(t_{f_i})$, respectively. If $P_t(t_{s_i})$ involves u paths, and $P_t(t_{f_i})$ involves v paths, then c^s_j and o^s_j denote the path condition and the relevant output of the j^{th} path of $P_t(t_{s_i})$ $(1 \leq j \leq u)$, and c^f_k and o^f_k denote the path condition and the relevant output of the k^{th} path of $P_t(t_{f_i})$ $(1 \leq k \leq v)$. The constraint encoding the condition for P_t to satisfy *MR* on g_i is:

[16] This study follows Nguyen et al. [2013] in their use of the term *repair constraint*.

$c_i = \bigvee_{j=1}^{u}(\bigvee_{k=1}^{v}(c_j^s \wedge c_k^f) \wedge ((o_j^s,\ o_k^f)\ \textit{satisfies MR}))$.

The first part of c_i, $(c_j^s \wedge c_k^f)$, indicates that when t_{s_i} exercises the j^{th} path of $P_t(t_{s_i})$, its follow-up test case t_{f_i} exercises the k^{th} path of $P_t(t_{f_i})$. The second part, $(o_j^s,\ o_k^f)$*satisfies MR*, denotes that for these two executions, their outputs satisfy the relationship specified by *MR*. Thus, if c_i can be successfully constructed and solved, then the solution suggests a way to create s' from s_t such that P_c will satisfy *MR* on g_i. Accordingly, the repair constraint specifying satisfaction of *MR* on all MTGs of G is:

$Cond = \bigwedge_{i=1}^{n} c_i$.

If *Cond* can be constructed and solved, then its solution can be applied to s_t to construct P_c. More importantly, such a P_c will satisfy the given MR on all MTGs of G. If it is not possible to construct and solve *Cond*, then this indicates that the current repair action cannot construct a candidate program that makes all MTGs of G non-violating.

The procedure of constructing a candidate program can be illustrated using the program P_{c_1} from Figure 6.1(b) (i.e., P_{c_1} is the currently considered PUR), which was repaired by applying *MR2* of program *median*: $median(x,\ y,\ z) = -median(-x,\ -y,\ -z)$, where x, y, and z are three valid integers. P_{c_1} violates *MR2* and yields the following two MFCCs.

$m_1 = (x < z) \wedge (y < z) \wedge (y - 5 < x < y)$;

$m_2 = (x > z) \wedge (y > z) \wedge (y < x < y + 5)$.

The MFCC constraint for *MR2* is $(m_1 \vee m_2)$, and the MTG set used to synthesise a candidate program contains four MTGs: $g_1 = ((-94,\ -91,\ -48), (94,\ 91,\ 48))$; $g_2 = ((8,\ 6,\ -58), (-8,\ -6,\ 58))$; $g_3 = ((-97,\ -99,\ -64), (97,\ 98,\ 64))$; and $g_4 = ((-64,\ 8,\ -95), (64,\ -8,\ 95))$. Solving m_1 and m_2 gives the two violating MTGs g_1 and g_2, respectively; while g_3 and g_4 are two non-violating MTGs whose executions cover the faulty statement of P_{c_1}.

Suppose the constant template is applied to the faulty statement of P_{c_1}, which means that s is 'if(a<=b-5)', then s_t becomes 'if(a<=b-*arg1*)', then P_t can be constructed from P_{c_1} by replacing s with s_t, and by adding a symbolic declaration for *arg1*. MTRepair then runs P_t on the four MTGs — for example: running P_t with g_1 includes the two concolic executions P_t(-94, -91, -48) and P_t(94, 91, 48). Exploring all paths of these two executions reveals that *MR2* is only satisfied on g_1 for two paths: one for P_t(-94, -91, -48) exercising the trace of ''1-2-3-4-14', and giving the output '-91'; and one for P_t(94, 91, 48) exercising

the trace of ''1-2-8-9-10-14', and giving the output '91'. Therefore, a repair constraint derived from g_1 based on the path conditions of these two paths is:

$$c_1 = (-94 \leq -91 - arg1) \wedge (-91 == -(91)).$$

Similarly, the repair constraints derived from the other three MTGs are:

$$c_2 = (-8 \leq -6 - arg1) \wedge (6 == -(-6)).$$

$$c_3 = (-97 > -99 - arg1) \wedge (-97 == -(97)).$$

$$c_4 = (64 > -8 - arg1) \wedge (-64 == -(64)).$$

Thus, the final repair constraint for synthesising a candidate program from P_t is $Cond = c_1 \wedge c_2 \wedge c_3 \wedge c_4$, the solving of which yields the solution $arg1 = 2$. Using this solution, s_t becomes 'if(a<=b-2)' (s'), and a candidate program is constructed by replacing the s in P_{c_1} with s' (which is P_{c_2} in Figure 6.1(c)).

6.3.3 Repairing a PUR

Although the ultimate goal of MTRepair is to construct a plausible repair that satisfies the given MR, for any currently considered PUR, the immediate target focuses on creation of a successful candidate program.

The procedure for repairing a currently considered PUR is implemented in the function RepairAPUR (Algorithm 6), which is iteratively invoked by the main procedure in MTRepair (line 7 of Algorithm 4). The input to RepairAPUR is the current PUR P, the given metamorphic relation MR, and the MFCC constraint of P (denoted M_p). The first step is to construct a test suite T from M_p, according to the method explained in Chapter 5 (line 2). This function then uses statistical fault localisation and T to construct a list of ranked suspicious statements (line 3). It then iteratively operates on P to construct candidate programs by replacing each of the suspicious statements with a synthesised statement: a candidate program is constructed using information from the given suspicious statement, the selected repair template, P, MR, and M_p, as described in Section 6.3.2 (line 7). Once a candidate program P_c is constructed, it is compared with the PUR P in terms of quality by invoking function EvaluateACandidate (line 9), as detailed in Algorithm 5. If P_c is a successful candidate program of P, the procedure terminates and returns P_c

Algorithm 6: The repairing of a PUR

```
1  Function RepairAPUR( P, MR, M_p)
2      T = GenTSFromMFCCs(MR, M_p);
3      stmt_list = FaultLocalization(P, T);
       /* Applying a statistical fault localization technique.                    */
4      for i ← 1, length(stmt_list) do
5          s = stmt_list[i];
6          for tpl ∈ tpl_list do
               /* tpl_list stores a list of templates.                            */
7              P_c = SynthesizeACandidate(P, M_p, MR, s, tpl);
8              if P_c is not null then
                   /* A candidate program P_c is constructed if the synthesis procedure is successful. */
9                  r = ValidateACandidate(P, P_c, MR);
10                 if r ≥ 1 then
11                     return P_c;
                       /* P_c is of higher quality than P with respect to MR, then MTRepair terminates
                          the repairing of P.                                      */
12                 end
13             end
14         end
15     end
16     return null;
```

to the main procedure (line 11); otherwise, it constructs other candidate programs using different suspicious statements or templates. If all possible candidate programs have been constructed and evaluated to be of lower quality than P, that is, MTRepair fails to repair P, then the function terminates and returns a *null* (line 16).

6.4 Implementation

To support the MTRepair approach, a prototype tool was developed, the architecture of which is shown in Figure 6.3. The tool accepts a faulty C program (the PUR) and an MR as inputs, and continues until a plausible repair satisfying the given MR is generated, or all possible candidate programs have been examined.

To repair a PUR with respect to the given MR, MTRepair first constructs the MFCC constraint for the PUR, using an *MFCC generator* (which is implemented with the symbolic execution engine KLEE [Cadar et al., 2008] and the constraint solver STP [Ganesh and Dill, 2007]). The generated MFCC constraint is then used by a *test case generator* that implements the test suite generation approach described in Chapter 5 to construct a test

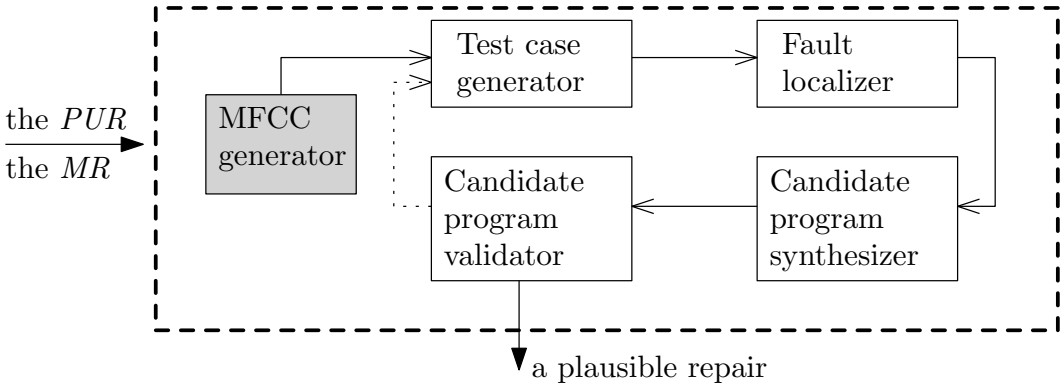


Fig. 6.3 The architecture of MTRepair tool.

suite that is used to locate faults and to synthesise candidate programs. In the current implementation, MTRepair applies the Tarantula technique [Jones and Harrold, 2005] to generate a ranked list of suspicious statements.

The *candidate program synthesizer* implements the methodology presented in Section 6.3.2 to generate candidate programs from the current PUR. It applies KLEE to conduct concolic execution and uses STP to solve the repair constraint. Note that the MFCC generator runs KLEE with symbolic inputs, but the candidate program synthesiser uses concrete inputs.

Once a candidate program is successfully generated, it is validated with the *candidate program validator*, which implements the methodology presented in Section 6.3.1. The validator invokes the MFCC generator to construct MFCCs for the candidate program. The final validation is conducted on the two MFCC constraints M_p and M_c (denoting constraints on the current PUR and one of its candidate programs, respectively) according to Proposition 1: if no MFCC is constructed from the candidate program, then the candidate program is reported as a plausible repair, otherwise, the following mechanism is used to identify the *refine* relationship between M_p and M_c:

If $cond_1$ and $cond_2$ are *satisfiable*, and $cond_3$ is *unsatisfiable*, then $M_c \sqsubset M_p$, where $cond_1$ is $(M_p \wedge M_c)$, $cond_2$ is $(M_p \wedge \neg M_c)$, and $cond_3$ is $(\neg M_p \wedge M_c)$.

In this way, the validator solves the conditions to compare a candidate program with its PUR. If a candidate program is of lower quality than its PUR, then it is ignored; if it satisfies the given MR, then it will be reported as a plausible repair; otherwise, it replaces the current PUR to become the program to be repaired in the next stage of the repair process.

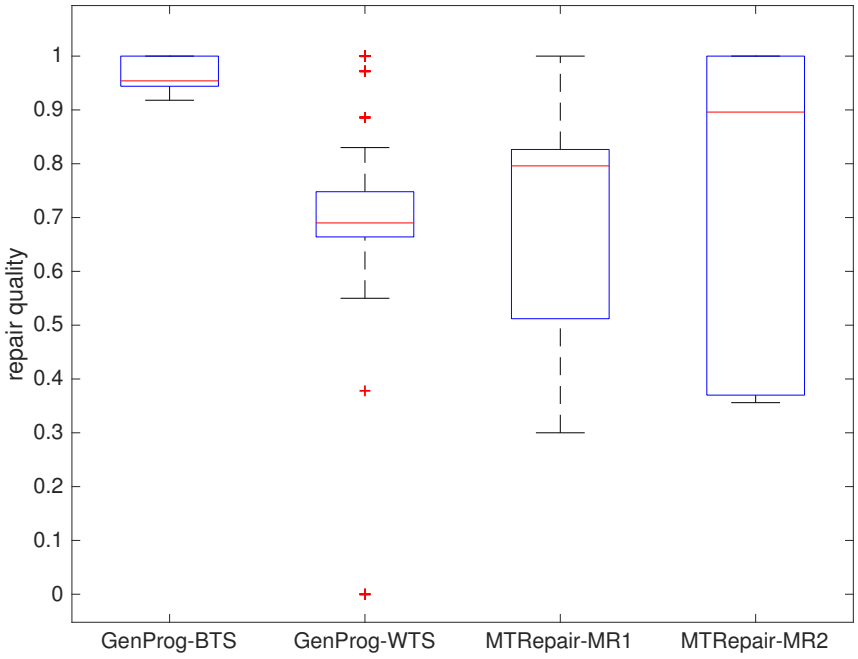
6.5 Evaluation

Currently, the MTRepair approach is implemented into a preliminary prototype to demonstrate the basic repair algorithm. In the evaluation, we selected programs from the IntroClass benchmark suite, which is a standard benchmark for evaluating early-stage new APR methods [Le Goues et al., 2015]. Specifically, the MTRepair approach was evaluated in an empirical study involving two subject programs (*median* and *smallest*) from the IntroClass benchmark suite [Le Goues et al., 2015][17]. The empirical study compared the effectiveness of MTRepair with that of GenProg in terms of the success rates and the repair quality (using the repair effectiveness metrics defined in Section 3.2 of Chapter 3). For a given input test suite, GenProg was run ten times (using different random seeds) on individual faulty programs, but MTRepair was run only once on each program with the given MR.

The success rates for both tools were calculated based on the number of faulty versions that were repaired. The repair quality was evaluated using Definition 3 (Section 3.1 in Chapter 3), and an evaluation test suite containing 500 randomly selected test cases was constructed for each subject program. The statistical techniques used in Chapters 4 and 5 were also used to compare the quality of repairs from the two tools: Wilcoxon rank-sum test [Wilcoxon, 1945] was used to check whether or not the quality of repairs from the two tools was significantly different; and the $\hat{A}_{12}$ statistic [Arcuri and Briand, 2011; Vargha and Delaney, 2000] examined the practical significance of the difference — detailed information about these two statistical techniques can be found in Sections 4.4.1 and 5.4.2 (in Chapter 4 and Chapter 5). Again, in the calculation of $\hat{A}_{12}$ statistic, the data related to GenProg was set as the first group, and that related to MTRepair was set as the second group.

The GenProg experiments used two input test suites, BTS and WTS (the black and white box test suites provided by the IntroClass benchmark suite). The repairs generated by GenProg in the experiments reported in Chapter 4 were reused, but this time the quality of these repairs was evaluated using the new evaluation test suite. The MTRepair experiments reused MRs for the subject programs (as reported in Section 4.3.1 of Chapter 4): MTRepair used two MRs (*MR1* and *MR2*) to repair faulty versions of each subject program. For ease of presentation, GenProg-BTS and GenProg-WTS denote the application of GenProg

[17] The programs *grade* and *syllables* were not included in the study because some of their MRs involve multiple follow-up executions; *digits* was excluded because the output is an array rather than a single value; and *checksum* was excluded because its input and output are of different types. The current implementation of MTRepair assumes that the MR involves only one source and one follow-up execution, and that the input and output of a PUR are of the same data type. It can only handle single output programs. Improving MTRepair to be able to handle a broader range of MRs and programs will be part of the future work.



(a) Distributions of the repair quality

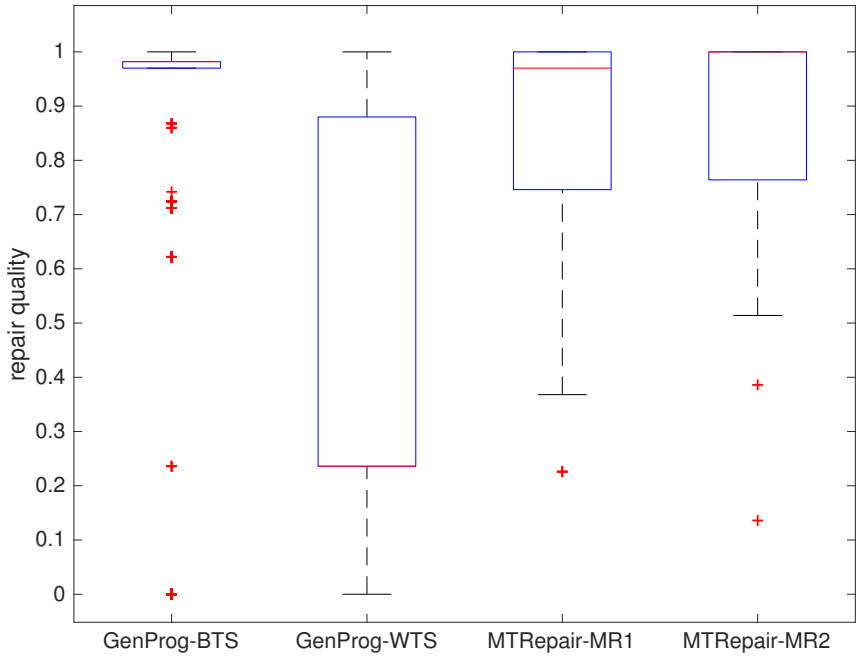
GenProg-BTS vs MTRepair-MR1	GenProg-BTS vs MTRepair-MR2	GenProg-WTS vs MTRepair-MR1	GenProg-WTS vs MTRepair-MR2
p <0.05 $\hat{A}_{12}$ = 0.880 **GenProg-BTS** is *better*.	p <0.05 $\hat{A}_{12}$ = 0.741 **GenProg-BTS** is *better*.	p >0.05 $\hat{A}_{12}$ = 0.510 *Similar*.	p >0.05 $\hat{A}_{12}$ = 0.451 *Similar*.

(b) Statistical analysis results

Fig. 6.4 Repair quality analysis for program *median*

with BTS and WTS, respectively; and MTRepair-MR1 and MTRepair-MR2 denote the application of MTRepair with *MR1* and *MR2*, respectively.

In the experiments on program *median*, the success rates of MTRepair-MR1 and MTRepair-MR2 were 0.993 and 0.474, respectively — those for GenProg-BTS and GenProg-WTS were 0.392 and 0.211, respectively (as reported in Table 4.4 of Chapter 4). The distribution of the quality of repairs constructed for program *median* is displayed in Figure 6.4(a). The statistical analysis on repair quality (Figure 6.4(b)) suggests that MTRepair (both MTRepair-MR1 and MTRepair-MR2) performed less well than GenProg-BTS, but comparably to GenProg-WTS. In summary, compared with GenProg-BTS, both MTRepair-MR1 and MTRepair-MR2 had higher success rates, but yielded repairs of lower quality. On the other hand, both MTRepair-MR1 and MTRepair-MR2 had higher success rates than GenProg-WTS, and they also constructed repairs of comparable quality to those generated by GenProg-WTS.



(a) Distributions of the repair quality

GenProg-BTS vs MTRepair-MR1	GenProg-BTS vs MTRepair-MR2	GenProg-WTS vs MTRepair-MR1	GenProg-WTS vs MTRepair-MR2
$p > 0.05$ $\hat{A}_{12} = 0.477$ *Similar.*	$p < 0.05$ $\hat{A}_{12} = 0.301$ **MTRepair-MR2** is *better.*	$p < 0.05$ $\hat{A}_{12} = 0.167$ **MTRepair-MR1** is *better.*	$p < 0.05$ $\hat{A}_{12} = 0.102$ **MTRepair-MR2** is *better.*

(b) Statistical analysis results

Fig. 6.5 Repair quality analysis for program *smallest*

Figure 6.5(a) shows the distribution of the quality of repairs for program *smallest*. In the experiment, the success rates for MTRepair-MR1 and MTRepair-MR2 were 0.923 and 0.838, respectively — those for GenProg-BTS and GenProg-WTS were 0.779 and 0.966, respectively (as reported in Table 4.4 of Chapter 4). Thus, both MTRepair-MR1 and MTRepair-MR2 had higher success rates than GenProg-BTS. A statistical analysis of the repair quality (Figure 6.5(b)) shows that MTRepair-MR1 was comparable to GenProg-BTS, but that MTRepair-MR2 outperformed GenProg-BTS. Compared with GenProg-WTS, both MTRepair-MR1 and MTRepair-MR2 had lower success rates, but they both yielded repairs of higher quality.

It can be observed that MTRepair had lower success rates only in comparison with GenProg-WTS on program *smallest*, and it had lower quality repairs only in comparison with GenProg-BTS on program *median*: in all other cases, MTRepair had higher success rates and higher or comparable repairs qualities, compared with both GenProg-BTS and

GenProg-WTS. In summary, MTRepair was overall more effective than GenProg for repairing the subject programs.

6.6 Related work

To date, a large amount of APR techniques have been developed, which are usually classified into two groups: the generate-and-validate techniques and the semantics-based techniques. A broad group of generate-and-validate APR techniques are based on iterative searching, where in each iteration, a candidate program is constructed and validated against the input test suite. This procedure is repeated until a candidate program that can pass all test cases of the input test suite is found and reported as a repair, or the stopping criterion is met. A well known method in this category is the Genetic Program Repair ("GenProg") [Le Goues et al., 2012a,b; Weimer et al., 2009], which uses genetic programming to repair programs. GenProg has achieved promising repair results on some real-world programs. Several studies have been conducted to improve GenProg, such as by designing better fitness functions [Fast et al., 2010], selecting better representations for program variants and operators [Le Goues et al., 2012c], and by designing new algorithms for constructing the search space [Weimer et al., 2013]. Inspired by GenProg, Qi et al. [2013, 2014] used a random search in the repair process, and applied test case prioritisation to reduce the repair cost. Seeing that GenProg may generate nonsensical repairs, Pattern-based Automatic program Repair (PAR) was proposed to apply fix patterns learned from human written patches and evolutionary computing to construct the repairs [Kim et al., 2013].

After inspecting the repairs constructed by three APR tools, Qi et al. [2015] found that "[t]he overwhelming majority of the reported patches are not correct and are equivalent to a single modification that simply deletes functionality (on page 24)." They presented an APR tool Kali that generates candidate repairs by only deleting functionality, and found that the repair effectiveness of Kali was at least as good as that of the three investigated APR tools. Long and Rinard [2015] applied a staged program repair (SPR) strategy and a condition synthesis technique to enrich the search space and improve the search efficiency. Prophet [Long and Rinard, 2016] is another approach that uses knowledge learned from previous successful patches to estimate the probability of a candidate repair being correct, and prioritises the candidate repairs in the search space according to their estimated probabilities. Prophet has been shown to produce more high quality repairs than SPR. Another approach by Tan and Roychoudhury [2015], relifix, looks at software regression errors. It takes two versions of the source code as input, together with a test suite

containing at least one failing test case that shows a regression error. The relifix approach employs a random search to iteratively construct and validate the candidate programs. Le et al. [2016b] proposed making use of historical bug fixes from open source projects to assess the fitness of candidate programs during a stochastic search based repair process. This proposed history driven APR technique was reported to be able to produce more good quality repairs than both GenProg and PAR.

Several other generate-and-validate APR methods that construct the set of candidate repairs without using iterative search, have also been developed. ClearView [Perkins et al., 2009], for example, identifies a set of correlated invariants that characterise normal and erroneous executions of the PUR, and constructs a set of candidate repairs that enforce the invariants. The PACHIKA tool [Dallmeier et al., 2009] takes a Java PUR and mines object behaviour models for both passing and failing runs. It then derives the set of candidate repairs by referring to the differences between the passing and failing models, and the candidate repairs are validated against the regression test suite.

In contrast, semantics-based APR techniques directly generate the repair without constructing or validating "candidate repairs." This is done by making use of various program analysis techniques to encode the conditions under which the fault can be repaired with respect to the input test suite, and then synthesising a repair to satisfy these conditions. SemFix [Nguyen et al., 2013] and DirectFix [Mechtaev et al., 2015] are both semantics based program repair methods: SemFix uses symbolic execution, constraint solving, and program synthesis to repair programs; and DirectFix uses constraint solving and component based program synthesis to construct the simplest patch. A comparison between SemFix and DirectFix showed that the DirectFix repairs are simpler and lead to fewer regression errors [Mechtaev et al., 2015]. Based on SemFix and DirectFix, a more recent technique, Angelix [Mechtaev et al., 2016], has been developed and applied to large scale real-world programs. NoPOL [DeMarco et al., 2014] uses satisfiability modulo theory to construct repairs for a class of predicate faults. CETI [Nguyen, 2014] transforms the task of program repair into a program reachability problem, and applies test input generation techniques to search for a solution to the reachability problem, which, if it exists, is then combined with the applied repair template to create a repair. BugFix [Jeffrey et al., 2009] and MintHint [Kaleeswaran et al., 2014] also use program analysis techniques to synthesise repair suggestions.

The majority of the APR techniques described above use an input test suite to encode the intended PUR functionality. Recently, Tan et al. [2016] introduced anti-patterns to generate-and-validate APR techniques in order to alleviate the incomplete specification caused by using the input test suite. A set of anti-patterns capturing disallowed modifications to a

PUR can be applied to any generate-and-validate APR techniques, and can be used in additional to an input test suite. During a repair process, such anti-patterns can help to filter out a candidate program that is incorrect or incomplete but that passes all test cases of the given input test suite. There is, therefore, a higher chance of generating a correct or complete repair (that is, a higher quality repair). These anti-patterns are used together with the input test suites, and should be identified prior to conducting the repair task. Compared with all the above techniques, MTRepair does not require an input test suite, using MRs instead.

Existing APR techniques neither compare a candidate program with its PUR in terms of quality, nor make use of successful candidate programs of higher quality than the PUR. The only study that discusses comparing a candidate program with its PUR was by Diallo et al. [2015, 2016], who used the concept of relative correctness to examine whether or not a candidate program was more correct than its PUR, with respect to the given specification. They also proposed rectifying the given program in a stepwise manner. However, their study focused on discussing and demonstrating the benefits of such a proposal, rather than providing an implementation. Moreover, although the concept of relative correctness appears to reveal similar information to the proposed concept of higher quality, the applications face different challenges.

6.7 Conclusion

In spite of the advances in APR, many challenges still remain. In this chapter, an APR approach for alleviating some of these challenges has been proposed. The approach, called MTRepair, was designed and implemented by based on the strengths of MT and MFCCs. It uses an MR instead of an input test suite as input, with a plausible repair (if successfully generated) satisfying the given MR. One of the innovations of MTRepair is in its use of an MFCC based measurement, which supports a candidate program validation procedure that compares a candidate program against its PUR, in terms of their quality. This validation procedure enables the exploration of successful candidate programs (that is, candidate programs that are of higher quality than the PUR), and also forms the basis for an incremental repair process. The effectiveness of MTRepair has been demonstrated through experiments on some subject programs of the IntroClass benchmark suite, with MTRepair exhibiting higher effectiveness than GenProg.

It is important to emphasise that the effectiveness of MTRepair is highly related to the effectiveness of the MR used. Therefore, similar to the challenge of selecting better

input test suites for test suite based APR, one challenge for MTRepair is the selection of effective MRs. While this study demonstrated only basic components of MTRepair, the MTRepair approach can be further improved and optimised by designing new strategies towards some of its components. On the other hand, to increase its practical impact, more empirical studies should be conducted using MTRepair to further examine its applicability and scalability.

Chapter 7

Conclusion

7.1 Summary

Although *test suite based automated program repair* (referred to as test suite based APR, or simply APR) has made significant progress in the past decade [DeMarco et al., 2014; Fast et al., 2010; Kim et al., 2013; Le Goues et al., 2012b,c,c; Mechtaev et al., 2015, 2016; Nguyen et al., 2013; Nguyen, 2014; Qi et al., 2014; Weimer et al., 2013, 2009], many challenges remain. This thesis has focused on some of these challenging APR problems, aiming to extend the scope of APR applicability and improving its repair effectiveness.

The first contribution of this thesis is *a series of formalisations of APR concepts.* These formalisations contribute to the characterisation of test suite based APR as well as the impact of input test suites on APR. Based on the formalisations, important evaluation metrics have been developed to enable the systematic evaluation of the repair effectiveness of APR techniques, and measurement of the effectiveness of APR input test suites.

The second contribution of this thesis is *a strategy enabling the application of APR techniques without the need for a test oracle.* This strategy is designed to alleviate the test oracle problem of APR in order to extend its scope of applicability. It integrates *metamorphic testing* (MT) with test suite based APR, using a set of *metamorphic test groups* (MTGs) as a substitute for an input test suite. Because of the characteristics of MT, the integrated techniques (referred to as APR-MT techniques) use MT checking mechanisms to determine the outcome of individual MTGs rather than a test oracle. A general framework to support the integration of MT and test suite based APR was developed,

yielding APR-MT techniques that no longer rely on a test oracle. Based on this framework, two APR-MT techniques, GenProg-MT and CETI-MT, were implemented by combining MT with the APR techniques GenProg and CETI, respectively. The effectiveness of the APR-MT techniques was investigated through an empirical study comparing APR-MT with conventional APR, using the IntroClass benchmark suite [Le Goues et al., 2015]. The empirical results showed both GenProg-MT and CETI-MT performing comparably to their corresponding APR techniques in terms of repair effectiveness: in other words, application of MT to test suite based APR not only alleviates the test oracle problem, but also delivers effective repair results. Therefore, the proposed strategy successfully extends the scope of applicability for test suite based APR techniques.

The third contribution of this thesis is *a novel input test suite generation approach for APR* that aims to enhance the APR repair effectiveness by using effective input test suites. This study started with an analytical investigation into the impact of input test suites on the repair effectiveness of APR techniques, which led to identification of the need for systematic input test suite generation approaches for APR. Based on this, a novel input test suite generation approach for APR was developed, making use of information from violated *metamorphic relations* (MRs) and *metamorphic failure-causing conditions* (MFCCs). An empirical study was conducted to compare the proposed approach with random and code coverage based test suite generation approaches, showing the superiority of the proposed approach in terms of the repair effectiveness. Further experimental analysis provided insights into the interplay between the input test suite construction and the relevant APR techniques, concluding that the effectiveness of an input test suite generation approach cannot be discussed without reference to the APR technique used.

The fourth contribution of this thesis is *a proposal for an MT based APR approach*, which was motivated by some challenging problems and issues faced by current APR techniques, including: the incomplete description of the intended functionality of the program under repair; the need for a more reliable validation of candidate programs; and the benefits of making use of a candidate program that is of higher quality than the program under repair. A novel semantics-based APR approach that makes use of the strengths of MT and MFCCs, referred to as MTRepair, has been developed. MTRepair has several distinct characteristics, including the use of MRs as input, the MFCC based validation procedure, and the incremental repair process. The repair effectiveness of MTRepair was demonstrated through experiments on some subject programs of the IntroClass benchmark suite [Le Goues et al., 2015].

Being based upon MT, the strategies and approaches presented in this thesis also face the challenge of identifying appropriate MRs, which is a fundamental challenge for

all applications of MT. However, much effort has been devoted to this issue, providing some useful guidelines for identifying effective MRs [Cao et al., 2013; Liu et al., 2014; Mayer and Guderlei, 2006]. Furthermore, there have recently been some approaches to systematically generating MRs [Chen, 2015; Liu et al., 2012; Zhang et al., 2014], some of which are even supported by automated tools. Any advances in the identification of effective MRs will also make the proposed approaches more effective and efficient.

In addition to the issue of effective MR identification, the proposed test suite generation approach and MTRepair approach are also influenced by the performance of the symbolic execution technique used. To enhance the capability of symbolic execution techniques, several strategies have been developed to process complex path conditions and to handle the path explosion problem [Boonstoppel et al., 2008; Păsăreanu and Visser, 2009]. Accordingly, an increasing number of symbolic execution engines have been developed [Cadar et al., 2008; Cadar and Sen, 2013; Godefroid et al., 2005; Sen et al., 2005]. The efficiency of the proposed approaches can be enhanced by applying more powerful symbolic execution engines. Furthermore, any limitations caused by symbolic executions may also be alleviated by developing new approaches to generate MFCCs.

7.2 Future work

The research presented in this thesis can be further explored in several directions.

Other applications of the proposed approaches. The investigation of the APR-MT technique applied MT to two APR techniques (CETI and GenProg), and the study of input test suite construction involved application to three APR techniques (Angelix, CETI and GenProg). GenProg is a generate-and-validate APR technique, but Angelix and CETI are semantics based. Due to the large variety of APR techniques, it will worthwhile to apply these two studies to a wider range of APR techniques to study their effectiveness. Furthermore, APR-MT, the input test suite construction approach, and MTRepair were all evaluated using selected subject programs. Although these subject programs have been commonly used in APR research, it is still important to *investigate the scalability of the approaches by applying them to more large-scale programs*. These follow-up studies will consolidate the strength of the proposed approaches, and also increase their practical impact.

Analysis of the impact of MRs on the proposed approaches. All of the proposed approaches are driven by MT, and therefore, as with all other MT based applications, MRs

play a critical role whose impact should be thoroughly investigated. An important future study will be to analyse the effectiveness of different MRs in order to derive guidelines for identifying effective MRs for the approaches. Furthermore, although the proposed approaches can support application of multiple MRs, the current studies only applied one MR at one time. It will therefore be necessary to investigate the application of multiple MRs, and analyse the degree to which diverse MRs can impact on the approaches. These studies will provide useful insights and guidelines for the practical application of the approaches.

Investigation of strategies to enhance the proposed approaches. The input test suite generation approach and the MTRepair approach make use of MFCCs, which are constraints encoding the MTGs that violate the relevant MR. In the presented studies, the MFCCs were constructed using semi-proving. Since semi-proving is built upon symbolic execution, the effectiveness and efficiency of the approaches are basically restricted by the strength of the underlying symbolic engines. However, because of the nature of MFCCs, it should be possible to find alternative construction methods. A future research direction, therefore, will be to explore the possibility of using other techniques to construct MFCCs, thereby enabling different implementations of the test suite generation approach and the MTRepair approach. A more effective and efficient MFCC generation method should increase the effectiveness of both the input test suite construction approach and the MTRepair approach. Moreover, studies focusing on this topic will increase the flexibility of the approaches, such that different appropriate implementations can be used for different application contexts. On the other hand, by its nature, MTRepair terminates with a successful candidate program (a program that is of higher quality than the current PUR) in the procedure of repairing an intermediate PUR, and then continues to repair such a successful candidate program if it is not a plausible repair. Obviously, the identification of a successful candidate program is crucial to the final outcome of MTRepair, and an identification of an inappropriate successful candidate program may lead to no repair, or a repair resulted from a lengthy sequence of repair actions. More advanced strategies assisting the identification of successful as well as appropriate candidate programs can improve the repair effectiveness of MTRepair.

References

Ammann, P. E. and Knight, J. C. (1988). Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425.

Arcuri, A. (2011). Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514.

Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 1–10.

Arcuri, A. and Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 162–168.

Artzi, S., Dolby, J., Tip, F., and Pistoia, M. (2010). Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, pages 49–60.

Assiri, F. Y. and Bieman, J. M. (2014). An assessment of the quality of automated program operator repair. In *Proceedings of the 7th International Conference on Software Testing, Verification, and Validation (ICST'14)*, pages 273–282.

Assiri, F. Y. and Bieman, J. M. (2016). Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal, DOI 10.1007/s11219-016-9312-z*.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software tesing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Barus, A. C. (2010). *An in-depth study of adaptive random testing for testing program with complex input types*. PhD thesis, Swinburne University of Technology.

Barus, A. C., Chen, T. Y., Grant, D., Kuo, F.-C., and Lau, M. F. (2011). Testing of heuristic methods: A case study of greedy algorithm. In *Proceedings of the Third IFIP TC 2 Central and East European Conference on Software Engineering Techniques (CEE-SET'08)*, pages 246–260.

Boonstoppel, P., Cadar, C., and Engler, D. (2008). Rwset: Attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and Practice of*

Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08), pages 351–366.

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

Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, pages 209–224.

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

Campos, J., Abreu, R., Fraser, G., and Amorim, M. d. (2013). Entropy-based test generation for improved fault localization. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE'13)*, pages 257–267.

Cao, Y., Zhou, Z. Q., and Chen, T. Y. (2013). On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*, pages 153–162.

Ceccato, M., Marchetto, A., Mariani, L., Nguyen, C. D., and Tonella, P. (2015). Do automatically generated test cases make debugging easier? An experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology*, 25(1):5:1–5:38.

Chan, W. K., Cheung, S. C., and Leung, K. R. P. H. (2005). Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the Fifth International Conference on Quality Software (QSIC'05)*, pages 470–476.

Chen, T. Y. (2015). Metamorphic testing: A simple method for alleviating the test oracle problem. In *Proceedings of the 10th International Workshop on Automation of Software Test (AST'15)*, pages 53–54.

Chen, T. Y., Cheung, S. C., and Yiu, S. M. (1998). Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

Chen, T. Y., Ho, J. W. K., , Liu, H., and Xie, X. Y. (2009). An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics*, 10(1):24–35.

Chen, T. Y., Kuo, F.-C., Ma, W., Susilo, W., Towey, D., Voas, J., and Zhou, Z. Q. (2016). Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55.

Chen, T. Y., Kuo, F.-C., and Zhou, Z. Q. (2005). An effective testing method for end-user programmers. In *Proceedings of the ICSE Workshop on End-User Software Engineering (WEUSE I)*, pages 21–25.

Chen, T. Y., Sun, C.-a., Wang, G., Mu, B., Liu, H., and Wang, Z. (2012). A metamorphic relation-based approach to testing web services without oracles. *International Journal of Web Services Research*, 9(1):51–73.

Chen, T. Y., Tse, T. H., and Zhou, Z. Q. (2001). Fault-based testing in the absence of an oracle. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 172–178.

Chen, T. Y., Tse, T. H., and Zhou, Z. Q. (2003). Fault-based testing without the need of test oracles. *Information and Software Technology*, 45(1):1–9.

Chen, T. Y., Tse, T. H., and Zhou, Z. Q. (2011). Semi-proving: An integrated method for program proving, testing and debugging. *IEEE Transactions on Software Engineering*, 37(1):109–125.

Dallmeier, V., Zeller, A., and Meyer, B. (2009). Generating fixes from object behavior anomalies. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09)*, pages 550–554.

DeMarco, F., Xuan, J., Le Berre, D., and Monperrus, M. (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'14)*, pages 30–39.

Diallo, N., Ghardallou, W., and Mili, A. (2015). Correctness and relative correctness. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 591–594.

Diallo, N., Ghardallou, W., and Mili, A. (2016). Program repair by stepwise correctness enhancement. In *Proceedings First Workshop on Pre- and Post-Deployment Verification Techniques*, pages 1–15.

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

Fast, E., Le Goues, C., Forrest, S., and Weimer, W. (2010). Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*, pages 965–972.

Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*, pages 947–954.

Fry, Z. P., Landau, B., and Weimer, W. (2012). A human study of patch maintainability. In *Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 177–187.

Ganesh, V. and Dill, D. L. (2007). A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, pages 519–531.

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

Gopinath, D., Malik, M. Z., and Khurshid, S. (2011). Specification-based program repair using SAT. In *Proceedings of the 17th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, pages 173–188.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*, pages 52–61.

Jeffrey, D., Feng, M., Gupta, N., and Gupta, R. (2009). BugFix: A learning-based tool to assist developers in fixing bugs. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC'09)*, pages 70–79.

Jiang, M., Chen, T. Y., Kuo, F.-C., Zhou, Z. Q., and Ding, Z. (2014). Testing model transformation programs using metamorphic testing. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14)*, pages 94–99.

Jobstmann, B., Griesmayer, A., and Bloem, R. (2005). Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, pages 226–238.

Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of 20th International Conference on Automated Software Engineering (ASE'05)*, pages 273–282.

Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. (2014). Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 266–276.

Kanewala, U. and Bieman, J. M. (2013). Techniques for testing scientific programs without an oracle. In *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE'13)*, pages 48–57.

Ke, Y., Stolee, K. T., Le Goues, C., and Brun, Y. (2015). Repairing programs with semantic code search. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, pages 295–306.

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 802–811.

Kuo, F.-C., Zhou, Z. Q., Ma, J., and Zhang, G. (2010). Metamorphic testing of decision support systems: A case study. *IET Software*, 4(4):294–301.

Le, V., Afshari, M., and Su, Z. (2014). Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, pages 216–226.

Le, X.-B. D., Lo, D., and Le Goues, C. (2016a). Empirical study on synthesis engines for semantics-based program repair. In *Proceedings of the IEEE 32nd International Conference on Software Maintenance and Evolution (ICSME'16)*.

Le, X.-B. D., Lo, D., and Le Goues, C. (2016b). History driven program repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 213–224.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012a). 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)*, pages 3–13.

Le Goues, C., Forrest, S., and Weimer, W. (2013). Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443.

Le Goues, C., Holtschule, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transaction on Software Engineering*, 41(12):1236–1256.

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

Le Goues, C., Weimer, W., and Forrest, S. (2012c). Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO'12)*, pages 959–966.

Lindvall, M., Ganesan, D., Árdal, R., and Wiegand, R. E. (2015). Metamorphic model-based testing applied on NASA DAT: An experience report. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 129–138.

Liu, C., Yan, X., and Han, J. (2006). Mining control flow abnormality for logic error isolation. In *Proceedings of the 6th SIAM International Conference on Data Mining (SDM'06)*, pages 106–117.

Liu, H., Liu, X., and Chen, T. Y. (2012). A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software (QSIC'12)*, pages 59–68.

Liu, L., Kuo, F.-C., Towey, D., and Chen, T. Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22.

Long, F. and Rinard, M. (2015). Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pages 166–178.

Long, F. and Rinard, M. (2016). Prophet: Automatic patch generation via learning from successful patches. In *Proceedings of the symposium on Principles of Programming Languages*.

Malik, M. Z., Ghori, K., Elkarablieh, B., and Khurshid, S. (2009). A case for automated debugging using data structure repair. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, pages 620–624.

Mayer, J. and Guderlei, R. (2006). An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 475–484.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). DirectFix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 448–458.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 691–701.

Monperrus, M. (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'14)*, pages 234–242.

Murphy, C., Shen, K., and Kaiser, G. (2009). Automatic system testing of programs without test oracles. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 189–200.

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 772–781.

Nguyen, T. (2014). *Automating program verification and repair using invariant analysis and test input generation*. PhD thesis, University of New Mexico.

Núñez, A. and Hierons, R. M. (2015). A methodology for validating cloud models using metamorphic testing. *Annals of Telecommunications*, 70(3):127–135.

Oliveira, V. P. L., Souza, E. F. D., Le Goues, C., and Camilo-Junior, C. G. (2016). Improved crossover operators for genetic programming for program repair. In *Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE)*, pages 112—127.

Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., and Zeller, A. (2014). Automated fixing of programs with contracts. *IEEE Transaction on Software Engineering*, 40(5):427–449.

Pei, Y., Wei, Y., Furia, C. A., Nordio, M., and Meyer, B. (2011). Code-based automated program fixing. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pages 392–395.

Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. (2009). Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 87–102.

Păsăreanu, C. S. and Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353.

Qi, Y., Mao, X., and Lei, Y. (2013). Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM'13)*, pages 180–189.

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 254–265.

Qi, Z., Long, F., Achour, S., and Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA'15)*, pages 24–36.

Röβler, J., Fraser, G., Zeller, A., and Orso, A. (2012). Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 309–319.

Segura, S., Durán, A., Sánchez, A. B., Berre, D. L., Lonca, E., and Ruiz-Cortés, A. (2015). Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163.

Segura, S., Fraser, G., Sanchez, A. B., and Ruiz-Cortés, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824.

Sen, K., Marinov, D., and Agha, G. (2005). CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272.

SIR (2005). http://sir.unl.edu.

Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. (2015). Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pages 532–543.

Stutzke, M. A. and Smidts, C. S. (2001). A stochastic model of fault introduction and removal during software development. *IEEE Transactions on Reliability*, 50(2):184–193.

Tan, S. H. and Roychoudhury, A. (2015). relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 471–482.

Tan, S. H., Yoshida, H., Prasad, M. R., and Roychoudhury, A. (2016). Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pages 727–738.

Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. (2010). Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, pages 61–72.

Weimer, W., Fry, Z. P., and Forrest, S. (2013). Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE'13)*, pages 356–366.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 364–374.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83.

Xie, X., Ho, J. W. K., Murphy, C., Kaiser, G., Xu, B. W., and Chen, T. Y. (2011). Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558.

Xie, X., Wong, W. E., Chen, T. Y., and Xu, B. W. (2013). Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879.

Zhang, J., Chen, J., Hao, D., Xiong, Y., Xie, B., Zhang, L., and Mei, H. (2014). Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 701–712.

Zhou, Z. Q., Xiang, S., and Chen, T. Y. (2016). Metamorphic testing for software quality assessment: A study of search engines. *IEEE Transactions on Software Engineering*, 42(3):264–284.

Zhou, Z. Q., Zhang, S., Hagenbuchner, M., Tse, T. H., Kuo, F.-C., and Chen, T. Y. (2012). Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243.

List of the author's publications

Jiang, M., Chen, T. Y., Kuo, F.-C., Ding, Z., Choi, E.-H., and Mizun, O. (2017a). A revisit of the integration of metamorphic testing and test suite based automated program repair. In *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET '17), accepted*.

Jiang, M., Chen, T. Y., Kuo, F.-C., Towey, D., and Ding, Z. (2017b). A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software*, 126:127–140.

Jiang, M., Chen, T. Y., Kuo, F.-C., Zhou, Z. Q., and Ding, Z. (2014). Testing model transformation programs using metamorphic testing. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE '14)*, pages 94–99.

Jiang, M., Chen, T. Y., Kuo, F.-C., Zhou, Z. Q., and Ding, Z. (2017c). Input test suites for program repair: Formalization and a novel generation method. *Submitted to ACM Transactions on Software Engineering and Methodology*.