Researches on Automatic Techniques for Specification-Based Testing and Fault Localization

WANG, Rong / 王, 榕

(開始ページ / Start Page)
1
(終了ページ / End Page)
143
(発行年 / Year)
2022-03-24
(学位授与番号 / Degree Number)
32675甲第547号
(学位授与年月日 / Date of Granted)
2022-03-24
(学位名 / Degree Name)
博士(理学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)
(URL)
https://doi.org/10.15002/00025230

DOCTORAL DISSERTATION REVIEWED BY

HOSEI UNIVERSITY

Researches on Automatic Techniques for Specification-Based Testing and Fault Localization

---

Rong WANG

Research Supervisor:
Prof. Yuji SATO

Thesis Supervisor:
Prof. Hiroshi HOSOBE

The Graduate School of Computer and Information Sciences

This dissertation is submitted for the degree of *Doctor of Philosophy*

Abstract

The existing specification-based techniques (SBT) has difficulty in generating an appropriate test suite without the knowledge of code structure to trigger different kinds of unintended behaviours hidden in programs. Symbolic execution, a powerful technique for automating software testing, instead takes advantage of internal code design to detect many types of errors like out of memory and assertion violations. However, it can encounter severe path explosion problem during the exhausted test data generation. Besides, by only using assertions, it may ignore some faulty paths due to not going deep into checking the functional correctness of a path. To address these problems, this research proposes a specification-based incremental testing method with symbolic execution, called SIT-SE, to provide a much more rigorous way to automatically check the functional correctness of all the discovered program paths within limited time. In this method, we introduce theorems instead of assertions for checking path correctness, and describe a Branch Sequence Coverage (BSC) algorithm together with checking levels for guiding a moderate path exploration. The proposed method carefully treats the relationship between a path condition and the specification in a theorem to reduce the monotonous path exploration, whereas traditional symbolic testing methods use assertions that are not sufficient to judge the correctness of a path during the long tedious path exploration. Moreover, we present a strategy of fault localization called TRIACFL with the support of the SIT-SE to give useful hints to pinpoint the faults in a small set of statements. To enrich our methodology of testing used in practice, we also describe a test data generation method that integrates formal specification with a genetic algorithm as supplementary to the SIT-SE for dealing with exceptional cases where some code is not available to testers. We conduct two experiments with the proposed methods, and the results demonstrate that these methods together facilitate an effective automatic bug detection.

There are three main contributions in our work. Firstly, we propose a method, SIT-SE, to provide a systematic way to automatically verify the correctness of all the representative program paths by integrating symbolic execution and formal specification. Secondly, we present a fault localization method with the SIT-SE, namely TRIACFL, to provide useful clues for the locations of real faults within a small scale of statements in programs. Thirdly, a test data generation method using the formal specification and a genetic algorithm (GA), is proposed to cope with the situations where the SIT-SE is not applicable.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my co-advisor Prof. Shaoying Liu for the continuous support during my Ph.D. study and research. His patience, enthusiasm and guidance helped me in all the time of research and writing of this dissertation. I would also like to thank my research supervisor Prof. Yuji Sato and my thesis supervisor Prof. Hiroshi Hosobe, for supporting me administratively and technically. Without their guidance and persistent help this dissertation would not have been possible.

Besides my advisors, I would like to thank Prof. Jianhua Ma and all the other professors of CIS who have provided me with useful suggestions for my research work.

I am also very grateful to the members in the lab, who enrich my daily life and give me kind assistance during my research period.

Last but not least, I wish to thank my parents. They have supported me unconditionally and financially. Thanks for all of your support!

Contents

List of Figures

List of Tables

Nomenclature

SE	Symbolic Excution
SBT	Specification-Based Testing
SBFL	Spectrum-Based Fault Localization
SOFL	Structured Object-Oriented Formal Language
CDFD	Condition Data Flow Diagram
SPF	Symbolic Java PathFinder
JPF	Java PathFinder
CFG	Control Flow Graph
BSC	Branch Sequence Coverage
SIT-SE	Specification-Based Incremental Testing Method with Symbolic Execution
TRIACFL	Triple Interaction-Based Fault Localization
SBT	Specification-Based Testing
GA	Genetic Algorithm
FSF	Functional Scenario Form

Chapter 1

Introduction

1.1 Software Quality Assurance

Software quality assurance is realized mostly by software verification and testing. This chapter discusses the quality assurance issue by focusing on software verification and testing, respectively.

1.1.1 Software Verification

Formal methods are established based on a variety of theoretical computer science fundamentals, including formal languages, logic calculi, automata theory, program semantics, type systems, and discrete event dynamic system. They are a particular kind of mathematically rigorous techniques frequently involving with the specification, development and verification of software systems.

Formal methods inspire software engineers to develop domain models and requirements specifications that are more rigorous and unambiguous than other conventional engineering methods. In formal methods, program refinement further helps the engineers transform an abstract formal specification (or domain model) into a concrete implementation (executable program). Moreover, for software verification, the use of formal specifications significantly strengthen the proof for the correctness of an implementation.

With the rapid deployment of safety-critical systems in many application domains [1], there is a growing concern about the quality of software adopted for controlling and/or monitoring of the systems [2]. Formal verification techniques, such as model checking [3, 4], and

theorem proving [5, 6], are developed to prove the correctness of a system against formal specifications based on mathematical methodologies. However, these techniques demand for a lot of efforts made by well-trained experts to build and analyze an abstract mathematical model that carefully describes the properties or the behavior of the system. Due to their intrinsic feature of abstraction, there always exists a gap between a model and an implementation under a complicated execution environment. Thus they cannot be directly used in testing real programs. Furthermore, formal methods have to deal with different problems in larger projects, such as the problem of state space explosion in model checking [7, 8], and the difficulty in automating tedious intricate proofs in theorem proving [9, 10].

1.1.2 Automatic Testing and Fault Localization

Despite that formal methods demonstrate their power in analyzing a system by taking advantage of mathematics, they are popular and widely researched in academia rather than in industry. Currently, as software development tends to be more of complexity and needs high-speed iterations, it is almost impossible to employ formal methods that call for huge manual effort for software verification in real world. Instead, automation of testing real programs has been seen as a potential solution to ensure the quality of software with moderate human labors [11].

On the one hand, testing automation techniques can be flexibly developed and applied in various sophisticated environments to detect various kinds of bugs; on the other hand, all the testing methods, intrinsically, cannot tell the absence of bugs even based on numerous executions of programs. To enhance the confidence of software quality, it is common to employ some exhausted path exploration strategies [12] or test selection techniques [13] in testing. Moreover, many techniques have been developed to involve formal methodologies. There are two main ways of such involvement: one is to derive tests from formal specifications [14, 15, 16, 17], and the other is to verify the correctness of executions based on formal specifications [18, 19]. These efforts together improve the testing for bug detection.

Furthermore, fault localization is an activity to pinpoint the causes by analyzing the false

executions identified in testing, which guides developers to fix bugs in right places. This activity usually demands for much human intervention, which is tedious, expensive and time-consuming [20]. Although a great many methods have been proposed to automatically produce a list of suspicious locations in the code, the precision of outcome is still quite a critical issue in practice.

1.2 Motivation

1.2.1 Path Explosion and False Paths Identification Problems

Symbolic execution-based method is one of the attractive techniques for detecting bugs by automatically searching various paths without predefined inputs. For every program path, symbolic execution summarizes the representation for the outcome over symbolic inputs, and continuously explores unchecked paths based on the solutions provided by a satisfiability modulo theories (SMT) solver that consumes the collected path constraints (or path condition) [21, 22]. Furthermore, specification-based symbolic execution, a technique that integrates formal methods into the correctness of implementations, makes use of such representations of the outcome to check whether or not they violate the assertions made from formal specifications [23, 24].

Although the intention of symbolic execution is to exhaustedly find as many paths as possible so that it can increasingly hit bugs, path explosion remains a severe problem and actually the correctness of each path is not well guaranteed. By using assertions, a path can be recognized as faulty if a test data that traverses that path violates the assertions. However, a faulty path cannot be recognized when the test data that traverses it satisfies the assertions. This is because among all the input values on the same symbolic path, some of them may violate the corresponding assertions and others not. The test data generated to introduce the faulty path may not violate the assertions. Thus more test data have to be generated to traverse the same path to trigger the violation of the assertions. For instance, suppose a program reaches the assertion $assert(x > 10)$ when the integer input x satisfies the path condition $x > 8$. The path with the path condition $x > 8$ will be seen as correct one for most cases except when

$x = 9$. Due to the intrinsic limitation of these assertions, many faulty paths may be mistakenly treated as correct ones and ignored for further testing.

To address these problems, we propose a specification-based incremental testing method with symbolic execution, called *SIT-SE*, providing a more advanced automatic verification for the correctness of each program path and in the meantime mitigate the heavy burden of path exploration. Our method mainly makes several improvements on the KLEE tool [23], a well-known symbolic execution engine which has been continually researched and updated all these years [1]. Compared to other testing methods that inevitably involve a great amount of effort for test design or static analysis, KLEE is able to systematically (exhaustedly) explore many possible program paths without requiring specific concrete input values during dynamic symbolic execution. We aspire to develop a method that takes advantage of KLEE meanwhile enhances its capability in bug detection. Since KLEE always adds the conditions from assertions into path constraints, a program path can be executed for several times in order to reach different branches of the same assertions. Whereas in the proposed method, we introduce *theorems* instead of assertions for checking the correctness of each path, and design an algorithm called *Branch Sequence Coverage* (BSC) algorithm for a moderate incremental path exploration.

1.2.2 Low Precision and High Costs in Fault Localization

Although many decades have witnessed great efforts that have been made to develop fault localization techniques, it is still not an easy job in most cases to precisely pinpoint the faults without human intervention [25]. In most ranking-based automated approaches, a long sequence of suspicious locations is first generated and then the reviewer comes up to inspect these locations. The costs of human inspection increase when there are many non-faulty candidates with the same or higher suspiciousness scores than that of actual faulty locations. This is a critical problem in existing techniques, especially when we deal with multiple-fault programs [26]. To mitigate the problem, we are motivated to propose a method, TRIACFL (Triple Interaction-Based Fault Localization), that mixes the automatic processes of both suspicious location generation and the human inspection. In this approach, the sequence of suspicious locations is dynamically changed regarding to the occasional lightweight human inspection,

[1] https://klee.github.io/publications/

which leads to a more effective fault localization within moderate costs of human inspection of suspicious locations.

Additionally, as the SIT-SE provides a rigorous way to automatically testing an implementation by verifying the correctness of all the "representative" paths on a moderate-scale generated test suite, we are motivated to integrate the SIT-SE into our approach to further reduce execution costs and improve the effectiveness of fault localization.

1.2.3 Problems in Test Selection for Unavailable Source Code

When it comes to testing a program containing some operations whose source code is unavailable, the SIT-SE may become less effective in this case because the symbolic execution cannot reach third-party libraries. We are motivated to develop a specification-based testing (SBT) method for test data generation as an supplement to the SIT-SE.

It is not easy for a pure SBT to generate various test data only from specifications to detect various bugs that are contained in the program. This is because different features and effects of the program output cannot be controlled and triggered by only input data suites that satisfy some portion of the specification (constraints over only input variables). Consequently, many faulty program paths would not be detected and thus the bug detection would be likely to fail in many cases. Most existing SBT methods like in [27, 28] only use original formal specifications as a pass/fail indicator of executions as well as for test data generation. The input generation there only takes into account the constraints over input variables in formal specifications without using the constraints over outputs.

To overcome the shortcomings of the existing SBT methods, we propose a new method that do mutation on original formal specifications for further test data generation in this research. The proposed method introduces dummy variables into some specified constraints in the specification to trigger bad behaviours of the program, and makes use of the constraints over both input and output variables to guide the test data generation, in contrast to the conventional SBT methods that concerns only constraints over input variables.

This method features the combination of three techniques, SBT, mutation testing, and genetic algorithm (GA). It is to obtain the enhanced (mutated) formal specifications without knowing the code structures by using a GA so that input data generated from those mutated

specifications are more likely to kill different kinds of mutants of the target program under test. The expected effect of the test data generated in this way is to detect various bugs probably occurring in the program that is being developed or optimized.

1.3 Contributions

We take great interested in finding functional bugs that distort the intention of formal specification and thus may cause unexpected behaviors in programs. Generally, it is hard to generate concrete input data to trigger the explicit violation for the output against the related formal specification, since this kind of input values may be the minority along a faulty path. In this research, we put forward a method with an advanced automatic analysis for each problematic execution that can be derived from any input values without the effort of test design. This method concentrates on effectively identifying false paths that may contain any incorrect arithmetical conditions or assignments. Also, we propose a method for further fault localization based on the discovered false paths and their summaries. Finally, we conduct experiments, targeting at procedural programs mostly, for evaluation of various methods. The experimental results demonstrate that the proposals outperform the baseline methods.

In detail, this research contributes to three aspects of software quality assurance as follows.

1.3.1 Bug Detection

We propose a method, called Specification-Based Incremental Testing Method with Symbolic Execution (SIT-SE), for verifying the correctness of program paths in a more rigorous way. By rigorous here we mean that the SIT-SE follows a well-defined process to carry out a testing that can achieve the effect the traditional SBT may not obtain in finding bugs. Compared with traditional specification-based testing (SBT) technologies, the SIT-SE is able to detect bugs in a more effective way with less effort. This advantage is reflected in three respects. Firstly, the SIT-SE can find all the bugs on a path by only executing it once while the traditional SBT may need to run more than one time to find them. Secondly, the SIT-SE can prove the correctness of a traversed path while the SBT cannot do so. Thirdly, the SIT-SE can systematically find all the representative paths in a given program but the SBT does not have any systematic way to explore paths.

In this method, we introduce checking levels for branch conditions to proceed an incremental testing, as well as design a algorithm called BSC to mitigate the burden of exhausted path exploration by chopping down redundant paths. This method outperforms conventional methods as to identifying more false paths based on a small-scale generated test suite.

The proposed method SIT-SE has advanced the state of the art in the following two respects: (1) KLEE is weak in checking the path correctness, whereas our method is more capable in that area; and (2) KLEE usually needs to generate considerable number of test cases, but our method can produce a much smaller number of test cases, meanwhile recognizing more faulty paths with bugs in relation to the correctness of the path that are sometimes mistakenly ignored by KLEE.

1.3.2 Fault Localization

We propose a new fault localization method, TRIACFL (Triple Interaction-Based Fault Localization) to enhance the precision of fault localization in the meantime reduce high costs of inspection.

TRIACFL makes use of false symbolic paths provided by the SIT-SE and analyzes the conditions and blocks along these paths. It provides a flexible framework for fault localization with the integration of three main modules. The three main modules are the SIT-SE, an elementary fault location generation algorithm, and an attentional shift-based human review, respectively. They are intimately interacted with each other based on the prediction of the number of faults in the code. According to the prediction, TRIACFL guides testers to either go for the human review and exclusion or to go for the human review and fixing.

We have conducted experiments to evaluate the performance of TRIACFL against the most commonly used technique called Spectrum-Based Fault Localization (SBFL) that has many variations and extensions [29]. Similar to TRIACFL, SBFL also automatically generates a list of ranked suspicious statements for further inspection. We use SBFL-SSE, a SBFL-based method with some extension, sharing the same test data generated from the SIT-SE, for a fair comparison with TRIACFL. In both single fault experiment and multiple fault experiment, the proposed method TRIACFL outperforms SBFL-SSE with respect to precisely pinpointing the faults within moderate efforts of human inspection.

1.3.3 Test Selection

We propose a new method, featuring the integration of the functional scenario-based testing [30], a genetic algorithm (GA) [31] and the mutation testing [32] without the knowledge of internal code structure, to find useful mutated formal specifications for further test data generation. The idea behind is quite different from other similar work, such as methods that deploy a GA to directly do mutation on input data for test selection [33, 34], and some researches that are to mutate the control flows by analyzing the internal design of the code [35, 36]. The proposed method mainly mitigate the problem when the SIT-SE may be ineffective or even not applicable for testing a program that contains unavailable source code.

Specifically, this method uses a GA to obtain mutated specifications where appropriate values are assigned to the unknown output and dummy variables in the variations of the original specifications. These mutated specifications, sensitive to small syntactic structural changes of program codes, are further used to generate tests for effective bug detection. The proposed method can effectively generate useful tests from mutated specifications to kill as many program mutants as possible (nearly 20% higher than the conventional) in our case study, which outperforms the conventional SBT method.

1.4 Structure of the Thesis

This thesis is organized as follows.

Chapter 1 introduces several aspects of software quality assurance, and presents the main motivation along with the contributions in this research.

Chapter 2 introduces related work in bug detection, fault localization, and test data generation, respectively.

Chapter 3 presents preliminaries in formal specification, symbolic execution, as well as the combination of both them.

Chapter 4 proposes a method, SIT-SE (Specification-Based Incremental Testing Method with Symbolic Execution), for bug detection.

Chapter 5 proposes a method that integrates the SIT-SE, TRIACFL (Triple Interaction-Based Fault Localization), for fault localization.

Chapter 6 proposes a mutated specification-based method using a genetic algorithm, for test data generation in regression testing.

Chapter 7 concludes the research and points out directions in future work.

Chapter 2

Related Work

2.1 Symbolic Execution and Specification-Based Techniques

Since the essentials of the research are concerned with symbolic execution and specification-based testing (SBT), we briefly introduce and discuss some related work on both techniques.

2.1.1 Symbolic Execution Techniques

Symbolic execution analyzes a program through algebraic computation by assuming symbolic values for inputs, against normal execution with concrete input values. Several approaches based on symbolic execution have been proposed for testing programs or model checking. Concolic testing performs dynamic symbolic execution of a program as the program is executed on concrete input values [37, 38]. In concolic testing, there are two states, a concrete state and a symbolic state for the input variables. The symbolic state is involved with symbolic computation (e.g., computing the symbolic state for the left side of an assignment) while the concrete state is used to evaluate the boolean conditional expressions to true or false for choosing the branches to execute. However, concolic testing might not guarantee the completeness, that is, some feasible program paths could be missed [21]. Our approach primarily performs the concolic execution driven by the BSC algorithm and can do advanced verification for the correctness of each tested path. The details of the SIT-SE are explained in Chapter 4.

Many tools have been built to support concolic testing. DART is one of the tools proposed in the work [39] and it was built to support concolic testing of C programs. CUTE (for C)

[40] and jCUTE (CUTE for Java) [38] are developed to handle multi-threaded programs with complex data structures. CREST [41] is an open source tool for C programs with flexibility of allowing users to heuristically explore all the program paths. EXE [42] is a symbolic execution tool for comprehensively testing complex software and KLEE [23] also performs mixed concrete and symbolic execution with several improvements compared to EXE. Our approach takes advantage of these tools and enhances their capability by combining concolic testing with formal specification for correctness verification of paths.

Unlike the principle of concolic testing, Symbolic Java PathFinder (SPF) uses concrete execution only for setting up the environment for symbolic execution [43]. It is a tool for performing symbolic execution of Java bytecode and handles inputs and operations on booleans, integers, reals, and compound data structures with a polymorphic class hierarchy [44]. In SPF, the number of loops is determined by the tester, different from the way of executing loops by a test data in our approach.

Some testing methods are proposed by using the engine of Java PathFinder (JPF). For example, a framework (on the top of SPF) based on symbolic execution proposed in [45] is used for checking concurrent systems with compound data structures. It mainly uses model checkers to perform symbolic execution and analyzes the whole program by building symbolic execution trees. However, it becomes critical to build a huge symbolic execution tree due to the high complexity of the path exploration. In comparison, our approach tests one path for one execution at a time, thus reducing the complexity of exploring long paths (containing loops).

2.1.2 Specification-Based Techniques

As far as specification-based testing is concerned, our work is related to the following studies. Offutt and Liu [27] describe a technique that can be used for automated test data generation from SOFL specifications. The technique basically addresses the issue of developing formalizable and measurable criteria for generating test cases from specifications. This technique is part of our approach to generating test data from predicates used in SOFL specifications.

A Java framework called Korat [46, 47] was developed by MIT Laboratory for Computer Science for automated specification-based testing. To test a method, given a bound on the

size of its inputs and its pre- and post-conditions, Korat can automatically generate all non-isomorphic inputs up to a given small size. However, there are some limitations hindering the use of Kerat, such as the fact that the instrumentation cannot be applied to inner classes and multidimensional arrays cannot be processed.

TestEra [48, 49] is a framework for automated specification-based testing of Java programs. It employs Alloy [50], a first-order relational language, and the Alloy Analyzer. It automatically produces test cases based on the model of correctness criteria for the program in Alloy and the specified relations between the Alloy models and Java data structures. However, there are two primary difficulties when using TestEra in a real project: the testers are only allowed to use two primitive data types, boolean and integers, and the tool lacks the support for inheritance and nested classes.

Testing-based formal verification (TBFV) proposed by Liu [51, 52, 53] is a backward-style method derived from the integration of specification-based testing and Hoare logic for formally verifying the correctness of program paths. Its essential idea is to convert the program correctness verification problem into a problem of verifying theorems based on automatic testing. However, this method becomes less ineffective in dealing with the assignments that has a side effect as well as the pre-condition of simple form (e.g., pre-condition:=True). Furthermore, the TBFV does not give clues to the fault locations along a path.

Different from the existing approaches on SBT, our approach not only combines black-box testing, specifically specification-based testing, with white-box testing (by analyzing the code for test data generation), but also has potential to automatically verify the correctness of all the representative paths (that cover all the feasible branches of conditions) by using the concolic testing tools.

2.2 Fault Localization Techniques

Next, we introduces several existing fault localization techniques that relate to our method, and point out some differences between these techniques and the proposed fault localization method with SIT-SE.

2.2.1 Spectrum-Based Fault Localization

Spectrum-based fault localization (SBFL) [29] has been developed to suggest which statements of programs are likely to be faulty by using the characteristics of runtime executions. The technique is mainly motivated by the assumption that faulty statements are more frequently executed by test cases triggering the bugs of a program while less frequently by test cases triggering no bugs. All the statements that are examined according to the executions would be ranked through computing the suspiciousness score by using some ranking metrics including *Tarantula* [54] and *Zoltar* [55] and others [56, 57, 58]. However, SBFL costs much time to analyze all the singular statements without recognizing different roles of statements in the code and also their execution orders. By contrast, our algorithm derived from the SIT-SE mainly works on the sequence of specific statements that determine which branches to execute.

2.2.2 Program Slicing

Program slicing, proposed by Weiser [59], is a technique useful in debugging by narrowing down the scale of incorrect statements. A slice, the reduced part of program, preserves all the potential terminations of a program. For the debugging, Korel and Laski [60] proposed a method to provide better information of faulty positions by considering some special slice that preserves the program's behavior for a specific input, which is well-known as *dynamic slicing*. But this technique inspects only one execution to a specific failed test case one time, and the analysis of other parts of the code still remains heavy work. As a comparison, the proposed method systematically looks into various inputs and execution paths, as well as analyzes both successful and failed test cases.

2.2.3 Assertions

Assertions are widely used in helping more precise fault localizations. An assertion inserted in the code is executed at a specific program state. Any violations of the assertions indicate that the faults are likely to locate in the statements before executing the assertions. Several tools like ALADDIN [61] and Bug-Assist [62] have been developed to detect errors in runtime executions. However, writing appropriate assertions for different points of a program calls for great manual efforts, and many other faults cannot be found since the assertions only relate

to some specific program behaviors. To detect more bugs against specifications, our method works based on the mechanism in SIT-SE that explores all the representative paths, and makes further analysis for the correctness of specific statements or blocks along paths.

2.3 Test Data Generation Techniques

In this section, we introduce several advanced techniques that relate to our methodology for test data generation.

2.3.1 Data Flow Analysis

Data flow analysis, a technique for computing the def-use associations for the control flow graph (CFG) of a program, are often used to develop different strategies for test data generation over a long history [63, 64, 65]. Many research works have proposed promising methods for automatic test data generation that integrates the data flow analysis with the heuristics, such as GAs [66, 67, 36], particle swarm optimization [68], and ant colony optimization [69]. Different from these techniques, our approach conducts the testing under the circumstance where the source code of thirty-party library under test cannot be accessed. Thus, these techniques rely too much on the knowledge and analysis of internal design (or code structure), but our approach focuses on generating test data without analyzing the source code when applying the GA to the formal specification. In addition, with respect to the usage of the GA, our approach uses the GA to search for the optimal mutated specifications that are later used for input data generation, while the techniques mentioned above use the GA to directly search for good input data.

2.3.2 Mutation-Based Testing

The techniques of mutation-based test data generation [70, 71] are used to select a set of "good" test data by executing designed incorrect versions of an original program with a great number of test data from the domain. Test data are selected if it can cause unintended behaviors for a

certain number of incorrect versions. These techniques mainly concentrate on designing appropriate mutation operators to introduce small modifications for different kinds of programming languages such as Java [72], C# [73], and C++ [74]. The incorrect versions, also called program mutants, are created by inserting the mutant operators into the original program. Compared with these techniques, our method selects a set of "good" mutated specifications as a seed for further test data generation by using not only program mutants but also the mutated specifications with the GA.

2.3.3 SBT with Heuristics

The SBT techniques, some of them integrated with heuristic search strategies, have been well developed to cope with different kinds of specifications, such as SOFL [27, 75, 28], Alloy [76, 28], protocol specifications [14, 77], and Object Constraint Language (OCL) specification [78, 79]. Among these specifications, we take an interest in the formal specification of pre-post style like SOFL and Alloy. On the one hand, the SBT techniques for both SOFL and Alloy generate test data only from the pre- and guard- conditions, and use the post-condition as an oracle to check if the outcome is correct. On the other hand, the SBT with SOFL still needs to be improved since a data suite generated only from the original SOFL specification is not sufficient enough to trigger different kinds of bad behaviors of programs. On the contrary, our approach uses both the pre- and post- conditions to generate input data, as well as selects the optimal mutated specifications to enhance the bug detection.

Chapter 3

Preliminaries

In this chapter, we discuss two basic concepts used in our method SIT-SE, which are formal specification and symbolic execution, respectively. Then we discuss how the existing techniques of SBT with symbolic execution work in bug detection, as well as point out their limitations in path correctness verification.

3.1 Formal Specification

In this thesis, we use the Structured Object-Oriented Formal Language (SOFL) to write formal specifications for two reasons. One is that we are well experienced of using it for many software development projects, and the other is that SOFL has offered a three-step specification approach that makes SOFL practical enough to be widely learned [80] and applied in both academia and industry [75].

In SOFL, a *process* is like an operation in general, representing a transformation from input to output. Its functional behavior is defined by a formal specification with pre- and post-conditions. Let S denote a process, then we use S_{pre} and S_{post} to represent its pre-condition and post-condition, respectively. The process specification prescribes that if the input variables satisfy S_{pre} before process S, the output variables, which are defined in terms of input variables, must satisfy S_{post} after the process. For the sake of readability, a formal specification is suggested, by the SOFL method, to write in a way that the post-condition S_{post} is defined as a disjunction $S_{post} = \vee_{i=1}^{n}(G_i \wedge D_i)$. Formally, the specification is converted into an equivalent expression called *functional scenario form* (FSF).

Definition 3.1.1. *FSF* of a process is the disjunction of functional scenarios: $\vee_{i=1}^{n}(T_i \wedge D_i) := S_{pre} \wedge (\vee_{i=1}^{n}(G_i \wedge D_i))(i = 1, \cdots, N)$.

In this disjunction, G_i is a predicate called *guard condition* that contains only input variables and D_i another predicate called *defining condition* that defines the output variables. Since both S_{pre} and G_i are the constraints only on input variables, $T_i := S_{pre} \wedge G_i$ is called a *test condition*. The test condition in conjunction with a defining condition, for example $T_i \wedge D_i$, is called a *functional scenario*. Thus the formal specification of operation S can be written as the disjunction of functional scenarios: $\vee_{i=1}^{n}(T_i \wedge D_i)$. The functional scenario $T_i \wedge D_i$ describes a single specific functional behavior: when test condition T_i is true, the output of the operation is defined using defining condition D_i.

Each functional scenario defines an independent function of the operation: when the test condition holds on the input variables, the output variables will be defined by the defining condition. According to the previous study by Liu and his colleagues in [81], such a functional scenario is usually implemented by one or more program paths.

process Abs (x: int) y: int
pre true
post x ≥ 0 ∧ y = x ∨
x < 0 ∧ y = -x
end_process

FIGURE 3.1: Simple example for formal specification.

Fig. 3.1 shows a simple process with its formal specification. Process *Abs* computes the absolute value for an integer. It has pre-condition $S_{pre} := true$ and post-condition $S_{post} := (x \geq 0 \wedge y = x) \vee (x < 0 \wedge y = -x)$. From the specification, we derive two functional scenarios whose details are given as follows:

$G_1 := x \geq 0, \quad D_1 := y = x,$

$G_2 := x < 0, \quad D_2 := y = -x,$

$T_1 \wedge D_1 := (true \wedge x \geq 0 \wedge y = x),$

$T_2 \wedge D_2 := (true \wedge x < 0 \wedge y = -x).$

Formal specifications can be used for specification-based testing (SBT). In general, SBT takes three steps to fulfill the task of testing a program: (1) generating test cases from the specification of the program, (2) executing the program with the test cases, and (3) analyzing

the test results to determine whether bugs are found. As described later in this research, the SBT we discuss in this thesis emphasizes the importance of taking advantage of the "divide and conquer" principle. Specifically, the whole pre-post style process specification is divided into a set of functional scenarios and each functional scenario is used as the basis for test case generation and for test result analysis. The test condition of a functional scenario, which contains only input variables, is used to generate test cases and the defining condition of the functional scenario, which contains output variables, is used to analyze the correctness of outcomes.

In addition, we assume that the formal specifications are complete and every program path is supposed to relate to the formal specifications. The completeness of formal specifications can be ensured by other techniques proposed in our work before, such as specification inspection or refinement [82, 83].

3.2 Symbolic Execution

Symbolic executions systematically explore multiple paths with symbolic input values, each summarized by a first-order boolean formula that describes the conditional branches and outcomes along that path. This is in contrast with a concrete (or normal) execution that results in a single control flow path after being performed on a specific input. In general, static symbolic execution simultaneously explores many possible paths to build a huge execution tree without using concrete inputs. All the branch conditions can be assumed to be true or false during path exploration. Especially, the iterations of a loop are determined by humans. The symbolic execution used in this research refers to concolic execution. Concolic execution, also called dynamic symbolic execution, performs mixed concrete and symbolic execution for programs. There are two states for variables: symbolic values and concrete values that are used for symbolic computation and for deciding which branches to execute, respectively. For example, when it comes to an assignment "$y := 2 * x - 1$" where x is an input variable with concrete value 2, a concrete execution simply stores 3 in the memory of y, whereas a concolic execution additionally maps y to the symbolic expression "$2 \cdot x - 1$" in the memory. We insert probes into the source program to extract symbolic path condition (or constraints) and outputs with expressions over symbolic input values.

We treat a *program path* as a sequence of statements (assignments and conditions used in conditional statements), produced from one execution of the program with a concrete test data, as illustrated below.

Program path.

$1 : assignment_1 / condition_1$

$2 : assignment_2 / condition_2$

$\ldots$

$n : assignment_n / condition_n$

We apply symbolic execution to such a path to derive the *symbolic representation* $\{Sta \wedge Cds\}$, where the state $Sta := \bigwedge_{z \in Z} (z = e)$ (e is an expression over symbolic inputs and Z is a set of typed variables) and the condition (or constraint) $Cds := \bigwedge_{p \in P} p$ (P is a set of conditional expressions over symbolic inputs).

The symbolic representation $\{Sta \wedge Cds\}$ of a path indicates that when Cds is satisfied by the input variables, Sta will be satisfied by both the input and output variables.

We explain how symbolic execution works next. Suppose the current symbolic representation derived from the path is $\{Sta \wedge Cds\} := \{ \bigwedge_{v \in V} (v = e) \wedge \bigwedge_{p \in P} p\}$ where e is an expression over symbolic inputs and V a set of non-input variables, then computing the next symbolic representation for the next statement i on the same path is illustrated by the following process:

{current symbolic representation}

i : *assignment / condition*

{next symbolic representation}.

The derivation of the next symbolic representation is divided into the following four situations:

1. After the assignment $z := E$, the current symbolic representation changes to

   $\{z = e^{'} \wedge \bigwedge_{v \in V \setminus \{z\}} (v = e) \wedge \bigwedge_{p \in P} p\}$,

   where $e^{'} = E[e/v]$, which denotes the substitution that any variable v occurring in expression E has been replaced by the corresponding expression e if v can be found in $Sta := \bigwedge_{v \in V} (v = e)$.

2. After a condition q : $true$ (i.e., q evaluates to true), the current symbolic representation changes to $\{ \bigwedge_{v \in V} (v = e) \wedge \bigwedge_{p \in P} p \wedge p^{'} \}$, where $p^{'} = q[e/v]$,
   where $q[e/v]$ denotes the substitution that any variable v occurring in condition q has been replaced by the corresponding expression e if v can be found in $Sta := \bigwedge_{v \in V} (v = e)$.
3. After a condition q : $false$ (i.e., q evaluates to false), the current symbolic representation changes to $\{ \bigwedge_{v \in V} (v = e) \wedge \bigwedge_{p \in P} p \wedge p^{'} \}$, where $p^{'} = \neg q[e/v]$.
4. After a RETURN statement that indicates the end of a program path, remove all the $v = e$ from $Sta := \bigwedge_{v \in V} (v = e)$ if v is a temporary (or local) variable not used in the post-condition.

This symbolic execution is slightly different from the concolic execution, since it allows the predicates over symbolic inputs to contain some algorithmic functions [19].

An *algorithmic function* is a function that is defined by an executable algorithm whose internal code structure is unknown or not available. For instance, the predicates $y = x + f(x)$ and $y = x^2 - f(3x - 1)$ contain executable expressions $f(x)$ and $f(3x - 1)$, respectively, where x is symbolic input, y is symbolic output and f is an algorithmic function (like a black-box). Such predicates containing any algorithmic functions are called *algorithmic predicates* in this chapter.

As an example, we apply the symbolic execution to a small case in Fig. 3.2. In this case, process $fuc1$ reads the external variable z and the input variable x and returns the modified z; process $fuc2$ is invoked in a conditional statement of process $fuc1$. We assume that the code of process $fuc2$ is unavailable and its formal specification is provided.

```
int z = 5;
int fuc1( int x ){
   int y = x - z;
   if ( fuc2 (y) > x ) {
      z = z + 1;
   }else{
      z = z - 1;
   }
    return z;
}
```

```
process fuc2(q:int) r:int
pre true
post q > 0 ∧ r = 2*q ∨
        q ≤ 0 ∧ r = -q
end_process
```

FIGURE 3.2: A small case 1.

Let ˜z denote the initial value of variable z before executing $fuc1$, thus ˜z can be regarded as an input variable of $fuc1$. An effect of $fuc1$ is to change ˜z to the output value represented by variable z. We make $\{z = \tilde{z}\}$ as the initial symbolic representation before executing the first statement in $fuc1$.

Fig. 3.3 displays the process of deriving the symbolic representations for all the statements on the path given in the figure. We explain the steps in the process next.

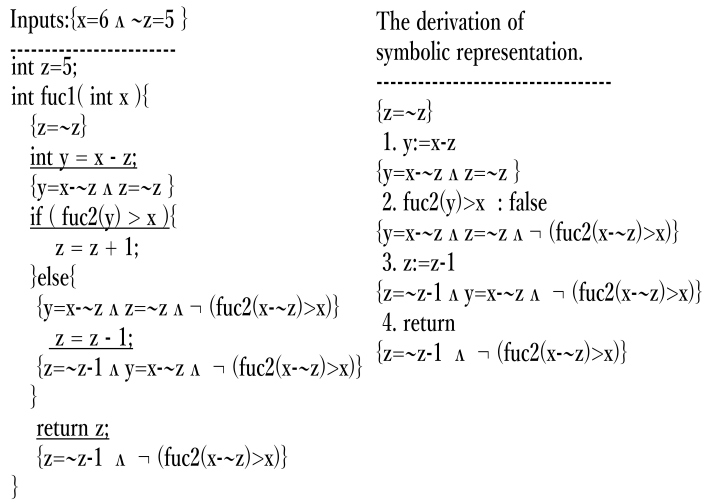


FIGURE 3.3: Derivation of symbolic representation along one path in case 1.

To obtain a program path, we assign concrete values for input variables x and ˜z: $\{x = 6 \wedge \tilde{z} = 5\}$. After the first assignment $y = x - z$ in the code, the symbolic representation changes to $\{y = x - \tilde{z} \wedge z = \tilde{z}\}$. Thus, temporary variable y can be represented by a symbolic expression over inputs x and ˜z. The symbolic expression before and after the assignment is illustrated as follows:

$\{z = \tilde{z}\}$

1. $y := x - z$

$\{y = x - \tilde{z} \wedge z = \tilde{z}\}$.

Further, we derive the next symbolic representation for the second statement $if\ (fuc2(y) > x)$ in the code. After substituting $x - \tilde{z}$ for y in $fuc2(y)$, we obtain the new condition

$fuc2(x - \tilde{z}) > x$.

Since the path is produced by the test case $\{x = 6 \wedge \tilde{z} = 5\}$, this condition evaluates to false because $fuc2(1) > 6$ evaluates to false based on the specification of $fuc2$. Thus, after the

branch condition $fuc2(y) > x : false$, the symbolic representation changes to

$\{y = x - \tilde{}z \wedge z = \tilde{}z \wedge \neg(fuc2(x - \tilde{}z) > x)\}$.

The process of obtaining the latest symbolic expression after the second statement is illustrated as follows:

$\{z = \tilde{}z\}$

1. $y := x - z$

$\{y = x - \tilde{}z \wedge z = \tilde{}z\}$

2. $if\ (fuc2(y) > x) : false$

$\{y = x - \tilde{}z \wedge z = \tilde{}z \wedge \neg(fuc2(x - \tilde{}z) > x)\}$.

Then we come to the third statement $z = z - 1$ in the code. According to situation 1), the symbolic representation changes to

$\{z = \tilde{}z - 1 \wedge y = x - \tilde{}z \wedge \neg(fuc2(x - \tilde{}z) > x)\}$

Finally, after the return statement, temporary variable y is removed from current symbolic representation. Thus for the path produced by executing the code with the test data, the final symbolic representation is

$\{z = \tilde{}z - 1 \wedge \neg(fuc2(x - \tilde{}z) > x)\}$.

This indicates that when input variables x and $\tilde{}z$ satisfy $\neg(fuc2(x - \tilde{}z) > x)$, the input and output variables will satisfy $z = \tilde{}z - 1$.

After performing the dynamic symbolic execution on the path, the derivation for the symbolic representation (where $\tilde{}z$ and x are symbolic inputs and z is the output) is finished as follows.

$\{z = \tilde{}z\}$

1. $y := x - z$

$\{y = x - \tilde{}z \wedge z = \tilde{}z\}$

2. $if\ (fuc2(y) > x) : false$

$\{y = x - \tilde{}z \wedge z = \tilde{}z \wedge \neg(fuc2(x - \tilde{}z) > x)\}$.

3. $z := z - 1$

$\{z = \tilde{}z - 1 \wedge y = x - \tilde{}z \wedge \neg(fuc2(x - \tilde{}z) > x)\}$

4. return

$\{z = \tilde{}z - 1 \wedge \neg(fuc2(x - \tilde{}z) > x)\}$.

Note that $fuc2$ is instrumented as an algorithmic function. According to its specification from Fig. 3.2, $fuc2(x - \tilde{}z)$ is equal to $2 * (x - \tilde{}z)$ because $x - \tilde{}z > 0$ given the concrete input values $\{x = 6 \wedge \tilde{}z = 5\}$ from Fig. 3.3. Thus the final symbolic representation can be further inferred as $\{z = \tilde{}z - 1 \wedge 2 * (x - \tilde{}z) \leq x\}$.

However, if the specification of an algorithmic function is unknown or not precise enough for such inference, the symbolic representation would be kept as before. In order to explore next path, we do negation for the path condition $\neg(fuc2(x - \tilde{}z) > x)$. Then the next test data should be generated to satisfy the condition $fuc2(x - \tilde{}z) > x$. However, no SMT solver can solve the predicate involving such an algorithmic function that is not defined using an explicit mathematical expression because the SMT solver can only deal with predicate formulas in which all of the terms (including function applications) must be explicitly defined using mathematical expressions. In our method, we continuously generate test data from the domain and invoke $fuc2$ until a suitable test data is found to satisfy $fuc2(x - \tilde{}z) > x$.

The techniques that employ a symbolic execution usually explore all the possible program paths by a depth-first search or other heuristic search strategies based on some stop criteria. Except the first initial input data that is generated from the domain, all the other input data are generated from the constraints formed based on the symbolic execution using an SMT solver. Each time the constraints for generating an input data are collected from the previous explored path with negation of some subexpression in the constraints. However, these techniques prefer an exhausted path exploration and inevitably encounter the problem of path explosion. Every path is explored by an input data, but the correctness of a path is not well verified. This deficiency causes to generate numerous test data for exploring paths and thus cost much time.

In contrast to the existing techniques, we propose a method that provides both a more rigorous way to verify the correctness of each path and the way to mitigate the problem of path explosion by using a new search strategy. More details of the proposed method are illustrated in the next section.

3.3 SBT with Symbolic Execution

Generally, the techniques of SBT with symbolic execution use assertions to verify the correctness of each path. An assertion, a boolean expression that relates to the specification, is

inserted into some specific point in a program before a test. The use of an assertion varies in performing different kinds of executions. When a normal execution reaches an assertion, an error will be detected if the assertion is evaluated to false (i.e., is violated) by using the concrete values of variables. Unlike the normal execution, when a symbolic execution reaches an assertion along a path, an SMT solver will be used to check if any values on that path violates this assertion.

We take the process $fuc2$ in Fig. 3.2 as an example. Suppose we have an incorrect implementation in C of $fuc2$ as follows. Two assertions, "$q > 0$ && $r == 2 * q$" and "$q <= 0$ && $r == -q$", are extracted from the specification, respectively. They are separately inserted into two different points near the exit of the code.

```
int fuc2(int q){
  int r;
  /* "q>2" should be "q>0" */
  if(q > 2){
      r = 2 * q;
      assert(q > 0 && r == 2*q); /*1*/
  }else{
      r = -q;
      assert(q <= 0 && r == -q); /*2*/
  }
  return r;
}
```

For a normal execution, concrete values are used to execute the code and determine if an assertion is violated. In the worst-case scenario, two test data, $t_1 : q = 3$ and $t_2 : q = -1$ for example, are generated from the domain to execute two program paths. No error can be detected at both *assert*s because neither t_1 nor t_2 violates the assertions.

By contrast, an SMT solver in symbolic execution (like in KLEE) is used to check if the following claims are correct, respectively.

1. For all q that $q > 2$, $q > 0 \wedge 2 * q = 2 * q$ is always true;
2. For all q that $q \leq 2$, $q \leq 0 \wedge -q = -q$ is always true.

Notice that r in the assertions will be replaced by its symbolic values $2 * q$ or $-q$ during the symbolic execution for the two paths. Claim 1) is correct, but claim 2) is incorrect because if $q = 1$ or 2, an error will be raised when the second *assert* (labeled by "/*2*/") is accessed during symbolic execution.

However, if the path with the second *assert* is found and explored by using symbolic input q together with the concrete value that satisfies $q \leq 0$, the second *assert* will never raise an assertion error for such input data. In view of the mathematical logic, this path is incorrect because $q \leq 2 \Rightarrow q \leq 0$ is false. Thus the related *assert* is supposed to raise an error regardless of which kind of input data that fetches it. This limitation hinders bug detection if we intend to test the program based on some specific functional scenarios to reduce time cost after an update of software. For instance, we intend to test $fuc2$ based on the second functional scenario: $q \leq 0 \wedge r = -q$.

```
/* use fuc2 with assertions */
/* use symbolic input q that q<=0 */
int r = fuc2 (q);
```

In this test, the second *assert* in $fuc2$ will never report any assertion violation because all the input data satisfying $q \leq 0 \wedge q \leq 2$ will not violate this *assert*. The use of assertions in existing methods does not always precisely reflect the requirement of specification, and thus these assertions are not sufficient enough to verify the correctness of a program path.

In the next chapter, we put forward a method to provide a more rigorous and systematic way for bug detection. In the proposed method, we use *theorems* instead of assertions for path correctness verification. A faulty path can be mistakenly seen as a correct one by assertions due to a single input data (with both symbolic and concrete states) that satisfies the post-condition. Conversely, such a faulty path can be identified by using theorems even when this single input data itself satisfies the related theorem. In our approach, a path is identified to be correct if and only if all the input values existing on this path satisfy the related theorem. Besides, there is no need to find every proper point of the code to insert assertions separately, which sometimes cannot be done due to some complex implementations or specifications, like in the case of testing a recursive function with multiple functional scenarios. The verification of the correctness for a path will be done after the related symbolic execution completely finishes.

As a comparison, for $fuc2$, the test for a specific path can proceed as follows.

```
/* use fuc2 with assertions */
/* use symbolic input q that q<=0 */
int r = fuc2 (q);
/* check a path with a theorem */
```

The explored path will be reported as false even if the second *assert* is never violated. From the perspective of safety-critical systems, it is essential to detect such kind of fault where the implementation mistakenly allows unexpected input data to exist on some specific paths regardless of which kind of input data used in the test. More details of the proposed method and theorems are described later.

Chapter 4

Grey-Box Testing: The SIT-SE for Bug Detection

We propose a specification-based incremental testing method with symbolic execution, called *SIT-SE*, to automate the verification of the correctness for each program path under a control of a moderate path exploration.

4.1 Principle of the SIT-SE

In the proposed method SIT-SE, every test data except the initial one is generated from some negation over a symbolic path condition, every path introduced by a test data is executed only once, and the correctness of each path is determined by the validity of a related theorem (mainly solved by an SMT solver) based on the specification.

The framework of the SIT-SE is illustrated in Fig. 4.1. To provide a proper environment for performing the symbolic execution on a program with a specified configuration (e.g., the checking levels used in the BSC algorithm), the program is instrumented manually by the tester or automatically by a script. For each path, we use one arbitrary test data that traverses this path to extract the symbolic representation as a summary of the path by performing symbolic execution. The symbolic representation is then used to form a theorem that describes the properties for all the input values on the path. Every path is found by a test data and checked by a theorem regardless of whether or not this test data itself satisfies the related theorem.

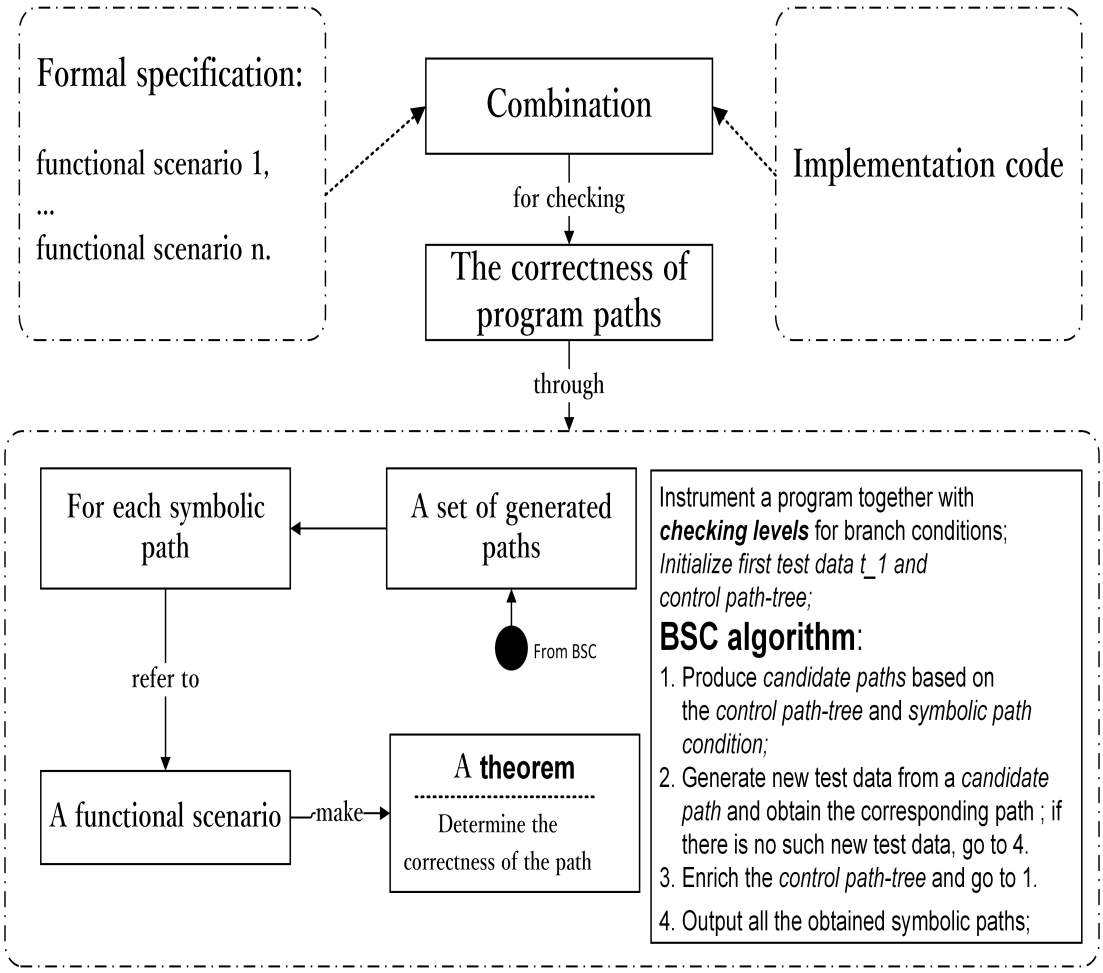


FIGURE 4.1: Framework of the SIT-SE.

Initially, we use the first test data that is an arbitrary one from the domain to start the process of bug detection. The BSC algorithm with predefined checking levels (determined by human) is used to automatically find other possible paths, each derived from a test data that is generated by using the existing path condition. In order to provide a more rigorous way to check path correctness, it is necessary to use one test data for one path to extract the symbolic representation for the path to make a theorem. We discuss how the theorem is formed and checked, as well as how all the suggested paths can be covered by the BSC algorithm below, respectively.

4.1.1 Theorem

A functional scenario may relate to many paths, and one path must refer to some functional scenario. In SIT-SE, all the representative paths are expected to be discovered and the correctness of each path needs to be checked by means of theorem proof.

A theorem in the SIT-SE is used to reason about the correctness of a program path, differing from Hoare logic [84] used for the correctness of the whole program. Such a theorem can be formed through the combination of the obtained symbolic representation and the formal specification of the code.

In the SIT-SE, for a process S under test, the correctness of a path is defined by the theorem:

$\exists! f \in F \cdot (S_{pre} \wedge Cds \Rightarrow f.T) \wedge (f.T \wedge Cds \wedge Sta \Rightarrow f.D)$,

where F is the set of all functional scenarios of the related operation and each functional scenario f has two attributes $f.T$ (test condition) and $f.D$ (defining condition).

According to this theorem, a path is correct if and only if there exists a unique functional scenario f in the specification such that the path contributes to the implementation of f (represented by $S_{pre} \wedge Cds \Rightarrow f.T$) and the test condition of f (i.e., $f.T$) together with the symbolic condition of the path (i.e., Cds) and its final state (i.e., Sta) will guarantee the defining condition of f (i.e., $f.D$). Note that since the test conditions of all the functional scenarios in the specification are pairwise exclusive, each path can only contribute to the implementation of one functional scenario. The pairwise exclusive property of functional scenarios is guaranteed by the algorithm for deriving the functional scenarios from a specification as described in [85]. On the other hand, if there does not exist any functional scenario for the path that satisfies the theorem, it will imply the existence of bugs on the path.

Checking Theorems

After the theorem is formed for the path, the next objective is to verify its validity. Fig. 4.2 shows a workflow for the verification. The first step is to verify whether the predicate $\exists! f \in F \cdot S_{pre} \wedge Cds \Rightarrow f.T$ holds or not, since it is the proviso given in the theorem for verifying the correctness of the corresponding path. If it holds, the next step is to verify whether the implication $f.T \wedge Cds \wedge Sta \Rightarrow f.D$ is satisfied by the symbolic representation of path. If it

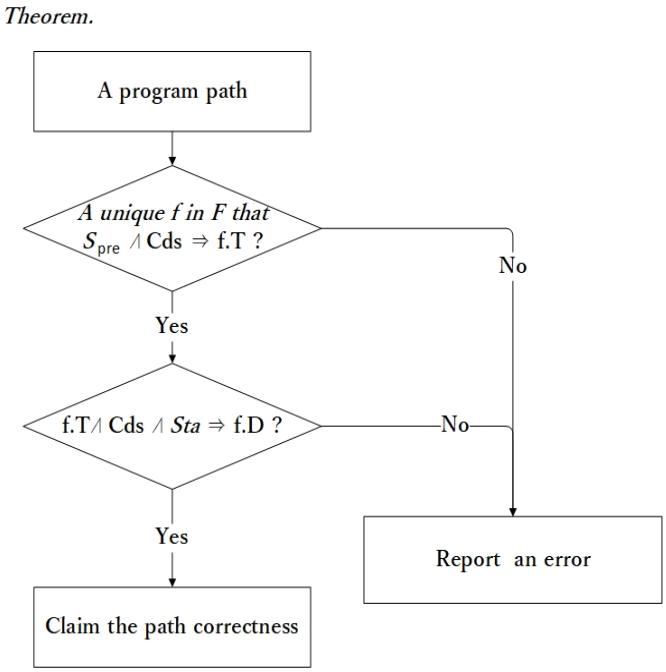


FIGURE 4.2: Checking the correctness of a path by theorem.

does not hold, that will indicate the existence of bugs on the path; otherwise, the correctness of the path will be ensured.

The theorem can either be formally proved (by human or a powerful theorem prover like Z3 [86] from Microsoft Research) or be verified with some confidence using predicate-based testing (that could be fully automated) like what has been done in [87].

To check an implication with the abstract form $X \Rightarrow Y$ (e.g., X can be $f.T \wedge Cds \wedge Sta$ and Y can be $f.D$) obtained from Fig. 4.2, we employ two steps as shown in Fig. 4.3. Firstly, we use an SMT Solver (Z3 in our work) to check the satisfiability of $X \wedge \neg Y$ (the negation of $X \Rightarrow Y$). There are three possible results: 1) the implication is proved to be true if $X \wedge \neg Y$ is determined to be unsatisfiable by Z3; 2) the implication is proved to be false if $X \wedge \neg Y$ is determined to be satisfiable by Z3; and 3) the satisfiability of $X \wedge \neg Y$ cannot be determined by Z3. Since Z3 is a software tool, it must have a limit in its capability of dealing with all kinds of predicates, but it is impossible for us as a user to estimate its capability quantitatively. When the proof of an implication is beyond the capability of the SMT solver, we will use the predicate-based testing to verify the implication instead. As discussed in [87], such predicate-based testing can be powerful enough and fully automated. This alternative is not discussed in detail here since

the current SMT solver Z3 seems to be powerful enough to deal with almost all of the classic cases in theorem proving.

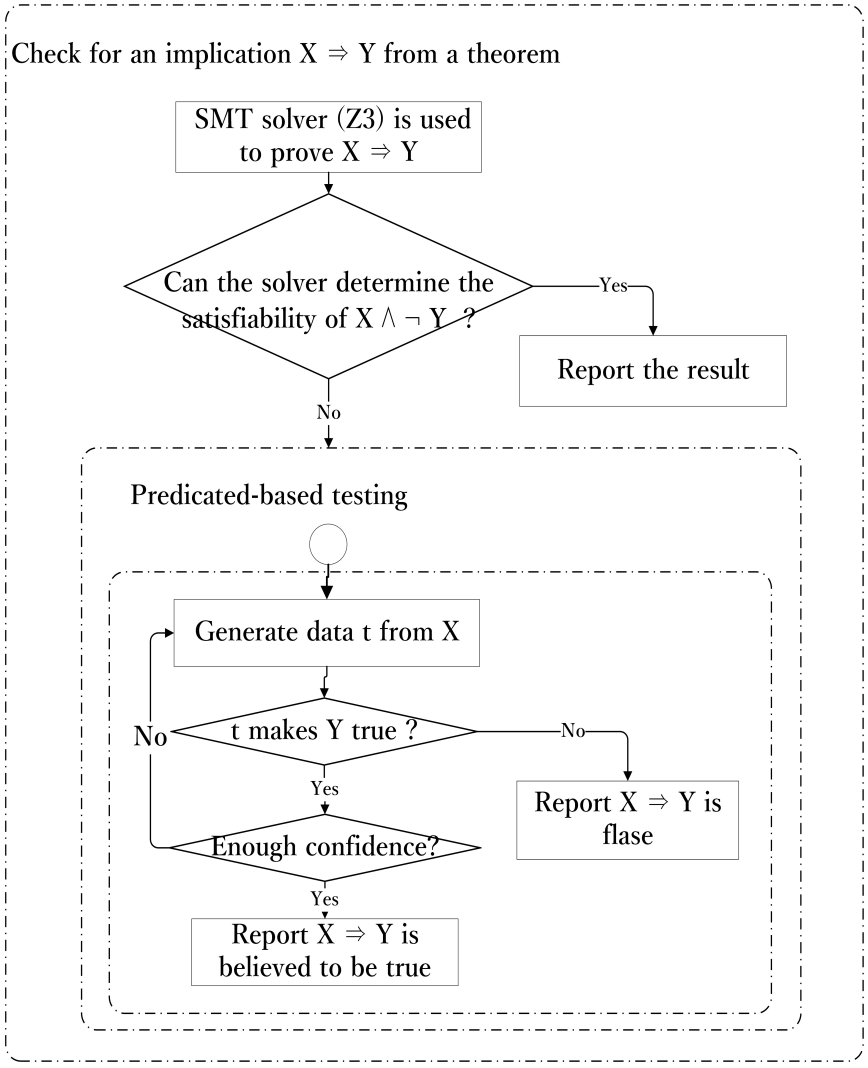


FIGURE 4.3: Checking an implication.

A Feature

There is a main feature that distinguish theorems from assertions.

Feature. *The correctness of a path is determined by the theorem for which the corresponding path condition is used.*

This feature ensures that the correctness of each path is verified through theorem proving. As described in Section 4.1.1, the theorem to be proved is formed based on the related functional scenario and the symbolic path condition and the final state of the related path traversed using one test data. Since once a test data satisfying the test condition of a functional scenario is used to run the program, one program path will be traversed and its symbolic path condition and final state can be automatically derived, as discussed previously. This ensures that the theorem about the correctness of the traversed path can be formed precisely. In our method, the SMT solver Z3 is used to prove the theorem. This is done by proving the conjunction of the hypothesis and the negation of the conclusion of the theorem to be a contradiction. In other words, the SMT solver concludes that no model for the conjunction can be found.

By contrast, KLEE checks if a single test data traversing that path violates the assertions. To recognize a faulty path, it has to find and generate another test data that traverses the same faulty path as well as violates the assertions when the previous test data that has introduced that faulty path does not violate the assertions.

In a word, theorems are stronger than assertions in verifying path correctness. Specially, in some cases, using assertions is not able to recognize a faulty path if the specification contains multiple functional scenarios. The process *spin* in Fig. 4.4 helps illustrate this point.

The process *spin* consumes an input p of type *Point* and returns an output pt of the same type. Two functional scenarios ($T_i \wedge D_i$) ($i = 1, 2$) can be extracted from the specification. An incorrect implementation of this specification is given in Fig. 4.5.

process spin(p: Point) pt: Point
pre true
post $(p.x \leq 0 \wedge p.y \leq 0) \wedge$
$(pt.x = p.y \wedge pt.y = p.x)$
end_process

Test condition *T1*: $p.x \leq 0 \wedge p.y \leq 0$
Defining condition *D1*: $pt.x = p.y \wedge pt.y = p.x$

Test condition *T2*: $p.x > 0 \vee p.y > 0$ *(it is ¬T1)*
Defining condition *D2*: *true*

FIGURE 4.4: The specification for process *spin*.

To explain this feature, we use assertions and theorems to check the code, respectively. First, Initialize a Point p and two symbolic values x (with name "x") and y (with name "y")

```
struct Point{
   int x,y;
};
struct Point spin(struct Point p ){
   // A bug, "||" should be "&&"
   if ( p.x <= 0 ||  p.y <= 0) {
      int tmp = p.x;
      p.x = p.y;
      p.y = tmp;
   }
   return p;
}
```

FIGURE 4.5: The incorrect code for process *spin*.

are assigned to the members of p, respectively. The input values are kept in [-20,20) to make the output easier to read without affecting the test.

```
static int x,y;
struct Point spin(struct Point p );
int main() {
 // initialized p with symbolic elements
    x=klee_range(-20,20,"x");
    y=klee_range(-20,20,"y");
    struct Point p= {.x=x,.y=y};
// using assertions based on the specification
    struct Point pt = spin(p);
    if (p.x  <=  0  &&  p.y  <=  0)
        assert(pt.x == p.y && pt.y ==
p.x);
    return 0;
}
```

FIGURE 4.6: Using assertions in KLEE.

As shown in Fig. 4.6, we use an *if expression* and an assertion (according to the specification) to test the output of process *spin* every time. However, the assertion would never be violated for any paths of *spin* reaching it. In fact, KLEE generates two test data, $t_1 : x = 12 \wedge y = 12$ and $t_2 : x = -20 \wedge y = -20$ to lead to two different paths, as well as reports no assertion violation errors.

By contrast, the two paths can be found to be faulty by using theorems. Fig. 4.7 shows that both paths are claimed to be incorrect because their path conditions cannot imply any test conditions in the first step of checking the theorem using the Z3 solver.

Although we can use assertions in another way like in Fig. 4.8, this way may require much

```
Using theorems:
test data t1: x = 12, y = 12;
The path goes along if else;
Path condition 1: ! (p.x <= 0 || p.y <= 0)
Theorem .
Path condition 1 ⇒ T1:
p.x > 0 ∧ p.y > 0 ⇒ p.x ≤ 0 ∧ p.y ≤ 0
False!
Path condition 1 ⇒ T2: (supposed to be true)
p.x > 0 ∧ p.y > 0 ⇒ p.x > 0 ∨ p.y > 0
False!
---------------------------------------------
test data t2: x = -20, y = -20;
The path goes along if then;
Path condition 2: (p.x <= 0 || p.y <= 0)
Theorem .
Path condition 2 ⇒ T1: (supposed to be true)
p.x ≤ 0 ∨ p.y ≤ 0 ⇒ p.x ≤ 0 ∧ p.y ≤ 0
False!
Path condition 2 ⇒ T2:
p.x ≤ 0 ∨ p.y ≤ 0 ⇒ p.x > 0 ∨ p.y > 0
False!
```

FIGURE 4.7: Using theorems.

```
struct Point spin(struct Point p ){
  struct Point p0 = p;
  // A bug, "||" should be "&&"
  if ( p.x <= 0 || p.y <= 0) {
    int tmp = p.x;
    p.x = p.y;
    p.y = tmp;
    assert(p0.x <= 0 && p0.y <= 0);
    assert(p.x == p0.y && p.y == p0.x);
  }else{
    assert(!(p0.x <= 0 && p0.y <= 0));
  }
  return p;
}
```

FIGURE 4.8: Using assertions in another way.

effort to work out the relationships between each exit of the process and the specification for some cases, or it just cannot work for some complex post-conditions thus the use of *if expression* is inevitable.

To test process *spin* in the way from Fig. 4.8, KLEE generates three test data,

$t_1 : x = 2 \wedge y = 2$, goes along if else;

$t_2 : y = -20 \wedge x = 12$ and

$t_3 : y = -20 \wedge x = 0$, goes along if then.

Only test data t_2 violates the assertion and the same path (that goes along if then) is executed twice, whereas by using the theorems all the test data will fail and both two paths will be claimed to be faulty.

Such a simple example vividly demonstrates the feature of theorems and also the effectiveness of using theorems.

4.1.2 Path Exploration

Unlike the traditional concolic testing that exhaustedly explores as many paths as possible by traversing all the nodes (each node represents a branch condition) along a path, we design an algorithm for path exploration, called *Branch Sequence Coverage algorithm* (*BSC* algorithm in short), doing negations for some representative constraints over a path condition and ensure that each path is symbolically executed for only once. Thus compared to the depth-first search used in KLEE, the BSC algorithm will cut off many uninteresting paths to explore.

Branch Sequence

To explain the BSC algorithm comprehensibly, we first need to introduce some necessary concepts, such as *control location* and *branch sequence*.

Given a program, a control location is a branch node fixed in the code that contains a condition. The condition can evaluates to true or false if the free variables are bound to specific values. For instance, the abstract program in Fig. 4.9 has four control locations l_i $(i = 1, 2, 3, 4)$ and every l_i possesses a condition (the condition is not explicitly written there) that cause the program to take different branches in its execution.

Every program path is related to a sequence of control locations:

$@l_1(N_1), \cdots, @l_m(N_m)$.

Where each element $@l_i(N_i)$ of the sequence denotes the control location l_i for its N_ith occurrence along this path; @ denotes either + or -, indicating a choice that the condition in l_i is evaluated to true (+) or false (-).

A *branch sequence*, derived from such a sequence of control locations, is defined as:

$@l_1(1), \cdots, @l_m(1)$.

Which only contains all the control locations for their first occurrence along the path (e.g., a loop condition may appear more than once when the loop is unfolded during an execution). Note that multiple symbolic program paths can share the same branch sequence, and the elements of a branch sequence can be symbolized differently according to the symbolic path.

For instance, suppose $z > x$ is the condition of a control location l_1 (for some "if"statement) in a program where z is a temporary variable and x is an input variable. Suppose along a specific symbolic path, $z > x$ can be evaluated to false for the first time where $z = x^2 - 1$, thus the element of its branch sequence, $-l_1(1)$, can be symbolized as $\neg(x^2 - 1 > x)$ (or $x^2 - 1 \leq x$).

Every path corresponds to a sequence of control locations (recording all the branch choices for the path), and a sequence of control locations can derive only one branch sequence.

Definition 1. *BSC is a coverage criterion requiring that all the feasible branch sequences for the code under test have been traversed.*

A program may have a great number of long paths (can be infinite in many cases) due to loops, but it has a relatively small scale of branch sequences for all of these paths. This is because for a program path, its branch sequence is one of the feasible combinations of different control locations occurring for the first time and the number of different control locations is finite and relatively small.

The *representative* paths, except the first randomly produced path, are all iteratively produced through the previously obtained symbolic paths and their branch sequences.

BSC requires all of theses paths (each path relates to a different branch sequence) to be traversed. Specifically, BSC will ensure that every statement of a sequential construct to be covered, every branch statement of a conditional construct to be covered, and the body of an iteration construct and the situation of its termination are covered.

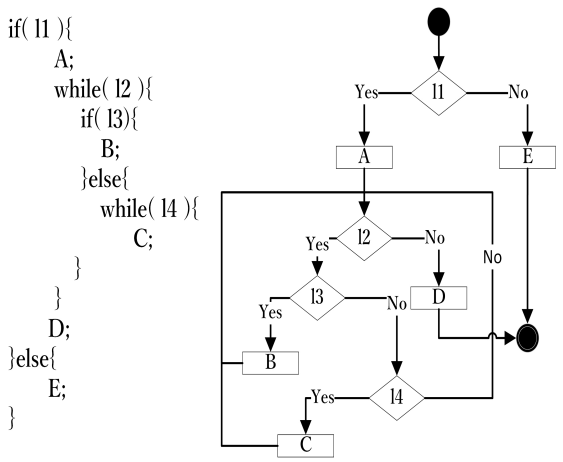


FIGURE 4.9: A program and its control flow diagram.

For example, Fig. 4.9 shows a program with nested loops and its control flow diagram

(CFD). For a randomly generated test data t_1, it may lead to a very long tedious path as showed in Fig. 4.10.

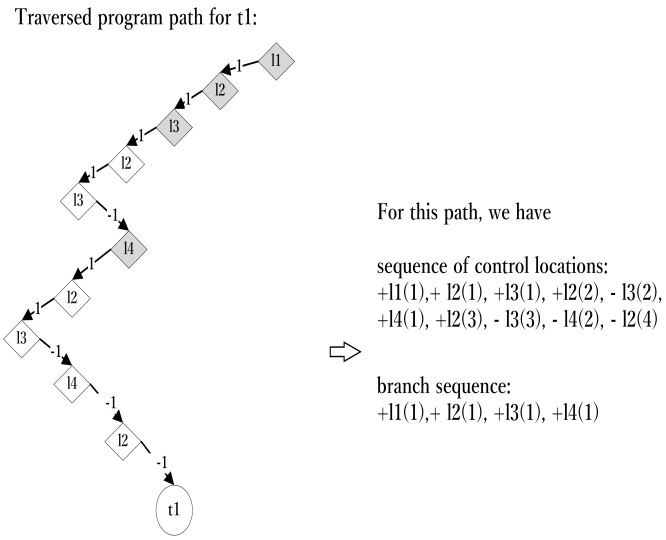


FIGURE 4.10: A traversed path and its branch sequence.

In the visualization of this traversed program path, the diamond denotes a control location l_i, the arrow labeled 1 starting from l_i denotes that the branch condition in l_i evaluates to true, and the arrow labeled -1 starting from l_i denotes that the branch condition in l_i evaluates to false. The grey diamonds refer to the control locations appearing for the first time along the path.

Fig. 4.10 also shows the branch sequence of this path, and we can see the branch sequence is quite shorter than the sequence of control locations for the program path. Although the program in Fig. 4.9 may have extremely large number of paths due to its loops, the number of branch sequences for all the paths is finite. All the feasible branch sequences referring to different paths (they are produced by some test data) for the program are listed in Fig. 4.11.

According to BSC, all the five branch sequences in Fig. 4.11 should be used for generating the five representative paths.

There are three primary characteristics for BSC. The first one is that BSC is inclined to explore more program paths than a full branch coverage requires. For example, test data t_1 and t_2 from Fig. 4.11 can be used to meet the branch coverage that explores only 2 paths as displayed in Fig. 4.12.

```
Branch sequences.

Test data t1:
+l1(1),+ l2(1), +l3(1), +l4(1)
Test data t2:
-l1(1)
Test data t3:
+l1(1),- l2(1)
Test data t4:
+l1(1),+ l2(1), -l3(1), +l4(1)
Test data t5:
+l1(1),+ l2(1), -l3(1), -l4(1)
```

FIGURE 4.11: All the branch sequences that BSC requires.

```
Only 2 test data are needed to meet
branch coverage.

Test data t1:
   Branch sequence:
     +l1(1),+ l2(1), +l3(1), +l4(1)
   Sequence of control locations:
     +l1(1),+ l2(1), +l3(1), +l2(2), -l3(2),+l4(1),
+l2(3), - l3(3), - l4(2), - l2(4)

Test data t2:
   Branch sequence:
     -l1(1)
   Sequence of control locations:
     -l1(1)
```

FIGURE 4.12: Two paths that branch coverage requires.

While BSC further requires more test data covering all the branch sequences in Fig. 4.11 to explore more paths. The second one is that BSC achieves the full path coverage for a program that contains no loops. The third is that BSC requires a smaller scale of test suite for the path exploration of a program with loops against to what a full path coverage requires (usually it is impossible to achieve a full path coverage for a program with loops).

Next we introduce the concept *control-path tree* used to implement BSC.

Control-Path Tree

Definition 2. *A control-path tree Tr is a binary tree established from branch sequences and it indicates the possibility of any unchecked paths. The tree can be graphically represented using diamond nodes, labeled arrows, and leaf nodes.*

A diamond node represents a control location l and it has two out-arrows labeled 1 and

-1, respectively. For convenience in the discussions below, the out-arrow of l is represented by $out(l)$. The $out(l)$ labeled with 1 denotes $l[p]$, and the $out(l)$ labeled with -1 denotes $\neg l[p]$, where $l[p]$ denotes the symbolic condition p located in l. The out-arrow can point to a diamond node or a leaf node.

The leaf node has two types, data node (relating to a test data that has been used to execute the program) and *null* node (without any test data). The path between the root node and a data node indicates the branch sequence for the related test data. The path between the root node and a *null* node is called a *seed track*, meaning that currently no test data is generated along this track.

Note that each seed track is the conjunction of the symbolized elements of the branch sequence, except where one symbolized element is negated.

Let Tkl be the set of all the control locations along a given seed track, P be the set of conditions based on Tkl, and let $Tk = \bigwedge_{l \in Tkl} out(l)$. Thus Tk for the seed track can be developed into a new path with the end of a data node. The sequence of control locations, branch sequence, symbolic path condition and seed track are obtained by executing the program with a test data t.

In order to produce a new path, firstly after running the program with a test data t and adding its branch sequence to the control-path tree, the symbolic path condition

$$\bigwedge_{i \in \{1,...,m\}, p \in P} l_i[p]$$

and Tk for the seed track

$$\bigwedge_{l \in Tkl} out(l)$$

are obtained for this test data.

Then a *candidate path* used to generate a new test data is constructed from the integration of both the symbolic path condition and the seed track:

$$\bigwedge_{i \in \{1,...,r-1\}, l_i \notin Tkl, p \in P} l_i[p] \bigwedge_{l \in Tkl} out(l),$$

where $1 \leq r \leq m$, and l_r is the control location of the last diamond along this track and it is also the control location of the rth position of the sequence of control locations; the set $\{1, ..., r-1\} = \emptyset$ if $r < 2$.

We can generate a new test data from such a candidate path to produce a new program path. In the next subsection, we will formally describe an algorithm to obtain all these kinds

of candidate paths for test data generation. Before that, we use the example in Fig. 4.13 to illustrate the process of developing a control-path tree and generating candidate paths.

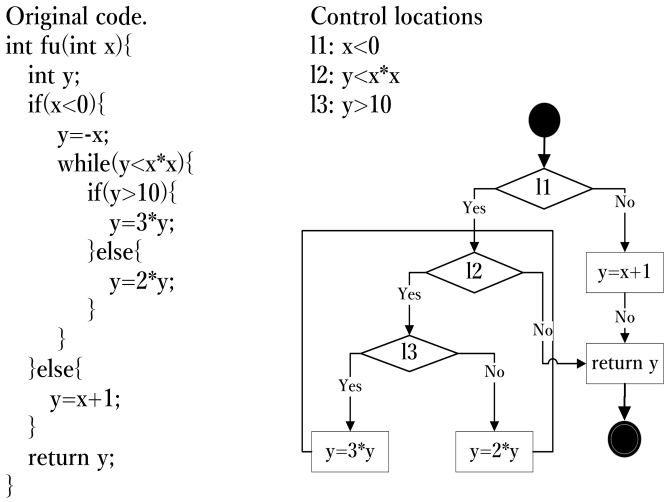


FIGURE 4.13: A program, control locations and CFD.

The left hand side of Fig. 4.13 shows a program code while the right hand side presents its three control locations and the control flow diagram (CFD). The control locations $l1$ and $l3$ relate to a "if" branch condition and $l2$ relates to a while loop condition.

We can obtain a symbolic path condition by running the program with a test data t_1: $x = -4$ (as an example). The control-path tree is established from the branch sequence of t_1 in Fig. 4.14.

There are three seed tracks in the control-path tree. Developing them further, we obtain three candidate paths as shown in Fig.4.15. For any new path derived from a new test data, we carry out a symbolic execution for it and compute its branch sequence. The new branch sequence will be added to the control-path tree.

For instance, Fig. 4.16 shows the modified control-path tree after incorporating the new branch sequence of test data t_2 that is generated from the candidate path $\neg(x < 0)$.

Pseudo-code for BSC

The BSC requires that all the generated candidate paths are explored. This implies that all the leaf nodes in the control-path tree are the nodes of test data. Such a tree is said to be *complete*.

Test data t1:
x = -4
Symbolic path condition:
x<0 ∧ -x<x^2 ∧ ¬ (-x>10) ∧
-2x<x^2 ∧ ¬ (-2x>10) ∧ ¬ (-4x<x^2)
Sequence of control locations:
+l1(1),+l2(1),-l3(1),+l2(2),- l3(2),- l2(3)
Branch sequence:
+l1(1),+l2(1),-l3(1)
Control-path tree:

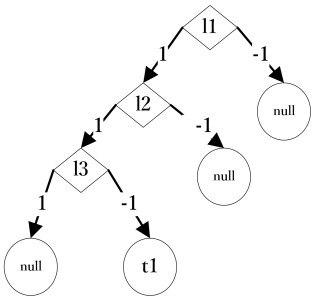


FIGURE 4.14: Control-path tree for example 1.

¬ *l1:*
¬ *(x<0)*
l1 ∧ ¬ *l2:*
x<0 ∧ ¬ *(-x<x^2)*

l1 ∧ *l2* ∧ *l3:*
x<0 ∧ *-x<x^2* ∧ *-x>10*

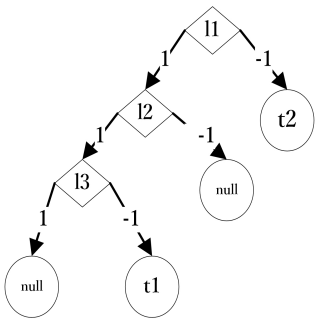
FIGURE 4.15: Candidate paths from test data t1.

The pseudo-code in Fig. 4.17 shows an algorithm for generating test data satisfying the BSC. The algorithm is characterized by the fact that only the initial test data is generated randomly but all of the other test data for traversing new paths are generated based on an available candidate path.

Both the BSC-based testing and the traditional concolic execution with depth-first search strategy perform mixed symbolic execution and concrete execution, but generally the former can explore fewer paths than the latter. This is due to the fact that the former has reduced the complexity of the path exploration through negating on branch sequences rather than on all the constraints over a symbolic path condition.

This means, as illustrated in Fig. 4.10, for a given traversed path, only the conditions obtained from a branch sequence (the conditions in the gray diamonds) would be negated to

Test data t2 from candidate path ¬ (x<0) :
x = 2
Symbolic path condition:
¬ (x<0)
Sequence of control locations:
- l1(1)
Branch sequence:
- l1(1)
Control-path tree:



There are no new seed traces that can be generated from t2

FIGURE 4.16: After symbolic execution for t2.

explore next paths in the BSC algorithm, but in the traditional concolic execution, the conditions in all the control locations (the conditions in all the diamonds) would be negated to explore next paths. We can see the BSC algorithm as a variant of the depth-first search. However, the BSC algorithm concentrates on searching for new paths with the undiscovered branch conditions and also undiscovered branchings. Unlike the depth-first search with the conventional concolic execution, the BSC algorithm only does the negations for the branch conditions obtained from the branch sequences along a program path.

4.1.3 Incremental Testing

To facilitate the exploration of more paths from the available path and therefore achieve more test data for their execution, the notion of *checking levels* to control locations needs to be introduced.

Checking levels are used to guide *incremental testing* to control the path exploration. Before *BSC* algorithm is applied, checking levels can be set to some values according to both experiences of testers and time limitation.

Algorithm *BSC*

```
Instrument a program;
Set checking levels for branch conditions;
INPUTS.   An initial test data: t1
          Original program code: P
OUTPUT.  Symbolic program paths: paths
-----------------------------------------------------------------
1:  paths={} ; D = {t1}
2:  while D is not empty:
3:     Let t=D.pop();
4:     sypath = derive_symbolic_path(P,t);
5:     paths.put( sypath );
6:     scl = Sequence_control_locations(sypath);
7:     bs = Branch_sequence(scl);
8:     if (Is_bs _not_in_cptree) :
9:          cptree = enrich_controlpath_tree(bs);
10:         cpaths.putall(
11:              get_candidate_paths(cptree,sypath)
12:          );
13:    for each p in cpaths:
14:         if (data_canbe_generated_from(p)):
15:              D.put(  generate_data_from(p)  );
16:    D.remove(t)
17:  return paths;
-----------------------------------------------------------------
scl: sequence of control locations
bs: branch sequence
```

FIGURE 4.17: Pseudo-code for BSC.

A checking level, denoted by $count_i$, is a non-negative integer. It is always attached to a control location l_i in the code. If $count_i > 0$, the first $count_i$ occurrences of control location l_i: $(\pm)l_i(1), \cdots, (\pm)l_i(count_i)$ in a path will be added to the branch sequence; otherwise, $count_i = 0$ means that all the occurrences control location l_i along a path should be added to the branch sequence.

We use the example from Fig. 4.10 to help explain it. By default, the checking level for each control location l_i is set to 1 so that only grey diamonds along the path will be used to form the branch sequence, as showed in Fig. 4.10.

However, if the checking levels for l_2 is specially set to 3, all the first three occurrences of l_2 will be used to form a different branch sequence, as displayed on the left side of Fig. 4.18; or if the checking levels for l_4 is further set to 2, all the first three occurrences of l_2 and the first two occurrences of l_4 will be used to form the branch sequence, as displayed on the right side of Fig. 4.18.

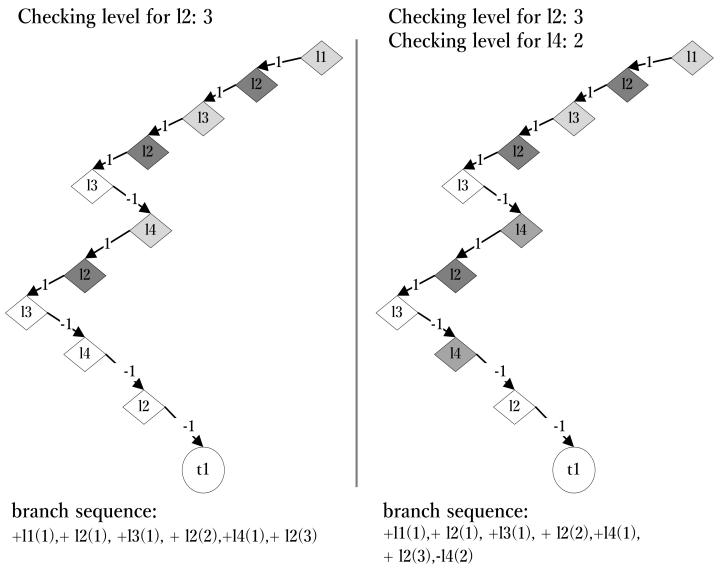


FIGURE 4.18: Different settings of checking levels: various shades of gray represent different checking levels.

In the BSC, branch sequences are used to build the control-path tree to explore paths. Since checking levels can affect the structure and the length of a branch sequence, different settings of them will lead to different ways of path exploration in the BSC.

We establish the basic rules for setting checking levels in a test:

Basic Rules for Settings.

1. *Generally checking levels for all the control locations are set to 1 by default, except some cases stated as follows.*

2. *For some control location relating to a loop condition, increase its checking level by 1 when the previous test cannot find faulty paths.*

3. *For control location that has access to elements of some vector (or array), set its checking level to 0.*

Rule 1) is used to guide a moderate path exploration meanwhile it assures that as far as possible all the control locations and their feasible branchings are visited. Rule 2) is used to incrementally explore new paths concerning loop conditions, because we think loop conditions should have more careful check than if conditions. Rule 3) is used to record different

features of symbolic elements of a vector to branch sequences, because different occurrences of the same control location can relate to different visits to that vector.

We use an example to illustrate how checking levels work by these rules. In Fig. 4.19, process *Mod_le0* is a variant of process *Mod* detailed in the next section, which is supposed to implement nearly half paths in *Mod* to calculate the quotient q and remainder r from dividing y by x. However, there is a fault in the control location $l_2 : x * r <= 0$, where $<=$ should be $<$.

```
void  Mod_le0(int y, int x){
  int q = 0;
  int r = y;
  if (x*y < 0) {
    // A bug,  "<=" should be "<"
    while(x*r <= 0){
        r = r + x;
        q = q - 1;
    }
  }
  printf("r = %d, q = %d", r , q );
}
```

```
Formal specification.
process  Mod_le0(y:int, x: int) r:int, q:int
pre  x*y < 0
post  y=q*x+r ∧ Abs(r)<Abs(x) ∧xr≥0
end_process
-----------------------------------
Control locations:
l1: x*y < 0
l2: x*r <= 0
```

FIGURE 4.19: Process *Mod_le0* for inputs with $x * y < 0$.

To test *Mod_le0*, set checking level 1 for both l_1 and l_2. Suppose we get $t_1 : y = 10 \wedge x = -12$, the first test data randomly generated from the domain $x * y < 0$, and obtain its branch sequence "$+l_1(1), +l_2(1)$". With this test data and the traversed path, we can form the following theorem for its correctness:

1) $(x * y < 0 \wedge x * y <= 0 \wedge x * (y + x) > 0 \Rightarrow x * y < 0) \wedge$

2) $(x * y < 0 \wedge x * (y + x) \wedge r = y + x \wedge q = -1 \Rightarrow$

$y = q * x + r \wedge Abs(r) < Abs(x) \wedge x * r \geq 0)$.

However its branch sequence cannot lead to other paths in this process because "$+l_1(1), -l_2(1)$" and "$-l_1(1)$" are not feasible branch sequences. As a result, we obtain only one successful test data t_1 in the BSC algorithm.

According to the rule 2), here comes to increasing checking levels in the test. For instance, we set the checking level to 2 for control location l_2. Then the branch sequence obtained from t_1 will be "$+l_1(1), +l_2(1), -l_2(2)$". Based on the BSC algorithm, we can get a new derived branch sequence "$+l_1(1), +l_2(1), +l_2(2)$" to generate a test data to lead to a new path, e.g., the

second test data $t_2 : y = -4 \wedge x = 1$. In this test, according to the BSC algorithm, two test data t_1 and t_2 can be obtained, one is successful and the other is failing.

By contrast, traditional concolic testing methods exhaustedly explore as many paths as possible using depth-first search or other heuristics (to change the order of paths to find). In the case of KLEE, it generates a total of 17 test data with one failing test data in this test.

Aside from these rules, testers can make their own plans by making additional rules in bug detection.

4.2 Case Study

This section uses a simple example, deliberately, to comprehensibly demonstrate how the SIT-SE works on the code implementing a process specification. The process is called *Mod*, computing the quotient q and remainder r from dividing y by x, and its specification is given in Fig. 4.20a. In the specification, the function *Abs* is used to compute the absolute value of an input. The implementation of the specification is also given in Fig. 4.20b in which the function *Abs* is implemented as a inline function.

There are two functional scenarios derived from the specification of process *Mod*.

1. The first functional scenario is $T_1 \wedge D_1$, where the test condition $T_1 := (x \neq 0 \wedge y \neq 0)$ and the defining condition $D_1 := (y = q \cdot x + r \wedge Abs(r) < Abs(x) \wedge xr \geq 0)$.

2. The second functional scenario is $T_2 \wedge D_2$, where $T_2 := (x \neq 0 \wedge y = 0)$ and $D_2 := (q = 0 \wedge r = 0)$.

There are four control locations according to the implementation code of the process *Mod*, as illustrated in Fig. 4.21. According to the basic rules for setting checking levels, the checking levels for all the control locations are set to 1.

In order to test all the program paths using the SIT-SE, let us start with an initial test data $t_1 : y = 3 \wedge x = 2$ (as an example) that is generated randomly to satisfy the pre-condition $x \neq 0$.

The symbolic execution for the path traversed by test data t_1 is shown in Fig. 4.22.

By executing the program with test data t_1 and meanwhile carrying out a symbolic execution of the traversed path, we can get the final state *Sta* and the path condition C_{t_1}, as well

process Mod(y:int, x: int) r:int, q:int
pre x ≠ 0
post x ≠ 0 ∧ y = q*x + r ∧ Abs(r) < Abs(x) ∧ x*r ≥ 0 ∨
y = 0 ∧ q = 0 ∧ r = 0
end_process

(A) The specification of *Mod*

```
void Mod(int y, int x){
    int r = y;
    int q = 0;
    if (y != 0){
        if (x*y > 0){
            while(Abs(x)<= Abs(r)){
                r = r - x;
                q = q + 1;
            }
        }else{
            do{
                r = r + x;
                q = q - 1;
            }while(x*r < 0);
        }
    }
    printf("r = %d, q = %d", r , q );
}
```

(B) The implementation of *Mod*

FIGURE 4.20: The process *Mod*

```
Control locations.
l1: y ≠ 0
l2: xy > 0
l3: Abs(x) ≤ Abs(r)
l4: xr < 0
```

FIGURE 4.21: Four control locations from the program.

```
The derivation of symbolic representation
for test data t1:
{Sta ∧ Cds} is {true} at the beginning.
1. r:=y
{r=y}
2. q:=0
{r=y ∧ q=0}
3. y != 0          :true
{r=y ∧ q=0 ∧ y≠0}
4. x*y>0           :true
{r=y ∧ q=0 ∧ y≠0 ∧ xy>0}
5. Abs(x)<=Abs(r)  :true
{r=y ∧ q=0 ∧ y≠0 ∧ xy>0 ∧ Abs(x)≤Abs(y)}
6. r:=r-x
{r=y-x ∧ q=0 ∧ y≠0 ∧ xy>0 ∧ Abs(x)≤Abs(y)}
7. q:=q+1
{r=y-x ∧ q=1 ∧ y≠0 ∧ xy>0 ∧ Abs(x)≤Abs(y)}
8. Abs(x)<=Abs(r)  :false
{r=y-x ∧ q=1 ∧ y≠0 ∧ xy>0 ∧ Abs(x)≤Abs(y) ∧
 Abs(x)>Abs(y-x)}
9. return
{Sta ∧ Cds} is
{r=y-x ∧ q=1 ∧ y≠0 ∧ xy>0 ∧ Abs(x)≤Abs(y) ∧
 Abs(x)>Abs(y-x)}
in the end.
```

FIGURE 4.22: Derivation of symbolic representation for the path by test data t1.

as the sequence of control locations and branch sequence. We have also constructed a control path-tree based on test data t_1 from which three seed tracks (the path between the root node and the null node) are obtained, as shown in Fig. 4.23.

Because the symbolic path condition C_{t_1} implies test condition T_1, i.e., $S_{pre} \wedge C_{t_1} \Rightarrow T_1$ where $S_{pre} := (x \neq 0)$, we can form the theorem

$T_1 \wedge C_{t_1} \wedge Sta \Rightarrow D_1$, that is,

$x \neq 0 \wedge y \neq 0 \wedge$

$xy > 0 \wedge Abs(x) \leq Abs(y) \wedge Abs(x) > Abs(y - x) \wedge$

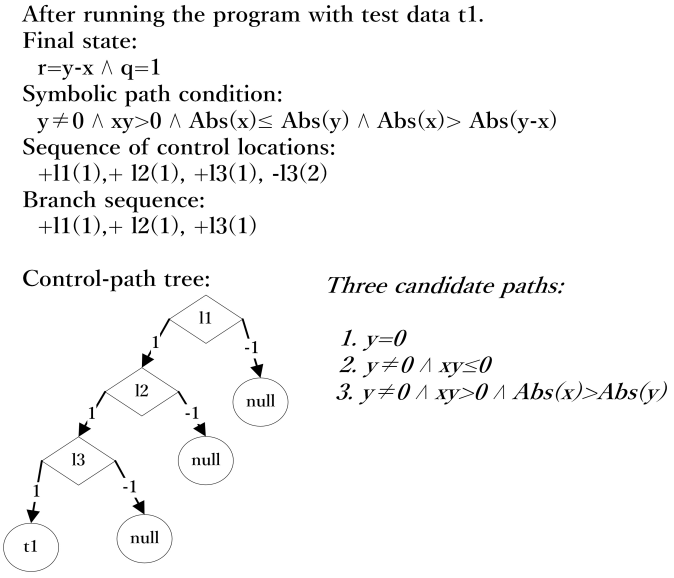


FIGURE 4.23: Results of running the program with test data t1.

$r = y - x \wedge q = 1 \wedge y \neq 0$

$\Rightarrow y = q \cdot x + r \wedge Abs(r) < Abs(x) \wedge xr \geq 0.$

By replacing r with 1 and replacing q with $y - x$ in the conclusion of the implication, respectively, we simplify this theorem into the following:

$x \neq 0 \wedge y \neq 0 \wedge$

$xy > 0 \wedge Abs(x) \leq Abs(y) \wedge Abs(x) > Abs(y - x)$

$\Rightarrow Abs(y - x) < Abs(x) \wedge x(y - x) \geq 0.$

The theorem now involves only input variables and has been formally proved to be valid. Therefore, the corresponding program path is also verified to be correct.

To verify other program paths, we generate one test data for each candidate path, respectively. As a result, the following three new test data are generated:

$t_2 : y = 0 \wedge x = 2$, satisfies $y = 0$.

$t_3 : y = 3 \wedge x = -2$, satisfies $y \neq 0 \wedge xy \leq 0$.

$t_4 : y = 3 \wedge x = 4$, satisfies $y \neq 0 \wedge xy > 0 \wedge Abs(x) > Abs(y)$.

To verify other program paths, do the similar thing to test data t_2, t_3, and t_4, respectively. In Fig. 4.24, we can see that a new test data t_5 can be generated based on test data t_3. After completing the testing of using all the five test data, the control-path tree is completed and no more new test data can be generated.

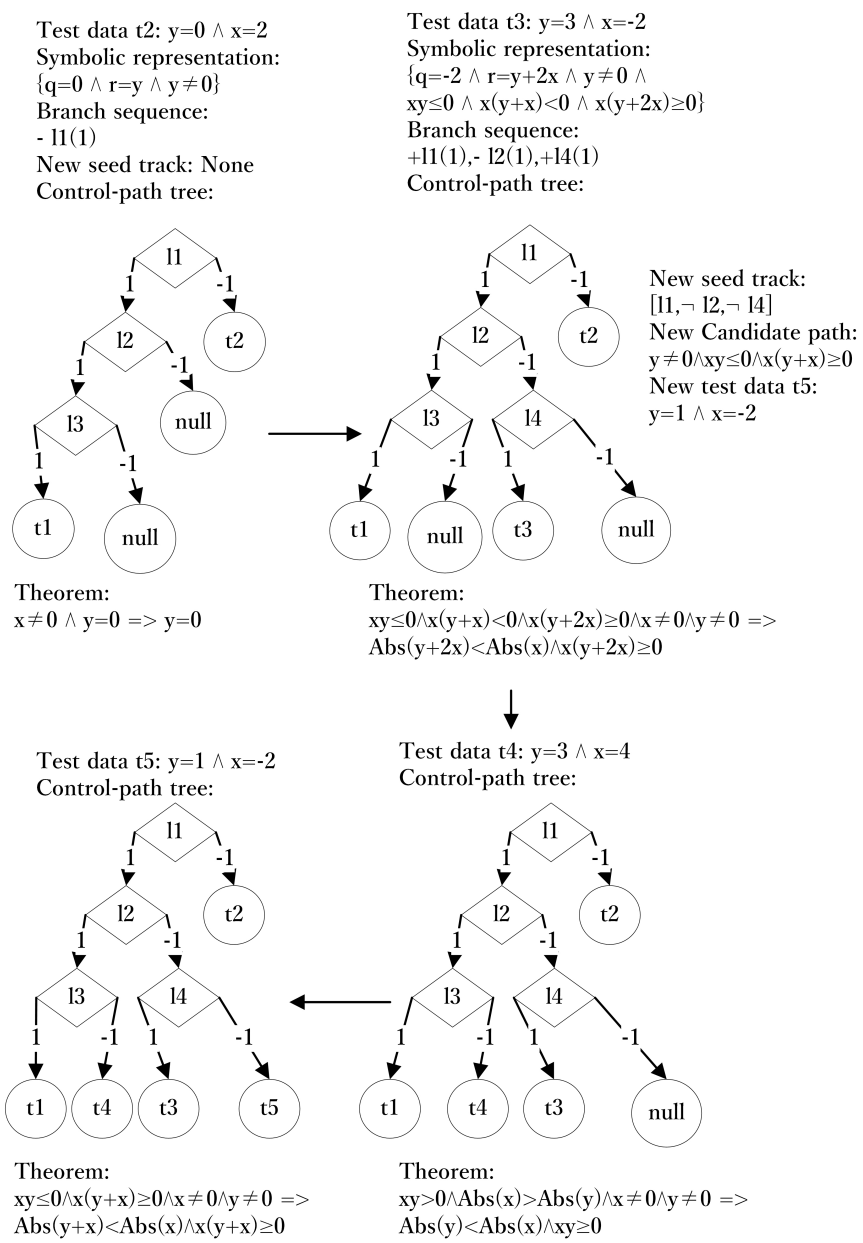


FIGURE 4.24: Apply SIT-SE to process Mod.

In summary, there are a total of 5 test data generated for the verification of five program paths. Four test data (t_1, t_3, t_4 and t_5) are referred to functional scenario $T_1 \wedge D_1$ and one test data (t_2) is referred to functional scenario $T_2 \wedge D_2$. All the discovered program paths are verified based on their theorems.

Different from these results, traditional concolic testing tool KLEE generates a total of 66

test data for *Mod* according to our experience with the tool, much more than the proposed method.

Now let us discuss how the SIT-SE can help uncover faults on a path. To this end, we first inject some fault into the path of interest, and then explain how it helps identify the fault.

A fault on the first path. Suppose a fault is injected into the first path, such as modifying $Abs(x) \leq Abs(r)$ to $Abs(x) < Abs(r)$ in the code. Use the same first test data t_1 to form a different theorem:

$x \neq 0 \wedge y \neq 0 \wedge$

$xy > 0 \wedge Abs(x) < Abs(y) \wedge Abs(x) \geq Abs(y - x)$

$\Rightarrow Abs(y - x) < Abs(x) \wedge x(y - x) \geq 0.$

This theorem can be formally proved to be invalid by the Z3 prover. As opposed to this result, KLEE incorrectly asserts t_1 to be a successful test data by only confirming that this test data satisfies the post-condition.

In addition, we can use the testing method to check the theorem. For instance, generate a test suite like:

$\{y = -7 \wedge x = -5\}, \{y = 4 \wedge x = 4\}, \{y = 6 \wedge x = 3\}.$

We can see the first three test data (including t_1) are not able to make the theorem false, but the test data $\{y = 6 \wedge x = 3\}$ makes the theorem to be invalid. This indicates that the SIT-SE is able to effectively capture much information of a path and uncover faults without repeatedly executing the faulty path.

4.3 Experiment

KLEE [23] is a well-known concolic testing tool that is commonly used for many researches [88, 89, 90, 91]. Since KLEE supports assertions for specification-based testing and leads to a per-path analysis like the SIT-SE, we mainly compare the performance of KLEE and the proposed method in this experiment.

We employs the mutation testing in our experiment as the means for the comparison. Mutation testing [92, 93] is used to design test data and measure the performances of existing techniques. Some faults are injected into the original programs in advance and these modified

programs are known as *program mutants*. The faults are created by applying *mutation operators* to original programs, making syntactic changes to all kinds of programming languages [94, 95, 96, 74]. For example, the mutation operator *AOR* describes an arithmetic operator replacement like replacing $+$ with $-$ in the original program. In our experiment, we selected some faults that were made during implementation, and insert them into the correct versions to make program mutants. The inserted faults reflect the effects of using traditional mutation operators, including arithmetic operator replacement, boolean relation replacement, missing of boolean subexpression, statement deletion, and swap of statements, as displayed in Table 4.1.

TABLE 4.1: Mutation operators used in the experiment.

Types of Bugs	**Mutation Operators**
* The inserted faults has following features: a) cause functional bugs b) no crashes of programs c) no infinite loops	Arithmetic Operator Replacement
	Boolean Relation Replacement
	Missing of Boolean Subexpression
	Statement Deletion
	Swap of Statements

Usually, a test data kills a program mutant if it makes the behaviour of the program mutant abnormal against the original program. This general principle is specialized in our case that a test data is regarded to kill a program mutant if the theorem for its path is proved to be invalid; and a test data kills a mutant if any assertion is violated in KLEE. The test data that kills the mutant is called failing test data and the corresponding generated path is called failing path; otherwise, they are called successful test data and successful path, respectively.

We develop a prototype tool to implement the proposed method in Python for automating the verification process in the experiment. But since the usability of the tool is not good enough for general public to properly use our prototype tool, we believe that it is inappropriate to publish it for now. But we will try to publish the tool in the future after it is developed to a rather mature level. Both the SIT-SE and KLEE use the constraint solver Z3. Different from KLEE working on LLVM bitcode, our tool works on the source code to facilitate quick and flexible scripting.

4.3.1 Preparation

We select 9 classic programs that implement various commonly used algorithms in most standard libraries (e.g., C++ standard library) for our experiments. These programs have different features on inputs, outputs, the structure of the code and formal specifications, as shown in Table 4.2 and in Table 4.3.

TABLE 4.2: Descriptions for programs under test

Programs	**Descriptions**
Mod	Get the remainder and quotient for 2 integers
Bsearch	Binary search for array
BubbleSort	Bubble sort for sorting array
HeapSort	Heap sort for sorting array
QuickSort	Quick sort for sorting array
Gcd_Stein	Steins algorithm to find GCD for 2 integers
Dijkstra	Algorithm to find the shortest paths between nodes
RedBlackTree	The implementation of Red-Black Trees. Check Red-Black Trees properties on the operations of insertion and deletion.
SetOp	Set operations for union,intersection,difference. The code involves binary search, quick sort and the check for several properties from set theory.

TABLE 4.3: The characteristics of programs under test

Programs	**Fucs**	**S_Input**	**S_Output**
Mod	2	y,x:int	r,q:int
Bsearch	1	v: array,x:int	ind:int
BubbleSort	1	v: array	v:array
HeapSort	3	v: array	v:array
QuickSort	1	v: array	v:array
Gcd_Stein	1	x,y:int	r:int
Dijkstra	2	graph: 2-array	dist:array
RedBlackTree	6	v: array	bst: tree
SetOp	6	v1,v2:array	v3:array

In Table 4.3, *Fucs* denotes the number of main functions involved in a program; *S_Input* and *S_Output* denote the symbolized inputs and outputs, respectively; the left side of colon ":" is variable and the right side of ":" is the type; and 2-array represents a two-dimensional array.

For each specification, a program resulted from a signature-preserved implementation of it is provided. The purpose is to ensure that all of the input and output variables used in the specification are preserved in the program in order to facilitate the formation and verification of the related theorems.

To evaluate the performance, three mutants are prepared for each program in the way that the injected faults will not cause execution crash and infinite loops, since both our method and the KLEE tool cannot manage those situations. We consider it reasonable to use 3 mutants for each program because most programs are relatively small and all the 3 faults with each being hidden in one or two statements are most likely to be made in practice. Additionally, all the injected faults are collected from the mistakes that were committed during the process of writing the programs by the programmers before our experiment.

4.3.2 Experimental Results

The experimental results are as follows.

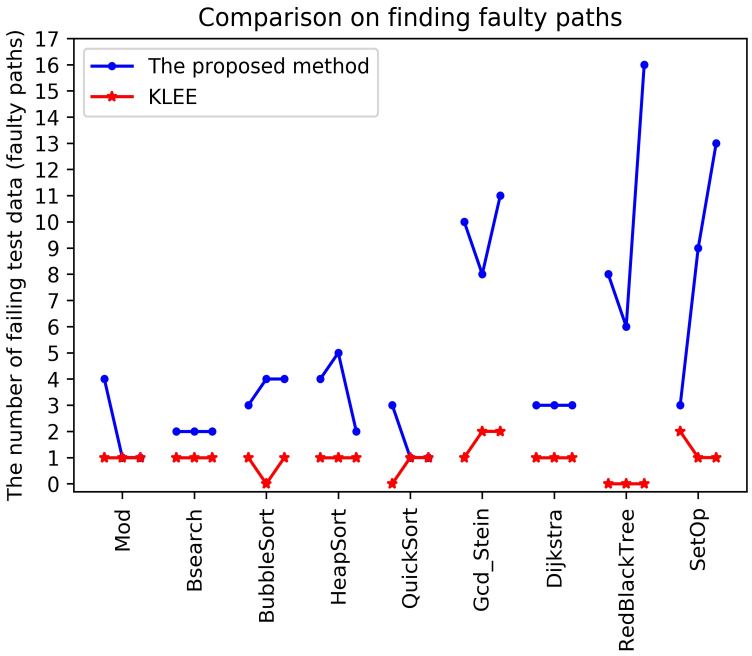


FIGURE 4.25: Performance on finding faulty paths

Fig. 4.25 displays the comparison on the number of failing paths discovered by the two methods. Each program label on the x-axis is surrounded by its own three mutants and the

y-axis represents the number of failing paths. Based on that we can see the proposed method is capable of identifying more faulty paths than KLEE. Moreover, KLEE only provides the contents of the failing test data, but our method provides both the contents and the features (in the form of theorems) of those faulty paths to help the tester clearly understand the locations of bugs in the code.

Specially, KLEE fails to generate failing test data for the program *RedBlackTree*. Actually, KLEE generates only one test data for each mutant that has a complex post-condition and data structures. It seems that KLEE may not be able to cope well with branch conditions involving some complex data structures.

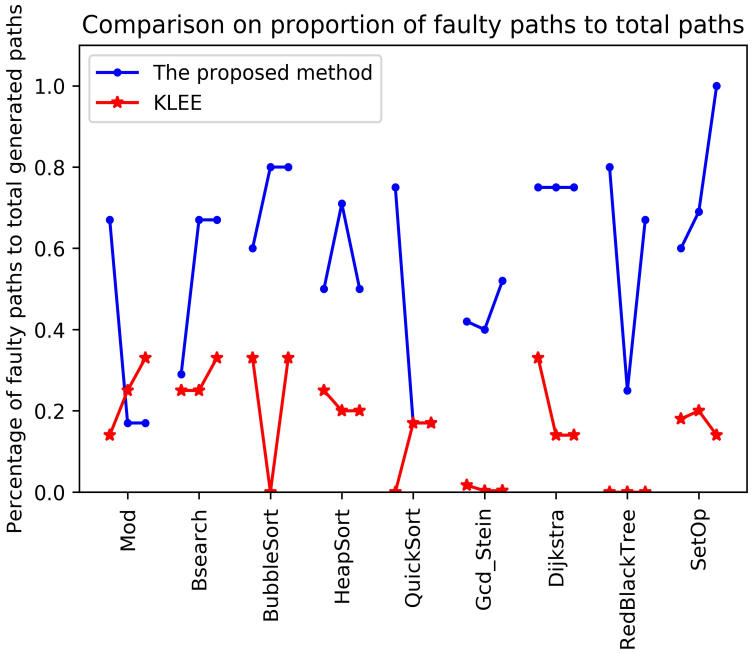


FIGURE 4.26: Faulty paths rates

Fig. 4.26 exhibits the proportion of faulty paths to all the generated paths after using the two methods. Compared to KLEE, the proposed method is better in recognizing faulty paths when searching unchecked paths. Furthermore, KLEE can find few faulty paths for relatively simple programs or cannot find any faulty paths for them (like the cases in *BubbleSort* and *Gcd_Stein*).

For KLEE, each path condition is only used to generate one test data and the correctness of a path is determined by whether or not such a test data satisfies the post-condition (through

assertions). Thus a faulty path can be mistakenly regarded as a correct one if the corresponding test data is successful. This can be a reason that explains why KLEE found a small number of failing test data.

By contrast, the path condition is further used in combination with the specifications to make a theorem for more rigorous verification of the path correctness in our method. Thus, a faulty path can be discovered even if the corresponding test data satisfies the theorem but a constraint solver can prove this theorem to be invalid. For such case, the "successful" test data is correctly regarded as failing by our method.

Furthermore, Fig. 4.27 and Table 4.4 show the comparison of both methods on the cost of testing time. This evaluation experiment was repeated three times and the best results of each method are recorded. Note that the testing time of the original correct program is also displayed before the sequence of the mutants of each program for comparison. The result shows that, our method can significantly reduce the testing time by KLEE when it comes to deal with complex code structures and post-conditions. Specially, although the KLEE tool rapidly completed the testing for program *RedBlackTree*, the test ended up generating only one successful path for each mutant.

In most cases, the SIT-SE takes moderate time to generate relatively a smaller scale of test suite that lead to the identification of more faulty paths. It is to sacrifice most time for advanced verification of each path while reducing heavy burden of dynamic executions due to the disciplined path exploration strategy. This will be a big merit when testing larger programs in the real world. Moreover, the performance of the tool can be further improved by optimizing the code in the future work.

4.3.3 Summary

The experimental results demonstrate that our method is able to generate more different failing test data with moderate cost of time by exploring relatively small scale of program paths. It can also provide more information by means of theorems of the existing bugs for process repairing. The result further shows that the proposed method can work well on both simple and complicated programs compared with the KLEE tool.

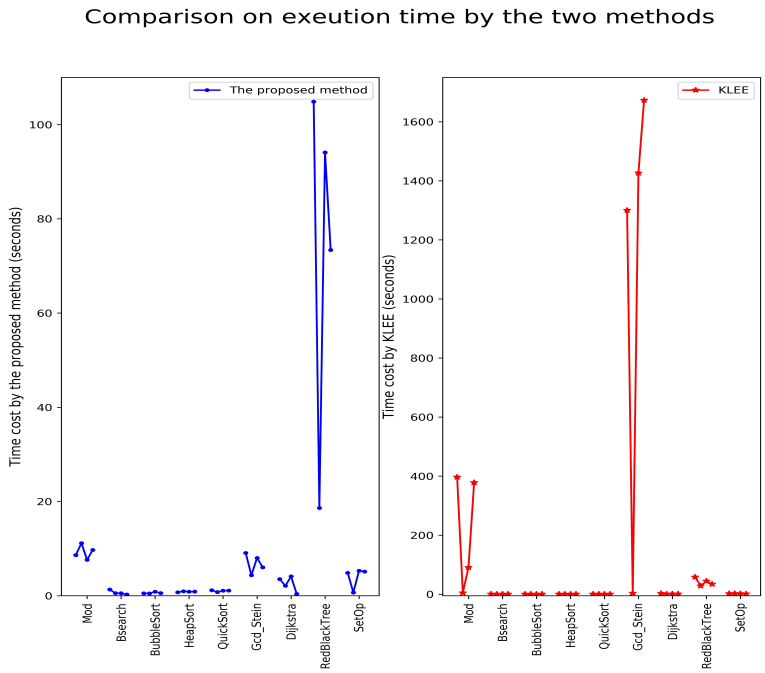


FIGURE 4.27: Testing time by the two methods

TABLE 4.4: Testing time by the two methods (Details).

Programs	**v0**	**v1**	**v2**	**v3**
Mod	8.61/396.64	11.14/4.73	7.61/90.66	9.72/377.90
Bsearch	1.30/0.52	0.52/0.29	0.53/0.37	0.25/0.25
BubbleSort	0.50/0.17	0.48/0.17	0.84/0.17	0.56/0.17
HeapSort	0.72/0.32	0.95/0.47	0.84/0.23	0.86/0.47
QuickSort	1.16/0.25	0.77/0.18	1.06/0.18	1.09/0.17
Gcd_Stein	9.08/1300.10	4.34/3.37	8.02/1426.33	6.02/1672.69
Dijkstra	3.52/2.52	2.11/0.53	4.09/1.14	0.39/0.57
RedBlackTree	104.88/58.22	18.625/28.34	94.09/44.62	73.38/33.90
SetOp	4.86/1.91	0.68/2.20	5.30/1.73	5.13/1.11
a/b. a: the proposed method; b: the KLEE method, **vi**: version i of the program.				

Apart from the above good points, our method is of flexibility in controlling the checking level for each control location by adding simple annotations, in contrast to KLEE. That can minimize the burden of path explorations and lead to an incremental testing. Also, our method is friendly to testers since it manipulates all the symbolic expressions in human-readable forms and the theorems are in Z3-style mathematical forms or integrated with programming language. Unlike our method, the KLEE tool performs the symbolic execution on LLVM bit-code

(not easily readable to humans) and consequently the way of negations for conditions to explore paths is quite different from the proposed method.

As formal specifications are not commonly used in most industries, the good experimental results for the proposed method also encourage programmers to write specifications for bug detection in practice. This can also help save testing time when the code is often modified based on the same specification to improve its efficiency.

4.4 Threats to Validity

We admit that the programs and mutants used in the experiments are of small scales. However, It is hard to find the real big software systems with both available code and the related formal specifications of pre-post style for every main process. This difficulty hinders the use of both methods properly and the comparison on their performance. We select a small but classic set of programs that differ from each other in several aspects. These programs cover various features of inputs, outputs, the code structures and also various complex formal specifications. The performance of the proposal can be further evaluated by a well-developed tool to support it in the future.

Moreover, the experimental results have shown that KLEE lacks the ability to do well with more complicated branch conditions and specifications for singular or multiple processes. For this reason, comparison of both methods on large scale programs cannot be done at present.

Since the experiments are all conducted in almost full automation for both methods, we have significantly reduced the human interference. It is a threat to validity that the formal specifications (written in SOFL language) are transformed to the program language in test by humans. We try to limit the threat by carefully implementing the program from the corresponding formal specification and all of the programs are well reviewed to ensure their quality before they are used for the experiment.

4.5 Conclusion

We propose a specification-based incremental testing method with symbolic execution (SIT-SE) to automatically verify the correctness of all the discovered representative program paths.

The approach is characterized by providing a grey-box testing resulted from an appropriate integration of formal specifications and the corresponding programs as well as an incremental testing strategy with flexibility to limit the cost of time.

In this approach, due to the formed theorem for encoding the correctness of the entire program path, the BSC algorithm adopted in the SIT-SE can considerably reduce the complexity of the path exploration, particularly for the program with complex nested loops.

Compared with the traditional symbolic execution techniques with SBT, our method SIT-SE has made new progresses by 1) offering a rigorous mechanism for proving the correctness of program paths, 2) and adding checking levels on branch conditions, 3) as well as developing the BSC algorithm to reduce the burden on exponential path exploration.

Chapter 5

TRIACFL: Triple Interaction-Based Fault Localization

In this chapter, a new method called TRIACFL (Triple Interaction-Based Fault Localization), is proposed to integrate the SIT-SE into further fault localization.

5.1 Principle of TRIACFL

To make use of the information on the correctness of program paths provided by the SIT-SE in fault localization, we propose a new fault localization approach, TRIACFL (Triple Interaction-based Fault Localization), featuring the integration of the three modules, the SIT-SE method, an elementary fault location generation algorithm, and an attentional shift-based review. In TRIACFL, four main steps are taken: 1) apply the SIT-SE to the implementation under test with the related formal specification to generate a test suite and symbolic paths with verification results; 2) if there is no false path found, finish the whole fault localization, otherwise, perform the elementary fault location generation algorithm on the results from the SIT-SE to produce a sequence of suspicious locations (statements and blocks); 3) proceed the attentional shift-based review to locate faults, mark the checked positions, as well as remove the faults if necessary; 4) go to 1) or 2) according to different situations.

Figure 5.1 shows the general framework of TRIACFL with more details. The tester, and three main modules including the SIT-SE (Module 1), elementary fault location generation algorithm (Module 2), and attentional shift-based review (Module 3, separated to 3.1 and 3.2),

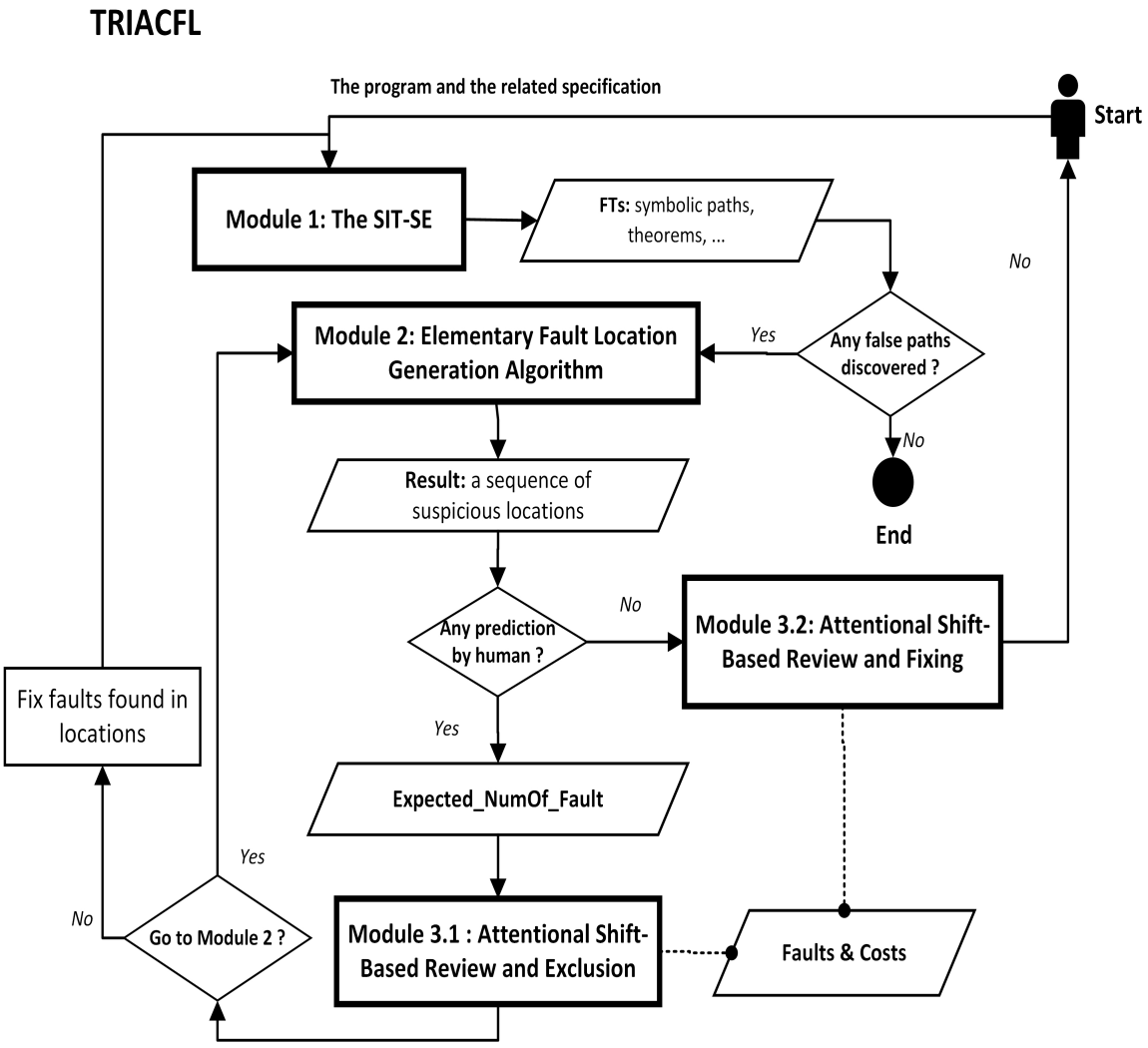


FIGURE 5.1: Framework of TRIACFL

collaborate closely with each other in a flexible way. The role of Module 1 is to test the program and send error information to Module 2. Module 2 analyzes the discovered false paths, and primitively produces a sequence of suspicious locations including branch conditions and code blocks. If the reviewer makes a prediction about the number (at least one) of faults that the program may contain, Module 3.1 is invoked with an expected number of faults, otherwise, Module 3.2 is executed. Both Modules 3.1 and 3.2 involve efforts made by the reviewer. The attention shifting mechanism in the review, taking as an argument the sequence of suspicious locations, would guide the reviewer to navigate the program to inspect most suspicious statements meanwhile mark checked positions for exclusion. In Module 3.1, if the reviewer

finds the expected number of faults, the faults found in locations are supposed to be removed and the revised program is to be tested again in Module 1. Otherwise, Module 2 is executed with the updated exclusion to produce a new sequence of suspicious locations. On the other hand, if no prediction is made, Module 3.2 suggests that a fault be removed once it is found by the reviewer, then the revised program will be sent again to Module 1. Finally, the fault localization will be over when there is no false path found by Module 1.

Due to the involvements of the prediction, inspection and fixing, these modules can be combined in mosaic ways. The circulation between Modules 2 and 3.1 (named circle 1) continually updates the sequence of suspicious locations by analyzing the same set of false symbolic paths form Module 1 and the iteratively increased exclusions for checked positions, until all the expected faults are found or no more fault can be found in this circle. As opposed to circle 1, the circulation between Modules 1, 2 and 3.2 (named circle 2) advocates pinpointing and removing the faults one by one, using different sets of false symbolic paths produced by Module 1. Both circulations can intertwine with each other through the prediction of flexibility from the reviewer. In circle 1, the fault localization may be of difficulty due to the mixed information of multiple faults, while in circle 2 extra costs will increase due to frequent executions of the program. In order to balance the trade of two circles, the prediction, made from the reviewer's experience or designed prediction strategies, can play an important role in coordinating both circles.

Spectrum-Based Fault Localization (SBFL) technique is one of the most popular approaches in this area. It can be automated to produce a list of ranked suspicious elements (usually, statements) by analyzing all the elements along paths. However, there may be too many statements at the same level of suspiciousness, which make the reviewer take much effort to inspect too many statements until pinpoint the faults. TRIACFL mitigates the burden of such labor costs because of the interaction between these modules. We will later give the descriptions for the elementary fault location generation algorithm, the attentional shift-based review, and the two main circles.

5.1.1 Elementary Fault Location Generation Algorithm

We categorize suspicious locations into two types, branch conditions and conditional blocks. For convenience, these locations are of following forms.

Definition 5.1.1 (Suspicious condition)**.** A suspicious condition is designated as $(l_C,)$, where l_C is the line number of symbolic condition C, indicating that condition C at line l_C may contain some fault.

Definition 5.1.2 (Suspicious conditional block)**.** A suspicious conditional block can be 1) designated as a basic block (l_C, I), where $I = 1$ if the symbolic condition C is evaluated to true or $I = -1$ otherwise, indicating that all the statements in "then" ($I = 1$) or "else" ($I = -1$) branch of condition C at line l_C may contain faults; or it can be 2) designated as $[(l_{C_i}, I_m), ..., (l_{C_j}, I_k)]$, the overlapped area between basic blocks.

The two types of fault locations play different roles in our approach. A suspicious code block, usually containing multiple suspicious lines, works as a guide to draw attention to the block after a certain branch choice, rather than requires an instant close inspection for all these suspicious lines. Such a code block (C, I) would become of smaller scale when the condition C locates more closely to the end of the program. Different from that of suspicious blocks, a suspicious condition calls for a close inspection of a certain line, which literally leads to take moderate effort.

A sequence of branch conditions (shorten as conditions) depicts the feature of input data along a path, conveying the abstract idea of the design from the beginning of the program. The faults right in the conditions can deteriorate substantial logic of the implementation, thus they influence the justifiability of the statements under these conditions. In our approach, suspicious branch conditions are supposed to be carefully checked by the order of their occurrences in executions, in order to ensure that the design of the structure is correct from up to bottom. Conversely, conditional code blocks (shorten as code blocks), each a set of all the statements after a certain condition affecting the computation of the output in detail, are inspected in a bottom-up way. This is because suspicious code blocks near the end of executions are of smaller scale of statements as well as are closer to the incorrect outcome against the specification.

We introduce two definitions before the description of the elementary fault location generation algorithm. Firstly, recall that a branch sequence from a symbolic execution of program is defined as:

$@l_1(1), \cdots, @l_m(1)$.

Where element $@l_i(1)$ is the control location l_i at the first occurrence along the program path, accompanied by a sign @, either + or -, indicating that the condition in l_i is evaluated to true(+) or false(-). Besides, such a $@l_i(1)$ can be symbolized to a predicate (over symbolic inputs), the symbolized condition in l_i along a certain path.

Then two definitions are given as follows.

Definition 5.1.3 (Reduced path-condition array)**.** A reduced path-condition array $RConds := (c_1, ..., c_i, ..., c_n)$, where each c_i $(i = 1, ..., n)$ is derived from symbolizing the ith element of a branch sequence in a symbolic execution.

Definition 5.1.4 (Reduced path array)**.** A reduced path array $PVec := (v_1, ..., v_i, ..., v_n)$, computed from a reduced path-condition array $RConds := (c_1, ..., c_i, ..., c_n)$, where $v_i = 1$ if c_i is evaluated to true, and $v_i = -1$ if c_i is evaluated to false.

Each reduced path array marks some key choices of branches along a symbolic path. All these arrays, together with the information of theorems for paths, are used to find suspicious fault locations.

Suspicious fault positions are checked and put in a sequence by the orders of occurrence along execution paths. In the process of finding a sequence of suspicious locations ordered by descending suspiciousness, several rules involve in making the sequence in the algorithm as follows.

1. A suspicious location excluded from the inspection should not be put in the sequence.
2. If a condition is not considered to be correct, the condition itself as a suspicious location should be put in the sequence and its related basic blocks are no longer considered to be put in the sequence.
3. If a suspicious code block $(C_r, 1)$ is found right behind the suspicious code block (C_l, I_l) in the current sequence, remove (C_l, I_l) from the sequence and put $(C_r, 1)$ in the sequence.

4. If a suspicious code block $(C_r, -1)$ is found right behind the suspicious code block (C_l, I_l) in the current sequence, keep both (C_l, I_l) and $(C_r, -1)$ in the sequence.

5. If a suspicious code block (C_r, I_r) is found right behind a suspicious condition in the current sequence, keep it in the sequence.

Rule 1) ensures that a suspicious location that is excluded by the tester should not be put in the sequence. Rule 2) suggests that a condition that is not considered to be correct will be inspected prior to the basic blocks of this condition. How to define an incorrect condition is later described in detail. Rule 3) is used to draw attention to the code block of smaller scale that is close to the incorrect outcome of the program. Particularly, if $(C_r, 1)$ is nested inside (C_l, I_l), it is reasonable to first look into $(C_r, 1)$, near the end of the execution and with fewer suspicious lines, rather than check the whole lines inside (C_l, I_l). Rule 4) is used to make a clear boundary for $(C_r, -1)$ by keeping (C_l, I_l) in the sequence, because "else" part of a condition is sometimes not explicitly specified by programmers. Rule 5) keeps both the suspicious condition and code block in the sequence if they are next to each other.

The fault localization algorithm with the SIT-SE is given in Algorithm 1, where all the rules are implemented in a well-organized structure. After checking all the positions along faulty paths, the algorithm produces a sequence of suspicious fault locations *Result*, in which each location features a dictionary of four elements. Two types of fault locations share first three features in the dictionary: "Fault", type of fault locations (a condition or block), "line", line number of branch condition, "Info", reduced path-condition. Additionally, a suspicious condition is with the forth feature "ToCheckLine", suggesting the way to check, while suspicious block is associated with other feature "ThenOrElse", suggesting it be a "then" or "else" branch. The value of its "ToCheckLine" will be assigned "self&up", which suggests that a reviewer should carefully check this condition itself and meanwhile pay attention to some few lines upon the condition ("up" region). This is because the code before the condition may affect the correctness of this condition. Note that there is no certain restriction to "up" region of a suspicious condition. The primary work on these suspicious conditions is to inspect the conditions themselves, after which a reviewer can pay attention to the code surrounding the suspicious conditions if no fault is found in conditions.

Algorithm 1 Elementary Fault Location Generation

Path:=[line, TorF] is the information for a path from SIT-SE. Where *line* is a unique identity number (line number) for each path, and *TorF* (valued True or False) indicates the verification result for the path.
RConds, a reduced path-condition array for a specific path.
PVec, a reduced path array for a specific path.
ExcluLines, an external list of excluded lines.
ExcluBlocks, an external list of excluded blocks.
INPUT:
FTs, the results obtained from SIT-SE, a list of triple (Path,RConds,PVec).
OUTPUT:
Result, a list of suspicious fault locations.

```
Elementary_Fault_location_Generation( FTs ):
    Result = []
    visited = [] #
    false_lines = [] # incorrect suspicious condition list
    for line, TorF, RConds, PVec in FTs:
        # Check lines on the faulty paths
       if TorF is False:
          for i in range(len(PVec)):
             if (line[i] not in visited) and (line[i] not in false_lines):
                if line[i] not in ExcluLines:
                   if is_Cond_false(RConds[i],PVec[i],FTs):
                      Result.append({'Fault': 'Cond','line':line[i],
                         'Info':RConds[i],'ToCheck':'self&up'}
                      false_lines.append(line[i])
                      visited.append(line[i]) #line visited
                if (line[i] not in false_lines) and
                     ((line[i],Pvec[i]) not in visited) and
                        ((line[i],Pvec[i]) not in ExcluBlocks):
                   if Pvec[i] > 0:
                      if Result != [] and Result[-1]["Fault"] == "Block":
                         Result.pop()
                         Result.append({'Fault': 'Block','line':Line[i],
                            'Info':RConds[i],'ThenOrElse':V[i]})
                      else:
                         Result.append({'Fault': 'Block','line':Line[i],
                            'Info':RConds[i],'ThenOrElse':V[i]})
                   else:
                      Result.append({'Fault': 'Block','line':Line[i],
                          'Info':RConds[i],'ThenOrElse':V[i]})
                   visited.append((line[i],Pvec[i])) # block visited
    return Result
```

The suspicious fault locations are defined by a sequence *Result* of conditions and basic blocks that are ordered by their descending suspiciousness. More details in dealing with *Result* are given in the process of human review.

In this algorithm, function *is_Cond_false* invoked at line 11 for Rule 2) is used to verify whether a branch condition on faulty paths is correct. This verification is mainly based on the following assumption.

Assumption 1 (Correct condition)**.** *A condition itself is considered to be correct if all the discovered paths going along 'then' (or 'else') branch of this condition have been verified to be correct by theorems of the SIT-SE, otherwise the correctness of the condition remains undetermined.*

As theorems are composed of branch conditions and specifications, their correctness can be deteriorated by any faulty branch conditions. On the other hand, a suspicious branch condition can be temporarily ruled out of doubt if itself (or its negation) makes all the corresponding paths correct. Note that such a branch condition can still be skeptical in a suspicious block even if it is not regarded as a singular suspicious condition.

In *is_Cond_false*, a symbolic condition on a false path can be identified as a suspicious one if its negation has occurred in a false path. We call it that this condition together with its evaluation value matches a false path. The pseudocode of *is_Cond_false* is given as follows.

```
1  # pre-condition: c is on a false path
2  is_Cond_false(c,v,FTs):
3    # look at the opposite branch
4    re_v = -v
5    Flag = False
6    for _, TorF, RConds, PVec in FTs:
7      if TorF is False:
8        for item in zip(RConds, Pvec)
9          # if (c,v) Matches a false path:
10         if (c, re_v) == item:
11           flag = True
12           return flag
13   return Flag
```

Since the proposed algorithm mainly works on the reduced path-conditions for all the paths, the complexity of the computation depends on the number of branch conditions located in the code and the number of examined paths generated from SIT-SE. Fortunately, both are relatively small against the number of total possible paths and the total lines of codes when the softwares under the test are of great complex and large scale.

5.1.2 Attentional Shift-Based Review

As mentioned previously, suspicious conditions and code blocks constitute a sequence of possible fault locations. They play different roles in the review process: suspicious conditions are mainly used for checking specific lines, while blocks are mainly used for attention shifting. We introduce two kinds of review, one is attentional shift-based review and exclusion in Module 3.1, another is attentional shift-based review and fixing in Module 3.2, respectively.

In Algorithm 2 of *attentional shift-based review and exclusion*, the elementary fault location generation algorithm first suggests a sequence of suspicious fault positions, then *Review_Exclu_phase* comes to further check the code with a prediction (expected number of faults) and give feedback to Algorithm 1. Repeatedly, Algorithm 1 returns a new sequence of suspicious ones with the feedback, and again calls for a review process. This kind of interaction aims to continually shrink the scale of suspicious positions and guide a more precise analysis for the cause of the faults within moderate effort.

The suspicious locations in *Result* is scanned and checked sequentially. When it comes to a suspicious block, the block is supposed to be carefully inspected by the reviewer in *inspect_line*1 if it is not excluded and can be straight-checked from *straight_check*. A block is called *straight-checked* if the body of the block only contains unchecked simple assignments. This kind of block is of simple form without conditional branches inside, thus it relieves the stress of careful inspection. Otherwise, the block is used for attention shifting that requires no inspection for its whole body. Different from suspicious blocks that are usually used for orienting attention, suspicious conditions are supposed to be checked immediately due to their short form of fewer lines. In *inspect_line*1, the review finishes if all the expected number of faults are found. The costs for all the faults are recorded in a dictionary

Algorithm 2 Attentional Shift-Based Review and Exclusion (Circle 1)

ExcluLines, an external list of excluded lines.
ExcluBlocks, an external list of excluded blocks.
INPUT:
Result, a list of suspicious fault locations.
Expected_NumOf_Fault, the expected number of faults (from the prediction).
OUTPUT:
FaultCosts, an external dictionary of form {FAULT_LOC: Times,...}.

```
ToTalCost = 0
Times = 0
Num_fault = 0
Review_Exclu_phase( Result, Expected_NumOf_Fault ):
    for susLoc in Result:
        if susLoc is a Block:
            if susLoc not in ExcluBlocks and
                    straight_check(susLoc):
                for line in susLoc.body:
                    inspect_line1(line)
                # inspect the condition (head) in the block
                inspect_line1(susLoc.head.cond)
                put susLoc in ExcluBlocks
            else:
                shift attention to susLoc
        else: # a condition
            inspect_line1(susLoc)

    new_Result = Elementary_Fault_Location_Generation(Result)
    if new_Result differs from Result:
        Review_Exclu_phase( new_Result, Expected_NumOf_Fault )
    else:
        EXIT Review_Exclu_phase


straight_check (susLoc):
    Ignore excluded lines and blocks in susLoc
    return True if input(is_plain) else False

inspect_line1(loc):
    if loc not in ExcluLines:
        if  loc is input(faulty): # by human
            Num_fault += 1
            FaultCosts.append({loc: Times+1})
            if Num_fault == Expected_NumOf_Faults:
                ToTalCost = sum(FaultCosts.items())
                put loc in ExcluLines
                END Review_Exclu_phase
            else:
                Times = 0
                put loc in ExcluLines
        else:
            put loc in ExcluLines
            Times += 1
```

$FaultCosts = \{FAULT\_LOC : Times, ...\}$ where *Times* for each fault $FAULT\_LOC$ is the cost of inspection for lines between the previous recognized fault and the current recognized fault.

When it comes to evaluate the performance of our approach, we mainly consider the cost of finding all the faults, which is influenced by the positions of these fault locations in the sequence $Result$. We record the time cost in *Times* for each fault once the fault is found. The number of suspicious locations and statements will be counted into *Times* until a fault location is encountered. For the careful inspection for a block, the head of the block, a condition, is supposed to be checked after the entire body of the block is checked. In the counting, any suspicious block will be ignored if it is rather to draw attention than to require a careful inspection (i.e., every statements inside the block are carefully checked). The total time cost of finding all the faults for a program mutant is calculated by the formula as follows.

In the proposed approach TRIACFL:

$TotalCost = \sum_{fa \in FAULT\_LOCS} FaultCost[fa]$

Since Algorithm 2 manipulates *Result* based on the same set of false paths from the SIT-SE, it saves the execution cost of the program during the fault localization.

Another kind of review in Module 3.2, called *attentional shift-based review and fixing*, is described in Algorithm 3.

The procedures in this review are similar to Algorithm 2, except the inspection procedure *inspect_line*2. In *inspect_line*2, a fault is supposed to be removed once it is found. After fixing the faulty line, we again apply the SIT-SE (Module 1) and elementary fault location generation algorithm (Module 2) to the revised program and obtain new *Result* for the review. The circle of Modules 1, 2 and 3.2 is repeatedly executed, until no more new false path found in Module 1. This kind of review requires frequent executions for the program when there are more than one fault in the program. Despite such execution cost, we can benefit from that it provides more precise information of fault locations by removing the faults one by one.

Next, we use a case study to demonstrate how TRIACFL works, as well as compare the approach with the out-of-the-state fault localization technique called spectrum-based fault localization (SBFL). The case study is separated to two parts. First we experiment with several mutants of which each with single fault, then we conduct a multiple fault experiment to show how the approach effectively pinpoint all the faults.

Algorithm 3 Attentional Shift-Based Review and Fixing (Circle 2)

CODE, SPECS, the code under test and the related specification.
INPUT:
Result, a list of suspicious fault locations.
OUTPUT:
FaultCosts, an external dictionary of form {FAULT_LOC: Times,...}.

```
ToTalCost = 0
Times = 0   # Time cost for each fault to be found

Review_fix_phase( Result ):
    # no new fault from SIT-SE
    while( no_new_fault_found):
        Review_fix( Result )

Review_fix( Result ):
    For susLoc in Result:
        If susLoc is a Block:
            If susLoc not in ExcluBlocks and
                    straight_check(susLoc)
                for line in susLoc.body:
                    inspect_line2(line)
                inspect_line2(susLoc.head.cond)
                put susLoc in ExcluBlocks
            else:
                shift attention to susLoc
        else: # a condition
            inspect_line2(susLoc)

#if no bugs detected by SIT-SE, do:
ToTalCost = sum(FaultCosts.items())
END_ALL

inspect_line2(loc):
    if Loc not in ExcluLines:
        if  loc is input(faulty): # by human
            FaultCosts.append({loc: Times+1})
            Times = 0
            put loc in ExcluLines
            Fix loc in the CODE # by human
            FTs = SIT_SE(CODE,SPECS)
            Result = Elementary_Fault_Location_Generation(FTs)
            Review_fix_phase(Result)
        else:
            put loc in ExcluLines
            Times += 1
```

5.2 Case Study

In this section, we apply TRIACFL to program mutants with single fault. Since there is only one fault in each mutant, we perform Algorithm 2 in the work.

Again recall that process *Mod* is to find the quotient q and remainder r from dividing y by x. The function *Abs* computes the absolute value of an input, would be invoked by *Mod*. Here is the implementation of *Mod* and *Abs* in programming language Python.

```
1   def Mod (y, x):
2       r = y;
3       q = 0;
4       if y!=0:
5           if x*y > 0:
6               while Abs(x)<=Abs(r):
7                   r = r - x
8                   q = q + 1
9           else:
10              while x*r < 0:
11                  r = r + x
12                  q = q - 1
13      return r, q
14  def Abs(x):
15      if x>=0:
16          return x
17      else:
18          return -x
```

There is a total of five control locations that locate in line 4, 5, 6, 10 and 15, respectively.

```
 4. if x>=0:                # in Mod
 5. if y!=0:                # in Mod
 6. if x*y > 0:             # in Mod
10. while Abs(x)<=Abs(r):   # in Mod
15. while x*r < 0:          # in Abs
```

As the bottom-up verification process in SIT-SE suggests, the low-level process *Abs* is supposed to be tested prior to that in *Mod*. Since *Abs* is of a simple structure, we suppose *Abs* has been checked and its correct version is used in *Mod* under test. Therefore the code in *Abs* is always excluded from suspiciousness, although they will participate in the derivation of symbolic paths in testing *Mod*.

5.2.1 Step-by-Step Analysis

To work on fault localization, we start with two versions of *Mod* (also called program mutants [97]) by injecting faults into two types of positions, a condition and an assignment in a block, respectively. Two program mutants are shown in Table 5.1.

TABLE 5.1: Two program mutants for fault localization

Mutants	**Line (No.)**	**Fault in statements**
1	6	while Abs(x) $<$ Abs(r):
2	8	q = q - 1

We obtain the first program mutant by modifying the statement at line 6 "while Abs(x) $<=$ Abs(r):" to the wrong one "while Abs(x) $<$ Abs(r):".

First of all, apply the SIT-SE method to mutant 1, and all the generated test cases are displayed in Table 5.2. Table 5.3 shows the related information of paths, where the column *Th* represents the validity of theorems, and *T* and *F* denote true and false, respectively.

TABLE 5.2: Test data generated for mutant 1 by SIT-SE

Path	**Test data: (y,x)**
1	(7,-2)
2	(0,1)
3	(-6,-16)
4	(7,6)
5	(1,1)
6	(-8,-3)

From the results of bug detection, we can see all the paths with incorrect theorems have passed through the faulty statement "while Abs(x) $<$ Abs(r)".

Four incorrect paths with false theorems are founded, and all the underlined statements from the four faulty paths may contain some faults, as displayed in Figure 5.2. Since *Abs*

TABLE 5.3: The results by applying SIT-SE to mutant 1

Path	RConds	PVec	Th
1	$(y! = 0, x * y > 0, x * y < 0)$	(1, -1, 1)	T
2	$(y! = 0,)$	(-1,)	T
3	$(y! = 0, x * y > 0, x >= 0, -x < -y)$	(1, 1, -1, -1)	F
4	$(y! = 0, x * y > 0, x >= 0, x < y)$	(1, 1, 1, 1)	F
5	$(y! = 0, x * y > 0, x >= 0, x < y)$	(1, 1, 1, -1)	F
6	$(y! = 0, x * y > 0, x >= 0, -x < -y)$	(1, 1, -1, 1)	F
Note: $y! = 0, x * y > 0, -x < -y, x < y, x * y < 0$ is from lines 4, 5, 6, 6, 10 in Mod, respectively; $x >= 0$ is from line 15 in Abs.			

has been excluded from suspiciousness, there are actually 7 statements that probably contain faults.

Suspicious statements and paths

```
def mod (y, x):
  r = y;
  q = 0;
  if y!=0:
    if x*y > 0:
      while Abs(x)<Abs(r):
        r = r - x
        q = q + 1
    else:
      while x*r < 0:
        r = r + x
        q = q - 1
  return r, q
```

Path 3 : ('y != 0', 'x*y > 0', 'x >= 0', '-x < -y')
PVec: (1,1,-1,-1)

Path 4: ('y != 0', 'x*y > 0', 'x >= 0', 'x < y')
PVec: (1,1,1,1)

Path 5: ('y != 0', 'x*y > 0', 'x >= 0', 'x < y')
PVec: (1,1,1,-1)

Path 6: ('y != 0', 'x*y > 0', 'x >= 0', '-x < -y')
PVec: (1,1,-1,1)

FIGURE 5.2: Incorrect paths in mutant 1 by SIT-SE

In order to further narrow down the scale of suspicious statements, we apply the proposed fault localization algorithm to these results from SIT-SE.

Initially, the sequence of suspicious fault locations *Result* is set to be empty.

Start with the path 3 that has a false theorem. The first element of its reduced path-condition array *RConds* is $y! = 0$ from line 4 in *Mod*. According to the related reduced path array *PVec*, we can see that $PVec[0] = 1$, thus "then" branch of $y! = 0$ that is on the incorrect path would be suspicious. In Algorithm 1, either a condition itself or some branch of the

Step 1.
Check the reduced path-condition array of Path 3
Rconds:= ('y != 0', 'x*y > 0', 'x >= 0', '-x < -y')
PVec:= (1,1,-1,-1)

Condition	Matched Paths & Th & PVec	Indication	Suspicious location
1. 'y != 0'	Path 2 & T & (-1,)	'y != 0' → True	Block: ('y != 0' , 1)
2. 'x*y > 0'	Path 1 & T & (1,-1,1)	'x*y > 0' → True	Block: ('x*y > 0' , 1)
~~3. 'x >= 0'~~	~~-~~	~~-~~	~~-~~
4. '-x < -y'	Path 6 & F & (1,1,-1,1)	'-x < -y' → False	Cond: ('-x < -y', 6)

Step 2.
Check the reduced path-condition array of Path 4
Rconds:= ('y != 0', 'x*y > 0', 'x >= 0', 'x < y')
PVec:= (1,1,1,1)

Condition	Matched Paths & Th & PVec	Indication	Suspicious location
~~5. 'x < y'~~	~~Path 5 & F & (1,1,-1,1)~~	~~'x < y' → False~~	~~Cond: ('x < y', 6)~~

Ordered suspicious fault locations :
Block: ('x*y > 0' , 1), Cond: ('-x < -y', 6)
Note that Block: ('y != 0' , 1) is removed.

FIGURE 5.3: Fault localization for mutant 1

condition (a block) is put in the sequence of suspicious fault locations. The correctness of condition $y! = 0$ is supposed to be checked by *is_Cond_false*. After following the instruction from *Match*, we can find that "else" branch of $y! = 0$ is not on any incorrect paths, due to the fact that only path 2 contains this branch and it is with a true theorem. In addition, *Result* is empty, and $y! = 0$ is not a condition that has been executed for more than once. Therefore, $y! = 0$ is not a suspicious condition but "then" branch of $y! = 0$ is supposed to be put in the sequence. This means that, *Result*:= [{'Fault': 'Block', 'line':4, 'Info':y! =0, 'ThenOrElse':1}].

Next, the second condition on path 3 to check is $x * y > 0$ at line 5 in *Mod*. The analysis for this condition is almost the same as that for $y! = 0$. Because all the paths (only path 1 in this case) that have passed through "else" branch of this condition are correct, $x * y > 0$ is not seen as a single faulty condition and the block $(x * y > 0, 1)$ is supposed to be put in the sequence.

In the meantime, the previous block in the sequence should be removed according to Rule 3). Thus the attention on the code can be drawn to a smaller block in the review process. Now, we have *Result*:= [{'Fault': 'Block', 'line':5, 'Info':x * y>0, 'ThenOrElse':1}].

The third condition on path 3 is $x >= 0$ that is from function *Abs*. Since *Abs* has been excluded from suspiciousness, here comes to check the next condition $-x < -y$ on path 3. This time path 6 with a false theorem is found to pass "then" branch of $-x < -y$, thus $-x < -y$ as a suspicious condition is put in the sequence. We have

Result:= [{'Fault': 'Block', 'line':5, 'Info':x * y>0, 'ThenOrElse':1},

{'Fault': 'Cond', 'line':6, 'Info':-x < -y, 'ToCheck': 'self&up'}].

After all the conditions on path 3 have been checked, we turn to check incorrect path 4. Since the first three conditions on this path are the same as those on path 3 and have been checked, just skip to check the forth condition $x < y$ at line 6. However, the condition $-x < -y$ at the same line has been put in the sequence, thus condition $x < y$ at line 6 is not to be inspected and *Result* remain unchanged.

Figure 5.3 describes key steps of fault localization.

The elementary fault location generation algorithm gives one suspicious block and one suspicious condition. Although the block $('y! = 0', 1)$ is the head of *Result*, its main role is to shift the attention. In the human review process, the actual careful inspection order for the suspicious fault locations is shown in Figure 5.4. In fact, the actual fault locates in the condition "while Abs(x) < Abs(r):" at line 6 in *Mod*, which would be removed after the reviewer carefully inspect it.

Human review process.
1. Shift attention to Suspicious block ('x*y > 0' , 1)
   {'Fault': 'Block', 'line': 5, 'Info': x * y > 0, 'ThenOrElse': 1}
2. Inspect *Suspicious condition* ('-x < -y', 6)
   {'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
   The fault is found at line 6.

FIGURE 5.4: Review process: the inspection order for mutant 1

To sum up, in our approach, firstly, the SIT-SE method detected a total of 4 incorrect paths with 7 suspicious statements; then the elementary fault localization algorithm found 2 suspicious fault locations by mainly checking 3 symbolic conditions on 1 incorrect paths; finally,

the human review process suggested line 6 (with two suspicious conditions), where the actual fault locates, be the first one for careful inspection.

To make program mutant 2, we insert a fault into the original program by modifying the assignment "q=q+1" in line 8 to "q=q-1". Then we apply the interactive approach to mutant 2, showing how the approach finds the fault that is in a block but not in the branch condition.

The results of bug detection by applying SIT-SE to mutant 2 are displayed in Table 5.4 and Table 5.5.

TABLE 5.4: Test data generated for mutant 2 by SIT-SE

Path	**Test data: (y,x)**
1	(6,-2)
2	(0,1)
3	(-3,-5)
4	(1,1)
5	(2,4)
6	(-2,-1)

TABLE 5.5: The results by applying SIT-SE to mutant 2

Path	**RConds**	**PVec**	**Th**
1	$(y! = 0, x * y > 0, x * y < 0)$	(1, -1, 1)	T
2	$(y! = 0,)$	(-1,)	T
3	$(y! = 0, x * y > 0, x >= 0, -x <= -y)$	(1, 1, -1, -1)	T
4	$(y! = 0, x * y > 0, x >= 0, x <= y)$	(1, 1, 1, 1)	F
5	$(y! = 0, x * y > 0, x >= 0, x <= y)$	(1, 1, 1, -1)	T
6	$(y! = 0, x * y > 0, x >= 0, -x <= -y)$	(1, 1, -1, 1)	F

There are two incorrect path 4 and path 6 with false theorems. The faulty paths made of underlined statements are described in Figure 5.5. Similar to the case of mutant 1, there are totally 7 statements that may contain the fault.

In order to reasonably further analyze all these paths and their theorems, we apply the proposed fault localization algorithm to these false paths. The algorithm mainly works on the two faulty theorems and their paths.

The procedure of applying the algorithm is described in Figure 5.6.

Because of Rule 3), the previous suspicious blocks in the sequence are removed, thus finally, *Result*:= [{'Fault': 'Block', 'line':6, 'Info':-x <= -y, 'ThenOrElse':1}. This suspicious block is actually where the fault "q=q-1" locates, which is also the first one to be inspected in the human review process, as shown in Figure 5.7.

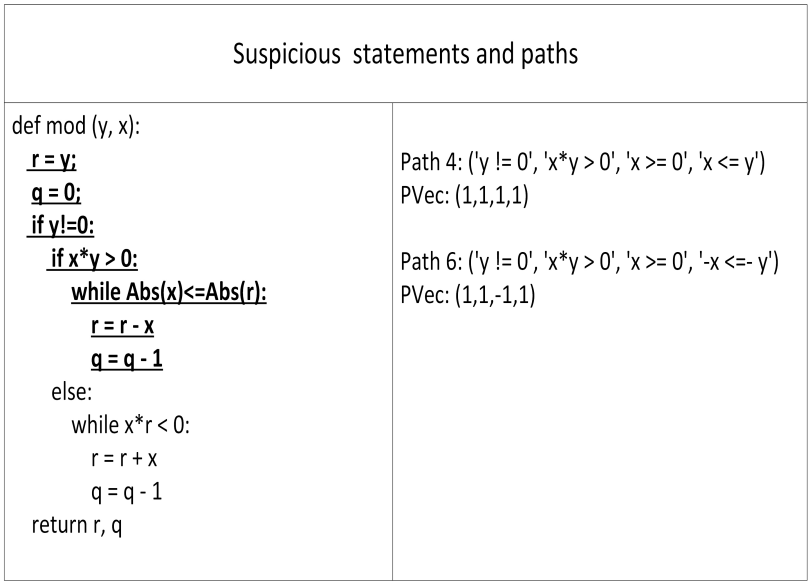


FIGURE 5.5: Incorrect paths in mutant 2 by SIT-SE

To sum up, in the interactive approach, firstly, the SIT-SE method detected a total of 2 incorrect paths with 7 suspicious statements; then the attentional shift-based review together with the fault location generation algorithm found 2 suspicious fault locations by mainly checking 4 symbolic conditions on 2 incorrect paths; finally, the human review process suggested a block starting from line 6, where the actual fault locates, be the first one for careful inspection. By using the approach, 7 suspicious statements found by the SIT-SE are reduced to only 2 statements. In the human review process, both suspicious statements are supposed to be inspected and the faulty one to be fixed.

5.2.2 Experimental Result with Single Fault

To evaluate the performance of our algorithm for fault localization, we compare it with the current technique of spectrum-based fault localization integrating with the SIT-SE (simply called SBFL-SSE).

For *SBFL-SSE*, 1) all the test data are generated the same way of the SIT-SE; 2) the pass/-failed information of a test data is determined by the validity of the related theorem, that is, a test data is failed if its theorem is proved to false or otherwise the test data is passed; we use three popular ranking metric *Tarantula* [54], *Ochiai* [56] and *Dstar* [98] to calculate the suspiciousness score for statements.

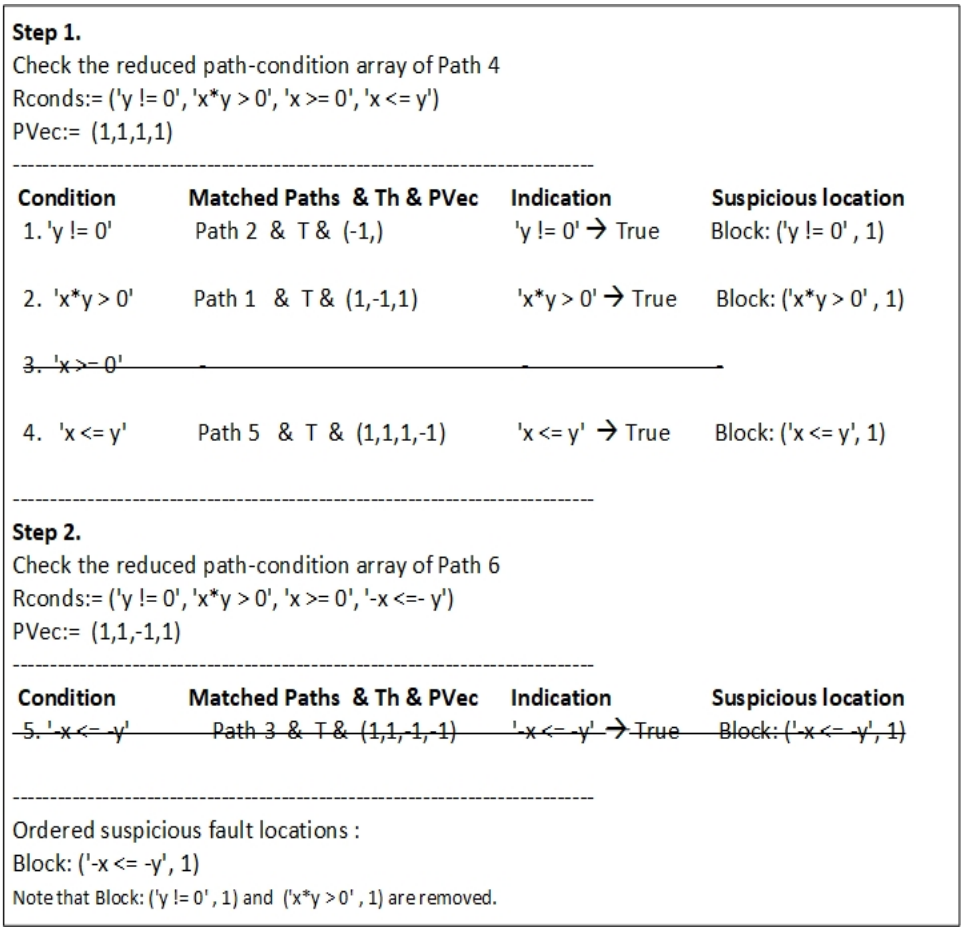


FIGURE 5.6: Fault localization for mutant 2

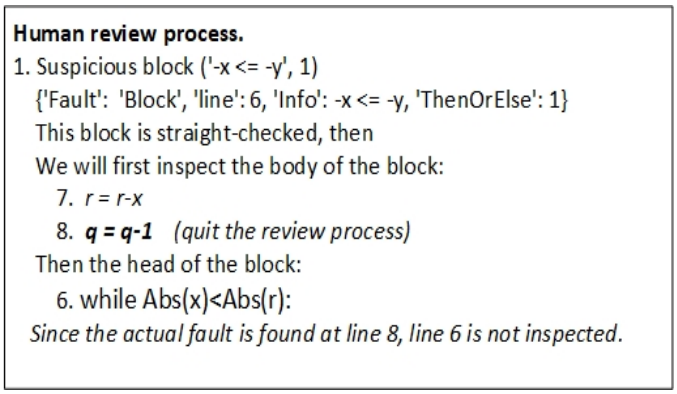


FIGURE 5.7: Review process: the inspection order for mutant 2

In comparison, the time cost for each recognized fault, also designated *Times*, is set to the rank of the faulty statement among all the suspicious statements. Thus for a program mutant,

its total time cost is calculated by the formula as follows.

In SBFL-SSE:

$TotalCost = \max\{fsta \rightarrow Times\}_{fsta \in Susp\_Stats} + NUM\_OF\_FAULTS - 1$,

where $Susp\_Stats$ is a set of ordered suspicious statements, $fsta \rightarrow Times$ is $Times$ for the suspicious statement $fsta$, and NUM_OF_FAULTS is the number of total faults in the program mutant.

For example, after applying SBFL-SSE to mutant 1, the rankings of suspicious statements are displayed in Table 5.6.

TABLE 5.6: Mutant 1: SBFL-SSE, using *Tarantula*

No.	Test data: (y,x)						S	R
line	(7,-2)	(0,1)	(-6,-16)	(7,6)	(1,1)	(-8,-3)		
2	+	+	+	+	+	+	0.5	5
3	+	+	+	+	+	+	0.5	6
4	+	+	+	+	+	+	0.5	7
5	+		+	+	+	+	0.67	4
6			+	+	+	+	1.0	1
7				+		+	1.0	2
8				+		+	1.0	3
9	+						0.0	9
10	+						0.0	10
11	+						0.0	11
12	+						0.0	12
13	+	+	+	+	+	+	0.5	8
P/F	P	P	F	F	F	F		

Where P denotes that the test data is passed, and F denotes that the test data is failed according to the theorems; the column S represents the suspiciousness score for the code lines, and the column R denotes the ranking of checking priority for the code lines that are likely to contain faults; $+$ means that the code line is covered by the test data.

In mutant 1, SBFL-SSE calculates the scores for all the 12 statements. The most three suspicious statements are line 6, 7 and 8. They are seen as the same level for inspection. Conversely, our approach mainly checks 3 lines (conditions) and precisely pinpoints the fault in the most suspicious location at line 6.

We make more single-fault mutants and apply both methods to them. Before the test, We assume that there is only one fault to find for both methods. Thus we apply *Review_Exclu_phase*

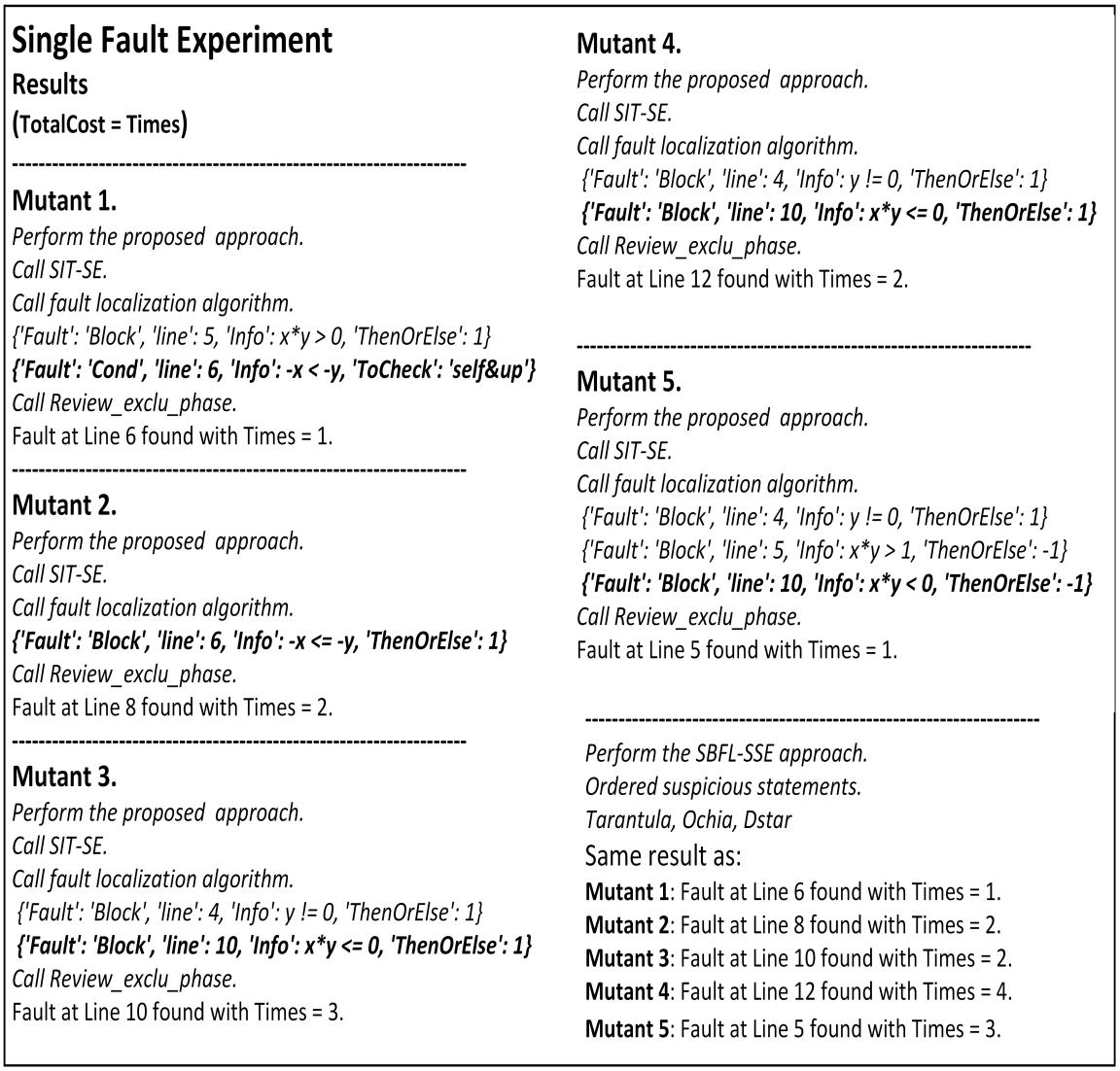


FIGURE 5.8: Costs by two methods in single fault experiment

of circle 1 to all these single fault versions to evaluate TRIACFL. The results from both methods are shown in Figure 5.8. Since there is only one fault in each mutant, the total cost for each mutant is just the value of *Times* for the first fault found. Three metrics, Tarantula, Ochiai and Dstar used in SBFL-SSE produce the same result, indicating there is no difference of performance between them in this experiment. Thus we mainly analyze the result from SBFL-SSE with Tarantula. Except the cost in mutant 3, all the costs from TRIACFL are not higher than that from SBFL-SSE.

Here is the explanation for the exception in mutant 3. When applying TRIACFL, the condition at line 10 is not recognized as a suspicious condition in the sequence. This is because is_Cond_false in the elementary fault location generation algorithm cannot conclude that this condition is incorrect by analyzing only 1 false path of total 7. Besides, the attentional shift-based review in TRIACFL always suggests that the body (line 11 and 12) of a block should be inspected before the head (line 10) of the block. On the other hand, SBFL-SSE assigns the same suspiciousness score to four statements at line 9, 10, 11 and 12, respectively. Although the inspection order of these statements is not clarified in most existing SBFL techniques, the statements with the same scores are inspected sequentially by their line number in SBFL-SSE. Thus line 10 with a fault is the second suspicious statement to inspect in SBFL-SSE, against that it is the third statement to inspect in our approach.

5.2.3 Evaluation and Summary

In the single fault experiment, we summarize the performance of two methods in terms of several aspects, as shown in Figure 5.9. To pinpoint the faults, TRIACFL costs the same or less time for inspection than that in SBFL-SSE in most cases. Additionally, TRIACFL outputs a much smaller scale of suspicious fault locations by analyzing smaller number of both lines and a small number of false paths, while SBFL-SSE produces much more suspicious locations (each with a score over 0.00) by analyzing and calculating suspiciousness scores for all the lines based on all the paths.

From the analysis of these results, our approach works more effectively and efficiently in fault localization, due to the integration of both the elementary fault location generation algorithm and the attentional shift-based human review. To sum, TRIACFL outperforms SBFL-SSE in the single fault experiment.

5.3 Experiment with Multiple Faults

We conduct a multiple faults experiment for both methods.

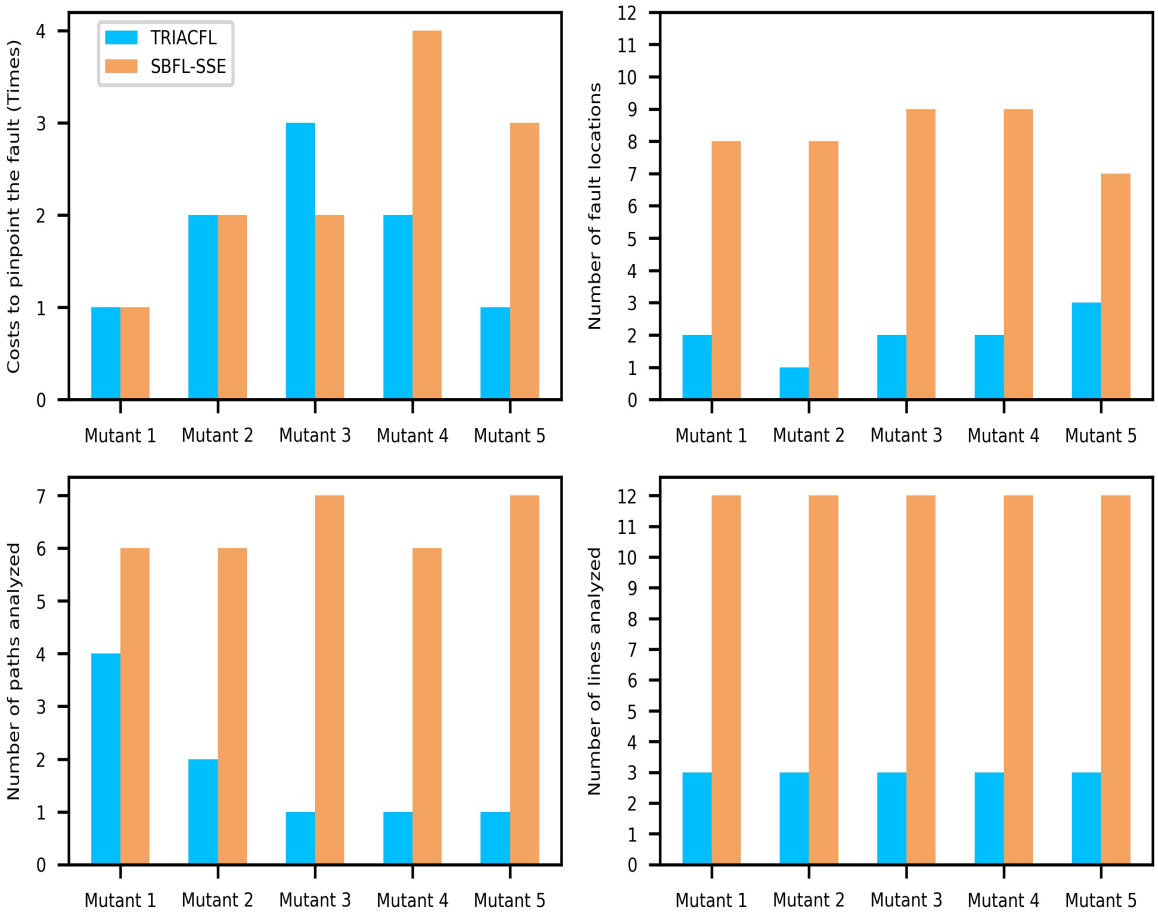


FIGURE 5.9: Evaluation for two methods in single fault experiment

5.3.1 Experiment Design and Preparation

Figure 5.10 describes the framework of the multiple faults experiment.

First of all, we prepare 10 program mutants for *Mod*, each with either 2, 3 or 4 faults in different locations in the code. In order to reduce the nondeterministic influence by the prediction in TRIACFL, we design two phases for TRIACFL, one implementing circle 1 of TRIACFL, and another implementing circle 2 of TRIACFL, as depicted in Figure 5.11 and 5.12. Circle 1 with a prediction (the number of inserted faults in a mutant) employs the attentional shift-based review and exclusion algorithm, executing the SIT-SE only once; Circle 2 employs the attentional shift-based review and fixing algorithm, considering no prediction from the reviewer.

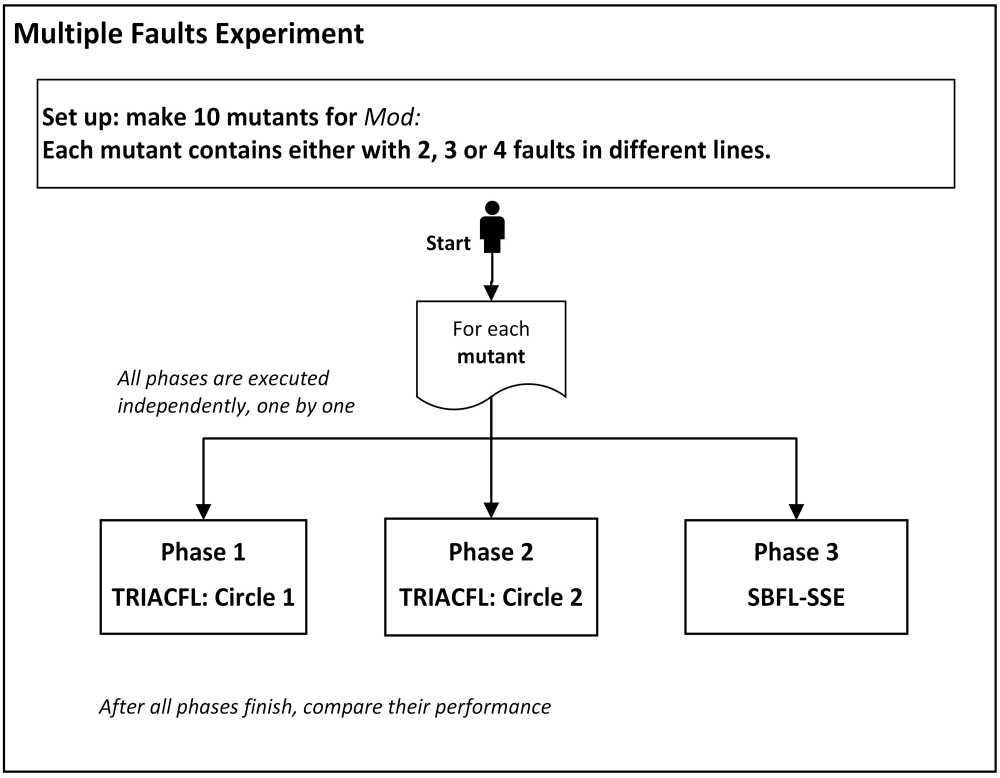


FIGURE 5.10: Design of a multiple faults experiment

As a comparison, phase 3 implements SBFL-SSE, which generates test data from the SIT-SE and then analyzes and calculates the suspiciousness scores for all the lines (statements) with metric formula Tarantula. Since both phase 1 and 3 apply the SIT-SE to each mutant once, we record the results from both phases in the same figures as a comparison. Then we mainly compare phase 1 and 2 to figure out how the prediction in phase 1 and the bug fixing in phase 2 affect the inspection costs in fault localization. Since there is no significant difference between the metric formulas Tarantula, Ochiai, and Dstar in SBFL, we consider reporting the results from Tarantula in this experiment.

All phases are executed independently, one by one. After all phases finish, compare their performance.

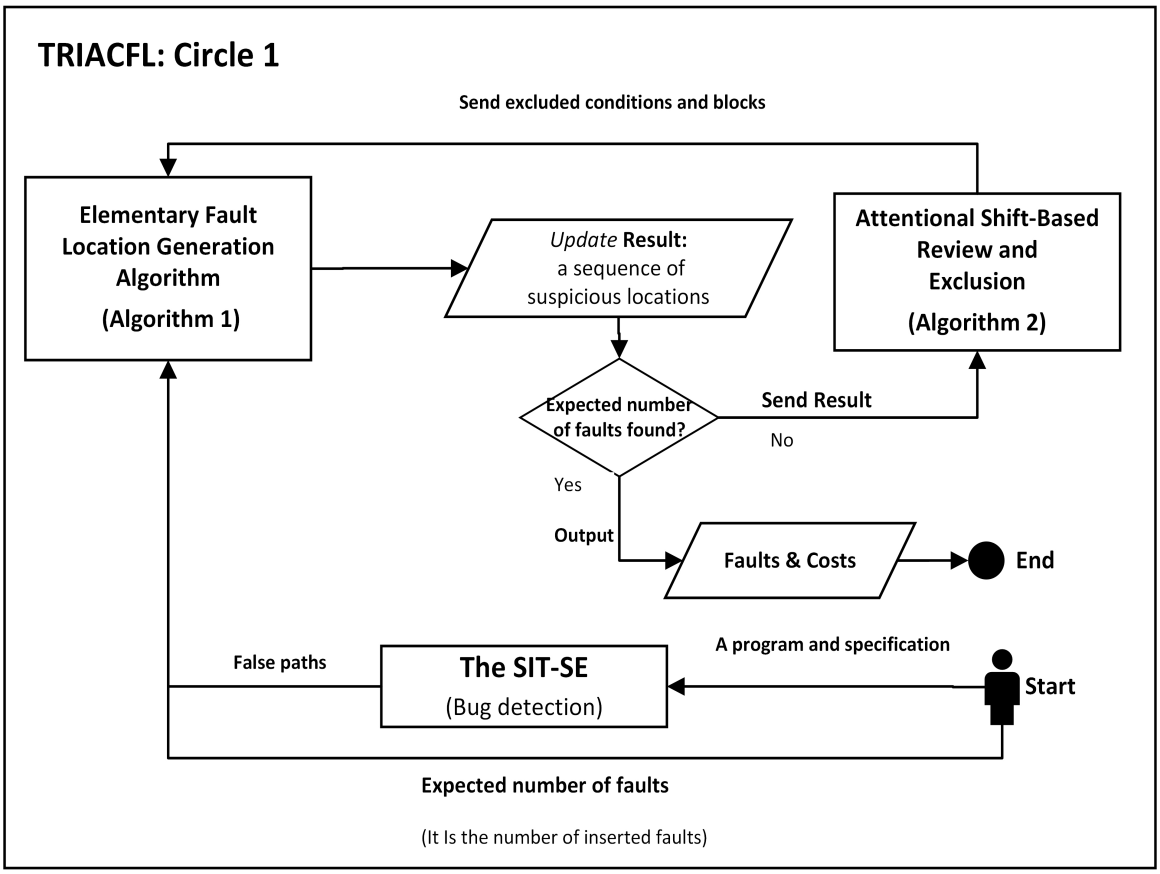


FIGURE 5.11: Circle 1 of TRIACFL

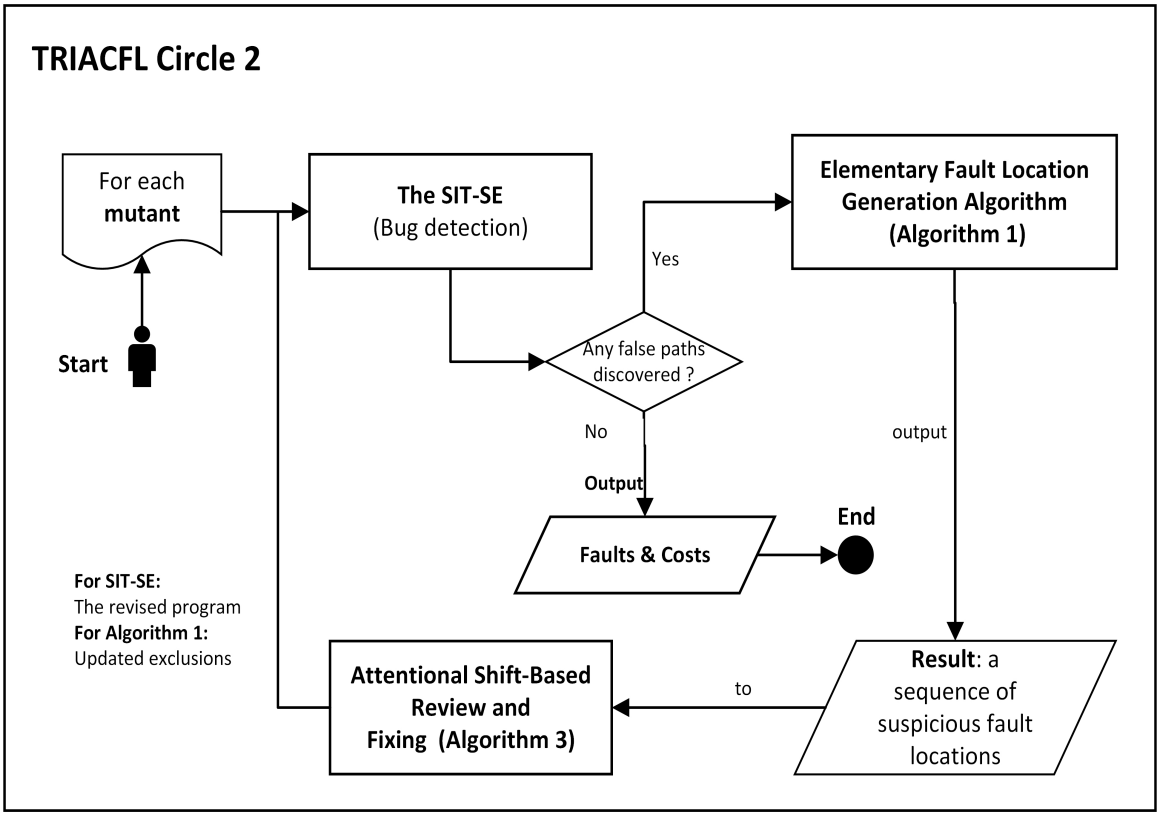


FIGURE 5.12: Circle 2 of TRIACFL

5.3.2 Experimental Result

The results from phase 1 and 3 are displayed in Figure 5.13, 5.14, 5.15, 5.16 and 5.17.

Mutant 1.
Perform Circle 1.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}
Call Review_exclu_phase.
Fault at Line 10 found with Times =1.
Call Review_exclu_phase.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Block', 'line': 10, 'Info': x*(y + x) <= 0, 'ThenOrElse': 1}
Call Review_exclu_phase.
Fault at Line 5 found with Times = 3.
Expected number of faults found.
TotalCost = 4.

Mutant 1.
Perform the SBFL -SSE.
Ordered suspicious statements.
line 9: 0.769
line 10: 0.769
line 7: 0.625
line 8: 0.625
line 11: 0.625
line 12: 0.625
line 5: 0.556
line 2: 0.500
line 3: 0.500
line 4: 0.500
line 13: 0.500
line 6: 0.357
Fault at Line 10 found with Times = 2.
Fault at Line 5 found with Times = 7.
TotalCost = 8.

Mutant 2.
Perform Circle 1.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_exclu_phase.
Fault at Line 12 found with Times = 3.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 6, 'Info': -x <= -y, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 0, 'ThenOrElse': -1}
Call Review_exclu_phase.
Fault at Line 8 found with Times =3.
Expected number of faults found.
TotalCost = 6.

Mutant 2.
Perform the SBFL -SSE.
Ordered suspicious statements.
line 7: 1.000
line 8: 1.000
line 9: 1.000
line 10: 1.000
line 11: 1.000
line 12: 1.000
line 5: 0.600
line 2: 0.500
line 3: 0.500
line 4: 0.500
line 6: 0.500
line 13: 0.500
End.
Fault at Line 8 found with Times = 2.
Fault at Line 12 found with Times = 6.
TotalCost = 7.

FIGURE 5.13: Results from phase 1 and 3: Mutant 1 and 2

Mutant 3.	**Mutant 3.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} {'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'} **{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}** **{'Fault': 'Block', 'line': 10, 'Info': x*(y + x) <= 0, 'ThenOrElse': 1}** *Call Review_exclu_phase.* Fault at Line 6 found with Times = 2. Fault at Line 10 found with Times =3. *Expected number of faults found.* **TotalCost = 5.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* **line 6: 1.000** line 7: 1.000 line 8: 1.000 line 5: 0.667 line 2: 0.500 line 3: 0.500 line 4: 0.500 line 13: 0.500 line 9: 0.286 **line 10: 0.286** line 11: 0.286 line 12: 0.286 Fault at Line 6 found with Times = 1. Fault at Line 10 found with Times = 10. **TotalCost = 11.**

Mutant 4.	**Mutant 4.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}** **{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}** *Call Review_exclu_phase.* Fault at Line 5 found with Times = 1. **{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}** *Call Review_exclu_phase.* Fault at Line 8 found with Times =2. *Expected number of faults found.* **TotalCost = 3.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* line 7: 1.000 **line 8: 1.000** line 6: 0.571 line 9: 0.571 line 10: 0.571 **line 5: 0.571** line 2: 0.500 line 3: 0.500 line 4: 0.500 line 13: 0.500 line 11: 0.000 line 12: 0.000 Fault at Line 8 found with Times = 2. Fault at Line 5 found with Times = 6. **TotalCost = 7.**

FIGURE 5.14: Results from phase 1 and 3: Mutant 3 and 4

Mutant 5.	**Mutant 5.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}** {'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': -1} **{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}** *Call Review_exclu_phase.* Fault at Line 5 found with Times = 1. Fault at Line 6 found with Times =1. *Expected number of faults found.* **TotalCost = 2.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* **line 6: 1.000** line 7: 1.000 line 8: 1.000 **line 5: 0.667** line 2: 0.500 line 3: 0.500 line 4: 0.500 line 13: 0.500 line 9: 0.286 line 10: 0.286 line 11: 0.000 line 12: 0.000 Fault at Line 6 found with Times = 1. Fault at Line 5 found with Times = 4. **TotalCost = 5.**

Mutant 6.	**Mutant 6.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} {'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1} {'Fault': 'Cond', 'line': 10, 'Info': x*y < 0, 'ToCheck': 'self&up'} *Call Review_exclu_phase.* *Exclude cond at line 10.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}** {'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': -1} *Call Review_exclu_phase.* Fault at Line 12 found with Times = 2. Fault at Line 5 found with Times =1. *Expected number of faults found.* **TotalCost = 3.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* line 9: 1.000 line 10: 1.000 line 11: 1.000 **line 12: 1.000** **line 5: 0.556** line 2: 0.500 line 3: 0.500 line 4: 0.500 line 13: 0.500 line 6: 0.000 line 7: 0.000 line 8: 0.000 Fault at Line 12 found with Times = 4. Fault at Line 5 found with Times = 5. **TotalCost = 6.**

FIGURE 5.15: Results from phase 1 and 3: Mutant 5 and 6

Mutant 7.
Perform Circle 1.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': x < y, 'ToCheck': 'self&up'}
Call Review_exclu_phase.
Fault at Line 5 found with Times = 1.
Fault at Line 10 found with Times = 1.
Fault at Line 6 found with Times =1.
Expected number of faults found.
TotalCost = 3.

Mutant 7.
Perform the SBFL -SSE.
Ordered suspicious statements.
line 6: 1.000
line 7: 1.000
line 8: 1.000
line 5: 0.667
line 2: 0.500
line 3: 0.500
line 4: 0.500
line 13: 0.500
line 9: 0.400
line 10: 0.400
line 11: 0.250
line 12: 0.250
Fault at Line 6 found with Times = 1.
Fault at Line 5 found with Times = 4.
Fault at Line 10 found with Times = 10.
TotalCost = 12.

Mutant 8.
Perform Circle 1.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 6 found with Times = 2.
Fault at Line 12 found with Times =2.
Checking 10 costs Times 1.
Exclude cond at line 5,6,10, block (line: 10,1)
Call fault localization algorithm.
{'Fault': 'Block', 'line': 5, 'Info': x*y > 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 6, 'Info': -x < -y, 'ThenOrElse': -1}
{'Fault': 'Block', 'line': 6, 'Info': x < y, 'ThenOrElse': 1}
Fault at Line 8 found with Times =2.
Expected number of faults found.
TotalCost = 6.

Mutant 8.
Perform the SBFL -SSE.
Ordered suspicious statements.
line 7: 1.000
line 8: 1.000
line 9: 1.000
line 10: 1.000
line 11: 1.000
line 12: 1.000
line 2: 0.500
line 3: 0.500
line 4: 0.500
line 13: 0.500
line 5: 0.444
line 6: 0.375
Fault at Line 8 found with Times = 2.
Fault at Line 12 found with Times = 6.
Fault at Line 6 found with Times = 12.
TotalCost = 14.

FIGURE 5.16: Results from phase 1 and 3: Mutant 7 and 8

Mutant 9.	**Mutant 9.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}** **{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}** **{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}** *Call Review_exclu_phase.* Fault at Line 5 found with Times = 1. Fault at Line 10 found with Times =1. Fault at Line 8 found with Times = 2. *Expected number of faults found.* **TotalCost = 4.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* line 7: 1.000 **line 8: 1.000** line 9: 0.667 **line 10: 0.667** **line 5: 0.571** line 2: 0.500 line 3: 0.500 line 4: 0.500 line 6: 0.500 line 11: 0.500 line 12: 0.500 line 13: 0.500 Fault at Line 8 found with Times = 2. Fault at Line 10 found with Times = 4. Fault at Line 5 found with Times = 5. **TotalCost = 7.**

Mutant 10.	**Mutant 10.**
Perform Circle 1. *Call SIT-SE.* *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}** {'Fault': 'Cond', 'line': 10, 'Info': x*y < 0, 'ToCheck': 'self&up'} **{'Fault': 'Cond', 'line': 6, 'Info': x < y, 'ToCheck': 'self&up'}** *Call Review_exclu_phase.* Fault at Line 5 found with Times = 1. Fault at Line 6 found with Times = 2. *Call fault localization algorithm.* {'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1} **{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}** **{'Fault': 'Block', 'line': 6, 'Info': -x < -y, 'ThenOrElse': 1}** {'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1} *Call Review_exclu_phase.* Fault at Line 12 found with Times =2. Fault at Line 8 found with Times =2. *Call Review_exclu_phase.* *Expected number of faults found.* **TotalCost = 7.**	*Perform the SBFL -SSE.* *Ordered suspicious statements.* **line 5: 1.000** **line 6: 1.000** line 7: 1.000 **line 8: 1.000** line 9: 1.000 line 10: 1.000 line 11: 1.000 **line 12: 1.000** line 2: 0.500 line 3: 0.500 line 4: 0.500 line 13: 0.500 Fault at Line 5 found with Times = 1. Fault at Line 6 found with Times = 2. Fault at Line 8 found with Times = 4. Fault at Line 12 found with Times = 8. **TotalCost = 11.**

FIGURE 5.17: Results from phase 1 and 3: Mutant 9 and 10

In these figures, for TRIACFL, the bold suspicious locations indicate that these locations are to be inspected, and the underlined suspicious locations indicate that these locations contain no faults after inspection and should be excluded from the sequence; for SBFL-SSE, the bold suspicious locations (lines) contain the inserted faults. In comparison with SBFL-SSE, circle 1 of TRIACFL produces fewer suspicious fault locations, as well as costs less time for inspection in all the multi-fault mutants.

Figures 5.18 and 5.19 display the results from applying circle 2 of TRIACFL to mutants, respectively. This circle demands for too many executions of the SIT-SE when there are too many faults in the program like mutant 10 with 4 faults. Although circle 2 doesn't require much effort made by human in the review, it may be inefficient due to frequently executing the SIT-SE after a fix when these faults locate near each other in the sequence of suspicious locations like that in mutant 7. Consequently, circle 2 generates more suspicious fault locations in total for all the rounds of restarting the bug detection with the SIT-SE in most cases.

Mutant 1.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}
Call Review_fix_phase.
Fault at Line 10 found with Times = 1.
fix up line 10.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': -1}
Call Review_fix_phase.
Fault at Line 5 found with Times =1.
fix up line 5.
Call SIT-SE.
End.
TotalCost = 2.

Mutant 2.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 12 found with Times = 3.
fix up line 12.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 6, 'Info': -x <= -y, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 8 found with Times =2.
fix up line 12.
Call SIT-SE.
End.
TotalCost = 5.

Mutant 3.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*(y + x) <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 6 found with Times = 2.
fix up line 6.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*(y + x) <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 10 found with Times =3.
fix up line 10.
Call SIT-SE.
End.
TotalCost = 5.

Mutant 4.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 5 found with Times = 1.
fix up line 5.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 6, 'Info': -x <= -y, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 8 found with Times =2.
fix up line 8.
Call SIT-SE.
End.
TotalCost = 3.

Mutant 5.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': -1}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
Call Review_fix_phase.
Fault at Line 5 found with Times = 1.
fix up line 5.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 5, 'Info': x*y > 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
Call Review_fix_phase.
Fault at Line 6 found with Times =1.
fix up line 6.
Call SIT-SE.
End.
TotalCost = 2.

Mutant 6.
Perform the interactive approach.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Cond', 'line': 10, 'Info': x*y < 0, 'ToCheck': 'self&up'}
Call Review_fix_phase.
No bug found.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 5, 'Info': x*y > 1, 'ThenOrElse': -1}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': -1}
Call Review_fix_phase.
Fault at Line 12 found with Times = 2.
Fault at Line 5 found with Times =1.
fix up line 5,12.
Call SIT-SE.
End.
TotalCost = 3.

FIGURE 5.18: Results from phase 2: Mutant 1-6 with 2 faults

Mutant 7.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': x < y, 'ToCheck': 'self&up'}
Call Review_fix_phase.
Fault at Line 5 found with Times = 1.
fix up line 5. Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 5, 'Info': x*y > 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 6 found with Times = 1.
fix up line 6. Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Fault at Line 10 found with Times =3.
End.
TotalCost = 5.

Mutant 8.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 6 found with Times = 2.
fix up line 6. Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 8 found with Times =2.
fix up line 8. Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 12 found with Times =2.
fix up line 12. Call SIT-SE.
End.
TotalCost = 6.

Mutant 9.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 10, 'Info': x*y <= 0, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 6, 'Info': x <= y, 'ThenOrElse': 1}
Call Review_fix_phase.
Fix up line 5. Call SIT-SE.
Call fault localization algorithm.
Fault at Line 5 found with Times = 1.
{'Fault': 'Block', 'line': 6, 'Info': -x <= -y, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 8 found with Times = 2.
Fix up line 8.
Call Review_fix_phase.
{'Fault': 'Block', 'line': 10, 'Info': x*y <= 0, 'ThenOrElse': 1}
Fault at Line 10 found with Times =4.
End
TotalCost = 7.

Mutant 10.
Perform circle 2.
Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 5, 'Info': x*y > 1, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 10, 'Info': x*y < 0, 'ToCheck': 'self&up'}
{'Fault': 'Cond', 'line': 6, 'Info': x < y, 'ToCheck': 'self&up'}
Call Review_fix_phase.
Fault at Line 5 found with Times = 1.
Fix up line 5. Call SIT-SE.
Call fault localization algorithm.
{'Fault': 'Block', 'line': 5, 'Info': x*y > 0, 'ThenOrElse': 1}
{'Fault': 'Cond', 'line': 6, 'Info': -x < -y, 'ToCheck': 'self&up'}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 6 found with Times =1.
Fix up line 6. *Call SIT-SE.*
Call fault localization algorithm.
{'Fault': 'Block', 'line': 6, 'Info': -x <= -y, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 8 found with Times = 2.
Fix up line 8. *Call SIT-SE.*
Call fault localization algorithm.
{'Fault': 'Block', 'line': 4, 'Info': y != 0, 'ThenOrElse': 1}
{'Fault': 'Block', 'line': 10, 'Info': x*y < 0, 'ThenOrElse': 1}
Call Review_fix_phase.
Fault at Line 12 found with Times =2.
Fix up line 12.
Call SIT-SE.End.
TotalCost = 6.

FIGURE 5.19: Results from phase 2: Mutant 7-9 with 3 faults, Mutant 10 with 4 faults

5.3.3 Evaluation and Summary

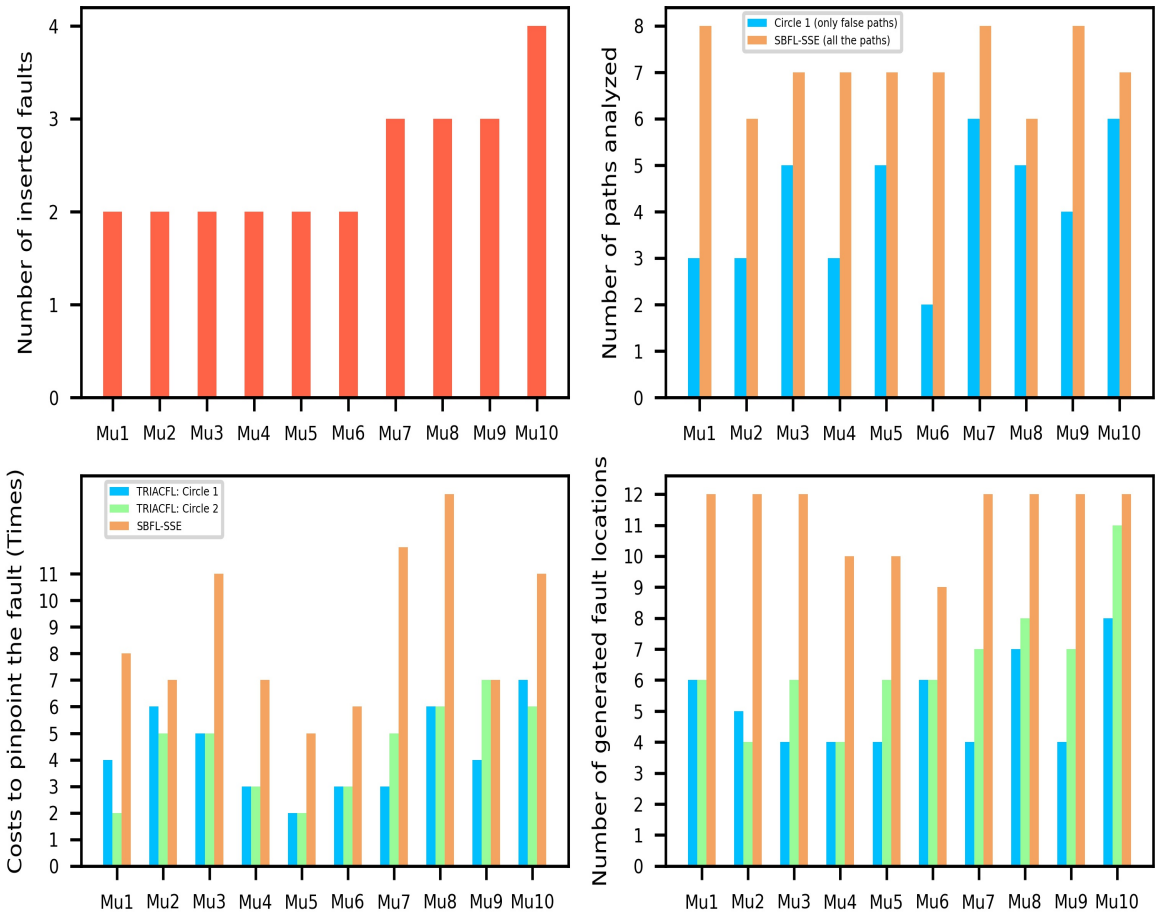


FIGURE 5.20: Performance of methods in multiple faults experiment

Since both circle 1 of TRIACFL and SBFL-SSE use one set of paths from SIT-SE for each mutant in fault localization, we compare them on the number of paths that need to be analyzed. Intrinsically, circle 1 of TRIACFL always analyzes branch conditions along all the false paths, while SBFL-SSE analyzes statements along all the paths. Thus circle 1 analyzes fewer paths than SBFL-SSE, as shown in Figure 5.20. Moreover, compared to SBFL-SSE, both circles 1 and 2 cost less time for inspection to pinpoint all the faults on the smaller scales of generated suspicious fault locations. Note that all the suspicious fault locations are counted every time the elementary fault location generation algorithm produces a sequence of them. Some suspicious locations, usually used for shifting attention, would be counted multiple times in

a mutant. Even if the elementary fault location generation algorithm would be performed multiple times on some mutants in both circle 1 and 2, our approaches still generate fewer suspicious fault locations than SBFL-SSE.

Comparing circle 1 and 2 of TRIACFL, we find that circle 1 costs a little more time in some two-fault mutants (mutant 1-2), but costs the same time as circle 2 in other two-fault mutants (mutant 3-6). In most three-fault mutants, circle 1 costs less time than circle 2 meanwhile analyzes smaller scales of suspicious fault locations. In mutant 10 with four faults, although circle 2 costs a little less time than circle 1, it has to generates and analyze much more suspicious fault locations due to more executions of both the SIT-SE and elementary fault location generation algorithm during the review and fixing. To sum, circle 1 takes the same labor costs for inspection as circle 2 in most cases, but circle 2 generally costs more execution (although they are automated) time than circle 1. It indicates that we would be better to make an appropriate prediction of the number of faults when applying TRIACFL in real world.

5.4 Conclusion

We proposed a fault localization method TRIACFL, which integrates three modules, the SIT-SE, the elementary fault location algorithm, and the attentional shift-based human review. Our approach provides a rigorous way to systematically analyze the correctness of key branch conditions and blocks based on the results from after applying the SIT-SE to programs.

We conduct a single fault experiment and a multiple faults experiment, respectively, to compare our approach with another method SBFL-SSE. SBFL-SSE is one of the existing popular technique SBFL families, which uses all the test cases generated by the SIT-SE. Both experimental results demonstrate that our approach can help testers to effectively pinpoint the fault on a smaller set of suspicious locations in the meantime inspect fewer lines of codes and fewer program paths. This sheds light on the effectiveness and efficiency of such approach when it comes to more complex and larger softwares.

Chapter 6

Supplement to the SIT-SE: A Mutated Specification-Based Approach with Genetic Algorithm

In this chapter, a mutated specification-based approach with genetic algorithm is proposed to mitigate the burden of test data generation in regression testing.

6.1 Genetic Algorithm (GA) with Mutated Specification

We first briefly introduce the basics of a genetic algorithm (GA) and then discuss how to obtain reformed specifications.

6.1.1 Description for GA

GA is a heuristic search method inspired from evolutionary biology and was first proposed by John Holland [99]. In general, a GA is involved in an iterative process with three steps: (i) create an initial population of solutions (called individuals) represented by a pre-defined chromosome that are typically encoded the solutions to a problem; (ii) in the existing population, select a group of individuals by a specified *selection* strategy based on a *fitness* function,

and generate the next population from applying two key genetic operators, *crossover* and *mutation* to those selected individuals; (iii) repeat step (ii) until the remaining individuals in the generation are good enough according to both the fitness function and the stopping criteria.

Since GA works well in finding optimal solutions for nonlinear problems and the specifications of pre-post style could be easily transformed to chromosomes with few efforts, we employ GA to find the best mutated specifications in this chapter. Later, we will first describe how to transform the original formal specifications into a chromosome, and then carefully describe the evolution in step (ii) in detail for our specific goal: to obtain all the mutated functional scenarios from the specification, each a constraint over only input variables.

6.1.2 Mutation Testing

Mutation testing, also called program mutation [92], is used to design test cases and evaluates the quality of existing testing techniques. In mutation testing, some small modifications are injected into the original program. Each mutated version is called *program mutant* and one test case is regarded as the good one if it kills the program mutants, that is, it makes the behavior of program mutants different from that of the original program.

In our approach, both programs and the specifications are mutated. The program mutants are used to evaluate the quality of mutated specifications. We search for good mutated specifications that can be used to effectively generate test data for bug detection.

6.1.3 Mutated Specifications

We use SOFL as the formal notation for specifications in this approach. There are two reasons: one is that SOFL, as a formal notation, is more comprehensible than other formal notations since it uses the comprehensible condition data flow diagrams for system structure as well as pre- and post- conditions for defining individual operations in the system; another reason is that SOFL is familiar to us and its use in industry has been increasing [100].

In SOFL, the *defining condition* describes the constraints over input and output variables after a method in the program performs. Generally, the defining condition is not used for directly generating input data in most of existing techniques because the values of outputs in defining condition are unknown to us before the execution of the program. We consider the

defining conditions as an important factor for test data generation from which the test data are sensitive to bad behaviors of the program.

Since defining conditions describe how output variables relate to input variables, they are often used to check whether an execution of the program is correct or not, rather than being used to directly generate input values. For a program, it is usually difficult to directly generate input values that satisfy a defining condition without knowing the corresponding output values. For instance, suppose input variable x and output variable y satisfy the defining condition ($x * y > x + y$), we cannot generate input x from ($x * y > x + y$) due to the unknown output y. Thus, usually ($x * y > x + y$) is not used to help generate the input values but can be used to check the result of executing the program with input x.

Nevertheless, by assigning appropriate values to the output variables in the defining conditions, we can get some useful mutated specifications that can be used to directly generate input values. For the defining condition ($x * y > x + y$) mentioned above, input data generated from ($x * 2 > x + 2$) (when $y = 2$) may be more likely to trigger bugs than from ($0 > x$) (when $y = 0$). Currently, it cannot be determined without further checking. However, ($x * 2 > x + 2$) is definitely better than ($x * 1 > x + 1$) (when $y = 1$) because the latter is always false and cannot be used to generate test data.

Our work mainly concentrates on developing a way to find appropriate output values for the specification. These output values are then used to build the mutated specifications that are the constraints over only input variables. Then, the mutated specifications can be directly used to generate input values in regression testing. To achieve that, we employ GA to seek such appropriate values of outputs from the defining condition.

Moreover, to obtain mutated specifications that are more powerful in bug detection, some extension is considered for reforming defining conditions before applying GA. In this extension, we make a slight change in defining conditions so as to induce the generated test data that satisfy those reformed ones to trigger as many bad behaviors of the program as possible.

In our method, *mutated specifications* are made from after applying GA to the original specification. More precisely, the mutated specifications can be obtained by following *two rules*:

1. Reforming the original specification by introducing dummy variables into defining conditions so that test data that satisfy those reformed ones can trigger bad behaviors of the

program;

2. Finding appropriate concrete values through GA and assigning them to the output and dummy variables that occur in the reformed specification.

Our goal is to obtain a new version of the specification from which the test suite can be generated to trigger as many bugs as possible in the program. Next, we will define the chromosome forms for the reformed specification, as well as describe the crossover operator and mutation operator. Then, we apply the GA for gaining the suitable mutated specifications that can do well in bug detection.

We define the form of chromosomes for a condition data flow diagram (CDFD) that is part of the SOFL language.

A condition data flow diagram (CDFD) is a directed graph that specifies how processes work together to provide functional behaviors [101]. Every process has its own pre- and post-conditions. For instance, Figure 6.1 displays a small CDFD that consists of two processes A and B where process A first consumes two input variables x and y and produces output z, and then process B consumes z and produces w.

The two separately defined processes A and B may not be automatically combined into a bigger process C since we can not always infer $C\_pre(x,y) \wedge C\_post(x,y,w)$ just from $A\_pre(x,y) \wedge A\_post(x,y,z) \wedge B\_pre(z) \wedge B\_post(z,w)$ unless we know the expression $z = Expr(x,y)$ in $A\_post(x,y,z)$, since, in that case, we can easily replace z with $Expr(x,y)$ and derive the following predicate expression:

$$C\_pre(x,y) \wedge C\_post(x,y,w) = A\_pre(x,y) \wedge A\_post(x,y,Expr(x,y)) \wedge$$
$$B\_pre(Expr(x,y)) \wedge B\_post(Expr(x,y),w).$$

However, the intermediate variables between two processes like variable z can not always be replaced in real CDFDs. Therefore, our discussion on test data generation from specifications focuses on a single process.

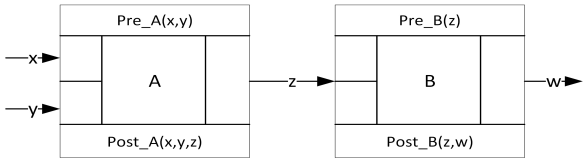


FIGURE 6.1: The process A and process B.

6.1.4 Chromosome Formation

In this approach, the specification is converted into an equivalent expression called *functional scenario form* (FSF). Currently, test data generation from a functional scenario only takes the test condition into account meanwhile leaving the defining condition untouched [27, 102, 33].

Now, we explain how to make a slight extension to change the form of defining conditions so as to allow bad behaviors to occur. To obtain a more flexible and useful reformed specification, we introduce *dummy variables*, $d_i (i = 1, \cdots, c)$, to the relationship of inputs and outputs from the defining condition. Then, we build an *output vector* from both the *dummy variables* and output variables.

Definition 6.1.1. An output vector $o^{'}$ is a vector constructed by output variables and dummy variables: $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$, where o_i $(i = 1, \cdots, n)$ are output variables, and d_i $(i = 1, \cdots, c)$ are dummy integer variables.

For example, let $f(inputs, outputs) \triangle 0$ (where $\triangle$ is a operator such as $=, >, <$ ...) be a relation in the defining condition D_i, by introducing dummy variables d_1 and d_2, we can construct an inequality $d_1 <= f(inputs, outputs) <= d_2$ and replace the relation $f(inputs, outputs) \triangle 0$ with this new inequality in D_i. Then, the output vector is formed as $o^{'} = (o_{1,} \cdots, o_n, d_{1,} d_2)$. In our work, we mainly make such change to only equality relations.

We change an equality relation to such an inequality relation because an equality relation is quite a strict condition that would drastically narrow down the exploration of input values for a single functional scenario when output values are determined by GA. Therefore, dummy variables need to be introduced for equality. For the inequality relation in the specification, dummy variables are not introduced to them because, compared with equality relation, inequality relation is not a too strict condition for the generation of input values. Thus, this kind

of relations are used to preserve some original features of specifications. In addition, the experimental results in Section 6.3 also indicate that additional dummy variables for inequality cannot help considerably improve the quality of the mutated specifications.

Definition 6.1.2. A chromosome $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$ is a reformed functional scenario $T_i \wedge D_i$, where some dummy variables are introduced to D_i. An individual (a mutated specification) is a constraint over symbolic inputs, established from the chromosome $[T_i \wedge D_i]_{o'}$ by assigning concrete values to the output vector $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$. A population is a group of such individuals. For convenience, the output vector $o^{'}$ is also called d-chromo, and each element of $o^{'}$ is called a genetic.

From this definition, a d-chromo $o^{'}$ with concrete values determines an individual formula $[T_i \wedge D_i]_{o'}$ that is a constraint on symbolic inputs. Such an individual is a reformed specification that can be used to generate test data for the program. Note that every time running the program on each test data, the correctness of the outcome is always verified by the original formal specification. In order to obtain good individuals to generate test data that are useful for bug detection, we apply the genetic manipulation to a group of individuals $[T_i \wedge D_i]_{o'}$ and find the appropriate d-chromo $o^{'}$. Each individual will be scored by evaluating the quality of the test data that are generated from it.

6.1.5 Genetic Manipulations and Selection

The genetic manipulation refers to the change of genetic structure in biology, but, in the GA, it indicates that a "child" solution is produced from a pair of "parent" solutions by using genetic operators like crossover and mutation.

In the existing population, a pair of individuals (solutions) are selected as parents to perform the *crossover* operator to produce their offspring. More specifically, as illustrated in Figure 6.2, first select two individuals $[T_i \wedge D_i]_{o_1'}$ and $[T_i \wedge D_i]_{o_2'}$ from the current population as parents and get their d-chromos $o_1^{'}$ and $o_2^{'}$, then swap each two genetics of the two d-chromos with possibility p $(0 < p < 1)$ to obtain two new individuals.

To perform the *mutation* operator, each genetic of an individual is mutated with possibility q $(0 < q < 1)$, as displayed in Figure 6.3. More clearly, for one individual $[T_i \wedge D_i]_{o'}$ with its d-chromo $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$, each genetic of it has the possibility q to be mutated:

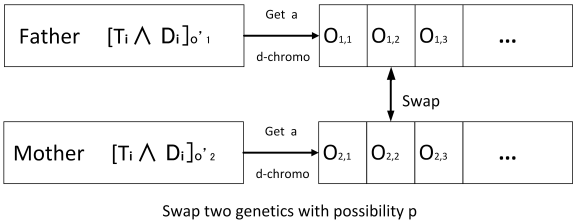


FIGURE 6.2: Crossover operator.

$o_i^{'} := o_i^{'} + \triangle$, where $\triangle$ is a different scalar of small value.

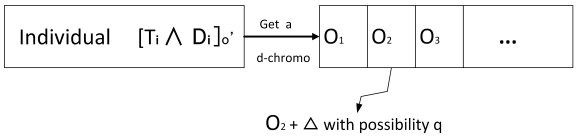


FIGURE 6.3: Mutation operator.

Fitness function *Grade* is used to evaluate an individual (a solution) $[T_i \wedge D_i]_{o'}$ by assigning a fitness value. Let $Datas = data\_suite\_from([T_i \wedge D_i]_{o'})$ where $data\_suite\_from$ generates a suite of input data from $[T_i \wedge D_i]_{o'}$ by using a constraint solver. Let $N\_kill_{i,o'} = (k_1, \ldots, k_m)$ where k_j is the number of test data that have been generated from $[T_i \wedge D_i]_{o'}$ and have killed the program mutant mu_j as well. A test case that kills a program mutant indicates that it fails based on the original specifications after it is executed by the program mutant. We consider both the killing rate of program mutants *Kill_rate*, and the killing rate of a data suite as important factors to compute the grade for $[T_i \wedge D_i]_{o'}$:

$$Grade([T_i \wedge D_i]_{o'}) = \frac{Kill\_rate(N\_kill_{i,o'}) \cdot Sum(N\_kill_{i,o'})}{(m \cdot (length(Datas) + 1))} \tag{6.1}$$

$$where \begin{cases} Kill\_rate(N\_kill_{i,o'}) = \dfrac{\Sigma_{j=1}^{m} I(k_j > 0)}{length(N\_kill_{i,o'})}, \\ I(k_j > 0) = \begin{cases} 1 & k_j > 0 \\ 0 & k_j \leq 0 \end{cases} \end{cases} \tag{6.2}$$

The factor $\Sigma_{j=1}^{m} I(k_j > 0)$ in *Kill_rate* is intended to encourage each mutated functional scenario to generate a test suite that can kill as many different kinds of program mutants as possible. The factor $Sum(N\_kill_{i,o'})$ as part of the numerator in *Grade* would inspire every

mutated functional scenario to generate a test data suite where most test data are effective enough to kill as many program mutants as possible. *Grade* is referred to the product of "the percentage of killed mutants per test suite" $Kill\_rate(N\_kill_{i,o'})/m$ and "the percentage of test cases that were able to kill mutants" $Sum(N\_kill_{i,o'})/length(Datas)$. To motivate each mutated functional scenario to produce a small scale of test suite *Datas* that is more capable of killing various program mutants, we add a penalty to *Grade* by using $length(Datas)+1$ instead of $length(Datas)$ in the denominator.

For a given chromosome $[T_i \wedge D_i]_{o'}$, its individual with appropriate d-chromo $o'_{i,best}$ is considered the best if and only if this individual possesses the highest value of *Grade* in the whole population. The GA is to find such best individuals for all these chromosomes $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$.

After all the individuals from the current population are evaluated, GA would select most of the best ones to form a new population for the next generation. This process is called *selection*. In our approach, we evaluate all the individuals and sort them by descending, then select individuals in the top 50 percent of the current population to breed the next generation.

As we can see, GA is used to find the best individuals separately for each chromosomes $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$. In order to evaluate all the best individuals represented by different chromosomes, the final formula of evaluation is made as follows:

$$Grade(\vee_{i=1}^{n}[T_i \wedge D_i]_{o'_{i,best}}) = \frac{Kill\_rate(N\_kill) \cdot Sum(N\_kill)}{(m \cdot length(Datass))} \tag{6.3}$$

$$where \begin{cases} N\_kill = \Sigma_{i=1}^{n} N\_kill_{i,o'_{i,best}}, \\ Datass = \{data\_suite\_from([T_i \wedge D_i]_{o'_{i,best}})\}_i \end{cases} \tag{6.4}$$

We use the final formula to find all the mutated functional scenarios that together hit the highest final grade (i.e., do the best in bug detection), each mutated one with well-tuned values for dummy variables and output variables. Additionally, this final grade is also used for comparison between our approach and other techniques. In the case study, our method is compared with the conventional specification-based method with respect to test data generation for bug detection.

6.2 Algorithm Summary

Our approach that incorporates GA accomplishes the goal of obtaining the mutated specifications by taking three key steps:

1. Inject faults into the original program to obtain a set of program mutants;
2. Use reformed specification $[T_i \wedge D_i]_{o'}$ as seed chromosomes. Each chromosome corresponds to a group of individuals that are generated by assigning concrete values to the output vector in the chromosome;
3. Apply the GA to each chromosome and select the best individuals (the best mutated specifications). According to the original specifications, determine whether or not a test case from a mutated specification (a constraint over inputs) kills the program mutants.

Figure 6.4 displays the whole evolution process of the GA. In the first round of evolution, a group of individuals are initialized and evaluated. Then, the best individuals in the top k ($k = 50\%$ in this chapter) of the group are selected to perform both crossover and mutation operators to produce their offspring for the next round. In the next round, all of the individuals are evaluated and the top k of them again prepare to breed a new generation by performing genetic manipulations. The population iteratively evolves in this process until there has been no improvement in the population or it reaches the predefined maximum number of generations.

In the mutation testing, we use Z3 [86], a widely used satisfiability modulo theories (SMT) solver, as our constraint solver to generate the data suite for each individual formula (i.e., each mutated functional scenario). The generation for a data suite takes three steps: (1) use Z3 to generate a test data that satisfies the logical formula, (2) exclude all the test data obtained previously from the logical formula; (3) go to step (1) to obtain another piece of test data unless enough test data are obtained. Each individual formula is evaluated by the fitness function that measures the quality of the test suite.

The pseudo-code of the algorithm is given in Algorithm 1.

Algorithm 1 GA to obtain mutated specifications.

Inputs: the functional scenarios from the specification: $T_i \wedge D_i$
Individuals: $o^{'} = (o_{1,} \cdots, o_n, d_{1,\cdots,}d_m)$ with concrete values
Outputs: the reformed specification $[T_i \wedge D_i]_{o'}$

```
run(){
    result = list()
    for [T_i ∧ D_i]_{o'} in functional scenarios:
        spec = T_i ∧ D_i
        population = initial(o')
        while(not enough iterations){
            one_step(spec)
      }
        best_individual = select_best_individual(population)
        reformed_specification = (spec, best_individual)
        result.append(reformed_specification)
}
one_step(spec) {
    # This function selects top 50% of the current population
    population = keep_good_individuals(population)
    do:
        father, mother = random_select_two(population)
        child1, child2 = crossover_operation(father,mother)
        child1, child2 = mutation_operation(child1,child2)
        population.put(child1,child2)
    until population increases enough
    do_valuation(population,spec)
}
do_valuation(population,spec){
    for individual in population:
        datas = data_suite_from(individual,spec)
        statistic_sum = kill_program_mutants(datas)
        individual.value = Grade(statistic_sum)
}
```

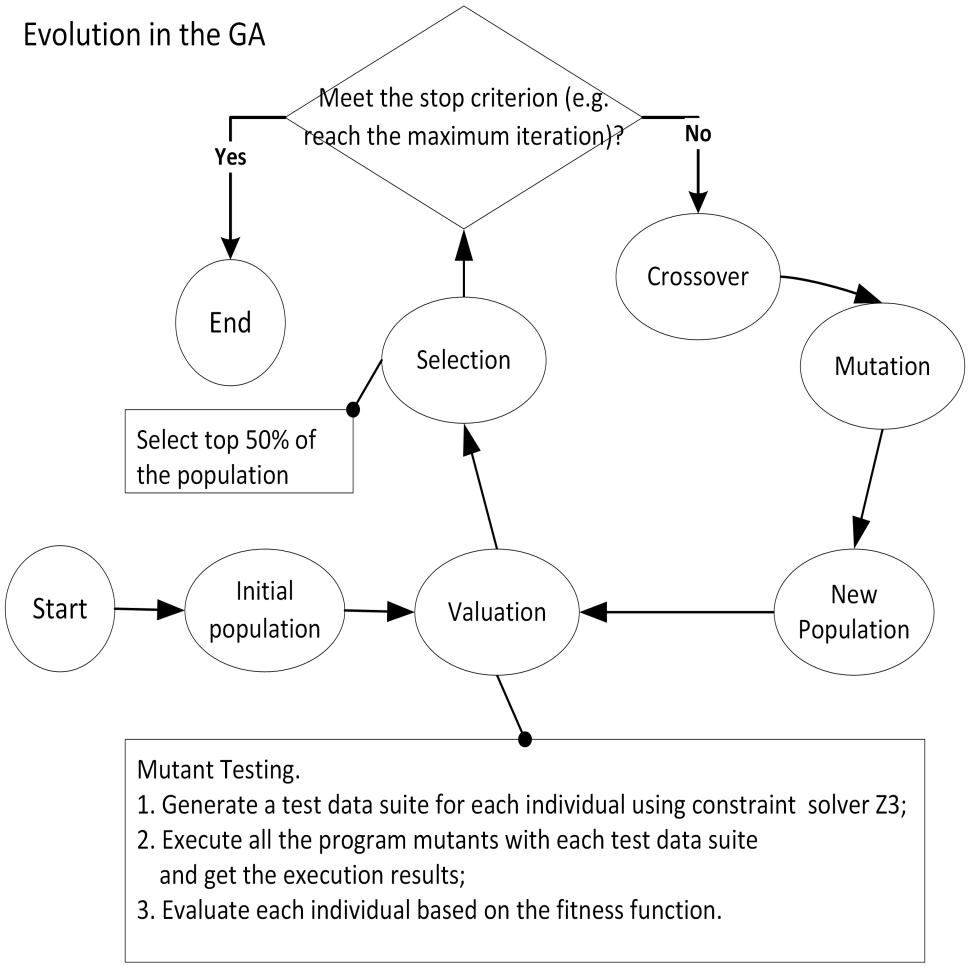


FIGURE 6.4: The evolution in GA.

6.3 Case Study

In this section, we apply the GA to two classic examples to demonstrate the effectiveness of the proposed method. The original specifications are used as test oracles for determining whether the outputs are correct or not during the evaluation of individuals.

We compare our method with the conventional method, called *original specification-based method*, which directly generates input data from the original specifications by using Z3. In the original specification-based method, neither dummy variables nor defining condition are used to generate input values, since the defining condition contains output variables with unknown values. The input data are directly generated from only test conditions (pre-condition and guard-condition over only input variables) by using Z3.

Sixteen program mutants are prepared for each program in the way that the injected faults will not cause execution crash and infinite loops since we only focus on the functional bugs in this research. Both methods generate a test suite of the same size 20 every time to execute these program mutants in the evaluation process. In configurations of the GA, we set the number of GA individuals to 20, the upper limit of the number of generations 100, the crossover rate 0.5, and the mutation rate 0.2.

6.3.1 Case Study 1: Mod

In this program, process *Mod* is to find the quotient q and remainder r from dividing y by x. For *Mod*, we give its formal specification in SOFL and the implementation in Python.

The formal specification of *Mod* is:

process Mod (y: int, x: int) r: int, q: int

pre $x \neq 0$

post $x > 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0 \vee$

$x < 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0 \vee$

$y = 0 \wedge q = 0 \wedge r = 0$

end_process

Its implementation in Python is:

```
def Abs(x):
```

```
    if x>=0:
        return x
    else:
        return -x
def Mod(y, x):
    r = y;
    q = 0;
    if y!=0:
        if x*y > 0:
            while Abs(x) <= Abs(r):
                r = r - x
                q = q + 1
        else:
            while x*r < 0:
                r = r + x
                q = q - 1
    return r, q
```

In this specification, *Abs* is a function for calculating the absolute value of its input. To shorten the explanation of each step, assume *Abs* is an inline executable predicate. Both $-7\ mod\ 5 = 3$ and $-7\ mod\ 5 = -2$ satisfy the classic definition $y = q * x + r \wedge Abs(r) < Abs(x)$. To avoid the ambiguity, the specification of *Mod* puts an additional condition $xr \geq 0$ in order to get only one result of $-7\ mod\ 5 = 3$.

In the specification, the pre-condition, guard-conditions, and defining conditions are listed as:

$S_{pre} := x \neq 0$;

$G_1 := x > 0 \wedge y \neq 0$; $D_1 := y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0$;

$G_2 := x < 0 \wedge y \neq 0$; $D_2 := y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0$;

$G_3 := x > 0 \wedge y = 0$; $D_3 := q = 0 \wedge r = 0$.

We can obtain the functional scenarios $T_i \wedge D_i := S_{pre} \wedge G_i \wedge D_i$ as follows:

$T_1 \wedge D_1 := x > 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0$;

$T_2 \wedge D_2 := x < 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0$;

$T_3 \wedge D_3 := x \neq 0 \wedge y = 0 \wedge q = 0 \wedge r = 0$.

For $T_3 \wedge D_3$, the input x and y are not related to the output q and r, so we do not need to apply GA to it. Since there is an equality $y = q * x + r$ in which inputs and outputs are related, we introduce two dummy variables d_1 and d_2. The chromosomes of *Mod* are displayed in Table 6.1.

TABLE 6.1: Chromosome forms for functional scenarios of process Mod.

Chromosome	D-Chromo	Dummy Vars
$[T_1 \wedge D_1]_{o'} : x > 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq d_2 \wedge$ $Abs(r) < x \wedge xr \geq 0$	$o' =$ (q, r, d_1, d_2)	d_1, d_2
$[T_2 \wedge D_2]_{o'} : x < 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq d_2 \wedge$ $Abs(r) < -x \wedge xr \geq 0$	$o' =$ (q, r, d_1, d_2)	d_1, d_2

Apply Algorithm 1 to these chromosomes. The results are displayed in Table 6.2.

TABLE 6.2: Results for process Mod after applying GA.

The Best Individual	Grade
$[T_1 \wedge D_1]_{o'}$	
$o' = (q, r, d_1, d_2)$	0.58
$o'_{1,best} = (-4, 0, -6, -6)$	
$[T_2 \wedge D_2]_{o'}$	
$o' = (q, r, d_1, d_2)$	0.55
$o'_{2,best} = (-7, 0, 9, 13)$	
$Total : \vee_{i=1}^{2} [T_i \wedge D_i]_{o'_{i,best}}$	0.59

To illustrate the effectiveness of data generation from the mutated specifications, Table 6.3 displays the results of the conventional method that generates the test data directly from the original specifications. For the original specification, we generate test data only from the test condition T_i consisting of both pre-condition S_{pre} and guard-condition G_i meanwhile ignoring the defining condition D_i because the defining condition D_i involves unknown output variables that can not directly help to generate test data.

For the proposed method, the final *Kill_rate* of $\vee_{i=1}^{2} [T_i \wedge D_i]_{o'_{i,best}}$ is 100%, same as the conventional method. It means that every program mutant has been killed by at least one piece of test data. The corresponding final *Grade* is 0.59, larger than the *Grade* of 0.37 with the original

TABLE 6.3: Results for process Mod with original specifications.

Original Specification	Grade
$T_1 : x > 0 \wedge y \neq 0$	0.32
$T_2 : x < 0 \wedge y \neq 0$	0.38
$Total : \vee_{i=1}^{2} T_i$	0.37

specification-based method, indicating that the test data generated from $\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$ are of high quality that are more likely to kill all the program mutants. The result suggests that it is plausible to use these best individuals of chromosomes to make four mutated specifications for test case generation in the further maintenance of the original program.

TABLE 6.4: Test data generated by the mutated/original specification to kill each of program mutants in Mod.

Two kinds of Specs		$N\_kill_{i,o'} = (k_1, \ldots, k_m)$
Mutated Spec:	1. d-chromo: (-4, 0, -6, -6)	(1, 16, 19, 3, 1, 19, 19, 1, 16, 20, 18, 1, 18, 19, 4, 19)
	2. d-chromo: (-7, 0, 9, 13)	(4, 14, 20, 2, 4, 16, 4, 4, 14, 20, 16, 4, 16, 20, 6, 20)
Original Spec:	1. original:	(6, 3, 7, 4, 6, 7, 7, 10, 3, 13, 5, 3, 5, 8, 10, 11)
	2. original:	(6, 6, 18, 6, 6, 12, 6, 6, 6, 18, 3, 4, 3, 7, 12, 9)

Both methods generate test data with different characteristics of killing program mutants, as shown in Table 6.4. Compared to the original specification, a single mutated functional scenario in the proposed method tends to produce a test suite that concentrates on killing the majority of program mutants, leaving the others that are to be further killed by other mutated functional scenarios. In this way, not only is a single mutated functional scenario more capable of generating effective test data, but all the mutated functional scenarios as a whole can be used to uncover as many faults as possible.

Comparing the reformed specifications with the original ones in Figure 6.5, we can find the *Grade* of reformed ones that are always larger than that of original ones. It means that the data suite generated from the mutated specifications is more likely to pinpoint bugs than that of original ones, although both of them share the same *Kill_rate* of 100%.

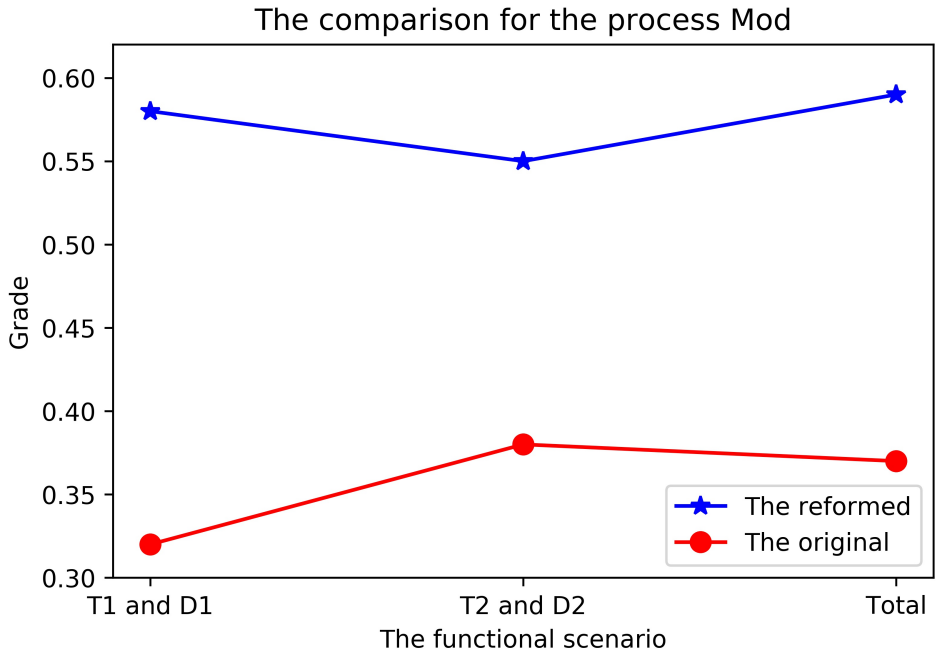


FIGURE 6.5: The grade of the mutated and original.

The Effect with Dummy Variables

We conduct additional experiments to figure out how dummy variables introduced into the different parts of defining conditions would affect the quality of the obtained mutated specifications. We make three versions of modifications to our approach as follows.

1. Version *V1*: Introducing dummy variables into only inequality relation;
2. Version *V2*: Introducing dummy variables into both equality and inequality relation;
3. Version *V3*: Putting no dummy variables in defining conditions.

For convenience, the approach with no modification, that is, having dummy variables for only equality relation, is called Version *V0*.

The previous experimental result for *V0* and the original, as well as the results from after applying variations of the approach *V1*,*V2*, and *V3* to process *Mod*, are together displayed in Figure 6.6.

According to Figure 6.6, three approaches with dummy variables *V0*, *V1*, and *V2* gain higher *Grade*s than the approach without dummy variables *V3*, and even *V3* seems to behave

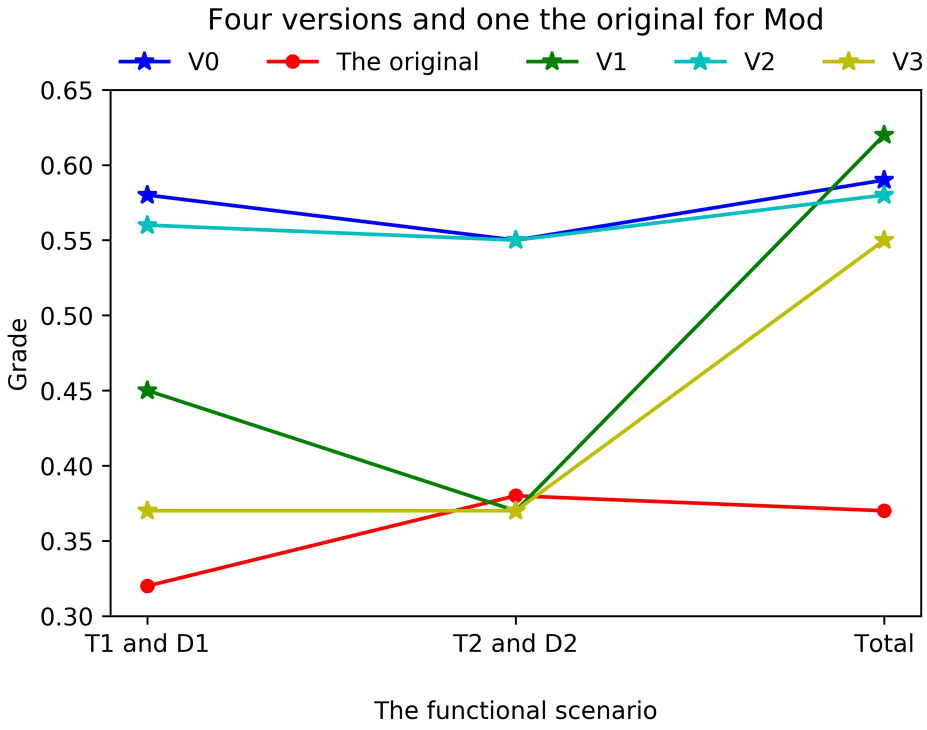


FIGURE 6.6: Results by four versions and the original for Mod.

better than the conventional method. There are no significant differences between the evaluation of *V0* and *V2*. However, *V2* would occupy more computation resources than *V0* due to the consideration of more dummy variables. It seems that *V1* gains a little better final *Grade* than *V0*, though its *Grade* for each single mutated functional scenario is not good enough.

In addition, by using an approach without dummy variables *V1* and *V3*, every obtained single mutated functional scenario demonstrates the strong capability to kill some specific program mutants while leaving other program mutants not killed, though the combination of all the functional scenarios in *V1* can reach 100% total *Kill_rate* while, for *V3*, the total *Kill_rate* unfortunately remains in 87.5%. This result demonstrates the importance of introducing dummy variables into equality relation in order to accomplish both good single and total *Grade*s and *Kill_rate*s.

In summary, it is necessary to introduce dummy variables into equality relations, and the additional dummy variables for inequality relation cannot significantly improve the proposed approach.

6.3.2 Case Study 2: Gcd

Process *gcd* is to compute the greatest common divisor of two inputs by using Stein's algorithm.

The formal specifications of *gcd* is:

process gcd (x: int, y: int) r: int

pre $x \geq 0 \wedge y \geq 0$

post $x > 0 \wedge y > 0 \wedge x \geq y \wedge r = gcd(y, x\%y) \vee$

$x > 0 \wedge y > 0 \wedge x < y \wedge r = gcd(y, y\%x) \vee$

$y = 0 \wedge r = x \vee$

$x = 0 \wedge r = y$

end_process

The implementation of process gcd in Python is:

```
def gcd(x, y):
  if x < y:
     x, y = y, x
  if (0 == y):
     return x
  if x % 2 == 0 and y % 2 == 0:
     return 2 * gcd(x//2, y//2)
  if x % 2 == 0:
     return gcd(x // 2, y)
  if y % 2 == 0:
     return gcd(x, y // 2)
  return gcd((x - y) // 2, y)
```

Process *gcd* is a recursive process and its post-condition contains itself, so it is difficult to generate data from this kind of post-condition. We transform the original post-condition to the following ones:

$T_1 \wedge D_1 := x > 0 \wedge y > 0 \wedge x \geq y \wedge x\%r = 0 \wedge y\%r = 0 \wedge x\%y\%r = 0;$

$T_2 \wedge D_2 := x > 0 \wedge y > 0 \wedge x < y \wedge x\%r = 0 \wedge y\%r = 0 \wedge y\%x\%r = 0;$

$T_3 \wedge D_3 := x \geq 0 \wedge y = 0 \wedge r = x;$

$T_4 \wedge D_4 := y \geq 0 \wedge x = 0 \wedge r = y.$

Table 6.5 shows the chromosomes of process *gcd*.

Apply the algorithm to all of the chromosomes; in the meantime, make use of the original post-condition to determine whether the outputs of codes are correct or not. The results are displayed in Table 6.6.

The final *Kill_rate* of $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ is 100%. The corresponding *Grade* is 0.46, which means roughly 46 percent of test data that are randomly generated from $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ can kill all the program mutants.

TABLE 6.5: Chromosome forms for functional scenarios of process gcd.

Chromosome	D-Chromo	Dummy Vars
$[T_1 \wedge D_1]_{o'} : x \geq y \wedge$ $(d_1 \leq x\%r \leq d_2) \wedge$ $(d_3 \leq y\%r \leq d_4) \wedge$ $(d_5 \leq x\%y\%r \leq d_6)$	$o' =$ $(r, d_1, d_2, d_3$ $d_4, d_5, d_6)$	$d_1, d_2, d_3,$ d_4, d_5, d_6
$[T_2 \wedge D_2]_{o'} : x < y \wedge$ $(d_1 \leq x\%r \leq d_2) \wedge$ $(d_3 \leq y\%r \leq d_4) \wedge$ $(d_5 \leq y\%x\%r \leq d_6)$	$o' =$ $(r, d_1, d_2, d_3$ $d_4, d_5, d_6)$	$d_1, d_2, d_3,$ d_4, d_5, d_6
$[T_3 \wedge D_3]_{o'} : y = 0 \wedge$ $(d_1 \leq x - r \leq d_2)$	$o' =$ (r, d_1, d_2)	d_1, d_2
$[T_4 \wedge D_4]_{o'} : x = 0 \wedge$ $(d_1 \leq y - r \leq d_2)$	$o' =$ (r, d_1, d_2)	d_1, d_2

TABLE 6.6: Results for process gcd after applying GA.

The Best Individual of Chromosome	Grade
$[T_1 \wedge D_1]_{o'}$ $o'_{1,best} =$ $(0, 1, 10, 10, 10, 3, 6)$	0.72
$[T_2 \wedge D_2]_{o'}$ $o'_{2,best} =$ $(8, 4, 6, 2, 2, 0, 8)$	0.58
$[T_3 \wedge D_3]_{o'}$ $o'_{3,best} = (5, -1, 20)$	0.0037
$[T_4 \wedge D_4]_{o'}$ $o'_{4,best} = (4, -3, 16)$	0.0037
$Total : \vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$	0.46

Conversely, the result for applying the method that generates test data directly from the original specification is displayed in Table 6.7.

TABLE 6.7: Results for process gcd with original specifications.

Original Specification	**Grade**
$T_1 : x > 0 \wedge y > 0 \wedge x \geq y$	0.29
$T_2 : x > 0 \wedge y > 0 \wedge x < y$	0.49
$T_3 : x \geq 0 \wedge y = 0$	0.0035
$T_4 : y \geq 0 \wedge x = 0$	0.0035
$Total : \vee_{i=1}^{2} T_i$	0.25

TABLE 6.8: Test data generated by the mutated/original specification to kill each of program mutants in Gcd.

Two kinds of Specs		$N\_kill_{i,o'} = (k_1, \ldots, k_m)$
Mutated Spec:	1. d-chromo: (0, 1, 10, 10, 10, 3, 6)	(0, 19, 20, 19, 19, 9, 20, 20, 19, 19, 19, 19, 19, 9, 9)
	2. d-chromo: (8, 4, 6, 2, 2, 0, 8)	(0, 1, 1, 18, 18, 16, 19, 19, 2, 18, 19, 4, 18, 18, 19, 19)
	3. d-chromo: (5, -1, 20, 7, 6, 3, 0)	(20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
	4. d-chromo: (4, -3, 16, 7, 8, 5, 3)	(20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
Original Spec:	1. original:	(0, 1, 13, 1, 1, 10, 14, 15, 10, 2, 2, 2, 7, 7, 8, 10)
	2. original:	(0, 7, 7, 9, 9, 12, 14, 15, 10, 10, 6, 11, 19, 19, 12, 15)
	3. original:	(19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
	4. original:	(19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

Like that in Mod, the experimental result in Gcd goes the same way in Table 6.8. The test data generated by both methods has different characteristics of killing program mutants. Compared to the original specification, a single mutated functional scenario in the proposed method tends to produce a test suite that concentrates on killing the majority of program mutants, leaving the others that are to be further killed by other mutated functional scenarios. In addition to that a single mutated functional scenario is more capable of generating effective test data for all program mutants, all the mutated functional scenarios as a whole can used to kill as many faults as possible.

Comparing the reformed specifications with the original ones in Figure 6.7, we can find that the first two reformed ones $[T_1 \wedge D_1]_{o'}$ and $[T_2 \wedge D_2]_{o'}$ have very high values of *Grade*, 0.72 and

0.58, respectively, higher than 0.29 and 0.49 with the original specifications. In addition, the *Kill_rate* of a sole $[T_i \wedge D_i]_{o'}$ $(i = 1, 2)$ is 94%, indicating that the test data generated from the first two reformed specifications are likely to pinpoint most bugs probably occurring in the program. Only a few program mutants (6% of total), with some faults that directly relates to the last two functional scenarios $T_3 \wedge D_3$ and $T_4 \wedge D_4$ (where $x = 0$ or $y = 0$), cannot be killed by the test data generated from either the first two reformed specifications or the first two original specifications. Due to the very simple forms and the limited functionality of the last two functional scenarios, there is no improvement of test data generation using our method against the original ones.

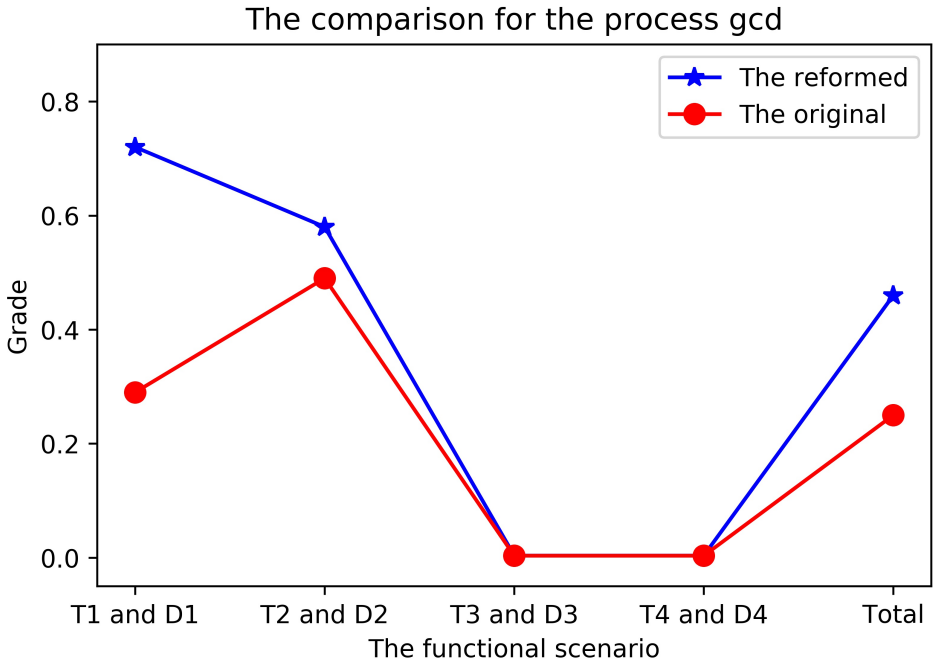


FIGURE 6.7: The grade of the reformed and original.

The results from both classic examples demonstrate that the input data generated from the mutated specifications are more likely to kill the mutants of programs than that from the original specifications.

The Effect without Dummy Variables

Like what we have done for process *Mod*, we conduct additional experiments with the approach without using dummy variable V3 since *gcd* only has equality relations in the defining

conditions. The results are shown in Figure 6.8.

FIGURE 6.8: Results by four versions and the original for gcd.

The experimental results are similar to that in *Mod*. $V0$ performs better than $V3$. $V3$ still encounters the problem that every single mutated functional scenario is not able to kill all the program mutants. It shows that the test data generated from those strict equality relations are less likely to trigger some bad behaviors of program.

6.3.3 Complexity of the Approach

We present an abstract analysis of the complexity for our approach. Generally, a GA complexity is on the order of $O(g * n * m)$ without the effect of the fitness function, where g is the number of generations, n the population size, and m the number of functional scenarios. Since our approach uses a fitness function involved in the mutation testing, we should take both the program execution time and the data suite generation time into consideration.

As the speed of the constraint solver to solve an individual formula (to generate a test suite) depends on the complexity of the functional scenarios (logical formulas) whose complexity cannot be easily determined, we associate the cost of using the constraint solver for a singular

individual with the number of input variables in, the number of output variables out, and the number of dummy variables d. In addition, the number of dummy variables relies on the number of equality relation in each functional scenario, which varies in different kinds of programs. We simply assume that each functional scenario has at least one equality relation. Thus, the complexity of using the constraint solver for each individual is $O(in + out + 2 * d * m)$. Moreover, the complexity of all the executions for program mutants is approximately $O(mu * sui)$ with mu the number of program mutants and sui the size of test suite. Finally, considering the complexity of the GA together with the mutation testing, the complexity for our approach is

$$O(g * n * ((in + out + 2 * d * m) * (mu * sui)) * m).$$

6.4 Conclusion

We propose a new method for effective test data generation based on mutated pre-post style formal specifications. The method is characterized by the integration of the functional scenario-based testing, a genetic algorithm and the mutation testing. In the approach, by assigning appropriate values to the unknown output and dummy variables to the variations for the original specifications, we can obtain useful mutated specifications that are sensitive to small syntactic structural changes of program codes.

We have also carried out two classic cases to evaluate the performance of our method. The results of case studies demonstrate that, for a complicated functional scenario, the proposed approach is capable of effectively generating useful test data to kill as many program mutants as possible, which outperforms the conventional data generation method.

In spite of the advantages of our method as mentioned above, there are also some limitations and disadvantages in the application of our method. First, the proposed method can only work on arithmetical relationships between inputs and outputs in which outputs affect the generation of inputs. Second, as the GA usually iterates many times and executes all the program mutants for every iteration, the cost would not be low. However, if we have enough computing resources for applying our method, it might be worth taking time to obtain good reformed specifications for the further maintenance of software.

In order to cope well with complex real programs, some additional extensions can be made to our approach. Firstly, by using the character encoding standard like US-ASCII [103], we can convert a String to a byte array so that the relationship that contains string variables can also be manipulated by our method. Moreover, since many research works exist concerning about the techniques of encoding complex data [104, 105, 106] that may occur in specifications like images and videos, it is possible to transform these specifications into appropriate arithmetical relationships so that our approach can be used in such cases. Secondly, although there exist specifications where the input and output variables are not specified by some explicit arithmetical equality relation, our method would still be applicable. Because instead of directly using these specifications, we can design some mutated arithmetical relationships (in form of inequality) of input and output that can not only approximate to the real properties of program but also leave open the possibility of occurrence of unexpected behaviors. Thirdly, when testing a big complex system, we can decompose it into a set of subroutines and focus on testing small procedures one by one using our approach. Thus, there is no need to repeatedly executing the whole system with our algorithm.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The SIT-SE contributes to a more rigorous testing of the consistency between an implementation and its formal specification by associating the correctness of symbolic paths with functional scenarios. This method can incrementally explore a moderate number of paths meanwhile strengthens the verification for each path by using theorems with a prover. The SIT-SE can significantly reduce the testing time due to the facts that test case generation can be fully automated and the number of test cases for detecting bugs is much smaller than that of traditional path testing. This is opposed to the general idea around most testing methods with expensive strategies of exhausted path exploration and weaker solutions to path correctness, as well as opposed to that in formal methods with expensive intricate proofs (sometimes no proof found) for an entire system.

More specifically, the SIT-SE provides a rigorous way to verify the correctness of paths by using theorems. It sets up checking levels for branch conditions to facilitate an incremental testing, as well as integrates the BSC algorithm that is designed to relieve the path explosion problem by cutting down redundant uninteresting paths. This method outperforms conventional methods as to identifying more false paths based on a small-scale test suite that is generated for checking program correctness.

Further, we have also developed a novel fault localization technique called TRIACFL. It utilizes false symbolic paths provided by the SIT-SE and analyzes the conditions and blocks along these paths. This method provides a flexible framework for fault localization where

the three modules, the SIT-SE, elementary fault location generation algorithm, and attentional shift-based human review are intimately interacted with each other. According to the prediction of the number of faults in the code, the approach guides testers to either go to the route of review and exclusion, or go to the route of review and fixing. TRIACFL outperforms the existing technique SBFL in both single- and multiple-fault experiments. It guides testers to effectively pinpoint the faults in a small sequence of suspicious fault locations within moderate human inspection costs.

When testing a program that contains some unavailable source code, the SIT-SE may be ineffective or even not applicable. A black-box testing method, featuring the integration of functional scenario-based testing, a genetic algorithm (GA) and mutation testing, is proposed to handle such situation. This method uses a GA to obtain mutated specifications where appropriate values are assigned to the unknown output and dummy variables in the variations of the original specifications. These mutated specifications, sensitive to small syntactic structural changes of program codes, are further used to generate tests for effective bug detection.

7.2 Future Work

In order to build a tool to support the SIT-SE for automatic bug detection, the most challenging thing is to write a mature interpreter for a variety of syntax types of formal specifications (SOFL in this thesis). Such an interpreter can facilitate a fully automated integration for formal specifications into symbolic program paths for further verification.

Besides, there are other promising topics for the tool to support the SIT-SE. Firstly, our method is inevitably influenced by the technique of symbolic execution. Although over last decades have seen many improvements on symbolic execution for managing all kinds of situations, it is still hard to work on many complex data structures and non-primitive types. Extending the symbolic execution would be a very challenging work in the future. Secondly, the effectiveness of our method is also confined by the limited capability of the constraint solver involved. To effectively use the constraint solver, we must develop some additional strategies for simplifying the quires to the solver. Thirdly, if the execution of the program does not terminate, for example, due to infinite loops, our method cannot be applicable in this situation. How to improve the capability of our method to deal with this challenge will be part

of our future work. Finally, our method is limited to dealing with the testing of sequential programs, but it may be extended to deal with the testing of concurrent programs in the future. The key issues to address would be how to carry out symbolic execution to derive the necessary information of the traversed program path and how to form the necessary theorem for the verification of the path correctness.

For example, in some cases, if we set up an appropriate test environment, the proposed method can be extended to be applied in that area. Specifically, suppose process S under test operates a shared variable x, and there is a program where multiple threads call S simultaneously, we suggest testing the program together with S by the following settings.

- Make a version of specification for the program where S is to be tested in the concurrency environment.
- For each branch condition that contains any shared variables, set its checking level to 0, so that all the accesses to such branch condition can be used to form a path condition for the program.
- Additionally, if one would like to check the behavior (against the specification) of a specific block (e.g., S or a thread) during symbolic execution of the program, one can monitor carefully both entry and exit of this block, as well as check the correctness of the path (only in this block) by forming theorems in runtime.

For more technical details, we will report the progress elsewhere after a serious investigation in the future.

Furthermore, it would be an attractive work to enrich the tool by adding the support of TRIACFL in fault localization. As the prediction of the number of faults in a program plays an important role in TRIACFL, we aspire to explore a variety of fault prediction models of high precision. For example, we can develop such models with the aid of many kinds of machine (and deep) learning techniques [107, 108]. Finally, we consider making some improvements to the proposed black-box testing method with a GA: 1) enhancing its capability of dealing with more complicated relationships between inputs and outputs where the values of outputs may not directly determine the inputs; 2) extending it by designing more useful mutated operators for formal specification. Achievement of effective solutions to all those challenges will

pave the way for us to develop a mature software tool to efficiently support all the proposed methods in the real world.

Bibliography

[1] C. Hobbs, *Embedded Software Development for Safety-Critical Systems*. Auerbach Publications, 2015.

[2] J. Din, R. Din, and Y. Y. J. . M. B. Jasser, "Towards employing metrics in measuring the quality of software safety critical systems and managing their development," *International Journal of Engineering and Technology*, vol. 7, no. 4.31, pp. 135–139, 2018.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[4] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[6] J. M. Schumann, *Automated theorem proving in software engineering*. Springer Science & Business Media, 2001.

[7] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets*, pp. 429–528, Springer, 1996.

[8] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*, pp. 1–30, Springer, 2011.

[9] M. Kaufmann and J. S. Moore, "Some key research problems in automated theorem proving for hardware and software verification.," *RACSAM*, vol. 98, no. 1, pp. 181–195, 2004.

[10] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[11] P. M. Bueno, W. E. Wong, and M. Jino, "Automatic test data generation using particle systems," in *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 809–814, 2008.

[12] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[13] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on software engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[14] E. Martins, S. B. Sabião, and A. M. Ambrosio, "Condata: a tool for automating specification-based test case generation for communication systems," *Software Quality Journal*, vol. 8, no. 4, pp. 303–320, 1999.

[15] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *International Conference on Formal Engineering Methods*, pp. 471–482, Springer, 2002.

[16] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications," in *Lectures on Runtime Verification*, pp. 135–175, Springer, 2018.

[17] S. Liu and S. Nakajima, "Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing," *IEEE Transactions on Software Engineering*, 2020.

[18] S. Liu and S. Nakajima, "Combining specification-based testing, correctness proof, and inspection for program verification in practice," in *International Workshop on Structured Object-Oriented Formal Language and Method*, pp. 3–16, Springer, 2013.

[19] S. Liu, "Testing-based formal verification for algorithmic function theorems and its application to software verification and validation," in *2016 International Symposium on System and Software Reliability (ISSSR)*, pp. 1–6, IEEE, 2016.

[20] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[21] C. Cadar and K. Sen, *Symbolic execution for software testing: three decades later*. ACM, 2013.

[22] D. Monniaux, "A survey of satisfiability modulo theory," in *International Workshop on Computer Algebra in Scientific Computing*, pp. 401–425, Springer, 2016.

[23] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[24] A. Giantsios, N. Papaspyrou, and K. Sagonas, "Concolic testing for functional languages," *Science of Computer Programming*, vol. 147, pp. 109–134, 2017.

[25] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[26] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 273–283, 2017.

[27] A. J. Offutt and S. Liu, "Generating Test Data from SOFL Specifications," *Journal of Systems and Software*, vol. 49, pp. 49–62, December 1999.

[28] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for alloy," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 264–275, IEEE, 2017.

[29] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016.

[30] S. Liu, "Testing-Based Formal Verification for Theorems and Its Application in Software Specification Verification," in *Proceedings of the 10th International Conference on Tests and Proofs (TAP 2016)*, (Vienna, Austria), pp. 112–129, LNCS 9762, Springer, July 5-7 2016.

[31] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[32] M. R. Woodward, "Mutation testing—its origin and evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993.

[33] Y. Sato and T. Sugihara, "Automatic generation of specification-based test cases by applying genetic algorithms in reinforcement learning," in *International Workshop on Structured Object-Oriented Formal Language and Method*, pp. 59–71, Springer, 2015.

[34] L. You and Y. Lu, "A genetic algorithm for the time-aware regression testing reduction problem," in *2012 8th International Conference on Natural Computation*, pp. 596–599, IEEE, 2012.

[35] P. B. Nirpal and K. Kale, "Using genetic algorithm for automated efficient software test case generation for path testing," *International Journal of Advanced Networking and Applications*, vol. 2, no. 6, pp. 911–915, 2011.

[36] M. R. Girgis, A. S. Ghiduk, and E. H. Abd-Elkawy, "Automatic generation of data flow test paths using a genetic algorithm," *International Journal of Computer Applications*, vol. 89, no. 12, pp. 29–36, 2014.

[37] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (New York), pp. 263–272, ACM Press, 2005.

[38] K. Sen and G. Agha, *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools*. Springer Berlin Heidelberg, 2006.

[39] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, pp. 213–223, ACM, 2005.

[40] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 263–272, ACM, 2005.

[41] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*, pp. 443–446, IEEE Computer Society, 2008.

[42] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[43] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 179–180, ACM, 2010.

[44] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder:symbolic execution of java bytecode," in *ASE 2010, Ieee/acm International Conference on Automated Software Engineering, Antwerp, Belgium, September*, pp. 179–180, 2010.

[45] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 553–568, Springer, 2003.

[46] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, pp. 123–133, ACM, 2002.

[47] D. Marinov, *Automatic testing of software with structurally complex inputs*. PhD thesis, Massachusetts Institute of Technology, 2005.

[48] S. Khurshid, *Generating structurally complex tests from declarative constraints*. PhD thesis, Massachusetts Institute of Technology, 2003.

[49] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 22–31, IEEE, 2001.

[50] D. Jackson, I. Shlyakhter, and M. Sridharan, "A micromodularity mechanism," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 62–73, ACM, 2001.

[51] S. Liu, "Utilizing Hoare Logic to Strengthen Testing for Error Detection in Programs," in *Proceedings of The Turing Centenary Conference*, (Manchester, UK), pp. 229–238, EPiC Series, June 2012.

[52] S. Liu, "A tool supported testing method for reducing cost and improving quality," in *IEEE International Conference on Software Quality, Reliability and Security*, pp. 448–455, 2016.

[53] S. Liu, "Testing-based formal verification for algorithmic function theorems and its application to software verification and validation," in *International Symposium on System and Software Reliability*, pp. 112–129, 2017.

[54] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 467–477, IEEE, 2002.

[55] A. Gonzalez, "Automatic error detection techniques based on dynamic invariants, master's thesis," *Delft University of Technology, The Netherlands*, 2007.

[56] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, IEEE, 2006.

[57] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 56–66, IEEE, 2009.

[58] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, pp. 21–30, IEEE, 2012.

[59] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[60] B. Korel and J. Laski, "Dynamic program slicing," *Information processing letters*, vol. 29, no. 3, pp. 155–163, 1988.

[61] R. E. Fairley, "Aladdin: Assembly language assertion driven debugging interpreter," *IEEE Transactions on Software Engineering*, no. 4, pp. 426–428, 1979.

[62] M. Jose and R. Majumdar, "Bug-assist: assisting fault localization in ansi-c programs," in *International conference on computer aided verification*, pp. 504–509, Springer, 2011.

[63] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE transactions on software engineering*, no. 4, pp. 367–375, 1985.

[64] E. J. Weyuker, "More experience with data flow testing," *IEEE transactions on software engineering*, vol. 19, no. 9, pp. 912–919, 1993.

[65] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[66] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software testing, verification and reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[67] M. R. Girgis, "Automatic test data generation for data flow testing using a genetic algorithm.," *J. UCS*, vol. 11, no. 6, pp. 898–915, 2005.

[68] N. Nayak and D. P. Mohapatra, "Automatic test data generation for data flow testing using particle swarm optimization," in *International conference on contemporary computing*, pp. 1–12, Springer, 2010.

[69] S. Biswas, M. S. Kaiser, and S. Mamun, "Applying ant colony optimization in software testing to generate prioritized optimal path and test data," in *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, pp. 1–6, IEEE, 2015.

[70] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 212–222, ACM, 2011.

[71] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[72] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[73] A. Derezinska and K. Kowalski, "Object-oriented mutation applied in common intermediate language programs originated from c," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 342–350, IEEE, 2011.

[74] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for c++ with the mucpp mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017.

[75] S. Liu, *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer Science & Business Media, 2013.

[76] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[77] K. L. McMillan and L. D. Zuck, "Formal specification and testing of quic," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 227–240, 2019.

[78] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from ocl constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[79] A. Jalila and D. J. Mala, "Automated optimal test data generation for ocl specification using harmony search algorithm," *International Journal of Business Intelligence and Data Mining*, vol. 16, no. 2, pp. 231–259, 2020.

[80] S. Liu, K. Takahashi, T. Hayashi, and T. Nakayama, "Teaching Formal Methods in the Context of Software Engineering," *Inroads SIGCSE Bulletin*, vol. 41, pp. 17–23, June 2009.

[81] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Transactions on Software Engineering*, vol. 24, pp. 337–344, January 1998. Special Issue on Formal Methods.

[82] S. Liu, "Capturing complete and accurate requirements by refinement," in *Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings.*, pp. 57–67, IEEE, 2002.

[83] S. Liu, "Utilizing test case generation to inspect formal specifications for completeness and feasibility," in *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pp. 349–356, IEEE, 2007.

[84] C. Hoare, "An Axiomatic Basis of Computer Programming," *Communications of the ACM*, no. 12, pp. 576–580, 1969.

[85] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima, "Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation," in *9th International Conference on Software Methodologies, Tools and Techniques (SoMet 2010)*, (Yokohama city, Japan), p. to appear, IOS International Publisher, Sept. 29- Oct. 1 2010.

[86] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[87] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software testing, verification and reliability*, vol. 13, no. 1, pp. 25–53, 2003.

[88] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 105–116, 2009.

[89] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[90] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, pp. 183–198, 2011.

[91] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, pp. 691–701, 2016.

[92] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[93] W. E. Wong, *Mutation testing for the new century*, vol. 24. Springer Science & Business Media, 2001.

[94] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th international conference on Software engineering*, pp. 827–830, ACM, 2006.

[95] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and W. E. Wong, "Component-based software: An overview of testing," *Component-Based Software Quality*, pp. 99–127, 2003.

[96] A. Derezińska and A. Szustek, "Cream-a system for object-oriented mutation of c# programs," in *Annals Gdansk University of Technology Faculty of ETI*, vol. 13, pp. 389–406, Information Technology, 2007.

[97] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[98] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[99] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[100] J. Luo, S. Liu, Y. Wang, and T. Zhou, "Applying sofl to a railway interlocking system in industry," in *International Workshop on Structured Object-Oriented Formal Language and Method*, pp. 160–177, Springer, 2016.

[101] S. Liu, *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[102] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, ACM, 2007.

[103] C. E. Mackenzie, *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[104] B. Basavaprasad and M. Ravi, "A study on the importance of image processing and its applications," *IJRET: International Journal of Research in Engineering and Technology*, vol. 3, p. 1, 2014.

[105] V. Barannik, S. Podlesny, D. Tarasenko, D. Barannik, and O. Kulitsa, "The video stream encoding method in infocommunication systems," in *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*, pp. 538–541, IEEE, 2018.

[106] T. Hur, J. Bang, J. Lee, J.-I. Kim, S. Lee, *et al.*, "Iss2image: A novel signal-encoding technique for cnn-based human activity recognition," *Sensors*, vol. 18, no. 11, p. 3910, 2018.

[107] Z.-Q. Zhou, Q.-X. Zhu, and Y. Xu, "Time series extended finite-state machine-based relevance vector machine multi-fault prediction," *Chemical Engineering & Technology*, vol. 40, no. 4, pp. 639–647, 2017.

[108] O. Al Qasem, M. Akour, and M. Alenezi, "The influence of deep learning algorithms factors in software fault prediction," *IEEE Access*, vol. 8, pp. 63945–63960, 2020.

Appendix A

List of Research Paper

Refereed Journal Papers

First Author

[1] R. Wang, S. Liu, and Y. Sato, "SIT-SE: A Specification-Based Incremental Testing Method with Symbolic Execution." *IEEE Transactions on Reliability*, vol. 70, no.3, pp. 1053-1070, 2021. (DOI: https://doi.org/10.1109/TR.2021.3078714)

[2] R. Wang, Y. Sato, and S. Liu, "Mutated Specification-Based Test Data Generation with a Genetic Algorithm." *Mathematics*, vol. 9, no. 4, p. 331, 2021. (DOI: https://doi.org/10.3390/math9040331)

[3] R. Wang, Z. Ding, N. Gui, and Y. Liu, "Detecting bugs of concurrent programs with program invariants." *IEEE Transactions on Reliability*, vol. 66, no.2, pp. 425-439, 2017. (DOI: https://doi.org/10.1109/TR.2017.2681107)

Refereed Conference Papers

First Author

[4] R. Wang, S. Liu, and Y. Sato, "A Fault Localization Approach Derived From Testing-based Formal Verification." in *2020 25th International Conference on Engineering of Complex Computer Systems* (ICECCS), pp. 165-170, IEEE, 2020.

[5] R. Wang, Y. Sato, and S. Liu, "Specification-based Test Case Generation with Genetic Algorithm." in *2019 IEEE Congress on Evolutionary Computation* (CEC), pp. 1382-1389, IEEE, 2019.

[6] R. Wang, and S. Liu, "Branch Sequence Coverage Criterion for Testing-Based Formal Verification with Symbolic Execution." in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion* (QRS-C), pp. 205-212, IEEE, 2019.

[7] R. Wang, T. Wakahara, "Practice in Caption Generation with Keras: The Design and Evaluation for Attention Models." in *Proceedings of the 2019 3rd International Conference on Deep Learning Technologies*, pp. 11-15, 2019.

[8] R. Wang, and S. Liu, "Tbfv-se: Testing-based formal verification with symbolic execution." in *2018 IEEE International Conference on Software Quality, Reliability and Security* (QRS), pp. 59-66, IEEE, 2018.

Corresponding Chapters with Papers

Chapter 1. Introduction

Chapter 2. Related Work

[8] R. Wang, and S. Liu, "Tbfv-se: Testing-based formal verification with symbolic execution." in *2018 IEEE International Conference on Software Quality, Reliability and Security* (QRS), pp. 59-66, IEEE, 2018.

[6] R. Wang, and S. Liu, "Branch Sequence Coverage Criterion for Testing-Based Formal Verification with Symbolic Execution." in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion* (QRS-C), pp. 205-212, IEEE, 2019.

[5] R. Wang, Y. Sato, and S. Liu, "Specification-based Test Case Generation with Genetic Algorithm." in *2019 IEEE Congress on Evolutionary Computation* (CEC), pp. 1382-1389, IEEE, 2019.

Chapter 3. Preliminaries

Chapter 4. Grey-Box Testing: The SIT-SE Method for Bug Detection

[1] R. Wang, S. Liu, and Y. Sato, "SIT-SE: A Specification-Based Incremental Testing Method with Symbolic Execution." *IEEE Transactions on Reliability*, 2021. (DOI: https://doi.org/10.1109/TR.2021.3078714) (Accepted)

Chapter 5. TRIACFL: Triple Interaction-Based Fault Localization

[4] R. Wang, S. Liu, and Y. Sato, "A Fault Localization Approach Derived From Testing-based Formal Verification." in *2020 25th International Conference on Engineering of Complex Computer Systems* (ICECCS), pp. 165-170, IEEE, 2020.

Chapter 6. Supplement to SIT-SE: a Mutated Specification-Based Approach with a Genetic Algorithm

[2] R. Wang, Y. Sato, and S. Liu, "Mutated Specification-Based Test Data Generation with a Genetic Algorithm." *Mathematics*, vol. 9, no. 4, p. 331, 2021. (DOI: https://doi.org/10.3390/math9040331)