SEMANTIC ANALYSES TO DETECT AND LOCALIZE SOFTWARE REGRESSION ERRORS

QI DAWEI
(B.Eng., Tsinghua University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

Qi Dawei

Qi Dawei

January 18, 2013

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the following individuals for their guidance, support and accompany during my Ph.D. time.

First of all, I would like to express my deepest gratitude to my thesis advisors, Dr. Abhik Roychoudhury and Dr. Zhenkai Liang. I am really grateful to them for guiding me throughout every step of my study despite my constant stubbornness and ignorance. Their insight and experience has been of immense value to my research. In addition, their passion for both research and life has also generated affluent influence on me. Without their guidance, this thesis would not have been possible.

I wish to thank my parents as well as other family members for their support and cultivation. Their endless love and care always brings me great strength. I feel really fortunate to have them in my life.

My study in NUS could not have been so wonderful without my colleagues and friends. I would like to especially thank Ju Lei, Liang Yun, Liu Shanshan, Wang Chundong, Ding Huping, Chen Jie, Lin Yong, Sun Tao, Hoang Duong Thien Nguyen, Marcel Boehme, Sudipta Chattopadhyay, Sandeep Kumar, Cristal Ngo, Jooyong Yi, Zhang Bingjun, Liu Xuan, Dong Xinshu, Patil Kailas, Dai Ting and Li Xiaolei to name a few. Thank them all for bringing so much joy and color to my life.

I would like to thank my thesis examiners in NUS, Dr. Khoo Siau Cheng and Dr. Stan Jarzabek. Thanks for their thoughtful comments and suggestions during my Ph.D. candidature. I am also thankful to Dr. Alessandro Orso for taking his precious time to be my external thesis examiner.

I also wish to thank Dr. Satish Chandra and Dr. Feng Qin for the insightful comments during our research collaboration. I am grateful to Dr. Xiangyu Zhang for the providing me the opportunity to visit his group as well as sharing many helpful advices.

The work presented in this thesis was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112) and a Defense Innovative

Research Programme (DIRP) grant (R-252-000-393-422) from Defence Research and Technology Office (DRTech). They are all gratefully acknowledged.

I am sure that there are many more to thank. My apologies to those that are overlooked.

RELATED PUBLICATIONS

1. QI, D., NGUYEN, H.D.T., AND ROYCHOUDHURY, A. Path Exploration based on Symbolic Output. *ACM Transactions on Software Engineering and Methodology* (accepted and to appear).

2. QI, D., YI, J., AND ROYCHOUDHURY, A. Software Change Contracts. *In Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering, New Ideas and Emerging Results track* (2012).

3. QI, D., ROYCHOUDHURY, A., LIANG, Z., AND VASWANI, K. DARWIN: An Approach to Debugging Evolving Programs. *ACM Transactions on Software Engineering and Methodology* 21, 3 (2012).

4. QI, D., NGUYEN, H.D.T., AND ROYCHOUDHURY, A. Path Exploration based on Symbolic Output. *In Proceedings of the joint meeting of ESEC and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2011).

5. QI, D., NGO, M.N., SUN, T., AND ROYCHOUDHURY A. Locating Failure Inducing Environment Changes. *In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (2011).

6. QI, D., ROYCHOUDHURY, A., AND LIANG, Z. Test Generation to Expose Changes in Evolving Programs. *In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (2010).

7. QI, D., ROYCHOUDHURY, A., LIANG, Z., AND VASWANI, K. DARWIN: An Approach for Debugging Evolving Programs. *In Proceedings of the joint meeting of ESEC and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2009).

TABLE OF CONTENTS

SUMMARY

Software maintenance takes up the major cost of a software project. When changes are made on software, extensive testing and debugging activities are performed to preserve software quality. Most of the existing testing and debugging techniques only focus on the current software version while ignoring previous program versions. However, we observe that previous software versions can be exploited to benefit these techniques in the following ways: (i) previous software versions serve as the reference for the current version with respect to unchanged software behavior, and (ii) previous analysis results can be reused given that they are not affected by software changes. We also observe that the recent advance in dynamic symbolic execution provides the adequate machinery for building semantic analysis techniques. Built upon these observations, we present several techniques for maintaining software quality in software evolution.

This thesis consists of the following five parts. First, we develop a test-suite augmentation technique based on code changes. We generate a test-input that has different outputs in the previous and current programs. The generated test-input can be helpful for comprehending program changes as well as testing the changed functionality of the program. Guided by program control structure and program changes, our technique was shown to be efficient in generating change-exposing test cases. Second, we present DARWIN, a debugging technique for evolving programs. When a bug is found through testing, we use DARWIN to locate the root cause of the bug. While using path condition to extract program semantics, our debugging technique could be more accurate than syntactic based approaches. Moreover, our DARWIN technique is also applicable to two entirely different implementations of the same specification. Third, we present a path partitioning technique that improves the state-of-the-art in dynamic symbolic execution. Program paths with the same input-output relationship are grouped in the same partition that is captured by relevant-slice condition. By improving dynamic symbolic execution, our debugging technique DARWIN achieves better efficiency and more accurate results. Fourth, we present the notion of change contracts, a contract language to document user intention of changes. Program requirement plays the most critical role in any testing and debugging activities. Unfortunately, a large volume of code does not come with any formal requirements, leaving program contract checking with full requirement unrealistic. Assuming that previous program versions are thoroughly tested, our proposed changed contract allows the users to only specify the changed program behavior. We have confirmed the usability of change contract via a user study with real-world program changes. Finally, we present a technique to localize program failure causes due to evolving execution environments.

As evidenced by comprehensive user studies and experiments on real-life programs, our proposed approaches ease various tasks in maintaining software quality, thereby helping to produce quality software in the face of never-ending software evolution.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

Software correctness is the top concern in the life-cycle of any software. The issue of software correctness exists in every step of the software life-cycle, including the initial requirement engineering, software writing, testing, debugging and bug-fixing. Failing to provide software correctness leads to costly remedial actions. It was estimated that software defects cost 59.5 billion in the Unites States annually [12]. In addition to economic cost, software defects may also cause severe damage to human lives. For example, a software bug in the radiation therapy device Therac-25 caused the death of at least five people and the injury of many more people [70].

Software usually undergoes continuous changes throughout its life. For example, on average, there is a change to the collection of software from Apache Software Foundation [10] every 6 minutes. The reasons of software changes are diverse. Software might be changed to accommodate the demand of new features. Software might also be changed to fix defects that were made earlier. Changes made to software are unavoidable and consume a large portion of the total software cost. It has been shown that about 75% of the total software cost goes to software maintenance cost [44]. Maintaining software correctness is challenging in this never-ending software evolution process. Even when software is perfectly correct with respect to previous requirements before a change is made, the changed software might be incorrect due to the change. The change can break existing functionality. The implementation of a new feature can also be flawed. Unsurprisingly, even bug-fixes are not always perfect, which leads to new bugs. A recent study showed that 14.8% to 24.4% of the bug-fixes in operating system code are buggy [122].

Various methods exist for guaranteeing software correctness in different stages of software construction. Immediately after software is written, static checking can be performed. Type checking, being the most basic static checking, has become the integral functionality of any compiler for strongly-typed programming languages. In addition to basic type checking, extended static checking (ESC) [42] has been proposed. In ESC, the implementation of a program is checked against the program requirements, which are given in pre/post-condition format. With the help of program requirements, it has been shown that ESC is able to find errors that are beyond the capability of basic type checking. In contrast to static checking, testing executes a program and checks whether the output is expected in order to find software bugs. Testing could be performed at different levels of abstraction. Unit testing focuses on the functionality of a unit, which is usually a function or a class. After the testing on each individual unit is done, system testing checks the functionality of the entire system as a whole. Once a bug is detected through testing, the process of debugging is started to locate the root-cause of the bug. If the root-cause of the bug can be located through debugging, programmers will fix the program and continue with the testing phase to check whether the bug-fix is correct as well as whether any other bugs still exist.

Testing and debugging techniques for evolving programs should make full usage of previous program versions. Although existing change-unaware testing and debugging techniques can be applied on the changed program, opportunities to exploit the previous versions for better results have often been ignored. For example, one major headache in debugging is finding a good reference execution — an execution that is close to the failing execution but passes. Debugging techniques usually search for the reference execution within the same program but with different program inputs [89, 110, 59]. If such techniques are applied to regression debugging, the perfect reference — execution in previous program versions with the same program input, is unfortunately ignored. As our research has shown, by taking previous version as the reference, a much more effective debugging technique can be built.

As another example, program contracts have been shown be to very useful in providing the oracle of correctness in testing and debugging activities [113, 37]. However, the large body of programs without associated contracts has prevented wide usage of program contracts. In the context of software evolution where the previous program versions have been extensively tested, we only need to provide the “contracts” for program changes as the previous versions implicitly serve as the contracts of unchanged program behavior.

As shown in the aforementioned scenarios, previous program versions provide valuable information for the quality assurance of the current program version. We observe that previous software versions can benefit the analysis of evolving software in the following two respects: (i) previous software versions serve as the reference of the current version for unchanged software behavior, and (ii) previous analysis results can be reused given that they are not affected by software changes.

In the few existing regression debugging techniques, syntactic analysis is the dominant tool being used (e.g. Delta debugging [123]). When applied on program evolution problems, syntactic analysis has to align the source codes (or execution traces) from different program versions. The alignment is usually done through heuristics (e.g. shortest edit distance) leading to errors in the analysis. Moreover, syntactic analysis is unable to identify semantically-equivalent program changes, leading to more noise in the analysis results. Recent advance in symbolic execution [51] provides excellent opportunities for precise semantic analysis across program versions feasible. In particular, path condition computed over a trace abstracts the semantics of computation in the trace into a logical formula. Precise comparison of logical formulae can be done efficiently by Satisfiability Modulo Theory (SMT) solvers.

Goal of this thesis. The overall goal of the research in this thesis is to improve the current techniques on detecting and localizing software regression errors. To detect regression errors, we aim to automatically generate test cases that reflect regression errors. To localize regression errors, we aim to reduce the manual debugging effort of programmers by providing them with a small set of program statements that are most suspicious to be the bug cause. While trying to develop techniques that advance state of the art, we want to investigate how information from previous program versions can be leveraged and how recent advances in automated theorem proving and Satisfiability Modulo Theory (SMT) constraint solving can be exploited. To achieve the goal, we propose several techniques for detecting and localizing software regression errors in software evolution in this thesis. Our techniques span across different stages of the software development process, including requirement formulation, testing and debugging.

In particular, we make the following contributions in this thesis:

- We propose a test-suite augmentation technique for evolving programs. Our technique tries to automatically generate test cases to expose the semantic changes introduced by syntactic changes. A test case generated by our technique has different outputs in the previous program version and the changed program version. Thus, the execution of the test case clearly explains the effect of program changes via concrete executions. Our technique uses targeted path exploration to first reach the program change and then propagate the effect of the change. During the propagation process, the execution of previous program version is always used as a reference to judge whether the effect of change is propagated. Our proposed notion of change-effect propagation tree helps us identify three different types of propagation termination. For each type, a solution is presented to carry forward the change effect.

- We propose a semantic analysis based regression debugging technique —DARWIN. The goal of DARWIN is to locate the root-causes of regression bugs. Different from existing syntactic analysis based techniques, DARWIN uses path condition as the semantic abstraction of an execution trace. By comparing path conditions from different program versions, we identify only semantic-changing differences across program versions. Through semantic analysis, DARWIN has been shown to achieve precise debugging results. Interestingly, as DARWIN is based on semantic analysis, it may still be applicable when the buggy program and the reference program are two entirely different implementations of the same specification.

- We propose a novel symbolic execution technique at the level of relevant slice and explore its usage in several program analysis tasks including path exploration, debugging and test-suite augmentation. The relevant slice condition generated during our symbolic execution technique enhances path condition by concentrating on input-output relationship. With the help of slicing, computation that does not affect program output is not included in the relevant slice condition. From another perspective, one relevant slice condition captures many different program paths sharing the same input-output relationship. We show that our technique can boost many dynamic symbolic execution based techniques where program output is the focus. We have proved the soundness of relevant-slice condition in capturing input-output relationship and the completeness of a path exploration algorithm based on relevant slice condition. Using relevant slicing condition as the underlying semantic analysis tool, our DARWIN technique is improved in both efficiency and effectiveness.

- We propose the notion of change contract to satisfy the need of a contract language for evolving programs. The correctness of any program is defined by its specification. However, providing full program contracts has always been a laborious task. Change contract alleviates this task by allowing users to specify only the changed program behavior. As a contract language for program changes, change contract has extensive usage in the presence of software evolution. In testing, change contract serves as the oracle to differentiate intended changes from real software regressions. In debugging, change contract provides the knowledge of correct behavior for the program being debugged. We have developed our change contract language for Java based on Java Modeling Language. We have also built a dynamic checking tool for the change contract language, which checks at runtime whether the program behavior satisfies its corresponding change contract.

- We propose a technique to debug program failures due to changes in a program's execution environment. The program's interaction with the reference environment (where the program passes) is first recorded. We then employ a systematic search process based on a semi-replay mechanism, which helps us locate the failure-inducing environment differences between the reference environment and the environment where the program fails.

Impact on current state of practice. Our proposed techniques can assist programmers in various tasks of detecting and localizing software regression errors, thereby helping programmers to build more reliable software with less regression errors. In the testing stage, with our test-generation technique, software regression errors can be detected with less manual effort from programmers. Instead of trying to come up with test cases to detect regression errors, programmers only need to check whether the generated tests reflect regression errors. In the debugging stage, with our DARWIN tool, programmers only need to examine the most suspicious statements as pointed out by DARWIN. With software change contracts, programmers could try to prevent software regression errors in the first place. Collectively, our techniques provide practical support for programmers in the never-ending battle with software regression errors.

Implications for future research. Through the studies in this thesis, we also demonstrate through concrete techniques that (i) previous program versions, when fully utilized, are of tremendous value in analyzing evolving software, and (ii) semantic analysis techniques such as dynamic symbolic execution are powerful instruments for evolving software analysis. This thesis provides the following implications for future research on detecting and localizing software regression errors: (i) techniques on detecting regression errors should focus on the changed program portion and its program dependencies, and (ii) when analyzing program changes, symbolic execution methods can help expose the semantic effect of the changes.

The remaining of this thesis is organized as follows. We first discuss the background knowledge and related work of this thesis in Chapter II and Chapter III. In Chapter IV, we present our test-suite augmentation technique that generates test cases to expose program changes. In chapter V, we propose a regression debugging technique – DARWIN which leverage dynamic symbolic execution as the semantic analysis tool. Chapter VI improves dynamic symbolic execution by focusing on input-output relationship of a program and proposes the notion of relevant-slice condition. Various usage of relevant-slice condition including path exploration and debugging is also discussed in Chapter VI. In Chapter 7.1, we present the change contract language that specifies the intended behavior of program changes. Our debugging technique for locating failure-inducing program execution environment is then presented in Chapter 7.2. Finally, Chapter VIII concludes this thesis and discusses some future research directions that can further extend this thesis.

CHAPTER II

Background

In this chapter, we recap the essential background knowledge for this thesis including two main parts: (i) control flow analysis and program dependency analysis, and (ii) symbolic execution.

2.1 Control Flow Analysis and Program Dependency Analysis

Basic control flow analysis and program dependency analysis [19, 18] has become the fundamental mechanism in many program analysis tasks, such as compiler optimization and program slicing. In this section, we give definitions of the basic concepts used in control flow analysis and program dependency analysis together with some illustrative examples.

Definition 2.1 (Basic Block)**.** *A basic block is a maximal sequence of program statements where the execution of the first statement implies the execution of all remaining statements in the basic block.*

```
int x, y;// x and y are both input variables
int o; // o is the output variable
scanf("%d",&x);
scanf("%d",&y);
if (x > 0){
  y = y +1;
  if (y > 0){
    o = 10;
  }else{
    o = 20;
  }
}else{
  o = 30;
}
```

Figure 2.1: A sample program to illustrate basic concepts

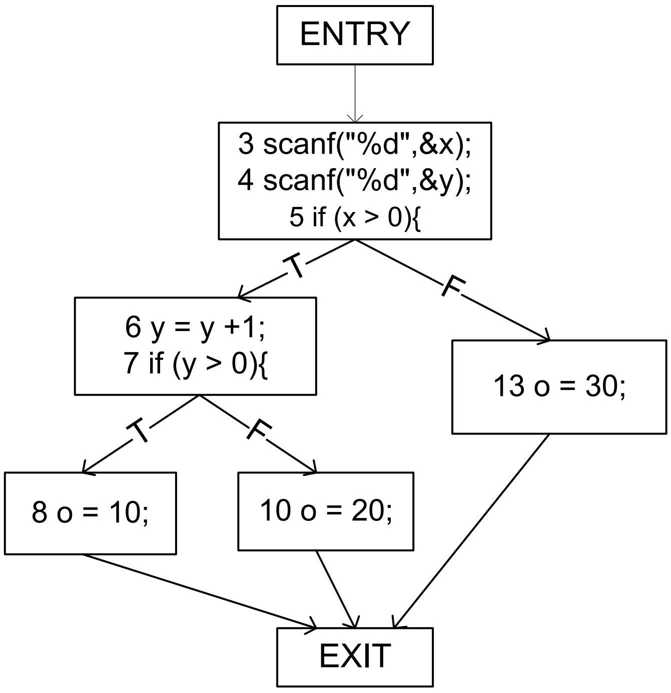


Figure 2.2: Control flow graph of the program in Figure 2.1

Definition 2.2 (Control Flow Graph (CFG))**.** *A control flow graph $G = (V, E)$ of a program P is a directed graph with each vertex $v \in V$ representing a basic block of program P and each edge $(v1, v2) \in E$, where $v1, v2 \in V$, representing the flow from the basic block of $v1$ to the basic block of $v2$. Typically, for the ease of analysis, two auxiliary vertices ENTRY (denoting the only point where control can enter the program) and EXIT (denoting the only point where control can leave the program) are added in a control flow graph.*

As an example, the control flow graph of the program in Figure 2.1 is shown in Figure 2.2. A label on an edge represents the direction (T stands for true, and F stands for false) of the conditional branch that makes the edge traversed. For example, the edge from basic block $\{6, 7\}$ to basic block $\{8\}$ can only be traversed when the branch[1] in line 7 is evaluated to true.

Definition 2.3 (Feasible Path)**.** *A program path π is feasible iff. there is at least one program input that traverses π.*

Therefore, a program path π is infeasible iff. there is no program input that traverses π.

[1]In this thesis, when the context is clear, we use "branch" to also refer to the test condition in a branching statement.

Definition 2.4 (Pre-dominator). *A basic block v_d is a pre-dominator of a basic block v iff. all paths from ENTRY to v have to pass through v_d.*

Definition 2.5 (Post-dominator). *A basic block v_d is a post-dominator of a basic block v iff. all paths from v to EXIT have to pass through v_d.*

The above definitions of dominance are also applicable to program statements.

Definition 2.6 (Static Control Dependency). *A statement v is statically control dependent on a branching statement b iff. (i) there is a nonnull path from b to v such that v post-dominates every node between b and v, and (ii) v does not post-dominate b [46]*

Based on the definition of control dependency, we can construct control dependence graph of a program.

Definition 2.7 (Control Dependence Graph (CDG)). *A control dependence graph $G = (V, E)$ of a program P is a directed graph with each vertex $v \in V$ representing a statement of program P and each edge $(v1, v2) \in E$, where $v1, v2 \in E$, representing that $v2$ is control dependent on $v1$.*

Figure 2.3 shows the control dependence graph of the program in Figure 2.1.

Definition 2.8 (Static Data Dependency). *A statement s is statically data dependent on a statement s_d iff. (i) there exists a variable x used in s such that x is defined in s_d, and (ii) there exists a a path from s_d to s that does not contain any definition of variable x.*

Different from static dependencies, dynamic dependencies concerns about the dependence relationship between statement instances during runtime.

Definition 2.9 (Dynamic Control Dependency). *Given an execution trace π and two statements α and β, a statement instance α_i of α is dynamically control dependent on a statement instance β_j of β in π iff. (i) α is statically control dependent on β, and (ii) β_j is executed before α_i in π, and (iii) there does not exist any statement instance γ_k (whose static statement is γ) between β_j and α_i such that (a) α is statically control dependent on γ (γ could be the same as β), and (b) γ_k is executed before α_i in π.*

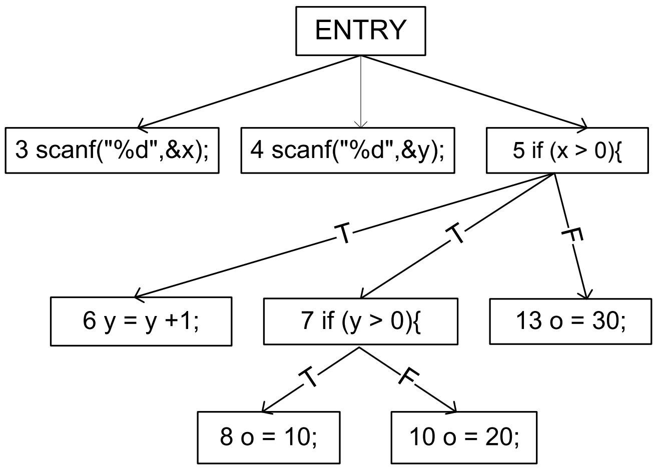


Figure 2.3: Control dependence graph of the program in Figure 2.1

Definition 2.10 (Dynamic Data Dependency)**.** *Given an execution trace π and two statements α and β, a statement instance α_i of α is dynamically data dependent on a statement instance β_j of β in π iff. (i) there exists a variable x such that x is defined in β_j and used in α_i, and (ii) there does not exist any statement instance γ_k (whose static statement is γ) between β_j and α_i such that γ_k defines variable x.*

2.2 Symbolic Execution and Dynamic Path Exploration

Symbolic execution was first proposed for program testing [66]. According to whether concrete executions are involved, symbolic execution techniques can be classified into either (i) static symbolic execution where no concrete execution is involved [66, 31] or (ii) dynamic symbolic execution where concrete execution is involved [51, 97]. We first explain the process of static symbolic execution. As opposed to concrete execution, symbolic execution takes symbolic program inputs. During symbolic execution, the program state is also maintained as symbolic and updated along program paths. For each variable in the program state, the symbolic value of the variable is represented as an expression of the symbolic input variables whose concrete values are not defined. When a branch is met during symbolic execution, both directions of the branch are tried out. The order of paths being traversed is subjective to the strategy of each symbolic execution technique. At every branch, we compute a *branch constraint*, which is a formula over the program's input variables which must be satisfied for the branch to be evaluated in the same direction as in the traversed path. For each path traversed in

Table 2.1: An example of dynamic symbolic execution and path condition computation for the program in Figure 2.1

Execution trace of $\langle x == 1, y == 1\rangle$	Concrete stores	Symbolic stores	Path condition
3 $scanf(\text{``}\%d\text{''}, \&x);$	$\{x \to 1, y \to undef\}$	$\{x \to x^s, y \to undef\}$	$true$
4 $scanf(\text{``}\%d\text{''}, \&y);$	$\{x \to 1, y \to 1\}$	$\{x \to x^s, y \to y^s\}$	$true$
5 $if(x > 0)\{$	$\{x \to 1, y \to 1\}$	$\{x \to x^s, y \to y^s\}$	$(x^s > 0)$
6 $y = y + 1;$	$\{x \to 1, y \to 2\}$	$\{x \to x^s, y \to y^s + 1\}$	$(x^s > 0)$
7 $if(y > 0)\{$	$\{x \to 1, y \to 2\}$	$\{x \to x^s, y \to y^s + 1\}$	$(x^s > 0) \land (y^s + 1 > 0)$
8 $o = 10;$	$\{x \to 1, y \to 2\}$	$\{x \to x^s, y \to y^s + 1\}$	$(x^s > 0) \land (y^s + 1 > 0)$

symbolic execution, a path condition is also maintained. The path condition of a path captures the set of inputs that traverse the path. More specifically, we have the following definition of path condition.

Definition 2.11 (Path condition)**.** *Given a program P, let π be a path in P. The path condition of π, say pc_π is a quantifier free first order logic formula which is satisfied by exactly the set of inputs that exercise π in program P.*

The satisfiability of a path condition is closely related to the feasibility of a path. In particular, a path is feasible iff. the corresponding path condition is satisfiable.

The result of symbolic execution is a set of executed paths and their corresponding path conditions. Concrete test cases traversing a particular path can be derived by solving the path condition of that path through SMT solvers. For each path, the symbolic values of program outputs, which are expressions of symbolic input variables, are also computed.

In the past decade, there has been a spurt of research on dynamic symbolic execution [51, 97]. Different from static symbolic execution, dynamic symbolic execution is driven by the concrete execution of concrete inputs[2]. In dynamic symbolic execution, we first execute the analyzed program with concrete input and derive the trace of the concrete execution. We then treat the program inputs as symbolic and propagate the symbolic program states along the trace. This is similar to static symbolic execution except that the path to be traversed is defined by the concrete execution trace. As in static symbolic execution, we can also compute path condition of the traversed path in dynamic symbolic execution.

We now use an example to illustrate the process of dynamic symbolic execution and path

[2]Hence, dynamic symbolic execution is sometimes called "concolic" (concrete and symbolic) execution by researchers.

condition computation. Let us take the example program in Figure 2.1 and input $\langle x == 1, y == 1\rangle$. We use x^s and y^s to denote the symbolic inputs of this program. The computation process is shown in Table 2.1. After executing each line, we show the concrete stores and the symbolic stores of the variables. In the last column, we show the path condition gathered up to the corresponding line. If a conditional branch is executed, the generated branch constraint is accumulated into the path condition as shown in the last column. For example, after line 6 is executed, the accumulated path condition is $(x^s > 0)$. Since line 7 is a conditional branch, the branch constraint $(y^s + 1 > 0)$ is generated and added into the path condition. So after executing line 7, the path condition becomes $(x^s > 0) \wedge (y^s + 1 > 0)$. The final path condition is simply the conjunction of all the branch constraints. In this example, two branch constraints $(x^s > 0)$ and $(y^s + 1 > 0)$ are generated from line 5 and line 7 respectively. Taking the conjunction of the two branch constraints, the final path condition is simply $pc = (x^s > 0) \wedge (y^s + 1 > 0)$. The path condition computed in this way only contains input variables. The path condition can guarantee that any input satisfying the path condition will follow the same path as $\langle x == 1, y == 1\rangle$. Although path condition is a conjunction of branch constraints, the assignments executed in the trace are also taken into consideration in the path condition. As we can see in the example, the assignment in line 6 is considered when computing path condition. The assignment in line 6 first affects the symbolic store of y. When y is used in the condition in line 7, the symbolic store of y is used to compose the branch constraints. If there is an error in line 6, the error can affect the branch constraint generated in line 7 and therefore affect the path condition.

Different program paths can be explored iteratively through dynamic symbolic execution [51, 97]. As explained earlier, a path condition is a conjunction of different branch conditions along the path. Suppose we have a path condition $pc = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_n$ that is computed from the execution of input t along path π. We can construct formulae of the form $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, $1 \leq i \leq n$. If any of the above formulae is satisfiable, the input that satisfies the formula will drive the program to another path. More specifically, if an input t' satisfies $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, the execution of t' will follow a path π', whose first $i - 1$ branch instances are evaluated to the same directions as those in path π and the ith branch instance is evaluated to different direction from the ith branch instance in path π. By iterating the above process, different program paths can be gradually explored.

The derivation of concrete program inputs from path conditions or modified path conditions is done through Satisfiability Modulo Theory (SMT) solvers. The path conditions as well as the modified path conditions are in the form of quantifier-free first order logic formula. Program input variables constitute the free variables of the path conditions. All free variables are implicitly existential quantified. Given a formula (e.g. path condition), an SMT solver may yield the following three types of result: (i) the formula is unsatisfiable, and (ii) the formula is satisfiable, and (iii) the SMT solver is unable to determine the satisfiability of the formula. In the case that the formula is determined to be satisfiable by the SMT solver, values of all the free variables that make the formula evaluated to true are also given. When the formula is a path condition whose free variables are input variables, the valuation of the free variables corresponds to a concrete program input. If the formula is determined to be unsatisfiable, that deviating path is in fact infeasible. If the SMT solver is unable to determine the satisfiability of a formula, some under-approximation can be done to simplify the formula as explained in the next paragraph.

Ideally, a symbolic execution engine should precisely model the semantics of every instruction and the SMT solvers should be able to solve any generated path condition formulae. However, this is hardly possible due the complexity of instruction set (e.g., X86 i386 instruction set) and lack of efficient solving theory for certain operations (e.g., multiplication). An useful technique to overcome these difficulties is to do under-approximation when precise modeling or efficient SMT solving is unavailable. One typical scenario is when dealing with the values returned from native calls in Java programs. Java native calls are calls to functions implemented in other languages. If symbolic execution is implemented at the Java virtual machine level, the native functions cannot be symbolically executed. Hence, the symbolic return values of native calls cannot be precisely modeled. In under-approximation, the return values from native calls are concretized using the their corresponding values at runtime. Another typical scenario is when certain operations are not supported by SMT solvers. Suppose we have an SMT solver that only supports linear arithmetic operations. Given a formula $x * y > x - z$, we could concretize the variable y using its runtime value. The SMT solver can then solve the concretized formula.

CHAPTER III

Related Work

In this chapter, we will discuss past research that most closely relates to this thesis. We first review existing research on software testing in the context of software evolution, including regression testing and test-suite augmentation. Then we recap the research on different software debugging techniques such as slicing, statistical debugging, delta debugging, etc. Subsequently, we will discuss the past research on program change comprehension. In the end, we will discuss existing research on software contracts.

3.1 Testing of Evolving Programs

After a program is changed, program testing is employed to maintain the correctness of the changed program. We will discuss two major research topics in the area of evolving program testing - regression testing and test suite augmentation.

3.1.1 Regression testing

To test evolving programs, there have been several research efforts under the banner of "regression testing". Even though regression testing in general refers to any testing process intended to detect software regressions (where a program's functionality stops working after some change), often regression testing amounts to retesting of tests from an existing test suite. In the past, there have been several research directions that go beyond retesting all of the tests of an existing test suite. One stream of work has focused on test selection [34, 91] – selecting a subset of tests from existing test suite for running on the modified program. Another stream of work proposes test prioritization [45, 101]

–ordering tests in an existing test suite to better meet testing objectives of the changed program. Researchers have also studied test suite reduction in regression testing. A test can be obsolete for mainly two reasons: (i) the tested functionality has been removed from the software; (ii) the tested functionality can also be tested by other test cases. Harrold et al. have built a general framework for eliminating test cases [54]. The technique requires the association of test cases and test requirements. The core idea of their technique is to incrementally select test cases that satisfy the largest number of remaining test requirements. During test suite reduction, the fault detection ability of the test suite should not be reduced. Wong et al. [115] gave an empirical study on the effect of test case elimination. Their experimental results showed that the effect of removing some redundant test cases has little effect on the fault detection ability of the test suite.

Recent research by Jin et al. [60] advanced regression testing by automatically generating test cases to stress program changes. When a program is changed, a set of unit test cases concentrating on the changed portion of the program is automatically generated. These test cases are executed on both the unchanged and changed programs. Any observed behavioral differences between the two versions are analyzed and presented to the user for further insepction.

3.1.2 Test suite augmentation

While the goal of regression testing is to maintain the correctness of existing functionality, the correctness of new features introduced by changes should also be checked through testing. To this end, recent research projects [92, 105, 121] have proposed test suite augmentation — developing new tests to stress test the effect of the program changes.

The technique proposed by Santelices et al. [92] focuses on generating criteria for test suite augmentation. Their technique starts from the change and generates a criterion, satisfying which guarantees that the change effect is propagated up to certain distance. Their result can either be used to select test cases from a large test pool or be used as the criteria to drive test generation techniques. The technique requires an input that reaches the change, but how such an input is obtained is not considered in their paper. Automatically constructing a change-reaching input could be difficult for large programs with a huge number of control flow paths. The authors later extended this technique by checking the satisfaction of the criteria at runtime without generating the full criteria beforehand [95]. In this way, the technique saves expensive symbolic effort and achieves better efficiency.

Another line of research on test suite augmentation employs dynamic symbolic execution as the underlying technique. Dynamic symbolic execution techniques from Directed Automated Random Testing (DART [51]) were originally designed to explore program paths with no specific focus on program changes. Recently, based on DART, some test suite augmentation techniques [105, 121] for evolving programs have been proposed. The technique by Taneja et al. [105] integrates heuristics in dynamic path exploration to avoid paths that do not stress the program changes. These heuristics include pruning paths that cannot lead to any change and avoid paths that are unlikely to propagate effects of changes. The technique proposed by Xu and Rothermel [121] identifies "dangerous edges" affected by the change in control flow graph. Subsequently, heuristics (built on top of symbolic execution using DART) are used to stress these edges. However, after the dangerous edges are identified, the entire analysis is carried out in the changed program. It is important to note that any test suite augmentation technique that analyzes only one program version is unlikely to find test cases to stress the changes.

Apart from dynamic symbolic execution, researchers have also employed genetic programming techniques in test suite augmentation [119]. In addition, Xu et al. [120] empirically compared different factors affecting the effectiveness of test suite augmentation techniques, such as the use of existing/new tests and the underlying test generation technique (genetic programming or dynamic symbolic execution).

3.2 Program Debugging

Researchers have paid a large amount of effort to promote the effectiveness of automated debugging techniques. In this section, we discuss a few major debugging techniques from past research.

3.2.1 Program slicing

Program slicing [114] computes a subset of the program that may affect the selected slicing criterion, which is a set of program statements. Program slicing can be either static or dynamic [109, 114]. A program slice is computed by traversing data dependency and control dependency in the program. In the case of dynamic backward slicing, the program execution trace needs to be stored.

For a non-trivial program, the execution trace can be huge. The JSlice tool [109] handles this problem by compressing the trace online and performing the slicing directly on the compressed trace. Some recent research of program slicing focused on reducing the size of slicing result [111, 118, 100, 127] without loosing the most relevant program statements.

Dynamic program slice only contains program statements that affect the slicing criteria through data and control dependence. However, researchers have found that some statements affect the slicing criteria indirectly by not being executed. This is captured as potential dependence. Researchers have proposed relevant slicing by taking potential dependence into consideration [16, 53].

3.2.2 Statistical debugging

Given a buggy program and a test-suite, we would expect some test cases fail and most of the test cases pass. Statistical debugging is built on the assumption that exercising the buggy statements and exposing buggy behavior are strongly correlated. Various metrics have been proposed to measure the likelihood of a statement being buggy [13, 62]. Typically, the more often a statement is executed in failing runs (and the less often it is executed in passing runs), the higher suspiciousness score is assigned to the statement. To leverage field data, Liblit et al. proposed to collect those statistical information from end-users [71]. A sparse sampling method is designed to reduce the overhead incurred on end-users and persevere the confidentiality of user data.

Apart from program statements [61], statistical debugging has been applied to many aspects of programs, including branches [73], instrumented predicates [72] as well as a combination of branches, statements and def-use pairs [96].

3.2.3 Delta debugging

Delta Debugging was first introduced by Andreas Zeller [123]. Initially, delta debugging targeted differences between two program versions. Suppose we have two program versions, delta debugging applies binary search on the the differences between these two versions until a small set of bug-inducing changes are identified.

Binary search makes delta debugging very efficient when the change set is large. However, some program changes may be correlated. One straightforward example is that a new program variable is added, followed by some statements using it. The variable definition must be applied together with the variable usage. A key observation in delta debugging is that by dividing the changes into smaller subsets, the chance of getting an inconsistent program decreases. In the extreme case, this may involve enumerating the changes one by one.

Since the initial research of delta debugging on source code, it has been extended to work on program inputs [125], program execution states [124] and thread schedules [36]. Recently, Misherghi and Su extended delta debugging on program input with hierarchical support [79]. If the input is highly structured, their approach can work more efficiently by quickly pruning away unrelated input portions.

Delta debugging has also been applied to isolate interactions between the components of a program [82, 30]. The Jinsi tool [30] captures and replays the inter/intra-components interactions in a system. Through delta debugging, Jinsi is able to isolate a failure-inducing sequence of method calls.

3.2.4 Other software debugging techniques

Apart from the aforementioned debugging techniques, many debugging techniques proceed by comparing the failing run with passing runs in the hope that the differences can reveal the cause of the failure [89, 110, 59]. The reference passing runs are thus critical for the accuracy of these techniques. Renieris and Reiss [89] propose to select the reference passing run that is most close to the failing run from a large pool of runs. The work of Wang and Roychoudhury [110] automatically generates the reference passing runs by systematically deviating from the failing run. Reserchers have also proposed to generate the reference passing runs by switching predicates in the programs [126] or by replacing values in the programs [59]. Most recently, Sumner et al. [104] empirically compared the advantages and disadvantages of different debugging techniques that are based on execution comparison.

Along a completely different direction, DebugAdvisor [22] assists the debugging process by searching for past debugging experience of similar bugs in the existing bug database. This is based on the observation that programmers tend to get useful hints from the solutions of past similar bugs. Given a current bug to resolve, similar previous bugs are selected according to the context information, including core dump, debugger output, etc.

The recent advance in SMT solving and dynamic symbolic execution has also found their application in software debugging. Given a failing program execution where the post-condition of the program is violated, the approach proposed by Jose and Majumdar [63] first converts a failing trace into a trace formula which captures all the operations along the trace. The trace formula together with the post-condition constitute an unsatisfiable formula. The approach then computes the minimal unsatisfiable core of the full formula (trace formula and post-condition) as the potential root causes of the bug. Intuitively, the unsatisfiable core contains program fragments that prevent the post-condition from being satisfied. These program fragments are likely to contain the bug. Recently, Chandra et al. proposed the angelic debugging approach [33]. The core idea of angelic debugging is to find the program statements that can be modified to rectify the program. Since modifying these program statements can lead to the correction of the program, they are more suspicious of being the bug causes.

3.2.5 Locating defects in execution environment

Program failures are not only caused by bugs in programs themselves but also by defects in their execution environments. Most of the existing research on localizing defects in execution environment has been focused on configuration errors. More specifically, errors in configuration files have been the major targets. Attariyan et al. [23] proposed a technique based on information flow analysis to locate the root cause of configuration errors. The approach instruments application binaries to monitor the information flow at runtime and uses this information to detect the relationship between erroneous behavior and errors in configuration files. The work from Su et al. [103, 102] also targets at misconfiguration problem. They use speculative execution to examine the effect of configurations and roll-back to earlier configurations when necessary.

Holmes et al. [55] have pointed out that a developer can evolve a program's behaviors not only by changing the program's source code but also by altering the execution environment. The authors have also introduced a program partitioning approach for categorizing program changes with regards to the program's behaviors. As such, the approach is able to tell the developers which groups of changes deserve deeper developer attention. The approach, however, does not focus on locating changes in the environment that are related to an observable error.

Recently, Clause and Orso [38] proposed a record and replay technique to support debugging of field failures. Their technique first records the interaction between program and its execution environment. The recorded interaction is then replayed during debugging so that the exact failure can be reproduced. The technique intercepts all the software interactions with the environment through the OS and produces an execution recording that consists of an event log and a set of environment data. As such, the execution can be replayed without the need of the original execution environment.

3.3 Program Change Comprehension

Researchers have also studied the issues in comprehending program changes [98] and have designed various techniques to support program change comprehension. Program differencing methods [56, 21] try to identify changes across two program versions. Past research has improved upon traditional text based differencing methods by taking program structure into consideration. For example, JDiff [21] compares the control flow graphs of the two versions, whereas Horwitz's approach [56] compares the program dependence graphs across program versions. The recent iDiff [80] approach represents a program as a graph of entities (classes and methods) and utilize the interaction among program entities to improve program matching. Researchers have also tried to ease program change comprehension by observing changes at different abstraction level. Jackson and Ladd [58] propose to identify changes in input-output dependence chains in programs. The LSdiff tool [65, 74] summarizes the differences between two program versions as structural changes.

To comprehend program changes, change impact analysis has also been studied in the past [88, 81, 14]. Typically, change impact analysis techniques follow program dependency from the changed code in order to compute all the impacted program entities. By leveraging the coverage data of test cases, change impact analysis techniques can also infer the set of test cases that are affected by program changes [88, 81]. For each impacted test case, the Chianti change impact analysis tool [88] also compute the set of changes affecting the behavior of the test case.

3.4 Dynamic Symbolic Execution

Symbolic execution was first proposed for program testing [66]. The behavior of many inputs sharing the same program path is represented by the behavior of one symbolic input in symbolic execution. Therefore, symbolic execution avoids redundant testing with inputs that behave the same. The practical usage of symbolic execution has been hindered by many obstacles, such as insufficient support from SMT solvers and imprecise modeling of complex program semantics. Recently, there is a spurt of research on dynamic symbolic execution [51, 97] and its usage on path exploration. By performing symbolic execution along program paths of concrete inputs, dynamic symbolic execution can always under-approximate by concretizing some symbolic values with their runtime values when it meets obstacles.

There exist several approaches that focus on improving the efficiency of dynamic path exploration. To avoid repeating symbolic execution of the same function, function summaries are generated and used when these functions are called [49]. When the input of a program is highly structured, the grammar of the input can be used to avoid generating large amount of invalid inputs [50]. Researchers have also used the notion of "path equivalence" to alleviate the path explosion problem. However, which paths are considered equivalent vary in different approaches. The difference in the definition of path equivalence originates from the different goals of these approaches. The goal of Boonstoppel et al. [26] is to explore all possible program states. Based on this goal, two paths are equivalent if the symbolic states of all live variables are the same. The goal of Kenneth [76] is to reach some critical locations in a program. Therefore, two paths are equivalent if they cannot reach any critical locations for the same reason (blocked by the same condition). Santelices and Harrold propose the notion of path family to capture equivalent program paths [94]. A program is statically decomposed into several path families, where each path family contains several paths that share similar behavior. Instead of analyzing each path individually, a program can be analyzed at the granularity of path family. The authors also propose the notion of "path family condition" for each path family, which could characterize that path family.

Differential symbolic execution [84] aims at efficiently computing semantic summaries of program changes. Given the previous program version and the current version, unchanged program portions are abstracted as uninterpreted functions. Symbolic execution is then performed on the two program versions after the abstraction takes place. By representing common program portions as uninterpreted functions, differential symbolic execution can concentrate on the program changes and compute concise summaries of program changes.

More recently, Directed incremental Symbolic Execution (DiSE) [85] was proposed to avoid redundant path exploration after program changes. The core observation is that some paths are not affected by program changes and hence these paths do not need to be re-explored after program changes. Therefore, by avoiding exploration of unchanged paths, DiSE can achieve better efficiency on the changed program as compared to techniques that try to explore all paths of the changed program.

Csallner et al. proposed Dysy [40] to generate program invariants through dynamic path exploration. For each path explored, the path condition serves as a pre-condition and the symbolic program output is treated as a post-condition. Thus, each explored path produces a program invariant which is defined as such a (pre-condition, post-condition) pair.

3.5 Software Contracts

Design by contract [78] was proposed by Bertrand Meyer and first realized in Eiffel programming language [77]. In the simplest form of design by contract, each method has its contract in the form of pre-condition and post-condition. A method has to guarantee its own post-condition whenever its pre-condition is satisfied. When a method invocation happens, it is then the caller's responsibility to guarantee the callee's pre-condition. Apart from Eiffel, Spec# [25] also incorporates design-by-contract into its core language. Different from Eiffel and Spec#, JML provides program contracts for the existing Java programming language [69, 68].

Inspired by the concept of Design by Contract, several research projects have focused on checking program code w.r.t. program contracts. Extended Static Checking [42, 47, 39, 24] aims at automated program contract checking at compile time. In extended static checking, verification conditions are generated from program code and the program contracts; these verification conditions are dispensed via automated theorem provers. Runtime assertion checking of contracts has also been studied [35]. In a typical runtime assertion checking system, program contracts are translated into checkable assertions and compiled into the associated program. By executing these assertions, the original program contracts are checked. The recently proposed hybrid checking approach [106] combines the power of static and dynamic checking of contracts. In this approach, a runtime assertion corresponding to a program contract is inserted into the compiled program and checked at runtime only when it cannot be statically proved to be true.

CHAPTER IV

Test Generation to Expose Changes in Evolving Programs

Regression testing is one of the most commonly known software engineering activities for developing reliable software. In simple terms, it stresses "program changes" as a program evolves from one version to another, checking whether new functionality introduced by the changes is correct and whether the changes result in errors in existing functionality. Often, regression testing involves re-testing using a new test-suite containing both existing test cases and new test cases as the program evolves.

For re-testing with existing test cases, because the test-suite of a program is often huge, it is inefficient to test the changed program with all existing test cases. Most of the past research efforts in regression testing focus on this inefficiency issue and provide solutions via test selection [34, 91] (selecting a subset of the tests to be run) or test prioritization [45, 101] (changing the order in which a set of given tests is run).

However, the evolution of a program often involves addition of new functionality, and thus the test-suite should also evolve with the evolution of the program. In this aspect, the key challenge is to generate test cases related to the changes. Recent work [92, 105, 121, 75] has studied test-suite augmentation for evolving software. The main task in test-suite augmentation is to find new test cases that stress the program changes and affect the program output. Suppose a program P (with a test-suite T) evolves to a program P', i.e., P is *changed* to produce P'. A test-suite augmentation method should generate test cases that make the effect of the *changes* visible in terms of observable program output. If these test cases do not appear in the existing test-suite T, we add them to T.

Let us now examine an intuitive way of generating test cases for stressing program changes. Consider an output variable *out* in programs P and P', and let the inputs of P and P' be in_1, in_2,

. . ., and in_k. By performing a strongest post-condition computation (using symbolic execution) on program P, we represent the output variable *out* in P as a formula $\varphi(in_1, in_2, \ldots, in_k)$. Similarly, by performing a strongest post-condition computation on program P', we represent the variable *out* in P' as another formula $\varphi'(in_1, in_2, \ldots, in_k)$. We can then solve

$$\varphi(in_1, in_2, \ldots, in_k) \neq \varphi'(in_1, in_2, \ldots, in_k)$$

and the solutions are test cases (assignments of values to inputs $in_1, in_2, \ldots, in_k$) that make the output values different in the two programs.

Although the above approach is straightforward, it does not scale. Since we need to perform static symbolic execution on the program (rather than dynamic symbolic execution on an execution path of the program), it is difficult for the approach to scale up to large real-world programs. In this chapter, we develop a scalable approach for test-suite augmentation. Our approach builds on the *execute-infect-propagate* (PIE) paradigm [108]: the new tests should (i) execute the program changes, (ii) infect the program state, and (iii) propagate the infection to the output.

From a high level, our approach works in two steps. The first step is to generate an input satisfying the *execute* property in the PIE paradigm. Given a change c (in source code from one program version to another), we find a path that reaches c in the control flow graph. We then perform dynamic symbolic execution along the path to find an input t that makes the program execute the path leading to c. The second step of our approach aims to generate test cases that satisfy the *infect* and *propagate* properties, in addition to the *execute* property. Since any infection in program states is reflected as different variable values (after the change) in the two program versions, we observe that state infection and propagation may be avoided if (i) variables affected (directly/indirectly) by a program change are defined but not used, or (ii) the uses of affected variable cannot propagate the change effect forward (by affecting other variables). If an assignment of some affected variable v is not used in the execution of test t, we find a new test t' (aided by symbolic execution along a path) that can execute the uses of v. If the use of variable v does not propagate the change effect forward, we find a new test t' that can propagate the effect in v.

The preceding describes our method in a nutshell. The key to the method's efficiency lies in our strategy in avoiding symbolic execution on programs. Our approach performs every symbolic execution along a program path. Note that symbolic execution along a program path has additional overhead in enumerating and searching for the "right" path. We use various analysis methods to guide us to the "right" path. When trying to execute the change, the shortest path in the control dependency graph guides us to efficiently locate and construct a path to the change. When trying to propagate the change effect, we identify the reasons for which the propagation terminates, and propagate the change effect to program output while detecting branch correlations on-the-fly (which allows us to avoid infeasible program paths).

Performing symbolic execution along a path also helps us avoid the memory alias problem: since the symbolic execution is along a program path, all memory references are disambiguated.

In our experiments, we tried our test-case generation approach on two programs: `tcas`, a small program with multiple versions (each encoding a different change) from the SIR repository [43] and `libPNG`, a large-scale library for manipulating PNG images (27977 lines of code). For both the subject programs, our method successfully generated test cases that stress the changes (by producing different program outputs) for almost all the program versions.

4.1 Motivating Example

In this section, we motivate the problem of test case generation in evolving programs with an example.

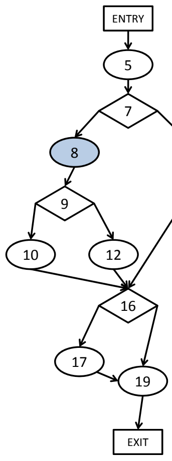
Figure 4.1 shows an example of evolving programs. In the example program shown in Figure 4.1a, the variable x is the input variable and the variable o is the output variable. The change is at line 8, where the assignment of variable y is changed from 3 to 2. We now need to synthesize a test case that stresses this change. Figure 4.1b and Figure 4.1c show the control-flow graph (CFG) and control-dependency graph (CDG) of the example program, respectively, in which the changed statement is marked in dark color.

In order to test the change, we need test cases that can drive the program to the changed statement in P', and can result in an output of P' different from that of P.

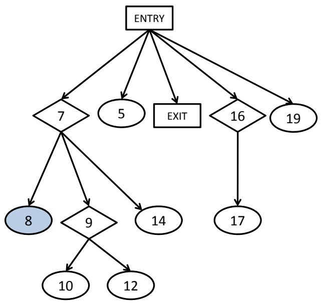
Reaching the change Suppose we have the following test case for the original program P: *x = -1*. The execution trace of the test case *x = -1* in the changed program is {*5, 7, 13, 14, 16, 19*}, which does not cover the change at line 8. Note that there are two branches in the trace, namely line 7 and

```
 1 int x; /* Input variable */
 2 int y;
 3 int o; /* Output variable */
 4
 5 input(x);
 6
 7 if (x > 0) {
 8     y = 3; //change: y = 2;
 9     if (x - y > 0)
10         o = y;
11     else
12         o = 0;
13 } else
14     o = -1;
15
16 if (x > 20)
17     o = 10;
18
19 output(o);
```

(a) Example Program



(b) Control Flow Graph



(c) Control Dependence Graph

Figure 4.1: A motivating example to illustrate our test generation approach.

line 16. If we execute these branches differently, we may drive P' to reach the change. However, some of the branches, such as line 16, do not help to reach the change. We use the CDG to identify such branches. From the CDG in figure 4.1c, we can see that node 7 can determine whether the change (node 8) is executed, while node 16 cannot. So we want to execute branch 7 differently. This analysis (for finding which branch to evaluate differently) can be automated via traversal of the CDG. In this example, we construct the formula $(x > 0)$ by flipping the evaluation of the branch in line 7. Solving this formula $x > 0$, we get an input, say *x = 1*, that stresses the change.

Affecting program output Using the new input *x = 1* to test both program versions, we get the same program output 0. Even though the change is executed, its effect is not propagated into the program output. We can see that propagation of the change effect (*y = 2*) stops at the branch in line 9 — this branch is evaluated to false in both the program versions for *x = 1*. To propagate the effect of the program change past this branch, we need the branch to be evaluated differently in P and P', which is expressed in the following symbolic formula.

$$(x > 0) \wedge (x - 2 > 0) \wedge \neg(x - 3 > 0)$$

An x satisfying this formula will execute the change, *and* evaluate the branch in line 9 differently in the two program versions. This gives us the answer *x = 3*. Executing P and P' using the input *x = 3*, we find that the program output is different in both program versions. Thus, we have generated a test case which executes the change, and propagates its effect to the program output.

4.2 Our Approach

Assumptions. In this chapter, we assume that all changes are independent and none of the changes is inside a loop. The meaning of independence is that the execution, infection and propagation of a change is not affected by whether other changes are executed. This assumption allows us to handle each change at a time.

Our goal is to generate test cases that make the effect of software changes observable through difference in program outputs. A test case for regression testing should drive the changed program to execute the changes, and the program states affected by the changes should result in difference in program outputs.

To meet the above requirements, our approach uses symbolic execution on program traces to guide the exploration of program paths. The exploration in our approach is guided by the execute-infect-propagate philosophy (execute the change, infect program state and propagate the infection to output), instead of program path coverage. Our approach is divided into two steps. First, given a program change, we use symbolic execution to find the constraints on program input variables that need be satisfied to execute the change. Second, given the constraints (and sample test inputs) generated in the first step, we tune the sample test inputs into test cases that not only execute the change but also propagate the effect of the change to the program output.

We now explain these two steps in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Driving the program to reach the changes

Our approach iteratively constructs the change-reaching test inputs using symbolic execution. In each iteration, a new test input is generated to drive the program execution closer to the targeted change. This process continues until we get a test input that executes the given program change.

The basic intuition of this step is as follows. We run the test-suite T_P of the old program P on the changed program P'. If any test $t \in T_P$ executes the changed statement $stmt$ in the changed program P', we return t. Otherwise, we collect the path condition of P' when it processes t and

Algorithm 4.1 Reaching the change

```
Input:
$P\ P'$ : original and modified program
$T_p$ : The existing test-suite for $P$
Output:
$t_{new}$: A test case that reaches the difference between $P$ and $P'$
$unexpanded = \emptyset$
$S = \emptyset$
$stmt = ...$ // this is the changed statement
$CDG_{p'} = computeCDG(P')$
$G_{stmt} = computeDistGraph(CDG_{P'}, stmt)$

// step1: run the existing test-suite
for all $t \in T_p$ do
    $ret = Execute(P', t)$
    if $ret \neq null$ then
        return $ret$
    end if
end for
// step2: construct new test case
while $unexpanded \neq \emptyset$ and not timeout do
    select $\varphi \in unexpanded$ with minimum distance
    remove $\varphi$ from $unexpanded$
    let $\varphi = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k)$
    construct $\theta = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k)$
    solve $\theta$
    if $\theta$ is satisfiable then
        let $t_\theta$ be an input that satisfies $\theta$
        $ret = Execute(P', t_\theta)$
        if $ret \neq null$ then
            return $ret$
        end if
    end if
end while
return $null$

procedure $Execute(P', t)$
    execute $P'$ with input $t$
    let $f = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$ be the path condition
    for all i from 1 to m do
        $\varphi_i \overset{def}{=} \psi_1 \wedge \ldots \psi_i \wedge \psi_{i+1}$
        if $dist(\psi_{i+1}) = 0$ then
            return $t$
        end if
        if $dist(\psi_{i+1}) \neq \infty$ and $\varphi_i \notin S$ then
            $S \bigcup = \varphi_i$
            $unexpanded \bigcup = \varphi_i$
        end if
    end for
    return $null$
end procedure
```

manipulate the path condition to generate a new input that "advances" the execution in P' towards the changed statement $stmt$.

Assume the path corresponding the test input t is π, which does not reach the changed statement $stmt$, and π's path condition is $\psi_1 \wedge \psi_2 \wedge \ldots \psi_m$. We look for inputs that can make P' deviate from π to reach $stmt$. The deviation can be made at any of the branches along π. If an input makes P' deviate from π at the k-th branch, it must satisfy the following condition:

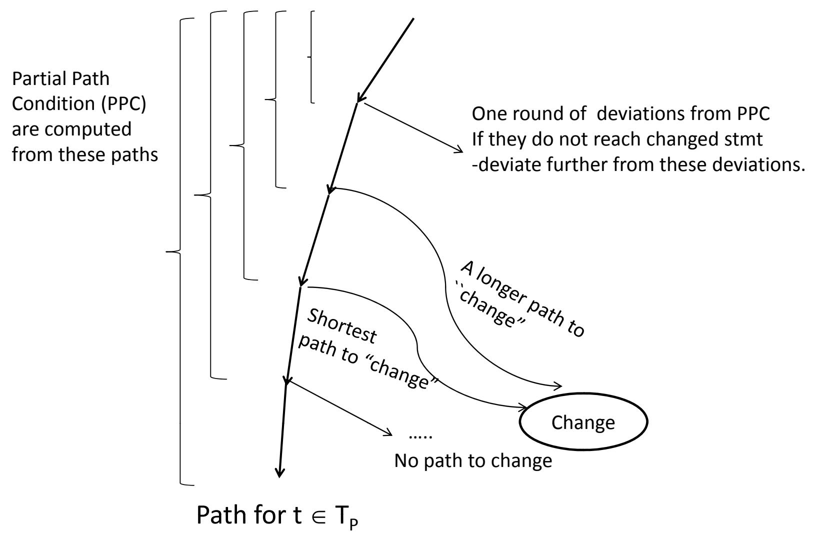
$$\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$$

That is, the new input satisfies the first $k-1$ branching conditions of π, but does not satisfy the k-th branching condition. If it is satisfiable[1], we can generate an input t_i satisfying the formula. The new input t_i leads to a different path, which can potentially make P' reach the target.

One of the major challenges faced by this intuitive solution is to handle the large number of branching conditions. The intuitive solution cannot handle large-scale software with limited time and computing resources. To address this problem, we observe that the negation of certain branching conditions cannot help to reach the change. For example, in our motivating example described in Section 4.1 (illustrated in Figure 4.1), negating the branch condition at line 16 will not help to drive the execution closer to the change. Therefore, we need to measure whether the negation of a branch condition can drive the execution in P' closer to the change. In our approach, the measure of closeness or proximity of a path π (w.r.t. the change $stmt$) is given by the *length of the chain of control dependencies that need to be traversed from π in order to reach the change $stmt$.*

In this step, we are working with program paths that have not yet reached the change $stmt$. The path conditions of such paths are referred to as *partial path conditions* (abbreviated as PPC) in our terminology. The term "partial" specifically emphasizes that the change has not yet been reached. Also, note that the path condition of any program path π is a quantifier-free first order logic formula satisfied by all test inputs that drive P' to execute along π. The notion of partial path conditions is central to our approach. Since we are trying to construct a feasible program path (*i.e.*, exercised by at least one program input) in the changed program P' that terminates at the given change $stmt$, our method works by constructing a path that gets "closer" to the change. These partial path conditions

[1]Note that this formula may be unsatisfiable. In that case, other deviations will be tried out based on heuristics explained later.



Partial Path Condition (PPC) are computed from these paths
One round of deviations from PPC If they do not reach changed stmt -deviate further from these deviations.
A longer path to "change"
Shortest path to "change"
Change
.....
No path to change
Path for t ∈ T_P

Figure 4.2: Summary of Algorithm 4.1

are then explored to see whether they can reach the changed statement $stmt$, otherwise we explore further deviations from these partial path conditions of t_i in a similar way (*i.e.*, by negating the last branch condition). This, in essence, is the heart of our algorithm for reaching the changed statement.

Algorithm 4.1 captures the core method for generating a change-reaching test case. The inputs to the algorithm are: the old program P, the changed program P', and the test-suite T_P for program P. There is a single change between the programs P and P', that is, P and P' differ via a unit change statement $stmt$. The output of the algorithm is a test input t_{new} that executes this changed statement $stmt$. The elements in set S and $unexpanded$ are PPCs (partial path conditions). Set S is used to maintain all the PPCs we have seen to avoid redundancy. Set $unexpanded$ contains all the PPCs that have not been tried out. In the algorithm, the CDG of P' is first computed, then the distance from each node to $stmt$ is computed from the CDG of P'. This is shown by the function $dist$ in the algorithm. Thus, let b be a program branch in the changed program P' whose condition is ψ. Then $dist(\psi)$ is the shortest path from b to the changed statement in the static inter-procedural control dependency graph of P'. The distance for a PPC φ_i ($\varphi_i = \psi_1 \wedge \ldots \psi_i \wedge \psi_{i+1}$) is the same as the distance of the last branch condition in φ_i, that is, $dist(\varphi_i) = dist(\psi_{i+1})$.

The algorithm first looks for a change-reaching test case in the existing test-suite T. If no existing test case can reach the target, our algorithm iteratively constructs such a test case in the second step. In each iteration of the second step, we choose a PPC that is closest to the target from the PPCs that have not been tried out. By deviating at the last branch of the path, we try to get closer to the target in each iteration.

From the description of Algorithm 4.1, we see that it is similar to the generational search strategy used in SAGE [52]. In SAGE, given a path π for a test, new paths are explored by negating each branch in π, similar to our algorithm. The main difference between our method and SAGE is in the way that we choose the branches to negate. Given a set of partial path conditions (obtained from the path condition $\psi_1 \wedge \ldots \wedge \psi_m$ of a program path)

$$\{\psi_1 \wedge \ldots \wedge \psi_i \wedge \psi_{i+1} \mid 0 \leq i < m\}$$

we use the *distance* between ψ_k and the changed statement $stmt$ to prioritize the selection of the branch condition to negate. The distance between ψ_k and $stmt$ is defined as the weighted shortest path from the program branch contributing to ψ_k to the changed statement $stmt$ in the static inter-procedural control dependency graph of the changed program P'. If the k-th branch condition has the smallest distance from $stmt$, it will be negated first.

4.2.2 Propagating the effect of a change to program outputs

Executing the changed statement is not sufficient for reflecting the change in the output. A change should first affect some program states, and the effect of the change should be seen from the output (via propagation of the affected states). In reality, a program can have a large number of paths. Therefore, any propagation technique is either path insensitive and hence imprecise, or path sensitive but not scalable to large programs. In this section, we propose a practical technique for propagating the effect of the change in an iterative way.

Why a change may not propagate to output To build a method that propagates the effect of a change to output, we investigate common reasons for which propagation failed to reach the output. We use P and P' to denote the original program and changed program respectively. The execution trace of input t in program P is denoted as $trace(P, t)$. For a variable definition statement instance s', we say the defined variable is "*affected*" when it has different value from the value defined in s. Statements s and s' are aligned statements in $trace(P, t)$ and $trace(P', t)$ respectively. Note that the effect of a change could have propagated to certain distance before the propagation stops. So we do not need to start the propagation from the change each time, we only need to intervene when the effect stops propagating in the execution. The change cannot affect the output when none of the affected variables can affect the output. Suppose an affected variable v is defined in s' in $trace(P', t)$, the reasons that the different value in v stops propagating, may now be enumerated as follows.

- The uses of v are never executed before v is redefined in $trace(P', t)$. As an example, consider the program in our example.

  ```
  y = 2; /* originally y = 3; */
  if (x > 0){ o = y;} else { o = 0;}
  ```

  Here, the changed statement affects the definition of `y`. However, in the execution trace for the input $x = 0$ (and thus $x > 0$ is false), the use of this definition is never executed. The output variable o is not affected by the change.

- Uses of v are executed before v is redefined in $trace(P', t)$, but the use cannot result in other affected variable. As an example, consider the following program. Let `x` be the input, and `o` be the output of this program fragment.

  ```
  y = 2; /* originally y = 3; */
  if (x - y > 0){ o = y;} else { o = 0;}
  ```

  Here `y` is the variable whose definition is affected by the change. The use of the definition is also executed in the form of the condition `(x-y > 0)`. However, for input $x = 0$, it does not make a difference in the control flow and the subsequently calculated output value for `o`.

Propagating changes to the output We handle the two reasons for which the effect of a change may not propagate to the program output as follows. If the uses of the affected variables are not executed in an execution trace π, we drive the program execution towards the use statements. This is achieved via a method similar to Algorithm 4.1, where we set the use statements as the target, instead of the changed statement.

In the second case, the uses of the affected variables are executed in the trace but no program variables are affected by the use. Suppose the affected variable v is defined in statement instance s', and v is subsequently used in statement instance m'. Obviously, m' was not able to propagate the effect of v forward. According to the type of m', we use the following steps to propagate the effect in v.

- If m' is a variable definition statement, we compute the so-called "*transfer condition*" [90] (Definition 4.12 below) of the statement. Intuitively, the transfer condition of an expression is the condition under which the value of the expression will be different if one of its operands is different. Given the transfer condition, we use it in symbolic execution to compute a test input which propagates the effect of the change, while following the same path.

  Definition 4.12 (Transfer condition)**.** *The transfer condition for $exp = oprand1$ op $oprand2$ with respect to $oprand1$ is the condition under which exp has different value if $oprand1$ has different value.*

  For example the transfer condition of `x + y` is *true* since if either operand is different, the sum is different. On the other hand, the transfer condition of `x * y` is $y \neq 0$ for a change in `x`.

- If m' is an branch and v is used as the condition in m', we compute the condition which makes m and m' to be evaluated *differently* in the two programs P and P' in order to produce different outputs in P and P' (m and m' are aligned statements in $trace(P,t)$ and $trace(P',t)$). We use symbolic execution to find an input that reaches m in programs P and m' in P' and then evaluates m and m' differently in the two programs.

Algorithm for Propagating Change The algorithm for propagating change effects is shown in Algorithm 4.2. The algorithm iteratively calls procedure *Propagate* to construct a new input that can propagate the change effect forward. The procedure *Propagate* first executes P and P' using the input t. Then it analyzes the execution traces of P and P'. An important concept here is the *change effect propagation tree (CEPT)* calculated by **CPTree** at line 11.

Definition 4.13 (Change effect propagation tree)**.** *Given a unit change, a CEPT CT is defined as follows. The nodes of a CEPT are statement instances in the changed program P'. There is* *an edge from α' to β' in CT if and only if (i) β' is dependent on α' (either control dependence or data dependence) (ii) the operands of β' have different values than those of β, where β is the corresponding statement of β' in P. Each leaf node in the CEPT is a place where the change effect propagation terminates.*

Note that the CEPT defined above is a polytree [83]. A polytree is a restricted DAG (Directed Acyclic Graph). While a DAG allows multiple undirected paths between two nodes as long as they do not form directed cycle, a polytree allows at most one undirected path between any two nodes. Compared to nodes in a tree, a node in a polytree can have more than one parent node, which represents that more than one operand in a statement is affected by the change.

Identifying terminating locations of effect propagation The CEPT is computed by dynamic forward slicing. During slicing, our approach compares the operand values of corresponding statement instances in both programs to determine whether the change effect has stopped propagating. If the operand values of a statement instance s' in P' are the same as those of the corresponding statement instance s in P, s' will not be included in the propagation tree, because it will not cause differences in the output. In other words, the change effect does not propagate to s'.

When comparing the operand values of corresponding statement instances in both P and P', if s' of P' has no corresponding statement instance in P, the operands for s' are treated as different from those in P. To compare the operand values, we use trace alignment to find the corresponding statement instance s (in execution trace of P) of s' (in execution trace of P'), which is the $align(P, P', t)$ function in Algorithm 4.2. For simplicity, we use $sat(\varphi)$ to represent an input instance that satisfies φ.

Propagating change effects further According to the type of the leaf nodes in the change effect propagation tree, we use different methods to drive the propagation of the change effect forward. If an affected variable is defined but never used, the procedure *PropNouse* is called. In this procedure, we first use def-use analysis to identify all the use locations of the defined variable, and use our Algorithm 4.1 to reach at least one of these locations.

If an affected variable is used but does not propagate the effect forward, according to the type of the use statement s', two different procedures are used for propagation. If the statement s' is a

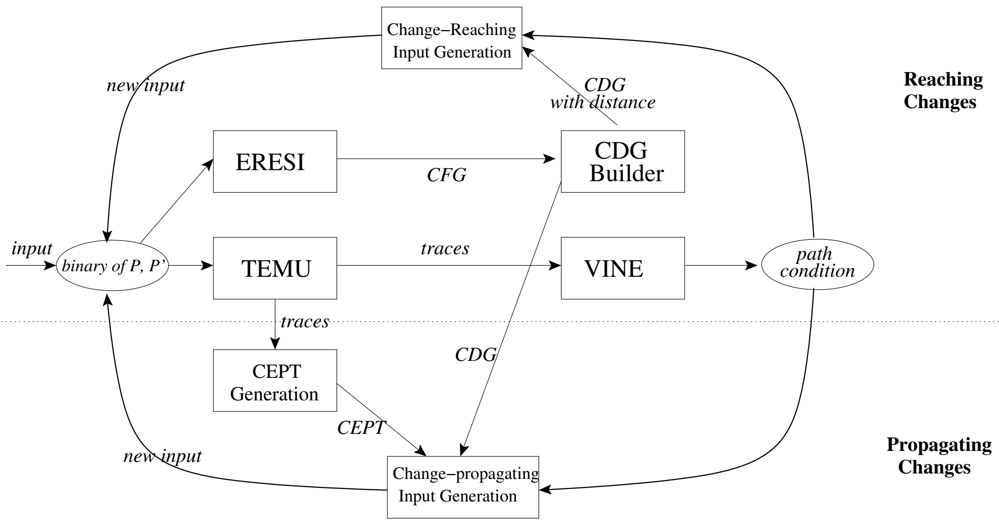


Figure 4.3: Component view of our approach (CEPT = Change Effect Propagation Tree, CDG = Control Dependency Graph).

variable-definition statement and the defined variable is used, it can only become leaf node (of CEPT) when the transfer condition is not satisfied. Procedure *PropTransfer* is invoked in this case. In this procedure, we first use symbolic execution to compute the transfer condition TC. Then we get a new input that satisfies TC by solving the formula $f' \wedge TC$ where f' is the partial path condition up to s'. If an affected variable is used as the condition in a branch, it becomes a leaf node when the branch is evaluated the same in both versions. In this case, we use *PropCjmp* to execute the branch differently in two versions. If the procedure *Propagate* cannot generate a change stressing test input by analyzing the new program version, we apply *Propagate* to the old program version.

4.3 Implementation

We implemented our approach on the x86 platform based on the BitBlaze [99] binary analysis framework. We show the component view of our implementation in Figure 4.3, where the components used in our solution are shown as boxes, and the data used by the components are shown as italic labels of edges. Some of the important intermediate data are shown in ovals.

SMT formula solving is used extensively throughout our approach. We used the Boolector SMT solver [28] for all our formula solving tasks.

Algorithm 4.2 Propagate the change effect

```
1: Input:
2: $t$: a change reaching input
3: $P$ $P'$: original and modified program
4: Output:
5: $t_{new}$: a input that have different output in $P$ and $P'$
6: procedure $Propagate(P, P', t)$
7:     $align(P, P', t)$
8:     let $stmt$ be the difference between $P$ and $P'$
9:     execute $t$ in $P'$ and $P$ to get the execution traces
10:    $T = CPTree(stmt, P', P, t)$
11:    for all leaf node $s'$ in $CT$ do
12:        if $s'$ is a variable definition statement then
13:            if variable defined by $s'$ is not used in $t$'s trace then
14:                $ret = PropNouse(s')$
15:            else
16:                $ret = PropTransfer(s')$
17:            end if
18:        else
19:            $ret = PropCjmp(s')$
20:        end if
21:        if $ret \neq null$ then
22:            return $ret$
23:        end if
24:    end for
25:    return $null$
26: end procedure
27: procedure $PropNouse(s')$
28:    $U = useSet(s')$ // All first uses of the definition of $s'$
29:    for all $u \in U$ do
30:        execute Algorithm 4.1 using $u$ as the target
31:        let $ret$ be the return value from Algorithm 4.1
32:        if $ret \neq null$ then
33:            return $ret$
34:        end if
35:    end for
36:    return $null$
37: end procedure
38: procedure $PropTransfer(s')$
39:    compute the transfer condition $TC$ for $s'$
40:    let the partial path condition up to $s'$ be $f'$
41:    if $f' \wedge TC$ is satisfiable then
42:        return $sat(f' \wedge TC)$
43:    else
44:        return $null$
45:    end if
46: end procedure
47: procedure $PropCjmp(s')$
48:    let $s$ be the corresponding statement instance for $s'$ in P
49:    let the partial path condition up to $s'$ for $t$ in $P'$ be $f' = \psi'_1 \wedge \ldots \psi'_{i'} \wedge \psi'_{i'+1}$, the path condition up to $s$ for $t$ in $P$ be $f = \psi_1 \wedge \ldots \psi_i \wedge \psi_{i+1}$
50:    let $\varphi = f \wedge \psi'_1 \wedge \ldots \psi'_{i'} \wedge \neg\psi'_{i'+1}$
51:    let $\varphi' = f' \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$
52:    if $\varphi$ is satisfiable then
53:        return $sat(\varphi)$
54:    else if $\varphi'$ is satisfiable then
55:        return $sat(\varphi')$
56:    else
57:        return $null$
58:    end if
59: end procedure
```

4.3.1 Architecture of our implementation

Reaching changes The top portion of Figure 4.3 illustrates the implementation of the first step of our approach: finding inputs to reach the change. Our approach first computes the static control-flow dependency graph (CDG): it uses the ERESI tool [2] to generate the static CFG, and then uses our module *CDG builder* to compute the inter-procedural CDG and distance graph from the static CFG. The distance from a node v to the target is defined as the shortest path from v to $target$ in weighted CDG. In a weighted CDG, auxiliary edges (such as function call to the start of the called function) are associated with weight 0. All other edges have weight of 1. We use Dijkstra's algorithm to compute the distance of all nodes (to the change) using one pass of the algorithm.

Next, our approach iteratively constructs an input to reach the change. Given the binary P' and a test case t, our approach generates an execution trace of P' using BitBlaze's TEMU component. TEMU is a whole-system emulator based on QEMU [4]. It emulates a PC system, which runs operating systems such as Windows and Linux. TEMU supports logging instructions executed in the emulated PC and tracking instruction operands that are dependent on program inputs (tainted operands) using taint analysis. Next, our approach uses BitBlaze's analysis component, VINE, to generate the path condition of the execution trace. The path condition is represented by VINE's intermediate language.

With the CDG and path condition, our approach uses our *change-reaching input generation* module to select a branching condition to negate based on the distance to the change statement in the CDG. It then generates a new input that drives P' to execute closer to the change. If P' reaches the change using the new input, our approach continues to the next step. Otherwise, the above process is repeated using the new input until it generates an input that leads P' to execute the changed statement.

Propagating effects of changes The bottom portion of Figure 4.3 illustrates the implementation of the second step of our approach: finding inputs to propagate the effects of the change to the program output. If the input generated in the previous step (the input that reaches and executes the change) cannot affect the output, our approach uses it to generate an execution trace of P' and compute the path condition. Then, our approach generates the Change Effect Propagation Tree or CEPT (refer Def. 4.13) to decide how to further propagate the change effect towards the output.

As described in our algorithm 4.2, trace alignment is needed to compute the CEPT. Our approach views each trace as an instruction sequence, and aligns the traces of P and P' using minimum editing distance. From the alignment result, we compare the operand values of P' with the corresponding instructions in P, which is required by our approach to decide whether a node is a leaf node in computing CEPT. For def-use analysis, we implemented a simple def-use analysis module without pointer aliasing support.

With the above information, our approach constructs the CEPT by performing forward slicing on the x86 instruction trace. Each instruction is treated as a statement in the slice. One of the practical challenges is caused by memory and register allocation in different program traces. The memory addresses of the same variable in two binary versions are often allocated differently by compiler or loader. Therefore, when such addresses become the instruction operands, the aligned instructions will have different operand values. For example, `mov EAX, [EBX]` will have different EBX value if the variable pointed to by the EBX is allocated at different addresses in the traces of P and P'. However, the difference in memory address does not imply a different program state, which is defined by the contents of variables, instead of addresses of variables. Similar issue happens with the stack registers ESP and EBP. To address this problem, our approach does not treat difference in memory address operands and stack register operands as different program states.

4.3.2 SMT solving optimizations

We note that in Algorithm 4.1 (for constructing an input which reaches the change), many of the formulae $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$ constructed for reaching the changed statement may be unsatisfiable. For example, when there is no dynamic data dependence chain between ψ_k and input variables (along the path which results in the partial path condition $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$), the formula $\psi_1 \wedge \ldots \wedge \psi_{k-1} \wedge \neg\psi_k$ is unsatisfiable. Naturally there is no point in submitting such formulae to the SMT solver. In this way, we reduce a significant amount of SMT queries, which leads to more than 90% reduction of SMT solving queries in our experiments.

4.3.3 Handling branch correlations

Most of the SMT formulae used in our algorithms are based on path condition. We noticed some branches along the path could be correlated. Suppose the branch we are trying to negate is b_β, there is an earlier branch b_α that is correlated with b_β. This correlation could make the constructed formula unsatisfiable. We handle the common cases of "immediate conflicts" between (branch, branch) and (assignment, branch) pairs as follows.

1. `if(x>2){ ... // x is not modified here}`

   `if(x<0){ //target }`

   Given a trace for say `x == 3` which evaluates the first branch to true and the second branch to false, suppose we want to flip the evaluation of the second branch statement to reach the target. However evaluating `x > 2` to true and then `x < 0` to true constitutes an infeasible path in the control flow graph.

   Solution: When we solve formula $\theta = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \neg\psi_k)$, if the branch corresponding to ψ_k is b_k, we perform a backward slicing on the trace from b_k. All the branch conditions that are not in the slice are removed from the path condition θ. In this way, we keep all the branches that are relevant for reaching b_k, and we also keep all the statements that are used for computing the branch condition. At the same time, branches that are not essential for reaching b_k but may prevent the negation of b_k are removed from the slice.

2. `if(x>0){ y = 1; } else{ y = 0; }`

   `if(y){ //target }`

   For the input $x = 0$, we can see y is set to 0 from the execution trace. To reach the target, we need to negate the first branch that the definition of y is dependent on.

   Solution: If we find a branch condition ψ_k is not "tainted" (dependent on the input via a chain of data dependencies), we cannot directly negate this branch. Suppose the last definition of ψ_k's variables in the trace is def, we use backward slicing to find all tainted branches that def is dependent on. By negating one of these branches, we may evaluate ψ_k differently.

4.4 Evaluation

To examine the efficacy of our approach, we evaluated our approach using two subject programs. In this section, we report our empirical evaluation results.

4.4.1 Experience with `tcas`

The first program we used to evaluate our approach was `tcas` from the SIR repository [43]. `Tcas` is an aircraft collision avoidance system. It has an original version (the golden program) and 41 changed versions with seeded bugs, exactly one line of a change for each bug. The `tcas` program from SIR reads inputs from command line, we modified the program to read inputs from a file. To stress our test generation method fully, we took an initial test-suite with only one randomly generated test case in this case study. Then we apply Algorithm 4.1 on this test-suite to generate test cases reaching the change.

Our technique uses 80 runs to reach all the 41 changes, about two runs to reach one change on average. Out of the 41 versions, in 8 versions, the inputs generated by Algorithm 4.1 already produced different program outputs; thus change effect propagation is not needed in these cases. These inputs are returned by our approach as the test cases to augment the test-suite.

The remaining 33 buggy versions of `tcas` needs to go through change effect propagation (as shown in Algorithm 4.2), to generate test cases. Our approach successfully generated test cases that show different program outputs (w.r.t the original `tcas` program) in 31 out of the 33 program versions. In the remaining two program versions, because of incorrect program alignment at line 7 of algorithm 4.2, the CEPT was not computed correctly. The node where change effect propagation terminates was not identified as a leaf node in the incorrect CEPT.

Now we discuss two cases where change effect propagation are needed in `tcas`. Through these examples, we show that how our techniques propagate the change effect forward towards the output.

Affected variable defined but not used. Figure 4.4 shows an example in version 3 of `tcas`, in which the change effect cannot propagate because the affected variable $intent\_not\_known$ is defined but not used. Note that in the example code, one line of source code is treated as multiple instructions in our technique. In the example, because the operator is changed from $\&\&$ to $||$, the variable $intent\_not\_known$ is evaluated to different values in two different versions in execution. However, because the value of $tcas\_equipped$ is false, the variable $intent\_not\_known$ is never used after its definition (note that a $\&\&$ is defined using short circuit evaluation, that is, if the first operand is false, the second operand is not used). To propagate the effect of the change, we employ our

```
//original version
intent_not_known = Two_of_Three_Reports_Valid
 && Other_RAC == NO_INTENT;
alt_sep = UNRESOLVED;
if (enabled && ((tcas_equipped
 && intent_not_known) || !tcas_equipped))

//changed version
intent_not_known = Two_of_Three_Reports_Valid
 || Other_RAC == NO_INTENT; /* logic change */
alt_sep = UNRESOLVED;
if (enabled && ((tcas_equipped
 && intent_not_known) || !tcas_equipped))
```

Figure 4.4: Variable $intent\_not\_known$ defined but not used

```
//orginal version
enabled = High_Confidence
 &&(Own_Tracked_Alt_Rate <= 600)
 &&(Cur_Vertical_Sep > MAXALTDIFF);

//changed version
enabled = High_Confidence
 &&(Own_Tracked_Alt_Rate <= 700)
 &&(Cur_Vertical_Sep > MAXALTDIFF);
```

Figure 4.5: Propagation stops because of branches

algorithm 4.1 to reach the statements where $intent\_not\_known$ is used. Algorithm 4.1 negated the value of $tcas\_equipped$ to execute the condition test on $intent\_not\_known$.

Propagation stops because of branches. From our experience in the experiments, it is very common that the propagation terminates because of a branch is evaluated similarly in both versions. Figure 4.5 shows the case in version 13 of `tcas`. The value compared with $Own\_Tracked\_Alt\_Rate$ is changed from 600 to 700. Only when $Own\_Tracked\_Alt\_Rate$ is in $(600, 700]$, the effect of the change propagates. Through symbolic execution, our technique found a value 604 for $Own\_Tracked\_Alt\_Rate$ such that variable $enabled$ is evaluated to different values in two versions.

Comparison with [105]. We compare our results with [105] — the only work that generates test cases to stress program changes. Other research efforts on this topic, such as [92], generate criteria for propagating effects of changes, but they do not generate test cases to reach *and* stress a program change. Therefore, we cannot compare our experimental results directly with those of [92].

In [105], they build their work based on PIE model. They provide heuristics to avoid exploring paths that (i) cannot lead to *execution* of the change (ii) cannot lead to state *infection* (iii) cannot lead to affected state *propagation*.

To compare with [105], we used the result from first 11 changed versions of the `tcas` program, the same versions used in the evaluation of [105]. For each version, our technique started with a random generated input, and interactively generated new inputs to reach the change. The technique in [105] used 95 runs in total to reach all the 11 changes. In contrast, our technique used only 32 runs to reach all the 11 changes. Note that we compare the number of runs for reaching the change, not the number of runs for propagating the change effect to the output. This is because, [105] does not report the number of runs for propagating the effect of the change to the output. Our technique can generate change-reaching inputs with much fewer runs, because the path exploration in our technique is guided by a target. We use the notion of distance in the control dependency graph to prioritize exploring shorter paths to the change.

In addition to the number of runs reported above, our technique also made less number of calls to the SMT solver. We used data tainting method to identify branches that cannot be executed differently, as is described in Section 4.3.2. In our experiments, we found that this optimization led to significant reduction in the number of calls to the SMT solver. More than 90% branch conditions are not tainted, and thus can be eliminated, in both of our case studies.

4.4.2 Experience with `libPNG`

In our second case study, we studied the changes between two versions of the `libPNG` program. `LibPNG` is a open-source library for manipulating PNG image files. It supports almost all the features of PNG file format. We used two consecutive versions from `libPNG`, v1.2.20 and v1.2.21. *Each version has a large code base, running into around 28000 lines of code (23.3% of which are executed in our experiments).*

We first remove all the obvious syntactic changes that do not affect the semantics of the program. After removing these changes, we are left with 10 changes. Each of these 10 changes are independent. Therefore, we can construct intermediate versions of `libPNG` by applying the 10 changes to version v1.2.20 one by one. We use c_i to denote change i and use v_0 to v_{10} to denote the intermediate versions (Version v_0 is v1.2.20 and v_{10} is v1.2.21). Version v_i is obtained by applying c_i on Version

```
//original version
png_byte red_high =
 (trans_value->red > 8) & 0xff;

//changed version
png_byte red_high =
 (trans_value->red >> 8) & 0xff;
```

Figure 4.6: An example change from `libPNG`

v_{i-1}. Because the changes are not correlated, if a change c_i can affect the output in v_{i-1} and v_i, it can also affect the output in v1.2.20 and v1.2.21.

Instead of randomly generating test inputs, we used the PngSuite [107] as the test-suite for evaluating Algorithm 4.1 on `libPNG`. PngSuite is a large collection of PNG files to test PNG applications. The creator of PngSuite aims to represent all the PNG formats when the suite was created in 1998. Because `libPNG` has been evolving with the evolution of PNG specification, some new features in `libPNG` cannot be fully tested by PngSuite any more. This is the exact situation where test-suite augmentation is needed due to program evolution.

`LibPNG` comes with a test driver to show how the library should be used. We modified the test driver to make the changes statically reachable. We tried to make the changes to the test driver minimal.

Eight out of all ten changes were reached by existing test cases in the PngSuite. For the remaining two changes, our algorithm was able to construct new PNG files that can drive the execution to the changes. A PNG file consists of multiple chunks with different information. Each chunk contains chunk type, chunk length, checksum, and the chunk data. Most functions in `libPNG` are chunk-specific. For example, a function for handling chunk of type `iTXt` is only called when there is a chunk of type `iTXt` in the input PNG file. These two changes appeared in functions that handle the `iTXt` chunk type. Since `iTXt` only appears in v1.2 of PNG specification, which was released in 1999, no PNG files in PngSuite (created in 1998) were able to test these two changes. Our algorithm automatically generated test inputs of the `iTXt` type.

After finishing Algorithm 4.1 for all the ten changes, we got ten change-reaching inputs (one for each change). Each of these ten inputs can only guarantee the corresponding change being executed, but they may not necessarily affect the program output. In fact, we found that out of the ten change-reaching inputs, seven affect the program output (that is, the output is different for these inputs in the two program versions), and the remaining three do not. Among these seven changes, six changes are bug fixes, and the other is a content change in the output message. We show one example of a bug fix in Figure 4.6. In the example, the bit-wise right shift operator $\gg$ was mistyped as greater than operator $>$ in the buggy version (original version). With the bug fix, the variable red_high had different values in two versions. This variable was later used to compute a PNG file as the program output. The output was already different because of this change, so change effect propagation was not needed in this case.

Three out of the ten change reaching test inputs needed to go through change effect propagation (Algorithm 4.2 in our approach). By employing Algorithm 4.2, we succeeded in altering two of these three change-reaching inputs, to produce test inputs which execute the change, *and* propagate its effect to the program output. In other words, we constructed a PNG file which executes the change and manifests its effect by producing different outputs in the two versions of `libPNG`.

For the last program change, we did not succeed in generating a change-stressing input. The change-effect propagation stopped at a conditional jump in this case. The formula constructed was not satisfiable because of branch correlation. The heuristics proposed by us to handle branch correlation (see Section 4.3.3) did not allow us to construct a satisfiable formula in this case. This is because, our heuristics handle "immediate" (branch, branch) and (assignment, branch) conflicts. It does not handle "transitive conflicts" where the branch correlation cannot be explained by a *pair* of assignments/branches. For example consider the code fragment `x = 1; y = x; if (y > 2)` Here the direction of evaluation of `y > 2` is fixed by the past two assignments, but there does not exist any pair of statements which can explain the infeasibility of the sequence of statements being executed for any input.

In summary, there are ten changes in our experiment with `libPNG`, our technique succeeded for nine of them. In the first step of our technique, we successfully get ten change-reaching inputs for all the changes. Eight of these ten inputs are from existing test-suite, the other two inputs are generated by our Algorithm 1. In the second step of our technique, our Algorithm 2 modifies the results from the first step to get inputs that have different output because of the changes. The second step succeeded on nine changes and failed on only one change.

4.5 Threats to Validity

One main threat to validity is the the choice of subject programs in our evaluation. We chose two subject programs with different sizes and used different setting for the initial test-suite. More experiments would help better evaluate our technique.

In our experiments with `libPNG`, we had to modify the original test driver to make all changes reachable. While modifying the test driver, we try to keep the changes minimal.

According to the PIE model, program state infection has to happen after a program change is executed. Our technique does not focus on state infection as we observed that program state infection typically happens immediately after the change is executed. However, in general, there might be cases where state infection does not always happen together with the execution of program change.

Our technique compares the program state based on the aligned execution traces. Currently, the trace alignment is based on the minimum edit distance. Other trace alignment techniques that exploit program structures could potentially give more precise results.

4.6 Summary

In this chapter, we present a test-suite augmentation method which stresses program changes. To stress a change c in a program, our technique automatically generates a new test case t that gives different outputs in two versions. Our technique works mainly in two steps. In the first step, we use distance in Control Dependency Graph to guide our path exploration towards the change. After a change-reaching input is constructed, our technique use *change effect propagation tree* to identify why a change cannot affect output, and then propagate its effect accordingly.

In the case of correct refactorings, we expect the technique to not generate any new inputs. Since such a change does not result in different program state, our test-suite augmentation method indeed does not generate any change-stressing inputs in such a case.

We have implemented our technique in a toolset based on BitBlaze [99]. To test the efficacy of our technique, we performed two case studies on `tcas` and `libPNG`. For almost all the changes we studied, our tool was able to generate a new test case that stresses the change and causes difference in program outputs. Compared with existing test-suite augmentation techniques, our technique is more goal-directed since we: (i) employ metrics like distance in the control dependency graph to reach the change quickly (this enables us to find a short program path to the change), and (ii) employ heuristics to handle correlated branches while propagating the effect of program change to output (this enables us to avoid searching through many infeasible program paths).

CHAPTER V

DARWIN: An Approach for Debugging Evolving Programs

As programs evolve from old versions to new versions, bugs are also constantly introduced in programs. Existing functionality could be broken by program changes, which is referred as software regression. The implementation of new program features may also contain defects. As mentioned earlier, many bug-fixes have been shown to be incorrect in real-life software projects.

Locating regression bugs has always been a notoriously difficult task. When a program is changed to a new version, usually most functionality of the program is to be preserved in the new version. If any of these functionality is accidentally broken, a regression bug occurs. Note that we do not require the full program requirements to stay unchanged, we only require the requirements for bug-manifesting inputs to be unchanged. The debugging process will only focus on inputs that can manifest the bug. A bug is considered to be regression bug whenever the specification for the bug-manifesting inputs is unchanged.

Problem Statement Specifically, we seek to solve the following debugging problem in this chapter. Let program P' be a buggy program whose buggy behavior can be manifested by input t. Let program P be the reference program on which the behavior of input t is correct, and P and P' are supposed to produce the same output for input t. Our goal is to find a set of program fragments in either P' or P that are responsible for the failing execution of input t in program P'.

We assume that the program requirements vis-a-vis existing features of the program do not change. However, in reality, program requirement do change over time. In that case, our technique is unable to differentiate software regression errors from intended program changes based on changed requirements. We will discusses this assumption more thoroughly in Section 5.5.

For regression bugs, the previous program version serves as an excellent reference when debugging the current buggy version, because the same program requirement is satisfied by the previous program version but not by the current buggy program version. An intuitive approach is to compare the execution trace of the bug-manifesting input from the previous program version and its execution trace from current program version. However, such syntax-based approach is unable to differentiate the semantics-preserving program changes from semantics-altering program changes. For example, even though switching the order of two independent statements may not result in any change in program semantics, a simple trace comparison approach will still report the trace differences caused by switching the statements as potential causes of the regression bug.

We propose a novel semantics-based regression debugging technique — DARWIN. Our proposed DARWIN approach employs semantic comparison as opposed to simple syntactic comparison. Our semantic comparison of two executions is based on the path conditions extracted from the execution traces. We notice that the path condition of an execution trace contains information of how the program output is computed. On the other hand, different from execution traces, path conditions are quantifier-free first order logic formulae, comparison of which assists us to automatically filter out logically equivalent computations even though they are syntactically different. As we will explain later in this chapter, DARWIN may still be applicable even when the reference program P and the buggy program P' are two entirely different implementation of the same specification.

5.1 Overview

In this section, we illustrate our DARWIN debugging technique using an example shown in Figure 5.1. The reference program is shown in Figure 5.1a, and the buggy program is shown in Figure 5.1b. Both programs have the same input variables $\{x, y\}$ and the same output variable out. Let us assume that the program requirement is not changed from the program in Figure 5.1a to the program in Figure 5.1b. Thus, for the same input, the current program in Figure 5.1b is supposed to produce the same output as the output of the program in Figure 5.1b. However, we observe that the output for input $\langle x == 5, y == 5\rangle$ in the current program is unexpected. The bug lies in line 6 of the program in Figure 5.1b, where the correct statement should be `if(x-y >=0)`. Assuming that the root-cause of the bug is unknown, our goal is to find the statements that are most likely to be the root-cause.

```
1  int x, y, out;
2  scanf("%d%d", &x, &y);
3  out = 0;
4  if(x+y == 10){
5    if(y < 8){
6      if(x > 4){
7        out = 1;
8      }else{
9        out = -1;
10     }
11   }
12 }
13 printf("%d", out);
```

(a) Previous program

```
1  int x, y, out;
2  scanf("%d%d", &x, &y);
3  out = 0;
4  if(y < 8){
5    if(x+y == 10){
6      if(x - y >= 1){
7        out = 1;
8      }else{
9        out = -1;
10     }
11   }
12 }
13 printf("%d", out);
```

(b) Current program

Figure 5.1: Sample programs (the program in Figure 5.1b contains a regression error in line 6, which should be `if(x-y >=0)`)

```
scanf("%d%d", &x, &y);
out = 0;
if(x+y == 10){
if(y < 8){
if(x > 4){
out = 1;
printf("%d", out);
```

(a)

```
scanf("%d%d", &x, &y);
out = 0;
if(y < 8 ){
if(x +y == 10){
if(x – y >=1 ){
out = -1;
printf("%d", out);
```

(b)

Figure 5.2: Execution traces of input $\langle x == 5, y == 5\rangle$ for the programs in Figure 5.1

Let us first examine the effectiveness of trace comparison on the above programs. We first collect the execution traces of input $\langle x == 5, y == 5\rangle$ on the two programs. The execution traces are shown in Figure 5.2. All the differences in the execution traces are highlighted in grey background to the user for further inspection. Even though only one statement is buggy, all executed statements in lines 4-9 are highlighted, which unnecessarily increases the programmer's manual inspection effort. Through this example, we notice two problems of trace comparison: i) we need to align the two execution traces from two different programs based on heuristics (e.g. shortest edit distance) during trace comparison. Any inaccuracy in the trace alignment will introduce false positives in the debugging result, and ii) trace comparison is unable to infer the semantic equivalence of two statements when they are syntactically different even though they are indeed equivalent in semantics. Presenting semantic equivalent statements across versions as potential bug causes only increases the manual effort of users in examining the automatic debugging result.

Different from the simple trace comparison approach, our DARWIN approach uses semantic comparison of the two executions based on path conditions. Let us first compute the path conditions of the input on the two program versions. We denote the path conditions from the previous version and current version as pc and pc' respectively. Following the steps in Chapter II, we get $pc = (x + y == 10) \wedge (y < 8) \wedge (x > 4)$ and $pc' = (y < 8) \wedge (x + y == 10) \wedge \neg(x - y >= 1)$. As the path condition is a conjunction of branch conditions, we use the following method to pinpoint the bug-causing branch condition, thereby locating the root cause of the bug. We construct the following formulae, each one of which represents a possible deviation (due to the negation of the last branch condition) from the path condition pc'.

- $pc \wedge \neg(y < 8)$
- $pc \wedge (y < 8) \wedge \neg(x + y == 10)$
- $pc \wedge (y < 8) \wedge (x + y == 10) \wedge (x - y >= 1)$

If any formula is satisfiable, the branch of the negated branch condition (the last branch condition) represents one semantic difference between the execution of input $\langle x == 5, y == 5\rangle$ in program P and the execution of the same input in program P'. The branch is considered as one suspicious root cause of the regression error. For the above formulae, we find that only the last formula $pc \wedge (y < 8) \wedge (x + y == 10) \wedge (x - y >= 1)$ is satisfiable. Hence, the branch `if(x - y >= 1)` in line 6 of the buggy program is considered as the potential root-cause. This concludes the process of DARWIN on the example and indeed it has found the root-cause of the bug.

To further explain the intuition behind our approach, let us consider the execution of a test input t' where $t' \models pc \wedge (y < 8) \wedge (x + y == 10) \wedge (x - y >= 1)$. For simplicity, let us use t to represent the initial buggy input $\langle x == 5, y == 5\rangle$. We then have $t' \models pc$ and $t' \models (y < 8) \wedge (x + y == 10) \wedge (x - y >= 1)$. Given that $t' \models pc$, the execution of input t' will follow the same path as that of our initial buggy input t in the previous program version. In the changed program version, the execution of input t' will follow the same path as that of our initial test input before the negated branch. Upon executing the negated branch, the executions of inputs t and t' start to differ. Without any change in semantics, the two inputs following the same path in previous program P should also follow the same program path in the changed program P'. Thus, the difference in execution paths of inputs t and t' in program P' signal changes in program semantics. Based on this intuition, our technique captures the semantic changes by reporting the branches where path deviations are possible as shown in the aforementioned example.

The technique we have discussed so far can only points to branch statements as the potential root causes of the bug. Thus, our technique is inherently suitable for bugs in program control flow. For data flow bugs that affect program control flow, our technique will point to locations where the program control flow is affected. In Section 5.2.3, we present a predicate instrumentation technique that allows our technique to directly locate the root causes of data flow bugs.

In summary, our DARWIN debugging approach proceeds as follows. Let us denote the reference program as P, the buggy program as P', and the buggy input as t.

1. We first symbolically execute input t on programs P and P' and compute the path conditions pc and pc' of the two executions respectively.

2. If $pc \wedge \neg pc'$ is satisfiable, we find the branch conditions in pc' that are not subsumed by pc (this process will be explained in details in Section 5.2). These branch conditions represent computation in program P' that does not occur in the execution of program P. The corresponding branches of all these branch conditions are considered as the potential bug causes.

3. If $pc \wedge \neg pc'$ is not satisfiable but $pc' \wedge \neg pc$ is satisfiable, we find the branch conditions in pc that are not subsumed by pc'. These branch conditions represent computation in program P that does not occur in the execution of program P'. The corresponding branches of all these branch conditions are considered as the potential bug causes. These program fragments in the trace of program P with no semantic equivalent fragments in the trace of P' help us find code-missing errors in the program P'.

4. If $pc \wedge \neg pc'$ and $pc' \wedge \neg pc$ are both unsatisfiable, our techniques fails. In this case, the two

path conditions are equivalent, $pc == pc'$. Hence, our technique based on comparing two path conditions is unable to locate any potential bug cause.

5.2 Detailed Methodology

In this section, we present the detailed methodology of our DARWIN debugging technique. We start with the core approach of semantically comparing two executions through path conditions. Then, we present a set of formula simplification rules that drastically boost the efficiency of our technique. Finally, we propose to use predicate instrumentation to uncover bugs that are not reflected in path conditions by default.

Notations We use the following notations in the upcoming discussion. Let ψ be a branch condition, we use $BrI(\psi)$ to denote the branch instance from which ψ is computed. We then use $Br(\psi)$ to denote the static branch in the source code of the branch instance $BrI(\psi)$.

5.2.1 Core method

Given the failing input t in the current buggy program P', we use semantic comparison of path conditions to locate the root-cause of the bug. Let π and π' be the execution traces of input t in programs P and P' respectively. We then compute the path condition of π and π'. Let us denote the path conditions as pc and pc'. Instead of comparing the traces directly, we compare the path conditions pc and pc'. The path conditions are both quantifier-free first order logic formulae on the program input variables. On a coarse-grained level, we can compare whether two path conditions are logically equivalent. However, our goal is to locate the bugs through comparing path conditions. Hence, a more fine-grained comparison is required.

Each path condition is a conjunction of a sequence of branch conditions. Let us denote the path condition pc' as $pc' = \psi_1 \wedge \psi_2 \ldots \psi_m$. Intuitively, we try to find any branch condition in pc' that is not subsumed by pc. That is, we try to find each ψ_i such that $\neg(pc \Rightarrow \psi_i)$. However, we should also pay attention to the order of branch conditions in pc'. A branch condition only has its meaning when the corresponding branch instance can be executed. Before executing the branch corresponding ψ_i, all branches before the branch of ψ_i are executed. Therefore, to assure that the branch of ψ_i is executed, we need to require $\psi_1 \wedge \psi_2 \wedge \ldots \wedge_{i-1}$ to be true. Together with the aforementioned condition, we consider a branch condition in pc' one difference between pc' and pc iff. $(\neg(pc \Rightarrow \psi_i)) \wedge (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1})$ is true. The above formula can be reduced to the following

$$\theta_i \overset{def}{=} pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$$

If θ_i is satisfiable, the branch $Br(\psi_i)$ is considered as a difference between the two executions and thus a candidate of the bug root-cause. In the end, the following set of statements will be presented as potential bug causes.

$$\{Br(\psi_i) | pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i \text{ is satisfiable}\}$$

Since there are m branch conditions in pc', there could be at most m entries in the above set. We have described the process of finding suspicious buggy locations in program P'. If we cannot find the root-cause of the bug in the current program P' through the above process, we use a similar process to find the semantic differences in the path condition pc. Let us denote the path condition pc as $pc = \varphi_1 \wedge \varphi_2 \ldots \wedge \varphi_n$. We then solve the following n formulae

$$\omega_i \overset{def}{=} pc' \wedge \varphi_1 \wedge \ldots \varphi_{i-1} \wedge \neg\varphi_i$$

For any ω_i that is satisfiable, we put the branch $Br(\varphi_i)$, which is in program P, into the set of suspicious program locations. Therefore, DARWIN will produce the following set of statements in program P to help explain the bug.

$$\{Br(\varphi_i) | pc' \wedge \varphi_1 \wedge \ldots \varphi_{i-1} \wedge \neg\varphi_i \text{ is satisfiable}\}$$

In the case that pc and pc' are equivalent, DARWIN fails to locate any potential bug cause.

Input validation Due to the complexity of computing path conditions for real-life programs and the limited ability of SMT solvers, we need to make under-approximation when computing path conditions and solving them using SMT solvers. For example, suppose we have an SMT solver that only support linear operations. The branch condition of `if(x * y)>0` will not be supported by the solver if `x` and `y` are both symbolic. Let pc be the ideal path condition for path π and let $\tilde{pc}$ be the computed path condition of path π. When $\tilde{pc}$ is an under-approximation of pc, we have the following relation: for any input t', if $t' \models \tilde{pc}$, then $t' \models pc$. Recall that we solve m formulae that are in the form of $\theta_i \overset{def}{=} pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$. Due to the negation on ψ_i, the computed θ_i is neither under-approximation or over-approximation of the precise θ_i. Therefore, to eliminate any error introduced by the approximation, we verify the solution of θ_i through concrete program execution. Suppose input t' satisfies θ_i. According to the definition of θ_i, $t \models pc$, which means that when executing input t on program P, the program should follow path π. Similarly, since $t' \models \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$, the execution of input t' in program P' should evaluate all branch instances in $\{BrI(\psi_k) | 1 \leq k \leq i-1\}$ to the same directions as in path π' and evaluate the branch instance $BrI(\psi_i)$ to different direction. Suppose we get an alternate input t' when solving $\tilde{\theta}_i$. We then validate that t' is indeed an solution of θ_i by concretely executing t' in programs P and P' and check whether the programs follow the desired paths. That is, (i) the execution of t' in program P follows the same path as the path of t, and (ii) the execution of t' evaluates all branch instances $BrI(\psi_k)$, $1 \leq k \leq i-1$ to the same direction as in the trace of t in P' and evaluates $BrI(\psi_i)$ to different direction.

Prioritizing suspicious causes We propose the following technique to prioritize the suspicious root-causes, thereby further improving the effectiveness of our technique. When an formula θ_i is satisfiable, we get an alternate input by solving it. Therefore, each statement in the set of potential root causes has an associated alternate program input. Our prioritization technique is based on the passing/failing status of the alternate input. We prioritize the branches whose associated inputs are passing. That is, if by solving θ_i we get input t', and the program P' passes when executing input t', then the branch $Br(\psi_i)$ is assigned with high priority. If an input t' satisfy $\theta_i \overset{def}{=} pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$, the programs P and P' should follow certain paths when executing t'. Specifically, the program P' should follow the same path as the path of input t before the branch $BrI(\psi_i)$ is met. On the branch $BrI(\psi_i)$, the inputs t and t' drive the branch to different directions in P'. The intuition behind this heuristic is that by negating the branch $Br(\psi_i)$, we have converted a failing execution (with input t) into a passing execution (with input t'), then the branch $Br(\psi_i)$

is more likely to be the root-cause of the bug. Similar ideas have also been presented in previous research (e.g. predicate switching [126]). Note that to apply this heuristic, an oracle (either automated or manual) checking whether an execution passes has to exist. As observed in our experiments, even without an automated oracle, the number of executions to check is usually small and manageable for programmers.

5.2.2 Formula simplification

Our DARWIN technique involves extensive satisfiability checking of formulae using Satisfiability Modulo Theory (SMT) solvers. Thus, the scalability of our method depends on the scalability of formula solving. We propose several techniques to improve the efficiency of formula solving specific to our problem domain.

Checking for unsatisfiable sub-formula. Recall that we are trying to solve formulae of the form $pc \wedge \neg pc'$ where pc and pc' are the path conditions collected from two program versions for a given test input t. Assuming $pc' \overset{def}{=} (\psi_1 \wedge \psi_2 \ldots \wedge \psi_m)$ we solve the m formulae $\theta_i \overset{def}{=} pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$. The key problem we face now is that the SMT solver may take substantial time to solve each of the θ_i formula. We note that common programming practices may make $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ unsatisfiable. For example, consider a check c being repeated many times in a program code. Clearly if ψ_j (for some $j \leq i$) and ψ_i are both c, an SMT solver will very quickly conclude that $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is unsatisfiable. In such situation, we do not need to solve the larger formula $pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$. Overall, instead of directly dispatching θ_i to the SMT solver (to check the satisfiability of θ_i) - we first dispatch $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ to the SMT solver and try to see whether the SMT solver declares it to be unsatisfiable within a short time bound. Our experience indicates that this is often the case, and in such a situation we do not need to solve the bigger formula $\theta_i \equiv pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$.

Slicing out unrelated symbolic variables. Secondly, using dynamic slicing, we can find the subset of symbolic input bytes that can affect the only branch (contributing to ψ_i) that we want to execute differently in both program versions. For unrelated symbolic input bytes, we use their value from the concrete execution when constructing the alternate inputs. Using concrete values for certain portions of our input greatly simplifies the formulae we need to solve and reduces the burden on the SMT solver.

We now describe the steps we employ to reduce the amount of time taken in checking satisfiability of θ_i.

1. We impose a short time bound (say 10 seconds), and within this time bound we let the solver check whether $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is satisfiable. If the solver says that $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is unsatisfiable, clearly θ_i is not satisfiable. If the solver does not terminate within the time bound or says that $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is satisfiable — we continue with the following steps.

2. We perform slicing on the (assembly level) execution trace π' corresponding to path condition pc' to find out the set of input bytes that ψ_i is dependent on. This is done as follows. Note that ψ_i is a primitive constraint corresponding to some branch instance b in the execution trace. Due to traceability links between sub-formula in the path condition and branches contributing to these formulae we can find the branch b contributing to ψ_i. Let l be the control location corresponding to b and $Vars$ be the variables appearing in the constraint ψ_i. We perform dynamic slicing [67, 17, 112] w.r.t. the slicing criterion $(l, Vars)$ on the assembly level execution trace π' corresponding to path condition pc'. During the traversal of the execution trace, the dynamic slicing algorithm maintains (i) a set of instruction instances (the slice), (ii) a set of variables δ whose values need to be explained. At the end of the slicing, we inspect the set of input fields (or bytes) which appear in δ. These are the input bytes on which ψ_i depends in the trace for pc'. Let this set of input bytes be In_{i+1}.

3. We assign all input bytes not appearing in In_{i+1} to the actual values used in the concrete execution of the test input t being debugged. We also use forward constant propagation along the execution trace π' to propagate these concrete values to other program variables (which do not correspond to program input). This greatly simplifies pc as well as $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ since many of the variables in the formulae get instantiated to concrete values. Let the simplified formulae be called $pc_{simplified}$ and $(\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i)_{simplified}$.

4. We check the satisfiability of $(\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i)_{simplified}$. If it is unsatisfiable, we can stop. Otherwise, we go to the next (and final) step.

5. Finally we solve the simplified formula $pc_{simplified} \wedge (\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i)_{simplified}$ using an SMT solver.

After concretizing the input bytes other than those in In_{i+1} and propagating constants, the formulae to be solved are greatly simplified owing to instantiation. This greatly reduces the solution time.

5.2.3 Predicate instrumentation

The core method described in Section 5.2.1 could only points us to branches in the programs as the bug candidates. Hence, errors that only affect program data flow but not control flow could not be detected using our core debugging method. We propose to use predicate instrumentation [71, 72] to introduce additional control flow that reflects information of the program data flow. We instrument the following two types of predicates in the programs being debugged.

1. For each function return value, we instrument predicates checking whether the return value is null if it is of pointer type and checking the sign of the return value if the type is integer.

2. For any variable x, before any assignment to variable x, we instrument predicates checking whether other $x == y$ for any variable y that i) shares the same type with variable x and ii) is live at the instrumentation location.

With the instrumented predicates, our DARWIN technique could find bugs in the program data flow that do not affect the program control flow originally. On the other hand, the benefit does come with the instrumentation cost, which is less than 20% in term of the increased trace size (see Section 5.4.7).

5.3 Implementation

We now describe our implementation setup. The overall architecture of DARWIN is summarized in Figure 5.3. We built DARWIN based on the BitBlaze platform [99]. Most of the modules used by DARWIN are contained in the recent open-source release of BitBlaze. However, BitBlaze does not have the modules for formula manipulation and optimization. We built these modules for DARWIN on our own.

5.3.1 Generating alternate inputs

DARWIN uses a symbolic execution engine for computing the path condition of a given program execution. Our execution engine is a part of the BitBlaze platform [99], which works on x86 binaries. Given an input, the platform concretely executes the program on the specific input and records the trace. It then performs symbolic execution to compute the path condition of the concrete trace recorded. The path condition represents a constraint denoting the set of inputs which execute the concrete trace.

The concrete execution is carried out by TEMU, a whole-system emulator based on QEMU [4]. TEMU can run Windows and Linux as its guest operating system, enabling us to analyze both Windows and Linux binaries. After the concrete execution, TEMU generates a trace of instructions executed by the program. The trace is also annotated with input dependence information, for example, whether the operand of an instruction is dependent on input (an operand is dependent on the input if there is a data dependence chain from the operand to an input). TEMU allows users to specify several types of inputs, such as network inputs, files, and keyboard inputs.

The path condition calculation is performed by the VINE component of BitBlaze. It first defines the bytes in the program input as symbolic variables: each byte in the input is a distinct variable. Then, it makes a forward pass through the trace recorded by TEMU, considering only *tainted* instructions i.e. instructions whose operands are (directly or transitively) dependent on the program input (via data dependencies). Note that such dependency information is present as annotations in the trace recorded by TEMU. For each tainted instruction in the trace, VINE translates the instruction to a sequence of statements in its own intermediate language, where the semantics of each instruction is preserved [27]. This translation helps the VINE tool deal with the complexity of the x86 instruction set. Finally, VINE performs a traversal of the trace in the intermediate language to compute the path condition.

Two points need to be noted about the BitBlaze execution engine, and its interplay with our debugging framework. First, the concrete and symbolic execution engines work on x86 binaries. Our path conditions are also computed at the level of binaries, rather than source code — thereby capturing the precise semantics of the program execution. On the other hand, the set of potential bug causes is computed at the level of source code for ease of understanding (by the programmer).

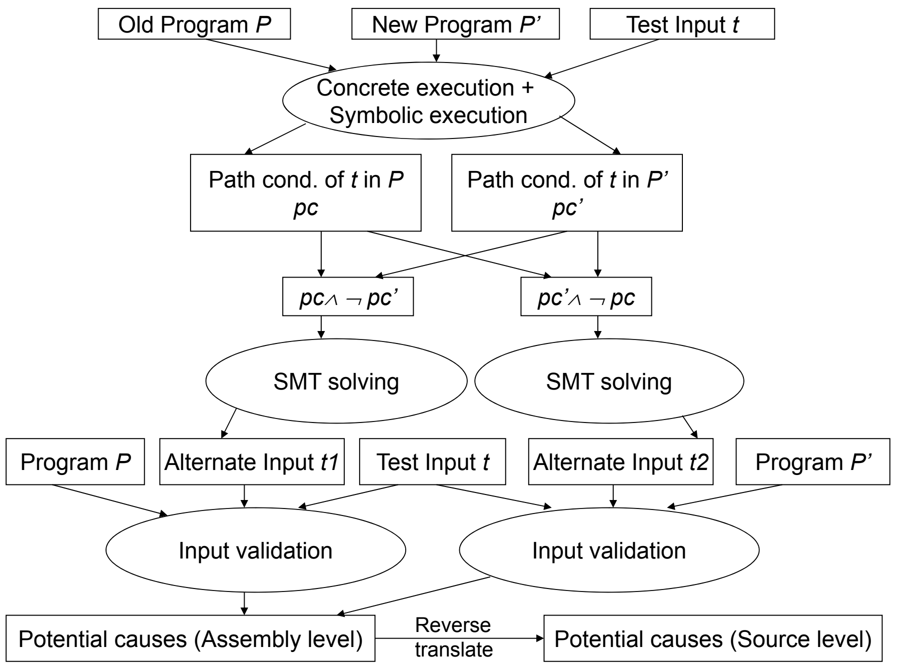


Figure 5.3: Architecture of our DARWIN toolkit. It takes an old program P, a new program P' and a test input t which passes in P but fails in P'. The output is a report explaining the behavior of test t. The entire flow is automated.

Secondly, the variables appearing in the path condition correspond to the different bytes of the program input.

Given program versions P, P' and a test input t which passes in P and fails in P' — we compute the path conditions pc, pc' of input t in programs P, P'. In fact, the symbolic execution engine in BitBlaze constructs these path conditions as formulae in the well-known SMT-LIB[87] format. The SMT-LIB format is supported by all the solvers that participated in the SMT annual competition. Thus, expressing the path conditions in the SMT-LIB format allows us to leverage a lot of state-of-the-art SMT solvers. It also allows us to benefit from the ongoing improvement in the solving ability of the existing solvers — we can use whichever solver is currently the fastest. The solver we are currently using is Boolector [28], the winner of the SMT competition in 2009 for quantifier free formulae with bitvectors, arrays and uninterpreted functions (the QF_AUFBV category). Indeed this is suitable for us, since our formulae do not have universal quantification and any variable is implicitly existentially quantified.

5.3.2 Reporting root causes

Given the solutions of $pc \wedge \neg pc'$ we first validate them. In case we find $pc \wedge \neg pc'$ to be unsatisfiable or none of the solutions of $pc \wedge \neg pc'$ can be validated, we solve $pc' \wedge \neg pc$ in a similar fashion. By following the steps mentioned in the previous section (solving either $pc \wedge \neg pc'$ or $pc' \wedge \neg pc$), we obtain a set of branches at the assembly level as potential root causes of the bug. Using standard compiler level debug information, these can be reverse translated back to lines in source code.

Accuracy of our reports We now discuss some low-level issues which make a *substantial* difference to the accuracy of our results. Given the path conditions pc and pc', let $pc' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$ where ψ_i are primitive constraints. As mentioned in the previous section, we solve the m formulae $\{\theta_i \mid 0 \leq i < m\}$ where $\theta_i \overset{def}{=} pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$. The VINE symbolic execution engine ensures that the path conditions contain only constraints from branches which are dependent on the program input. In practice, this greatly cuts down on the number of ψ_i constraints, and hence the number of θ_i formulae that need to be dispatched to the SMT solver. Since each θ_i formula contributes at most one statement in our report, we get a smaller sized report by reducing the number of θ_i. As mentioned in Section 5.2.1, if the number of root causes is still high (due to large number of alternate inputs), we *prioritize* statements obtained from successful alternate inputs over other statements since these are more likely to reveal the real root cause.

5.4 Debugging Experience

We report our experience in using DARWIN to locate error causes in real-life case studies.

5.4.1 Experience with `libPNG`

We first describe our experience in debugging the `libPNG` open source library [3], a library for reading and writing PNG images. We used a previous version of the library (1.0.7) as the buggy version. This version contains a known security vulnerability, which was subsequently identified and fixed in later releases. A PNG image that exploits this vulnerability is also available online. As the reference implementation or stable version, we used the version in which the vulnerability was fixed (1.2.21). Assuming this vulnerability was a regression bug, we used our tool to see if the vulnerability could be accurately localized.

The bug we localized is a remotely exploitable stack-based buffer overrun error in `libPNG`. Under certain situations, the `libPNG` code misses a length check on PNG data prior to filling a buffer on the stack using the PNG data. Since the length check is missing, a buffer overrun may occur. What is worse, such a bug may be remotely exploited by emailing a bad PNG file to another user who uses a graphical e-mail client for decoding PNGs with a vulnerable `libPNG`. In Figure 5.4, we show a code fragment of `libPNG` showing the error in question. If the first condition `!(png_ptr->mode & PNG_HAVE_PLTE)` is true, the length check is missed, leading to a buffer overrun error. A fix to the error is to convert the `else if` in Figure 5.4 to an `if`. In other words, whenever the length check succeeds, the control should return.

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

Figure 5.4: Buggy code fragment from `libPNG`

We now explain some of the issues we face in localizing such a bug using approaches other than ours. Suppose we have the buggy `libPNG` program and a bad PNG image which causes a crash due to the above error. If we want to perform program differencing methods (such as source code "diff") to localize the bug, there are 1589 differences in 28 files. Manually inspecting these differences requires a lot of effort. Existing *Semantic diff*[58, 88, 56, 20] techniques based on program control and data structures could only provide limited help to the manual inspection. Because of the very large number of source code differences, the number of semantic differences would still be large. Moreover, given a coarse-grained semantic difference such as *method change*[88], one still needs to inspect more details to tell whether this change indeed causes the bug.

If we want to localize the error by an analysis of the erroneous execution trace starting from the observable error — it is very hard to even define the observable error. Even if the buffer being overrun is somehow defined as the observable error, tracking program dependencies from the observable error can be problematic for the following reason. The libPNG library is used by a client which inputs an image, performs computation and outputs to a buffer (the one that is overrun due to error inside libPNG). In this case, we are debugging the sum total of the client along with the libPNG library. Since almost all statements in the client program and many statements in libPNG involve manipulation of the buffer being overrun itself — a dynamic slicing approach will highlight almost the entire client program as well as large parts of the libPNG library.

If we want to employ statistical bug isolation methods (which instrument predicates and correlate failed executions with predicate outcomes), the key is to instrument the "right predicate". In this case, the predicates in question (such as !(png_ptr->mode & PNG_HAVE_PLTE)) contain pointers and fields. Hence they would be hard to guess using current statistical debugging methods which usually consider predicates involving return values and scalar variables.

If we want to perform debugging by comparing the failing trace with a passing trace from the same program P', we two two choices. First, we can compare the trace of the bad PNG image (which exposes the error) with the trace of a good PNG image (which does not show the error). The question then is how do we get the good PNG image? Even if we have a pool of good PNG images from which we choose one – making the "right" choice becomes critical to the accuracy of root cause analysis. Second, we can compare the trace of the bad PNG image from program P and the the trace of the bad PNG image from program P'. In this case, any executed program changes would show up in the differences of the traces.

Different from the existing techniques, DARWIN uses semantic analysis to effectively locate the bug root-cause. Specifically, DARWIN first computes the path conditions of the bad PNG image on the two libPNG versions 1.0.7 and 1.2.21. Let these be pc_{buggy} and pc_{fixed} respectively. We find that $pc_{fixed} \wedge \neg pc_{buggy}$ is unsatisfiable, so we solve for $pc_{buggy} \wedge \neg pc_{fixed}$. By solving this formula we get 9 branches. All these 9 branches passed the input validation process, hence we report 9 statements as potential root causes.

We prioritize these 9 statements as follows. Among the 9 alternate inputs of these 9 statements, we find out which of them are successful i.e., the program output for a successful input should be the same in both the program versions. Only one of our 9 alternate inputs is found to be successful. The branch instruction contributed (to the result) by this input corresponds to the branch length >

(png_uint_32) png_ptr->num_palette thereby pointing directly to the cause of failure. This branch is (mistakenly) not executed in the buggy libPNG version 1.0.7

Discovering New Errors Interestingly, in the process of this debugging we found other potential problems in libPNG. As mentioned earlier, DARWIN obtained 9 alternate inputs, only one of which exhibits bug-free behavior, and pointed us to the error. Interestingly, the other branch instructions point us to other deviations between the two versions of libPNG. For example, by following one of these 8 instructions we find that the two versions of libPNG use different functions to retrieve the length field of a chunk from the input. In version 1.0.7, we have

```
length = png_get_uint_32(chunk_length);
```

while in version 1.2.21 we have

```
length = png_get_uint_31(chunk_length);
```

In particular, the code for png_get_uint_31 is as follows.

```
png_get_uint_31(png_structp png_ptr, png_bytep buf)
{
   png_uint_32 i = png_get_uint_32(buf);
   if (i > PNG_UINT_31_MAX)
     png_error(png_ptr, "PNG unsigned integer out of range.");
   return (i);
}
```

Thus, png_get_uint_31 first uses png_get_uint_32 and then performs a length check. If png_get_uint_32 is directly used to find the length of a chunk, a length check w.r.t. the constant PNG_UINT_31_MAX is missing. We also report the branch instruction containing this missing length check, thereby pointing to another potential error in libPNG.

5.4.2 Experience with miniweb-apache

In our second case study, we study the web-server miniweb [57], an optimized HTTP server implementation which focuses on low resource consumption. The input query whose behavior we debugged was a simple HTTP GET request for a file, the specific query being "GET x". Ideally, we would expect miniweb to report an error as x is not a valid request URI (a valid request URI should start with '/'). However, `miniweb` does not report any errors, and returns the file `index.html`. We then attempt to localize the root cause of this observable error.

We found that even the latest version of `miniweb` contains the error. Therefore, we cannot choose another version of `miniweb` as the reference implementation. We chose another HTTP server `apache` [1] as the reference implementation. `Apache` is a well-known open-source secure HTTP server for Unix and Windows. Since both `apache` and `miniweb` implement the HTTP protocol, they should behave similarly for any input accepted by both implementations. Further, `apache` does not exhibit the bug we are trying to fix. It reports an error on encountering the input query "`GET x`".

We generate the path conditions of "`GET x`" in both `apache` and `miniweb`. Let these be f_{apache} and $f_{miniweb}$ respectively. We find $f_{apache} \wedge \neg f_{miniweb}$ to be unsatisfiable. However, by solving $f_{miniweb} \wedge \neg f_{apache}$ we can get alternate input queries. By following our methodology described in Section 5.2, we get exactly 5 potential root causes whose corresponding alternate inputs are:

`GET /`, `GET \`, `GET *`, `GET .` and `GET %`

Based these 5 branches, we were able to localize the bug immediately. The `miniweb` program does not check for '`/`' in `GET` queries and treats the query "`GET x`" similar to "`GET /`" thereby returning the file `index.html`.

Discovering New Errors Only one of our five alternate inputs was successful, exhibiting the same output in both program versions. The branch instruction corresponding to this input pointed us to the missing check for '`/`'. The other statements pointed us to other missing checks in `miniweb`. Indeed, we can locate that `apache` contains checks for each of these 5 characters while `miniweb` misses the check for all 5 of them, leading to potential errors.

In a Broader Perspective Our experiments with `apache-miniweb` also give us a broader perspective on the applicability of our method. Even if all versions of a program exhibit a given error (as was the case with `miniweb`), we can still use DARWIN to localize the error. We only need a reference program which is intended to behave similarly to the program being debugged, and does not exhibit the bug being localized. In our experiments, the `apache` web-server was the reference program.

5.4.3 Experience with `savant-apache`

`Savant` [5] is a full-featured open-source web-server for Windows. We notice that `savant` does not report any errors when faced with an input query of the form "`GOT /index.html`", a typo from the valid HTTP GET request "`GET /index.html`". We cannot choose another version of `savant` as the reference program because the latest version of `savant` also exhibits this error. As reference program, we choose the `apache` webserver, which reports an error for the query "`GOT /index.html`". Both `savant` and `apache` implement the HTTP protocol, and are expected to behave similarly.

In this case study, DARWIN found 46 branches. Out of these, the associated alternate input of only one branch is successful, that is, the input produces the same output in both `savant` and `apache`. This is the input "`GET /index.html`". The corresponding branch pinpointed the error to missing checks in `savant`. The `savant` program does not check for all the three letters 'G', 'E', 'T' in HTTP GET requests for HTTP protocol version HTTP/0.9 (which is the default assumed since we do not explicitly specify a HTTP protocol version in the query "`GOT /index.html`"). Indeed, we found that `savant` reports an error if we provide "`GOT /index.html HTTP/1.0`" as input. In HTTP/0.9 there is only one command, namely GET. The error lies in the fact that `savant` does not check for the string "`GET`", and assumes any given string to be the GET command.

Discussion Our experiments with `savant` also illustrate another additional feature of DARWIN — the ability to rectify program inputs. The process of alternate input generation in DARWIN can help correct errors in an almost correct program input such as the input "`GOT /index.html`". In this case, the input fix was easy and could have been done manually as well. In the future, we plan to conduct experiments with programs like web browsers to see if an almost correct HTML file (where the incorrectness in the file is hard-to-see) can get rectified through alternate input generation.

Table 5.1: Properties of the subject programs

Programs	LOC	Trace size (# instructions)	# Branches in trace	# Tainted instructions
libPNG v1.0.7	31,164	87,336	13,635	2999
libPNG v1.2.21	36,776	108,769	15,472	2592
Miniweb	2,838	270,856	26,201	331
Savant	8,730	121,714	16,212	1613
Apache	358,379	60,380 (miniweb)	5,388 (miniweb)	264 (miniweb)
		74,002 (savant)	9,672 (savant)	6889 (savant)
TCPflow (unpatched)	895	56838	7210	7753
TCPflow (patched)	934	58079	7375	7860

5.4.4 Experience with `TCPflow`

We use two versions of the `TCPflow` program, namely `TCPflow_0.21.ds1-2` and the same version with the patch `10_extra-opts.diff`, which is supposed to provide the user with some extra options. TCP is the most popular transport layer protocol and `TCPflow` is a program which captures and displays data sent through TCP connections. The statistics about the `TCPflow` program are given in Table 5.1.

The `TCPflow` program is designed to help users analyze TCP packets over the network. If we capture the raw TCP packets transmitted over the network — there is a TCP header inside each TCP packet. Inside each raw packet, we also have the header for the network layer protocol (usually the IP protocol). Thus, it is non-trivial to manually distinguish which parts in a raw packet correspond to the real data being transmitted. Moreover, there can be multiple active TCP connections at the same time. As a result, it is hard to determine which packets are from the same connection. `TCPflow` is a program which solves these problems. It analyzes the raw data (TCP packets) from TCP connections and outputs the actual data being transmitted over the network. A TCP connection is associated with source IP address, destination IP address, source port and destination port. The output from `TCPflow` is also classified by the connections.

`TCPflow` can read input both from network and file. If the input is from network, then it captures the data that is being transmitted and analyzes the data. In our experiment, the input is from a file which is generated by `tcpdump`.

```
47 45 54 20 2F 69 6E  64 65 78 24 68 74 6D 20 0D  |GET /index.htm .|
0A 0D 0A                                          |...             |
```

Output from the unpatched version of TCPflow

```
00 47 45 54 20 2F 69 6E  64 65 78 24 68 74 6D 20  |.GET /index.htm |
0D 0A 0D 0A                                       |....            |
```

Output from the patched version of TCPflow

Figure 5.5: Output from the `TCPflow` program

The bug we investigate is introduced by the patch `10_extra-opts.diff`. We provided two packets from the same connection to `TCPflow`: an SYN packet to setup the connection and a simple HTTP request packet. Figure 5.5 shows the output from both versions of the `TCPflow` program, where only the HTTP request payload is shown, and the headers from TCP layer and IP layer are excluded.

```
// unpatched version of the TCPflow
void handle_tcp (packet_t packet) {
   if( this packet has no data) {
        return;
   }
   if ((state = find_flow_state(current_flow)) == NULL)
        state = create_flow_state(flow, seq);
   offset = seq - state->ins;
   //write data from offset;
}
// patched version of the TCPflow
void handle_tcp (packet_t packet) {
   if( this packet has no data) {
        if ((state = find_flow_state(current_flow)) == NULL)
            state = create_flow_state(flow, seq);
        return;
   }
   offset = seq - state->ins;
   //write data from offset;
}
```

Figure 5.6: Schematic Code fragment from `TCPflow`

The two versions of `TCPflow` we use are `TCPflow_0.21.ds1-2` and the same version with the patch `10_extra-opts.diff`. Although the patch is supposed to provide some extra options to the user, it actually introduces a bug into the code. Figure 5.6 is a simplified code pattern from `TCPflow`. For each TCP connection, a struct named `flow_state_t` is used in the program to maintain some data associated with the connection. The program processes the packets one by one from the start to the end. So, for our program input, the SYN packet is processed before the data packet. The bug appears because the manner in which empty packets are handled is changed by the patch.

In the unpatched version of the program, if we see an empty packet and no other packets from the same connection have been seen before, the packet is simply ignored (the struct `flow_state_t` for the connection is not created at all). However in the patched version, empty packets are not ignored (the struct `flow_state_t` for the TCP connection is still created). Note that in TCP connections, each transmitted packet has a sequence number which is used by the sliding window protocol to make sure the packet is transmitted to the destination. In our case the sequence number of the data packet is just the sequence number of the SYN packet increased by one. Given a TCP connection, the corresponding struct `flow_state_t` has one critical member field named `ins` which is used to store the initial sequence number the program has seen for this connection. When a `flow_state_t` is created, `ins` is assigned with the sequence number of the *current* packet being handled.

Since the SYN packet has no data inside and the manner of handling such packets are different in the two program versions, the `flow_state_t` are created with different `ins` values in two program versions. In the un-patched version, because the SYN packet is ignored, the `flow_state_t` is only created when the data packet is seen, so the `ins` field is equal to the sequence number of the data packet. In the patched version, the `flow_state_t` for this connection is created when the SYN is seen, so the `ins` field is equal to the sequence number of the SYN packet. Note that the `ins` field is later used to calculate the offset in the output file when the data is written out. The offset is calculated via the statement

```
offset = seq - state->ins;
```

where `seq` is the sequence number of the packet being written. So, while writing the data packet in the unpatched version, the value of `seq` is equal to the value of `state->ins`; they are both set to the sequence number of the data packet. However, in the patched version, the `seq` is the sequence of the data packet, the `state->ins` is the sequence of the SYN packet. So the offset is 1 in the patched version, making the program write from the second byte in the output file. As a result there is an additional 0x00 (in the first byte of the buggy output) as shown in Figure 5.5.

Once again, we emphasize the bug we described above (and detected using DARWIN) is a *real-life bug* appearing in a patch of the `TCPflow` program. The bug happens because the authors of `TCPflow` forgot to modify the update of `state->ins` field after the manner of handling empty packets was changed. In fact, this bug is only observed when the input to `TCPflow` contains at least one empty packet. When we attempted to localize the root cause of this bug using DARWIN the root causes we reported were extremely accurate. Only 6 statements are reported as potential root causes from and one of them points to a branch condition which checks for empty packets.

Over and above the accuracy, making DARWIN work on the `TCPflow` program presented us with a substantial challenge in terms of scalability. Although the `TCPflow` program contains only 1000 lines of code, its path condition size was the largest among all our four case studies (see Table 5.1). Part of the reason for this comes from the frequent usage of libraries during the execution of `TCPflow`. The execution of the libraries bloats up the trace size and creates substantial time overheads for symbolic execution. Recall that we are trying to solve a formula of the form $pc_{unpatched} \wedge \neg pc_{patched}$ where $pc_{unpatched}, pc_{patched}$ are the path conditions of our chosen program input on the un-patched and patched versions of `TCPflow`. Assuming $pc_{patched} \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$, we actually solve m formulae $\{\theta_i \mid 0 < i \leq m\}$ where

$$\theta_i \overset{def}{=} pc_{unpatched} \wedge \psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$$

Without the optimizations mentioned in Section 5.2.2, solving each θ_i takes up to 30 minutes, and there are around 2000 θ_i formulae to solve!!

Let us now examine the impact of the different optimizations mentioned in Section 5.2.2. In the experiment with `Tcpflow`, we use one additional optimization technique to further shorten the formula solving time. We only solve those θ_i formulae where ψ_i corresponds to a branch in the source code. The effect of this technique is discussed in the next paragraph. By considering only θ_i formulae from the source code, there are still 86 formulae left to solve. The estimated time to solve these formulae comes to 2 days (since the solving of each θ_i formulae in the `TCPflow` program seems to take about 30 minutes). However, recall that in the first step of our formula simplification (see Section 5.2.2), we check whether $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is satisfiable in a time-bounded fashion. In other words, we set a time limit (10 seconds for our experiments), and see how many of the θ_i formulae can be proved to be unsatisfiable within this time limit. Clearly, if $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$

is unsatisfiable, θ_i cannot be satisfiable! We find that 64 out of the 86 formulae are proved to be unsatisfiable in this fashion. Thus, we are left with $(86 - 64)$, that is, 22 formulae to solve. The time to solve these formulae without any further optimization comes to around 12 hours. As mentioned in Section 5.2.2, we further employ dynamic slicing and constant propagation to reduce the burden of the SMT solver. By using all of the formula simplification steps mentioned in Section 5.2.2, the total time taken by the SMT solver is reduced to only 10 minutes. The total debugging time (which includes tracing as well) comes to 33 minutes. The final result from DARWIN contains only 6 statements including the line containing the error cause.

5.4.5 Experiment with latent bug

In this section, we report our experience with a latent injected bug to show a special feature of our debugging method. We want to demonstrate the scenario where the actual bug exists in the old stable program, however it only gets manifested in the new changed program. Note that in such scenarios change analysis based debugging methods such as [123] will not work — since they seek to report a subset of the changes (between the old and new programs) as the cause of error. For trace comparison method in the same program, the issue of generating/selecting the trace to compare still exists. However our method, being based on semantic analysis of the old and new programs, can still locate the error cause.

```
if((state->ins != seq) && !(IS_SET(flags, TH_ACK))){
      return;   /* ERROR here: should be printf("Warning: xxxxxx\n"); */
}
```

Figure 5.7: Injected bug in `TCPflow`

We use the unpatched and patched versions of the `TCPflow` program as described in Section 5.4.4. The injected bug is shown in Figure 5.7. The code in Figure 5.7 is injected in both versions of `TCPflow`. In the unpatched version of `TCPflow`, whenever the code is executed, `state->ins` is always equal to `seq`, the second condition `!(IS_SET(flags, TH_ACK))` is never evaluated and the `return` statement is never executed. However, in the patched version, because of other code modifications, `state->ins` can be not equal to `seq`. As a result, the `return` statement is executed, manifesting the error.

Although we have the same buggy code in both versions, the injected code is actually executed

Table 5.2: Performance of DARWIN's extended debugging method(m=minutes, s=seconds)

Programs	Time in step 1	Time in step 2	Time in step 3	Time in steps 4&5	Total Time
libPNG(v1.0.7-v1.2.21)	3m 57s	1m 49s	7m 44s	4s	13m 34s
Miniweb-Apache	2m 4s	1m 1s	2m 42s	1s	5m 48s
Savant-Apache	2m 27s	1m 11s	5m 2s	10s	8m 50s
TCPflow(unpatched-patched)	7m 9s	57s	20m12s	3m32s	31m 50s

differently in the two versions. This difference is caused by other modifications in the patched version. Change analysis based delta debugging [123] cannot expose such error causes since the error is in a line which was *not* changed across versions.

Using DARWIN, the difference in program executions is captured in the path conditions $pc_{unpatched}$ and $pc_{patched}$ of the unpatched/patched program versions. The branch `!(IS_SET(flags, TH_ACK))` appears in $pc_{patched}$ but not in $pc_{unpatched}$. So, our technique is able to construct an alternate input that satisfies $pc_{unpatched} \wedge \neg pc_{patched}$ by negating the branch `!(IS_SET(flags, TH_ACK))`. Thus one of our θ_i formulae corresponds to a deviation in the branch `!(IS_SET(flags, TH_ACK))`, since this is a branch recorded in the path condition. This deviation results in `!(IS_SET(flags, TH_ACK))` being selected as a potential root cause. On the whole, we identify 10 potential root causes. Clearly, the inclusion of the branch `!(IS_SET(flags, TH_ACK))` as a potential root cause helps the programmer diagnose the issue.

5.4.6 Performance of our debugging method

In this section, we evaluate the performance of our debugging method. The properties of our subject programs in terms of trace size and other statistics appear in Table 5.1.

Recall from Section 5.2.2 that our debugging method with formula simplification involves five steps. The steps are: (i) constructing and checking the satisfiability of the $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ (ii) slicing on the f' (iii) concretize all the inputs that are not in the slicing result and perform constant propagation, (iv) check the satisfiability of the simplified formula $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ after constant propagation (v) solving the simplified formula $pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ (if the simplified $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ is found to be satisfiable in step iv).

Table 5.2 summarizes the time taken in these steps by DARWIN for all programs including

`TCPflow`. The input validation only compares whether two execution traces are the same or different, no formula generation is needed. It takes hardly any time to validate the inputs in all our case studies.

In the first step of our method, we construct the path conditions in the two program versions, and then construct several formulae θ_i. We also use a very short time to check the satisfiability of $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$. We count the time taken to generate the traces and raw path conditions into this step. The total time taken in this step was less than 7 minutes in all the case studies. In the second step, we use dynamic slicing to find out the relevant input bytes for each formula. The time taken is less then 2 minutes in all the case studies. In the third step, we concretize all the irrelevant input bytes and perform constant propagation to simplify the formulae. The time taken by this step was less than 21 minutes in all our case studies. In the last two steps, we first check the satisfiability of the simplified formula $\psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ after constant propagation. If it is satisfiable, we solve the whole formula $pc \wedge \psi_1 \wedge \ldots \psi_{i-1} \wedge \neg\psi_i$ (which also has been greatly simplified by now due to constant propagation). The time taken by this step was less than 4 minutes in all the case studies.

Overall, DARWIN took less than 32 minutes in all the case studies.

5.4.7 Additional overheads due to predicate instrumentation

Our debugging method is most suited for debugging branch errors (errors in program branches) and code-missing errors. For errors in assignments, our technique needs to be augmented with predicate instrumentation as discussed in Section 5.2.3. Our predicate instrumentation is geared to expose assignment errors as mentioned in Section 5.2.3. We introduce branches with branch conditions checking the following —

- function return values at each function return site, and
- binary constraints describing equality of a program variable `x` with other variables of the same type, at each assignment to `x`. Thus, if `x`, `y` are of the same type — we introduce branches to check `x == y`.

Table 5.3 shows the overhead for our predicate instrumentation. The additional branches and instructions are introduced because of our predicate instrumentation. We only show the numbers for `TCPflow` (a program with high instrumentation overhead) and `miniweb` (a program with low instrumentation overhead). The overhead in terms of number of additional branches and instructions is less than 20%. The instrumentation is done at source code level, and hence library code is not instrumented. This also prevents the instrumentation overhead from blowing up.

Table 5.3: Overhead of Predicate Instrumentation

Programs	Additional branches (%)	Additional Instructions (%)
TCPflow	17.78%	16.48%
Miniweb	4.06%	3.83%

5.5 Threats to Validity

In this section, we discuss certain threats to validity of the results presented in this chapter. This also clarifies any implicit assumptions on which our debugging method may be built.

- One key assumption of our approach is the program requirements vis-a-vis the buggy input do not change. The program requirements for the buggy input define the supposed behavior of the program execution with the buggy input. In reality, what commonly happens is that the program requirements vis-a-vis existing features do not change (although new features may be added). In such a case, our assumption is guaranteed to be satisfied. In fact a typical scenario where DARWIN is applicable may be described as follows. A program version P evolves to a new program version P' because the customers want some new features to be added. However in trying to program the new features, the code for the old features mistakenly gets affected. Thus, a test case t which used to pass in program P, fails in the new program P'. In other words, in going from program P to program P' there is code evolution but no evolution of requirements. The requirements for the old features (those supported by both P and P') remain unchanged. DARWIN is most suited to explain and root-cause such errors resulting from *code evolution*.

  Note that the above assumption does not conflict with our claim that DARWIN works with two different implementation of the same specification. Suppose P and P' are two different implementations such as `miniweb` and `apache`. As long as the behavior of the buggy input is supposed to be the same in both P and P', we can use P as a reference implementation to debug P'.

To illustrate the issue with a more concrete example, consider a banking system P supporting some basic features like “login”, “logout”, “view balance” and so on. Suppose now the customers of the banking system demand a new feature for transferring funds between accounts. In trying to implement this system and produce a new banking system P', the programmer may make mistakes and incorrectly modify the account balance. As a result, the “view balance” functionality, which used to work correctly earlier, may not work correctly any more, leading to an observable error. DARWIN is most suited for explaining the root-cause of such observable errors. Consider an alternate scenario where the requirements of the banking system itself being changed. Suppose the “view balance” functionality earlier used to be interpreted as viewing of the account balance, and is now changed to display the account balance for current accounts and displays the account balance minus \$50 (the minimum deposit) for savings accounts. In this situation, the requirements of the “view balance” feature itself has changed. DARWIN approach is not suited to explain any errors resulting from such evolution of software requirements.

- Path conditions serve as the basis of our debugging technique. In particular, the approach hinges on the observation that the path conditions pc, pc' of the test input t being debugged are different in the two program versions. What if pc and pc' are logically equivalent? This means that the effect of the error being debugged is not observable by a difference in control flow. Our DARWIN approach is not inherently suited to explain such errors. Thus, the approach is most suited for explaining errors that manifest as changes in control flow. In Section 5.2.3, we proposed some methods to introduce more control flow paths to handle assignment errors that do not affect control flow. Even with heavy instrumentation, our solution cannot guarantee that all such errors will be correctly diagnosed.

- Regarding the scalability of our technique, the size of generated SMT formula largely depends on the number of tainted instructions in the execution trace. This is because only the tainted instructions are analyzed in the path condition generation and all subsequent steps of our tool. From our experience in the experiments, we found that the number of tainted instructions depends on the input size as well as the size of the program. Since SMT solving is extensively used in our approach, the scalability of our approach is also directly tied to the scalability of the SMT solvers. We believe that there are generally two ways to increase the scalability of our approach. First, we can use various methods to reduce our SMT formula size. The high-level idea is to remove something unrelated. In the chapter, we have presented a means that concretizes some unrelated input bytes and backward slices out unrelated components. For a particular program, the user may know which modules/functions are trustable. These information can be used to reduce the formula size further. If a program has large structured input, the technique from [125] would be useful to simplify the input before applying our tool. Secondly, the scalability of the SMT solvers is increasing all the time. This could also benefit our approach.

- Although our DARWIN tool is built based on the C binary executables, our technique can be generalized to other programming languages. As we will see in Chapter VI, the DARWIN technique is also implemented for Java. As long as the errors can affect program flow and program requirements vis-a-vis the buggy input are the same, our technique should apply.

- Finally, there are some limitations regarding our experiments. Long program execution with large input size would produce large SMT formulae. We did not perform any experiments on programs of this kind. For programs with large structured inputs, we suggested that some input simplification techniques should be adopted. We did not perform any experiments to evaluate the effectiveness of these simplification techniques on DARWIN. For errors in assignment, one may need to follow dependency links to find the root cause if our instrumentation technique in section 5.2.3 is not used. Some manual code inspection is needed in this case. We did not perform any case studies to evaluate this manual effort. However, as suggested by the result in subsection 5.4.7, the instrumentation overhead is affordable. Therefore, users could employ the instrumentation technique to expose errors in assignment.

5.6 Summary

In this chapter, we presented DARWIN a debugging methodology and tool for evolving programs. DARWIN takes in two programs and explains the behavior of a test input which passes in the stable program, while failing in the buggy program. The stable program and buggy program can be two completely different implementations of the same specification. DARWIN handles hard-to-explain code missing errors by pointing to code in the stable program. We have conducted experiments using four real world applications such as the Apache web server, libPNG (a library for manipulating PNG images), and TCPflow (a program for displaying data sent through TCP connections). Our experience with real-life case studies demonstrates the utility of our method for localizing real bugs.

Developers are often faced with hard-to-locate bugs when a large software system changes from one version to another. As long as the program requirements vis-a-vis existing features do not change, DARWIN can truly be an useful automatic debugging assistant for developers.

The alternate inputs generated by our method can also help *detect* new errors, apart from localizing a given observable error. This can also help test-suite augmentation of evolving programs — when a program changes we can find out potentially new test cases to be tested for stressing the change.

CHAPTER VI

Path Exploration based on Symbolic Output

Our proposed DARWIN method focused on path condition for software debugging. While providing a unique opportunity to explore dynamic symbolic execution for debugging, it also raises the question about the level at which the symbolic execution should be performed. In DARWIN, symbolic execution is performed at the path level. However, we note that debugging only concerns about whether output is expected whereas a full path contains computation that do not affect the "unexpected output". The example in Figure 6.1 explains one such scenario.

Suppose the observed value of `x` in Figure 6.1b is unexpected for `inp1 == inp2 == 0` because of a "bug" in line 2 (say, the condition should be `inp1 >= 0`). The path condition of the above input in the current program is $pc' = \neg(inp1 > 0) \wedge \neg(inp2 > 0)$, while the path condition of the above input in the previous program is $pc = \neg(inp1 >= 0) \wedge \neg(inp2 > 0)$. In our DARWIN technique, the two path conditions pc and pc' are compared to localize the cause of the regression error. However, it is clear that the constraint $\neg(inp2 > 0)$ corresponding to the branch in line 6 is

```
1  ... // input inp1, inp2
2  if (inp1 >= 0)
3    x = inp1 + 1;
4  else
5    x = inp1 - 1;
6  if (inp2 > 0)
7    y = inp2 + 1
8  else
9    y = inp2 - 1;
10 ... // output x, y
```

(a) Previous program

```
1  ... // input inp1, inp2
2  if (inp1 > 0)
3    x = inp1 + 1;
4  else
5    x = inp1 - 1;
6  if (inp2 > 0)
7    y = inp2 + 1
8  else
9    y = inp2 - 1;
10 ... // output x, y
```

(b) Current program

Figure 6.1: An example of computation unrelated to observable error

unrelated to the observable error (unexpected value of `x`). Although our DARWIN technique can still locate the root-cause of the regression bug, unnecessary effort is wasted in symbolic execution and SMT solving due to the inclusion of the branch constraint from line 6 that is unrelated to the observable error. Instead, we could remove the branch constraint $\neg(inp2 > 0)$ from the two path conditions and compare the "reduced path conditions": $\neg(inp1 > 0)$ and $\neg(inp1 >= 0)$. By following the same method as in DARWIN, we could locate the error cause with less effort by comparing the "reduced path conditions". In general, taking out branch constraints unrelated to the observable error from path conditions not only helps save the effort required in symbolic execution but also helps reduce unnecessary noise the debugging results.

Similar issues appear in symbolic execution based software testing, e.g. Directed Automated Random Testing or DART [51] which try to achieve path coverage in test-suite construction. While focusing on checking the correctness of the program output in testing, a program output can be defined as a symbolic expression in terms of the program inputs. Thus, given a program P, instead of enumerating all program paths, we can seek to enumerate all the different possible symbolic expressions which describe how the output will be computed in terms of the inputs. Of course, the symbolic expression defining the output (in terms of the inputs) will be different along different program paths. However, we expect that the number of such symbolic expressions to be substantially lower than the number of program paths. In other words, a large number of paths can be considered "equivalent" since the symbolic expressions describing the output are the same.

We now use an example to demonstrate the opportunity brought by this new persecutive in software testing.

To illustrate our observation, let us consider the program in Figure 6.2. The output variable `out` can be summarized as follows.

- If $x - y > 0$ and $x + y > 10$, then $out == x$
- If $x - y \leq 0$ and $x + y > 10$, then $out == y$
- If $x + y \leq 10$, then $out == 2$

The summary given in the preceding forms a "semantic signature" of the program as far as the output variable `out` is concerned. Note that there are *only three* cases in the semantic signature -

```
1  int foo(int x, int y, int z){//input variables
2    int out; // output variable
3    int a;
4    int b = 2;
5    if(x - y > 0) //b1
6      a = x;
7    else
8      a = y;
9    if(x + y > 10) //b2
10     b = a;
11   if(z*z > 3)  //b3
12     System.out.println("square(z) > 3");
13   else
14     System.out.println("square(z) <= 3");
15   out = b;
16   return out; //slicing criteria
17 }
```

Figure 6.2: An example program

whereas there are *eight paths* in the program. Thus, such a semantic signature can be much more concise than an enumeration of all paths.

In this chapter, we develop a method to compute a semantic signature for a given program by performing symbolic execution at the level of relevant slice [53]. Our semantic signature is computed via dynamic path exploration. While exploring the paths of a program, we establish a natural partitioning of paths *on-the-fly* based on program dependencies - such that only one path in a partition is explored. Thus, for the example program in Figure 6.2 only three execution traces corresponding to the three cases will be explored. For test-suite construction, we can then construct only three tests corresponding to the three cases in the semantic signature.

Our path partitioning method is based on the notion of "path equivalence". We consider two program paths to be "equivalent" if they have the same relevant slice with respect to the program output. A relevant slice is the transitive closure of dynamic data, control and potential dependencies. Data and control dependencies capture statements which affect the output by getting executed; on the other hand, potential dependencies capture statements which affect the program output by *not* getting executed. In Figure 6.2, even if line 10 is not executed, the assignment to `out` in line 15 is potentially dependent on the branch in line 9. This is to capture the fact that if line 9 is evaluated differently, the assignment in line 10 will be executed leading different values flowing to the variable `out` in line 15. We base our path partitioning on relevant slices to capture all possible flows into the output variable - whether by the execution of certain statements or their non-execution.

The contributions of this chapter can be summarized as follows. We present a mechanism to partition program paths based on the program output. The grouping of paths is done by efficient dynamic path exploration - where paths sharing the same relevant slice naturally get grouped together. We show that our smart path exploration is much more time efficient as opposed to full path exploration via path enumeration. Our efficient path exploration method has immediate benefits in software testing. Since our path exploration naturally groups several paths together - it is much more efficient than the full path exploration (as in Directed Automated Random Testing or DART) as evidenced by experiments. Moreover, since several paths are grouped as "equivalent" in our method (meaning that these paths compute the output similarly), the test-suite generated from our path exploration will also be concise.

Secondly, we show the application of our path partitioning method in reasoning about program versions, in particular, for debugging the root-cause of software regressions. While trying to introduce new features to a program, existing functionality often breaks. Given two program versions P, P' and a test t which passes in P while failing in P' — we seek to find a bug report explaining the root cause of the failure of t in P'. In chapter V, we presented the DARWIN approach for root causing software regressions. The DARWIN approach constructs and composes the path conditions of test t in program versions P, P' in trying to come up with a bug report explaining an observed regression. In this work, we show that computing and composing the logical condition over a relevant slice (also called *relevant-slice condition* throughout the chapter) produces more pin-pointed bug reports in a shorter time — as opposed to computing and composing path conditions. The reason for obtaining shorter bug reports in lesser time comes from the path conditions containing irrelevant information which are filtered out in *relevant-slice conditions*. Hence *relevant-slice conditions* are smaller formulae, which are constructed and solved (via Satisfiability Modulo Theory solvers) more efficiently.

Finally, we show two applications of the "semantic signature" produced by our path partitioning method. We first apply the "semantic signature" on test-suite augmentation, which is to augment the existing test-suite after a program changes. We compare the semantic signatures of the previous and current program versions. Differences in the signatures lead to test cases that have different outputs in the two versions. These test cases are used to augment the existing test-suite.

6.1 Overview

We begin with a few definitions. Let us first define slice conditions, which are path conditions computed over slices.

Definition 6.14 (Dynamic Slice Condition)**.** *Given a program P, a test input t and a slicing criteria C — let π be the execution trace of t in P. Let $\pi \mid_C$ denote the projection of π w.r.t. the dynamic slice of C in π. In other words, a statement instance s in π is included in the projection $\pi \mid_C$ if and only if s is in the backward dynamic slice of C on π. The dynamic slice condition of C in π is the path condition computed over the projected trace $\pi \mid_C$.*

Slice conditions are weaker than path conditions, that is, $pc_\pi \Rightarrow dsc_{(\pi,C)}$ where $dsc_{(\pi,C)}$ is the dynamic slice condition of any slicing criteria C in π. We now refine dynamic slice condition to *relevant-slice condition* - the central concept behind our path partitioning. But first, let us recall the notion of potential dependencies and relevant slices [15, 53].

Definition 6.15 (Potential Dependence [15])**.** *Given an execution trace π, let s be a statement instance and br be a branch instance that is before s in π. We say that s is potentially dependent on br iff. there exists a variable v used in s such that (i) v is not defined between br and s in trace π but there exists another path σ from br to s along which v is defined, and (ii) evaluating br differently may cause this untraversed path σ to be executed.*

An example of potential dependence for the program in Figure 6.2 is shown in Figure 6.3.

We now introduce the notion of a relevant slice, and *relevant-slice condition*, a logical formula computed over a relevant slice.

Definition 6.16 (Relevant slice)**.** *Given an execution trace π and a slicing criteria C in π, the relevant slice in π w.r.t. C contains a statement instance s in π iff. $C \rightsquigarrow s$ where $\rightsquigarrow$ denotes the transitive closure of dynamic data, control and potential dependence.*

Note that our definition of relevant slice is slightly different from the standard definition of relevant slice [15, 53]. In standard relevant slicing algorithm, if a statement instance s is included only by potential dependence, the statement instances that are only control dependent by s are not included in the relevant slice. We have removed this restriction to simplify the definition of relevant

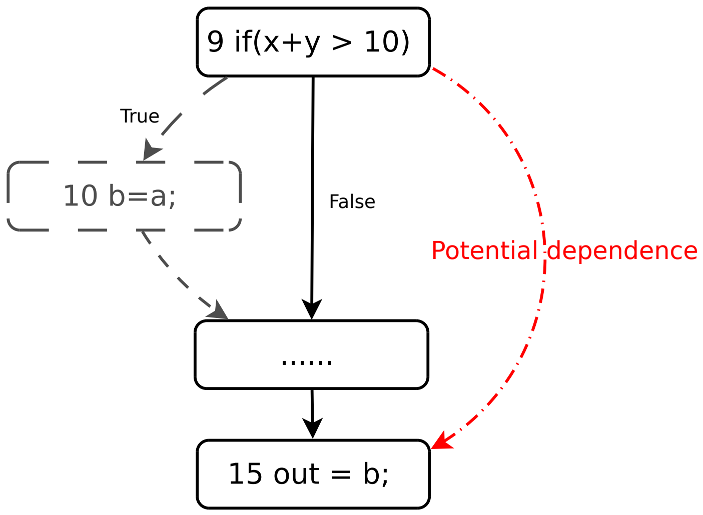


Figure 6.3: Example of potential dependence. The solid arrows denote the execution path. According to Definition 6.15, (i) the variable b is not defined between line 9 and line 15 but there exists a path (though line 10) along which b is defined, and (ii) evaluating the branch at line 9 differently may cause the path through line 10 to be executed. Therefore, line 15 is potentially dependent on line 9.

slice, it is simply the transitive closure of three kinds of program dependencies — dynamic data dependencies, dynamic control dependencies and potential dependencies. In the rest of this chapter, all appearances of relevant slice and *relevant-slice condition* refer to this simplified definition of relevant slice.

Definition 6.17 (Relevant slice condition)**.** *Given an execution trace π and a slicing criteria C in π, the relevant slice condition in π w.r.t. criterion C is the path condition computed over the statement instances of π which are included in the relevant slice of C in π.*

```
int foo(int x){ //input variable
  int a = 0;
  x = x - 1;
  if(x > 0)
    a = 1;
  out = a;
  return out;
}
```

Figure 6.4: Example to show path condition and *relevant-slice condition* computation

We take the example program in Figure 6.4 to show that the effect of assignments is also considered in *relevant-slice condition* computation (just as assignments are considered in path condition computation). Let the slicing criteria be the value of `out` in line 7. The relevant slice for input $\langle x == 0 \rangle$ is [2,3,4,6,7] and the corresponding *relevant-slice condition* is $\neg(x - 1 > 0)$. That is, the effect of the assignment in line 3 is considered.

We use the simple program in Figure 6.2 to illustrate the advantage of using *relevant-slice condition* in dynamic path exploration. The slicing criterion is the variable `out` at line 16. Since each statement is executed once, we do not distinguish between different execution instances of the same statement in this example.

We use the executed branch sequence annotated with directions to represent an execution trace. For example, the trace for input $\langle x == 6, y == 2, z == 2 \rangle$ of the program in Figure 6.2 is denoted as $[b1^t, b2^f, b3^t]$. Let us take the input $\langle x == 6, y == 2, z == 2 \rangle$ to see the differences between path condition, dynamic slice condition and *relevant-slice condition*. Given the trace $[b1^t, b2^f, b3^t]$ corresponding to input $\langle x == 6, y == 2, z == 2 \rangle$, the path condition along this execution is $(x - y > 0) \wedge \neg(x + y > 10) \wedge (z * z > 3)$.

For the execution path of $\langle x == 6, y == 2, z == 2 \rangle$, the dynamic backward slice result w.r.t. the slicing criteria at line 16 is [4,15,16] - it contains no branches. The path condition computed over the statements in the dynamic slice (or the dynamic slice condition) is simply the formula $true$.

Different from dynamic backward slicing, relevant slicing also includes the statement instances that could potentially affect the slicing criteria. For example, if evaluating a branch differently could affect the slicing criteria — such a branch is included in the relevant slice, even though it is not contained in the dynamic backward slice. In the example program, the branch at line 9 can potentially affect the value of *out* in the slicing criteria. This is because if the branch in line 9 is evaluated differently (to true), the variable b is re-defined (in line 10) which affects the output variable out. Hence the relevant slice contains line 9. The entire relevant slice is [4,9,15,16], and the *relevant-slice condition* on it is $\neg(x + y > 10)$. Any input t satisfying the *relevant-slice condition* $\neg(x + y > 10)$ has the same symbolic expression for the output `out`, which in this case turns out be the constant value 2.

As mentioned earlier, program paths can be partitioned based on the input-output relation. *Relevant-slice condition* perfectly serves this purpose. If two paths have the same relevant slice with output being the slicing criteria, then they have the same input-output relation. The path partitions of the program in Figure 6.2 are shown in Figure 6.5. The grey nodes in Figure 6.5 are the statements

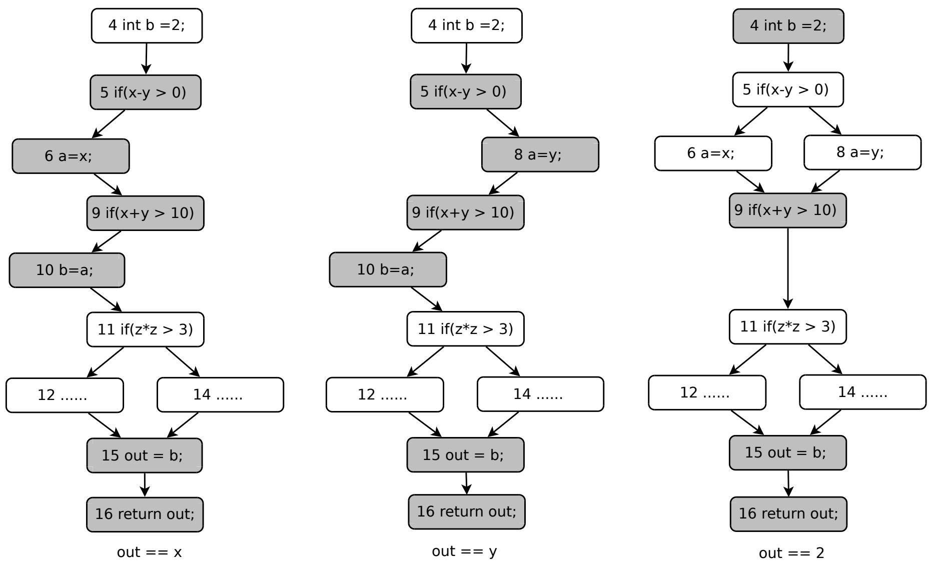


Figure 6.5: Path partitions of the example in Figure 6.2

that are contained in the relevant slice w.r.t. to the unique slicing criteria at line 16 in Figure 6.2. As we can see from Figure 6.5, based on the relevant slice, we can group the eight program paths into three path partitions.

Just like the DART approach [51] uses path conditions to dynamically explore paths in a program, *relevant-slice condition* can be used to explore the possible symbolic expressions that the program output can be assigned to. How would such an exploration proceed? Suppose we simply use *relevant-slice condition* to replace path condition in DART's path exploration. Given a *relevant-slice condition* $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ — we construct k sub-formulae of the form of $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $1 \leq i \leq k$. The path exploration is done by solving these formulae to get new inputs and iteratively applying this process to the new inputs. Note that each sub-formula shares a common prefix with the *relevant-slice condition*. Now, we examine the effectiveness of this simple solution on the program in Figure 6.2. Depth-first exploration strategy is used, and path exploration terminates when no new sub-formulae are generated. Let the initial input be $\langle x == 6, y == 2, z == 2\rangle$, the path for this input is $[b1^t, b2^f, b3^t]$. The entire path exploration process is shown in Table 6.1. The "from" column of Table 6.1 can be understood as follows. If the "from" column contains $\alpha.\beta$, it means that the current input is generated by negating the βth branch constraint of the *relevant-slice condition* in the αth row.

Recall that we expect the following three symbolic expressions for out to be explored.

- $x - y > 0 \wedge x + y > 10$: `out` $== x$
- $\neg(x - y > 0) \wedge x + y > 10$: `out` $== y$
- $\neg(x + y > 10)$: `out` $== 2$

As we can see from Table 6.1, no path having *relevant-slice condition* $\neg(x - y > 0) \wedge (x + y > 10)$ is explored. Therefore, this feasible *relevant-slice condition* is missed by the exploration process. In addition, the *relevant-slice condition* $\neg(x + y > 10)$ is explored several times. Thus, we cannot simply replace path condition with *relevant-slice condition* in DART's path exploration.

Let us examine closely what went wrong in the path exploration of Table 6.1. In particular, the input in the third row is generated by negating the second branch condition of the *relevant-slice condition* in second row in Table 6.1. That is, when we solve $(x - y > 0) \wedge \neg(x + y > 10)$, we get an input $\langle x == 6, y == 2, z == 2\rangle$ whose *relevant-slice condition* is $\neg(x + y > 10)$. The branch condition $(x - y > 0)$ disappears in the new *relevant-slice condition* because the corresponding branch is not contained in the relevant slice anymore. In contrast, DART follows certain path-prefixing properties — if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is the prefix of a path condition (for some program input), the path condition of any input satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ will have $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. Such a property does not hold for *relevant-slice condition*. Hence, simply replacing path condition with *relevant-slice condition* in DART not only causes redundant path exploration but also makes the exploration incomplete (in terms of possible symbolic expressions that the output variable may assume).

We have developed a path exploration method which avoids the aforementioned problems. While exploring (groups of) paths based on *relevant-slice condition*, our method re-orders the constraints in the *relevant-slice condition*. The path exploration is based on reordered *relevant-slice condition*. A reordered *relevant-slice condition* satisfies the following property (which also holds for path conditions): if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of a reordered *relevant-slice condition*, the reordered *relevant-slice condition* of any input satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ has $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix.

Table 6.1: Path exploration based on *relevant-slice conditions* for example in Figure 6.2

No.	From	Input	Path	RSC	Path condition
1		$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$	$(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$
2	1.1	$\langle 6,5,2\rangle$	$[b1^t, b2^t, b3^t]$	$(x-y>0)\wedge(x+y>10)$	$(x-y>0)\wedge(x+y>10)\wedge(z*z>3)$
3	2.2	$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$	$(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$
4	2.1	$\langle 2,6,2\rangle$	$[b1^f, b2^f, b3^t]$	$\neg(x+y>10)$	$\neg(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$

Table 6.2: Path exploration based on reordered *relevant-slice conditions* for example in Figure 6.2

No.	From	Input	Path	RSC	Reordered RSC
1		$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$	$\neg(x+y>10)$
2	1.1	$\langle 6,5,2\rangle$	$[b1^t, b2^t, b3^t]$	$(x-y>0)\wedge(x+y>10)$	$(x+y>10)\wedge(x-y>0)$
3	2.2	$\langle 5,6,2\rangle$	$[b1^f, b2^t, b3^t]$	$\neg(x-y>0)\wedge(x+y>10)$	$(x+y>10)\wedge\neg(x-y>0)$

6.2 Our Approach

In this section, we give our path exploration algorithm based on *relevant-slice condition*. We then give theorems on the completeness of our path exploration algorithm. Throughout this chapter, we assume that the slicing criteria is in a basic block that post-dominates the entry of the program. More discussion of this assumption is provided at the beginning of Section 6.2.2.

First we introduce the following notations.

Notations We use C to denote the unique slicing criteria. When used in a dynamic context, C refers to the last executed instance of the slicing criteria. Given a test case t, we use $\pi(t)$ to denote the execution path of t. We use $rs(sc,\pi)$ to denote the relevant slice on path π w.r.t. slicing criteria sc. We use $rsc(sc,\pi)$ to denote the relevant slice condition on path π w.r.t. slicing criteria sc. We use $reordered\_rsc(sc,\pi)$ to denote the reordered sequence of $rsc(sc,\pi)$. We use $br(\psi)$ to denote the branch instance of a branch condition ψ. We use $bc(b)$ to denote the branch condition generated by b. Given a *relevant-slice condition* or reordered *relevant-slice condition* θ and a branch condition ψ, we use $\theta\backslash\psi$ to denote the result of removing ψ from θ. Recall that θ is a conjunction of branch conditions. If ψ is contained in θ, ψ is deleted from the conjunction to get $\theta\backslash\psi$. Otherwise, $\theta\backslash\psi$ is the same as θ.

6.2.1 Path exploration algorithm

We now present our path exploration method which operates on a given program P. All relevant slices and *relevant-slice conditions* are calculated on the same program P with respect to a slicing criteria C (which refers to the program output).

We group paths based on *relevant-slice condition*. As explained in the last section, a DART-like search based on *relevant-slice conditions* is incomplete, that is, not all possible symbolic expressions that the output may assume will be covered. For this reason, we reorder the *relevant-slice conditions*.

Our path exploration algorithm RSCExplore is shown in Algorithm 6.1. The core of the algorithm is the *reorder* procedure, which reorders the *relevant-slice conditions*. When we compute the *relevant-slice condition*, we get a sequence of branch conditions – ordered according to the sequence in which they are traversed. We use the $reorder$ function to reorder the branch conditions, after which the path exploration will be performed based on the reordered sequence of branch conditions.

The $reorder$ procedure is given in Algorithm 6.1. The reordering works in a quick-sort-like fashion. In each call to *reorder*, we split the to-be-reordered sequence into two sub-sequences. Suppose the last branch condition in the sequence is from branch instance b_k. Then b_k is used as the "pivot" in the splitting process. If a branch instance b is in the backward relevant slice of b_k, then the branch condition of b is placed before the branch condition of b_k. Otherwise, the branch condition of b is placed after the branch condition of b_k. Then we recursively call the *reorder* procedure to reorder the two sub-sequences.

We show the *reorder* procedure in action in Figure 6.6. Note that our reordering is done on branch conditions in a *relevant-slice condition*. Since there is a unique branch condition for each branch instance in the execution trace, the example in Figure 6.6 is on branch instances for simplicity. On the left of Figure 6.6, the dependencies among all the branch instances are provided. If there is an arrow from bj to bi, then bi is in the relevant slice of bj. The "pivot" in each reorder step is marked in dark; the other branches are reordered w.r.t. to the "pivot". For example, initially `b6` is the pivot and we reorder `b1`, ... `b5` depending on whether they are in the relevant slice of `b6`.

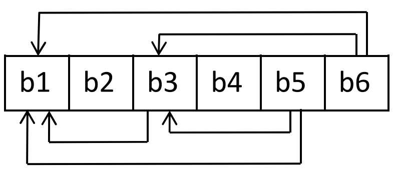
In Algorithm 6.1, we use a stack to maintain the to-be-explored partial *relevant-slice conditions*. The main algorithm keeps on processing the formulae in the stack when it is not empty. In each iteration, the algorithm pops out one partial *relevant-slice condition* from the stack, and checks

Algorithm 6.1 RSCExplore:path exploration using *relevant-slice condition*

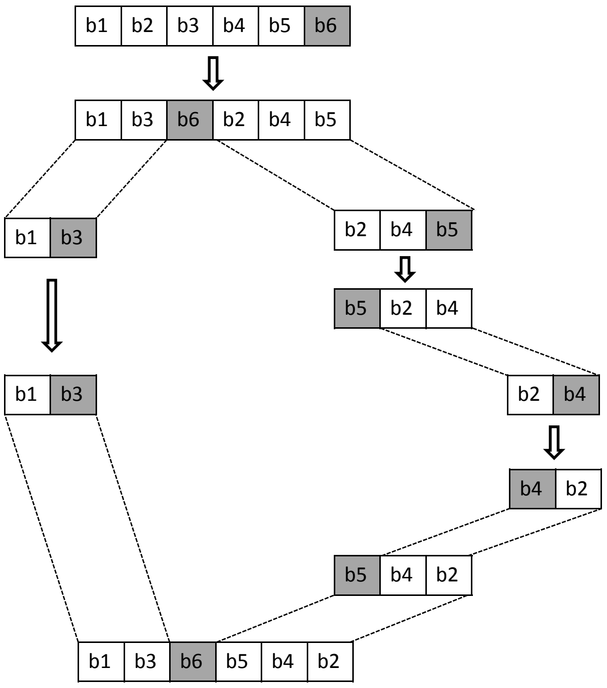
```
Input:
$P$ : The program to test
$t$ : An initial test case for $P$
$C$ : A slicing criterion
Output:
$T$: A test-suite for $P$

$Stack = null$ // The stack of partial rsc to be explored
$Execute(t, 0)$
while $Stack$ is not empty do
    let $\langle f, j\rangle = pop(Stack)$
    if $f$ is satisfiable then
        let $\mu$ be one input that satisfies $f$
        put $\mu$ into $T$
        $Execute(\mu, j)$
    end if
end while
return $T$

procedure $Execute(t, n)$
    execute $t$ in $P$ and compute relevant-slice condition $rsc$ w.r.t. $C$
    let $rsc = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{m-1} \wedge \psi_m$
    let $rsc' = reorder(rsc)$
    suppose $rsc' = \psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{m-1} \wedge \psi'_m$
    for all i from n+1 to m do
        let $h = (\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i)$
        push $\langle h, i\rangle$ into $Stack$
    end for
    return
end procedure

procedure $reorder(seq)$
    if $|seq| \leq 1$ then
        return $seq$
    end if
    let $seq$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$
    $seq_1 = true$, $seq_2 = true$
    for all i from 1 to k-1 do
        if $br(\psi_i)$ is in relevant slice of $br(\psi_k)$ then
            $seq_1 = seq_1 \wedge \psi_i$
        else
            $seq_2 = seq_2 \wedge \psi_i$
        end if
    end for
    return $reorder(seq_1) \wedge \psi_k \wedge reorder(seq_2)$
end procedure
```

Algorithm 6.2 Augmented reorder

```
 1: procedure $reorder(seq, p)$
 2:     let $seq$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$
 3:     if $|seq| == 1$ then
 4:         assign the priority of $b(\psi_1)$ as $p@[b(\psi_1)]$
 5:     end if
 6:     if $|seq| \leq 1$ then
 7:         return $seq$
 8:     end if
 9:     $seq_1 = true$, $seq_2 = true$
10:     for all i from 1 to k-1 do
11:         if $b(\psi_i)$ is in relevant slice of $b(\psi_k)$ then
12:             $seq_1 = seq_1 \wedge \psi_i$
13:         else
14:             $seq_2 = seq_2 \wedge \psi_i$
15:         end if
16:     end for
17:     assign the priority of $b(\psi_k)$ as $p@[b(\psi_k)]$
18:     $seq'_1 = reorder(seq_1, p@[b(\psi_k)])$
19:     $seq'_2 = reorder(seq_2, p)$
20:     return $seq'_1 \wedge \psi_k \wedge seq'_2$
21: end procedure
```



(a) Dependencies among branch instances (Arrows denote both direct and indirect dependencies).



(b) Reorder process. Each arrow represents one single reorder step. The "pivot" nodes are highlighted in grey.

Figure 6.6: *Reorder* process for relevant slice condition from relevant slice [b1,b2,b3,b4,b5,b6]

whether it is satisfiable or not. If it is satisfiable, we get a new input μ by solving the formula. The new input μ could lead to some unexplored *relevant-slice condition*. The *relevant-slice condition* for the execution trace of input μ is then explored, as shown by the procedure *Execute* in Algorithm 6.1. Given the execution trace of μ, the *relevant-slice condition* over this trace w.r.t. the slicing criteria C is first computed. The *relevant-slice condition* is reordered using the *reorder* procedure, and the to-be-explored partial *relevant-slice conditions* are pushed into the stack.

The second parameter of $Execute$ is used to avoid redundancy in path search. When $Execute$ is called with parameters t and n, let the reordered *relevant-slice condition* $reordered\_rsc(C, \pi(t))$ be $\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{m-1} \wedge \psi'_m$. For any partial *relevant-slice condition* $\varphi_i = \psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i$, $1 \leq i \leq n \leq m$, we know that φ_i has been pushed into the stack a-priori. So the for-loop in the *Execute* procedure starts from $n + 1$ to avoid these explored partial *relevant-slice conditions*.

The path exploration of Algorithm 6.1 when employed on the program in Figure 6.2 leads to the *relevant-slice conditions* shown in Table 6.2. If the "from" column of Table 6.2 contains $\alpha.\beta$, it means that the current input is generated by negating the βth branch constraint of the reordered *relevant-slice condition* in the αth row. The path exploration based the reordered *relevant-slice condition* explores all possible *relevant-slice conditions* of the program.

We now use the same example program in Figure 6.2 to explain that our technique is different from employing path condition based path exploration on the static slice of the program. Given a slicing criteria, we could first perform static slicing on the program with respect to the slicing criteria. Since the static slicing result is also a complete program, we can enumerate all paths of the static slice. Applying this approach on the program in Figure 6.2, the static slice result contains all lines except for lines 11-14. As there are two branches in the static slice, path exploration based on path condition explores all four feasible paths. In contrast, our technique generates only three *relevant-slice conditions* as shown in Table 6.2.

6.2.2 Proofs

Assumptions We assume that the SMT solver used to solve *relevant-slice conditions* is sound and complete (more discussion on this assumption is given in Section 6.5). As mentioned earlier, we assume that the slicing criteria is in a basic block that post-dominates the entry of the program — this is the location of the program output. This assumption makes sure that the slicing criteria is

```
int foo(int x){
  if(x > 0){
    return x+1;//slicing criteria
  else
    return x+2;//slicing criteria
}
```

(a) Program before transformation

```
int foo(int x){
  int ret;
  if(x > 0){
    ret = x+1;
  else
    ret = x+2;
  return ret;//slicing criteria
}
```

(b) Program after transformation

Figure 6.7: Example of program transformation that makes the slicing criteria post-dominates the program entry

executed for any program inputs. If a program does not satisfy this assumption, a simple program transformation can produce an equivalent program that meets this assumption. We give one example of transformation in Figure 6.7. If the program contains *multiple outputs*, the slicing criteria can simply be a set of primitive criteria of the form

$$\langle output\ variable, output\ location\rangle$$

Note that slicing can be performed on such a criteria (which is a set) without any change to our method.

Execution Index In the following proofs as well as our implementation, we need to align/compare different paths. Hence, it is critical to determine whether two statement instances from different paths are the same. We use the concept of execution index [117], which is defined as:

Definition 6.18 (Execution Index [117]). *Given a program P, the index of an execution trace π of P, denoted as EI^{π}, is a function of execution points in π, that satisfies: $\forall$ two execution points $x \neq y, EI^{\pi}(x) \neq EI^{\pi}(y)$.*

Two statement instances in different paths are the same iff. they have exactly the same "execution index". Given two paths π and σ and a statement instance s in π, we say s also appears in σ iff. in σ there is a statement instance s' such that $EI^{\pi}(s) == EI^{\sigma}(s')$. In its simplest form, we use the path from root to s in the Dynamic Control Dependence Graph of π as the execution index of s in π.

Additional Notations Used in Proofs Over and above the notations introduced earlier, we use the following notations in our proofs. The immediate post-dominator of a branch b is denoted as $postdom(b)$. We use $\rightarrow_d$ to denote dynamic data dependence, and $\rightarrow_c$ to denote dynamic control dependence. We use $\rightarrow_p$ to denote potential dependence. We use $\rightsquigarrow_d$ to denote transitive data dependence and $\rightsquigarrow_c$ to denote transitive control dependence. When no subscript is specified, we use $\rightarrow$ to denote any type of direct dependence and $\rightsquigarrow$ to denote the transitive closure of $\rightarrow$.

We use $\rightsquigarrow_s$ to denote a *special* kind of transitive dependence. Let u be a statement instance and b be a branch instance in path π, then $u \rightsquigarrow_s b$, iff. (i) there exist a variable v used at u, (ii) there is no definition of v between $postdom(b)$ and u and (iii) there is at least one static definition of v that is statically transitively control dependent on the static branch of b. There could potentially many static definitions of v that are statically transitively control dependent on static branch of b. Depending on whether these definitions of v are executed, there are two different scenarios when $u \rightsquigarrow_s b$. If all the definitions of v are not executed, then u is potentially dependent on b. Otherwise, u is data dependent on the last definition of v (say it is d), and d is control dependent on b. In both cases, there is a dependence chain from u to b.

Transformations For the ease of our proofs, we statically transform any program in the following way. Note that the transformations do not affect the program semantics, and do not impact the generality of our proofs. (i) We add a dummy statement nop at the start of the program. The statement nop means no operation. (ii) If the slicing criteria (output statement(s)) are not a branch, we add a dummy branch that contains a dummy use for each variable appearing in the output statement(s). We use this branch as the slicing criterion C.

6.2.2.1 Priority sequence and shortened priority sequence

We need to prove the following property for *relevant-slice condition*: if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of a reordered $reordered\_rsc(C, t)$, the reordered *relevant-slice condition* of any input t' satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ has $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. There are two important facts to prove:(i) Each $b(\psi_k)$, $1 \leq k \leq i$, is included in the relevant slice in path $\pi(t')$. (ii) The relative order of branch conditions in $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ is not changed. To prove these two facts, we need to find out what is not changed between $reordered\_rsc(C, t')$ and $reordered\_rsc(C, t)$. In the following, we define a shortened priority sequence $sp(b)$ for each branch instance b. The shortened priority sequence has the following two properties:

1. Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reordered\_rsc(C, \pi(t))$. If $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1}$, then $sp(b(\psi_i))$ is the same in $\pi(t)$ and $\pi(t')$.

2. Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$ by the reorder algorithm in Algorithm 6.2, then $sp(b_x) > sp(b_y)$.

The first property means that the shortened priority sequence for the corresponding branch instance of each branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is not changed between $reordered\_rsc(C, t')$ and $reordered\_rsc(C, t)$. The second property means that the shortened priority sequence essentially defines the order of branch conditions in a reordered *relevant-slice condition*. We explain how the shortened priority sequence is computed in the following.

To define the shortened priority sequence, we define the priority sequence first. In Algorithm 6.2, we have an augmented reorder algorithm. When $reorder$ is invoked from $Execute$, the value for the second parameter of the augmented reorder procedure is an empty list. The @ symbol in Algorithm 6.2 means list concatenation. Given the same parameters, the augmented reorder algorithm computes the same reordered sequence as the one in Algorithm 6.1. In the augmented reorder procedure, a priority sequence is computed for each branch instance along with the reorder process. Recall that the reorder process is done in a quick-sort-like fashion. When we divide the input sequence of the reorder procedure using ψ_k as the "pivot", if $b(\psi_i)$ is in the relevant slice of $b(\psi_k)$, then $b(\psi_k)$ is added to the end of the priority sequence of $b(\psi_i)$.

Let t be an input and b_x be a branch instance in path $\pi(t)$. Let the priority number for b_x in $\pi(t)$ be $p(b_x) = [\hat{b}_x^1, \hat{b}_x^2, \ldots, \hat{b}_x^\sigma]$. From this priority sequence, we form a new shortened priority sequence $sp(b_x)$ by selecting only the branches $\hat{b}_x^i$ such that $\hat{b}_x^i$ satisfies: there does not exists any $\hat{b}_x^j$ in $p(b_x)$ such that $\hat{b}_x^i \rightsquigarrow_c \hat{b}_x^j$. We denote the new shortened priority sequence as $sp(b_x) = [b_x^1, \ldots, b_x^\alpha]$. Note that the last branch instance in both $p(b_x)$ and $sp(b_x)$ is always b_x itself. Because of our transformation, if b_k is in $rs(C, \pi(t))$, then the first branch instance in both $p(b_x)$ and $sp(b_x)$ is from the slicing criteria C.

Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$, then one of the following two cases must be true: (i) There is a branch instance b in $p(b_x)$, where b is

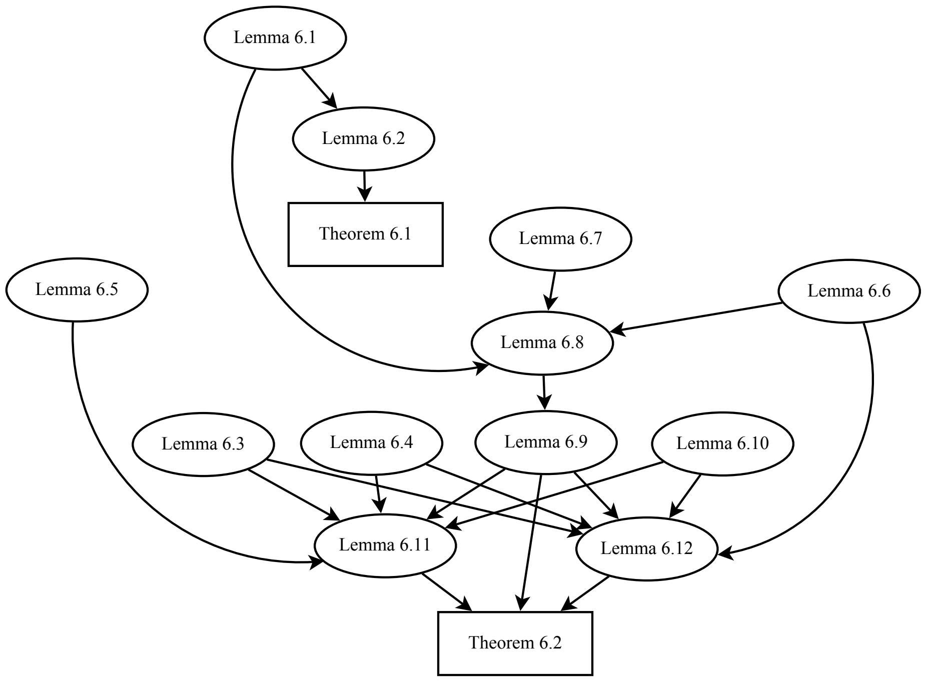


Figure 6.8: Structure of the proofs

after b_y in time order and $b \not\rightsquigarrow b_y$. (ii) $b_y \rightsquigarrow b_x$. In the first case, when b is used as the "pivot" in reorder algorithm, $bc(b_x)$ is reordered before $bc(b_y)$. In the second case, since $b_y \rightsquigarrow b_x$, then b_x should always be before b_y in the entire reorder process.

Suppose $sp(b_x) = [b_x^1, b_x^2, \dots, b_x^\alpha]$ and $sp(b_y) = [b_y^1, b_y^2, \dots, b_y^\beta]$. Let k be the maximal number that satisfies: for each i, $i \leq k$, $b_x^i == b_y^i$. Then we say $sp(b_x) > sp(b_y)$ if either (i) $k == min(\alpha, \beta)$ and $b_y \rightsquigarrow b_x$ or (ii) $k < min(\alpha, \beta)$ and $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$ or (iii) $k < min(\alpha, \beta)$, $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$ and b_x^{k+1} is after b_y^{k+1} in time order. By this definition, it is impossible to have both $sp(b_x) > sp(b_y)$ and $sp(b_y) > sp(b_x)$.

6.2.2.2 Proof structure

We prove two theorems in this chapter about *relevant-slice condition* and our path exploration algorithm based on *relevant-slice condition*. The proof structure is shown in Figure 6.8. For ease of understanding, we give the outline of our proofs and the relations between lemmas and theorems in the following.

In Theorem 6.1, we show that a *relevant-slice condition* could guarantee the unique symbolic values of the variables used in the slicing criteria. Symbolic value can be computed by dynamic symbolic execution. Each symbolic value is an expression in terms of the program inputs. Let s be a statement instance in the path of input t, and v be a variable used in s. The symbolic value of v in s is a expression in terms of input variables. If the symbolic value of v is concretized with t, it must be the same as the value of v in s when the program is run concretely with input t. To prove Theorem 6.1, we actually prove the stronger Lemma 6.2. Let t and t' be two inputs and s be a statement instance in $\pi(t)$. In Lemma 6.2, we show that if $t' \models rsc(s, \pi(t))$, then the relevant slice w.r.t. s in $\pi(t')$ would be exactly the same as that in $\pi(t)$. Theorem 6.1 could be easily derived from Lemma 6.2.

In Theorem 6.2, given any feasible path π, we show that our path exploration algorithm would explore a path π' that shares the same *relevant-slice condition* with π. This is concretized by showing that the algorithm gradually gets a sequence of *relevant-slice conditions* with each one closer to the *relevant-slice condition* of π than the previous one. Recall that the path exploration process is by iteratively negating a branch condition in a *relevant-slice condition*. Suppose we solve φ to get a new input t', we need to prove that the *relevant-slice condition* on $\pi(t')$ still contains φ as a prefix. Otherwise, the path exploration process would be out of order. Although this is obviously true for path condition, it is not for *relevant-slice condition*. According to the result of relevant slice, some branch constraints do not appear in *relevant-slice condition* even they are in path condition. This important property of reordered *relevant-slice condition* is proved in Lemma 6.12. Let t and t' be two inputs. In Lemma 6.12, we prove that if $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reodered\_rsc(C, \pi(t))$, then $reordered\_rsc(C, \pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. Let the target reordered *relevant-slice condition* be $g = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{n-1} \wedge \varphi_n$ and $reodered\_rsc(C, \pi(t))$ be f. If the first different branch condition between f and g is at location k, we prove that $\psi_k == \neg\varphi_k$ in Lemma 6.11. Combining with 6.12, we show that we could indeed get closer to π (having longer common prefix with $reordered\_rsc(C, \pi)$) by negating the k^{th} branch condition in f. All the lemmas from Lemma 6.3 to Lemma 6.10 are used to gradually prove Lemma 6.11 and Lemma 6.12.

6.2.2.3 Full proofs

Lemma 6.1. *Let t and t' be two inputs and s be a statement instance in $\pi(t)$. Suppose s is not in $\pi(t')$. Let b_s be the last branch instance in $\pi(t)$ that satisfies: $s \rightsquigarrow_c b_s$ and b_s is in both $\pi(t)$ and*

$\pi(t')$. Then b_s is evaluated differently in $\pi(t)$ and $\pi(t')$.

Proof. Let the control dependence chain from s to b_s in $\pi(t)$ be $s \rightsquigarrow_c b \rightarrow_c b_s$, where $b \rightarrow_c b_s$ is the last link in $s \rightsquigarrow_c b_s$. Note that b could be same as s. Assume to the contrary that b_s is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Then b must also be executed in $\pi(t')$. Therefore, b also satisfies: $s \rightsquigarrow_c b$ and b is in both $\pi(t)$ and $\pi(t')$. Since b is after b_s in time order, this contradicts that b_s is the last branch instance that satisfy this condition. Therefore, b_s is evaluated to different directions in $\pi(t)$ and $\pi(t')$. □

Lemma 6.2. *Let t and t' be two inputs and s be a statement instance in $\pi(t)$. If $t' \models rsc(s, \pi(t))$, then s will be executed in $\pi(t')$, the variables used in s in $\pi(t')$ will have the same symbolic values as in $\pi(t)$, $rs(s, \pi(t'))$ is exactly the same as $rs(s, \pi(t))$ and each branch instance in $rs(s, \pi(t))$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$.*

Proof. We prove this lemma by induction. Given the path $\pi(t)$, suppose it is a sequence $[s_0, s_1, \ldots, s_{n-1}, s_n]$.

Initial Step: According to our transformation, the statement instance s_0 must be from nop. Then $rsc(s_0, \pi(t))$ is $true$. It is obvious that s_0 satisfies Lemma 6.2.

Inductive Step: The induction hypothesis is: for each statement s_j, $j < i$, s_j satisfies Lemma 6.2. We need to prove that s_i also satisfies Lemma 6.2.

First we prove that s_i will be executed in $\pi(t')$. Let s_j be the statement prior to s_i such that $s_i \rightarrow_c s_j$. Then each statement in $rs(s_j, \pi(t))$ is also in $rs(s_i, \pi(t))$. So we have $rsc(s_i, \pi(t)) \Rightarrow rsc(s_j, \pi(t))$. Since $t' \models rsc(s_i, \pi(t))$, $t' \models rsc(s_j, \pi(t))$. By the induction hypothesis , s_j will be executed to the same direction in $\pi(t)$ and $\pi(t')$. This implies that s_i will be executed in $\pi(t')$.

The core of the inductive step is to prove that $rs(s_i, \pi(t'))$ is exactly the same as $rs(s_i, \pi(t))$. This is proved in two directions. (i) If $s_i \rightsquigarrow s'$ in $\pi(t')$, then $s_i \rightsquigarrow s'$ in $\pi(t)$. (ii) If $s_i \rightsquigarrow s'$ in $\pi(t)$, then $s_i \rightsquigarrow s'$ in $\pi(t')$.

Now, we prove that given any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$. Suppose in $\pi(t')$, $s_i \rightarrow s_k \rightsquigarrow s'$ where s_k is another statement instance in $\pi(t')$. We first prove that $s_i \rightarrow s_k$ in $\pi(t)$ in two steps: (i) s_k appears in $\pi(t)$. (ii) $s_i \rightarrow s_k$ in $\pi(t)$.

We first prove that s_k appears in $\pi(t)$. We prove this by contradiction. Assume to the contrary that s_k does not appear in $\pi(t)$. We find the last control dependence ancestor of s_k that is in both

$\pi(t)$ and $\pi(t')$. Let this statement be s_u. This means that s_u is the last statement in both $\pi(t)$ and $\pi(t')$ such that s_k is transitively control dependent on s_u in both the execution traces $\pi(t), \pi(t')$.

According to Lemma 6.1, the branch in s_u must be evaluated differently in $\pi(t)$ and $\pi(t')$. According to the type of $s_i \rightarrow s_k$ in $\pi(t')$ we have the following cases.

- (a) $s_i \rightarrow_c s_k$. The existence of s_i in $\pi(t)$ contradicts that s_k is not in $\pi(t)$.
- (b) $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$. In this case, the existence of s_i in both paths $\pi(t), \pi(t')$ indicates that $s_i \not\rightarrow_c s_u$ in $\pi(t')$ since s_u is evaluated differently in $\pi(t)$ and $\pi(t')$. Similarly, $s_i \not\rightarrow_c s_u$ in $\pi(t)$. This means that s_i appears after $postdom(s_u)$ in both execution traces $\pi(t), \pi(t')$.

  Suppose $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$ is caused by the use of variable v at s_i (in case of multiple such variables, we choose one randomly). There should be no definition of v between $postdom(s_u)$ and s_i in $\pi(t)$. Otherwise the definition would also appear in $\pi(t')$, making $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ impossible in $\pi(t')$. In $\pi(t)$, according to the definition of $\rightsquigarrow_s$, $s_i \rightsquigarrow_s s_u$ in $\pi(t)$. Therefore, s_u is in the relevant slice of s_i in $\pi(t)$. By the induction hypothesis , s_u is then evaluated to the same direction in $\pi(t)$ and $\pi(t')$, contradicting our original assumption that s_u is evaluated differently in $\pi(t)$ and $\pi(t')$.

Therefore, in both cases, we achieve a contradiction - thereby establishing that s_k must appear in $\pi(t)$.

Given that s_k is in $\pi(t)$, we prove that $s_i \rightarrow s_k$ in $\pi(t)$. According to the type of $s_i \rightarrow s_k$ in $\pi(t')$ we have the following cases. (a) $s_i \rightarrow_c s_k$ in $\pi(t')$. — The existence of s_i and s_k in $\pi(t)$ already shows that $s_i \rightarrow_c s_k$. (b) $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$. — Suppose the dependence between s_i and s_k is caused by the use of variable v (in case of multiple such variables, we choose one randomly) at s_i in $\pi(t')$. Then $s_i \not\rightarrow s_k$ could only happen in $\pi(t)$ because v is redefined by another statement instance between s_k and s_i in $\pi(t)$. Suppose the last definition of v before s_i in $\pi(t)$ is at statement instance s_n. So we have $s_i \rightarrow s_n$ in $\pi(t)$. By the induction hypothesis, s_n will be executed in $\pi(t')$. The variable v will still be redefined by s_n in $\pi(t')$, which contradicts that $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$. Therefore, we have proved that in both case, $s_i \rightarrow s_k$ in $\pi(t)$.

We have now proved that $s_i \rightarrow s_k$ in $\pi(t)$. According to induction hypothesis, the relevant slice of s_k is the same in $\pi(t)$ and $\pi(t')$, that is, $rs(s_k, \pi(t)) == rs(s_k, \pi(t'))$. Thus, for any statement instance s' such that $s_k \rightsquigarrow s'$ in $\pi(t')$ — we must have $s_k \rightsquigarrow s'$ in $\pi(t)$. Therefore, $s_i \rightarrow s_k \rightsquigarrow s'$ in

$\pi(t)$. Thus, we have proved that given any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$.

Next, we prove that given a statement instance s' in $\pi(t)$, if $s_i \rightsquigarrow s'$ in $\pi(t)$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t')$. Suppose $s_i \rightarrow s_j \rightsquigarrow s'$ in $\pi(t)$. According to induction hypothesis, s_j appears in $\pi(t')$ and $s_j \rightsquigarrow s'$ in $\pi(t')$. So we only need to prove that $s_i \rightarrow s_j$ in $\pi(t')$. According to the dependence type of $s_i \rightarrow s_j$, we have the following two cases.

- (a) $s_i \rightarrow_c s_j$ in $\pi(t)$. The existence of both s_i and s_j already implies that $s_i \rightarrow_c s_j$ in $\pi(t')$.

- (b) $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$. We need to prove that the dependence between s_i and s_j still appears in $\pi(t')$. Suppose $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$ is caused by the use of variable v at s_i. We prove $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t')$ by contradiction. Assume to the contrary that this is not the case in $\pi(t')$. This could only happen if v is redefined between s_j and s_i in $\pi(t')$. Suppose the last definition of v before s_i in $\pi(t')$ is statement instance s_n, so $s_i \rightarrow_d s_n$ in $\pi(t')$. We have already established that for any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$. Thus, s_n must also appear in $\pi(t)$, and $s_i \rightarrow_d s_n$ in $\pi(t)$. This contradicts that $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$ (simply by the definition of dynamic data dependencies and potential dependencies). So if $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ is in $\pi(t)$, $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ is also in $\pi(t')$,

Therefore, we have proved by induction that $rs(s_i, \pi(t'))$ is exactly the same as $rs(s_i, \pi(t))$ (inductive step).

Since the entire slice of s_i is exactly the same in two paths, the symbolic values of the variables used in s_i must be exactly the same in $\pi(t)$ and $\pi(t')$. Suppose s_i uses α variables $v_1, v_2, \ldots, v_{\alpha-1}, v_\alpha$ to define variable $\bar{s}_i$. Let the corresponding definition of these variables be at $s^i_1, s^i_2, \ldots, s^i_{\alpha-1}, s^i_\alpha$. Note that each definition s^i_x, $1 \leq x \leq \alpha$, is same in both $\pi(t')$ and $\pi(t)$ since s^i_x is in the relevant slice of s_i. According the induction hypothesis, the symbolic value of each v_x is the same in $\pi(t)$ and $\pi(t')$, where $1 \leq x \leq \alpha$. Moreover, the definition of $\bar{s}_i$ is computed in exactly the same way (using the same operations) from $v_1, v_2, \ldots, v_{\alpha-1}, v_\alpha$ in $\pi(t)$ and $\pi(t')$. Therefore, the symbolic value of $\bar{s}_i$ in s_i is the same in $\pi(t)$ and $\pi(t')$. We know that $t' \models rsc(s_i, \pi(t))$. Therefore, t' should satisfy the branch constraints corresponding to the branches appearing in the relevant slice

$rs(s_i, \pi(t)) == rs(s_i\pi(t'))$. Therefore each branch instance in $rs(s_i, \pi(t))$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. This completes the proof. □

Theorem 6.1. *If the relevant-slice conditions of two paths π_1 and π_2 w.r.t. C are the same, then the variables used in the slicing criteria C have the same symbolic values in π_1 and π_2.*

Proof. Let t_1 and t_2 be two test inputs whose execution traces are π_1 and π_2 respectively. According to the theorem statement, we have $rsc(C, \pi_1) == rsc(C, \pi_2)$. Since $t_2 \models rsc(C, \pi_2)$, $t_2 \models rsc(C, \pi_1)$. According to Lemma 6.2, the symbolic values of the variables used in C are exactly the same in $\pi(t_2)$ and $\pi(t_1)$, where $\pi(t_2) == \pi_2$ and $\pi(t_1) == \pi_1$. Therefore, the variables used in the slicing criteria C have the same symbolic values in π_1 and π_2. This completes the proof. □

Lemma 6.3. *Let t and t' be two inputs, given a branch instance b in $\pi(t)$, if $t' \models rsc(b, \pi(t)) \backslash bc(b)$, b will be executed in $\pi(t')$ and the variables used in b in $\pi(t')$ would have the same symbolic values as in $\pi(t)$.*

Lemma 6.4. *Let t be an input. Let $reordered\_rsc(C, \pi(t))$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ in path $\pi(t)$. Then for any i, $1 \leq i \leq k$, $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \Rightarrow rsc(b(\psi_i), \pi(t)) \backslash \psi_i$.*

Lemma 6.5. *Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$ by the reorder algorithm in Algorithm 6.2, then the shortened priority sequences $sp(b_x) > sp(b_y)$.*

Proof. Suppose $sp(b_x)$ is $[b_x^1, b_x^2, \ldots, b_x^{\alpha-1}, b_x^\alpha]$ and $sp(b_y)$ is $[b_y^1, b_y^2, \ldots, b_y^{\beta-1}, b_y^\beta]$. When $sp(b_x) \neq sp(b_y)$, let k be the maximal number that satisfies: for each i, $i \leq k$, $b_x^i == b_y^i$.

If $k == min(\alpha, \beta)$, it must be either $k == \alpha$ or $k == \beta$. If $k == \alpha$, then we have $b_x == b_x^\alpha == b_x^k == b_y^k$ and $b_y^k \rightsquigarrow b_y$. Therefore, we have $b_x \rightsquigarrow b_y$, which could not be possible when $bc(b_x)$ is reordered before $bc(b_y)$. Therefore, when $k == min(\alpha, \beta)$, it must be $k == \beta$ and $b_y \rightsquigarrow b_x$.

If $k < min(\alpha, \beta)$, we enumerate the relation between b_x^{k+1} and b_y^{k+1}.

1. $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$. This is possible.

2. $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$. We prove that it is not possible to have $bc(b_x)$ being reordered before $bc(b_y)$. This is proved through (i) $b_y^{k+1} \not\rightsquigarrow b_x$ (ii) There is no branch b after b_y^{k+1} such that b is in $p(b_x)$, but $b \not\rightsquigarrow b_y$. We first prove $b_y^{k+1} \not\rightsquigarrow b_x$ by contradiction. Assume to the contrary that

$b_y^{k+1} \rightsquigarrow b_x$. According to the process of computing $p(b_x)$, if $b_y^{k+1} \rightsquigarrow b_x$ and $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$, then b_y^{k+1} is in $p(b_x)$. However, according to the definition of $sp(b_x)$, if $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$ and b_y^{k+1} is in $p(b_x)$, then b_x^{k+1} is not in $sp(b_x)$. This contradicts that b_x^{k+1} is in $sp(b_x)$. Next, we prove that there is no branch b after b_y^{k+1} such that b is in $p(b_x)$, but $b \not\rightsquigarrow b_y$. This is obvious since for each b after b_y^{k+1} and b is in $p(b_x)$, we have $b \rightsquigarrow b_x^{k+1} \rightsquigarrow b_y^{k+1} \rightsquigarrow b_y$, contradicting $b \not\rightsquigarrow b_y$. Therefore, we know that there is no branch b after b_y^{k+1} such that b_y is reordered after b_x by using b as the "pivot". So when $bc(b_y^{k+1})$ is used as the "pivot" in the reorder algorithm, either $bc(b_x)$ is already after $bc(b_y^{k+1})$ hence after $bc(b_y)$, or $bc(b_x)$ is still before $bc(b_y^{k+1})$. If $bc(b_x)$ is still before $bc(b_y^{k+1})$, given the "pivot" $bc(b_y^{k+1})$ and the fact that $b_y^{k+1} \not\rightsquigarrow b_x$, $bc(b_x)$ will be reordered after $bc(b_y^{k+1})$ hence after $bc(b_y)$. This contradicts that $bc(b_x)$ is before $bc(b_y)$.

3. $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$. We prove that b_x^{k+1} is after b_y^{k+1} in time order using contradiction. Assume to the contrary that b_x^{k+1} is before b_y^{k+1} in time order. According to whether b_y^{k+1} is transitively dependent on b_x^{k+1}, we have the following two cases: (i) $b_y^{k+1} \rightsquigarrow b_x^{k+1}$, then there is at least one branch b that is between $postdom(b_x^{k+1})$ and b_x^k such that b is in $sp(b_x)$. this contradicts that the $(k+1)th$ element in $sp(b_x)$ is b_x^{k+1}. (ii) $b_y^{k+1} \not\rightsquigarrow b_x^{k+1}$. We first show $b_y^{k+1} \not\rightsquigarrow b_x$. Assume to the contrary that $b_y^{k+1} \rightsquigarrow b_x$. Then when b_y^{k+1} is used as the "pivot" in the reorder process, b_x is before b_y^{k+1} and b_x^{k+1} is after b_y^{k+1}. Therefore, b_x and b_x^{k+1} are in two different sub-sequences, making it impossible to have b_x^{k+1} in the shortened priority sequence of b_x. This contradicts that b_x^{k+1} is in $sp(b_x)$. Given $b_y^{k+1} \not\rightsquigarrow b_x$, when b_y^{k+1} is used as the "pivot" in the reorder process, either $bc(b_x)$ is already after $bc(b_y^{k+1})$ hence after $bc(b_y)$, or $bc(b_x)$ is still before $bc(b_y^{k+1})$. If $bc(b_x)$ is still before $bc(b_y^{k+1})$, given the "pivot" $bc(b_y^{k+1})$, $bc(b_x)$ will be reordered after $bc(b_y^{k+1})$ hence after $bc(b_y)$. This contradicts that $bc(b_x)$ is before $bc(b_y)$. So it is impossible to have b_x^{k+1} before b_y^{k+1} in either case.

So we have either (i) $k == min(\alpha, \beta)$ and $b_y \rightsquigarrow b_x$ or (ii) $k < min(\alpha, \beta)$ and $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$. or (iii) $k < min(\alpha, \beta)$ and $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$ and b_x^{k+1} is after b_y^{k+1} in time order. This is exactly the definition of $sp(b_x) > sp(b_y)$. □

Lemma 6.6. *Let t be an input and b and b_k be two branch instances in $\pi(t)$. Suppose in $\pi(t)$, $sp(b_k)$ is $[b_k^1, b_k^2, \ldots, b_k^i]$. If $b_k^j \rightsquigarrow b$, where $1 \le j < i$, and $postdom(b)$ is after $postdom(b_k^{j+1})$, then $bc(b)$ is before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$.*

Proof. We first prove that b is not in $p(b_k)$. Based on the possible position of b in $\pi(t)$, we have the following two cases:(i) b is after $postdom(b_k^{j+1})$. Since $b_k^j \rightsquigarrow b$, b could only be between b_k^j and $postdom(b_k^{j+1})$. According to the process of computing the shortened priority sequence, any branch instances between b_k^j and $postdom(b_k^{j+1})$ cannot be in $p(b_k)$. (ii) b is before $postdom(b_k^{j+1})$. Since $postdom(b)$ is after $postdom(b_k^{j+1})$, it must be $b_k^{j+1} \rightsquigarrow_c b$. Therefore, we have $b_k^{j+1} \rightsquigarrow_c b$ and b is in $p(b_k)$. This cannot happen given b_k^{j+1} is in $sp(b_k)$.

According to the reorder algorithm, if $bc(b_k)$ is reordered before $bc(b)$ and b is not in $p(b_k)$, then there must be a branch instance $\hat{b}_k^u$ in $p(b_k)$ such that $\hat{b}_k^u$ is after b and $\hat{b}_k^u \not\rightsquigarrow b$. We will prove that such a $\hat{b}_k^u$ cannot exist. Such a $\hat{b}_k^u$ should be after $postdom(b)$, otherwise $\hat{b}_k^u \rightsquigarrow_c b$. Since $postdom(b)$ is after $postdom(b_k^{j+1})$, $\hat{b}_k^u$ is after $postdom(b_k^{j+1})$. However if $\hat{b}_k^u$ in $p(b_k)$ is after $postdom(b_k^{j+1})$, $\hat{b}_k^u \rightsquigarrow b_k^j \rightsquigarrow b$. So it is not possible to have any $\hat{b}_k^u$ in $p(b_k)$ such that $\hat{b}_k^u$ is after b and $\hat{b}_k^u \not\rightsquigarrow b$. This means that $bc(b_k)$ cannot be reordered before $bc(b)$. Therefore, $bc(b)$ is before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$. $\square$

Lemma 6.7. *Let t and t' be two inputs. Let the reordered relevant-slice condition in $\pi(t)$ be $reordered\_rsc(C, \pi(t)) = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Then if $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$, $k \le i$, for any j, $1 \le j \le k$, $b(\psi_j)$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$.*

Lemma 6.8. *Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ is a prefix of $reordered\_rsc(C, \pi(t))$. Let $b(\psi_k)$ be b_k. Suppose the shortened priority sequence for b_k in $\pi(t)$ is $sp(b_k) = [b_k^1, b_k^2, \ldots, b_k^i]$, where $b_k^i == b_k$. If $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, then (i) For any j, $1 \le j \le i$, b_k^j also appears in $\pi(t')$. (ii) For any j, $1 \le j < i$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t'))$ is also in $rs(b_k^j, \pi(t))$. (iii) For any j, $1 \le j < i$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t))$ is also in $rs(b_k^j, \pi(t'))$.*

Proof. We prove the claims in the lemma one by one.

For any j, $1 \le j \le i$, b_k^j also appears in $\pi(t')$. Suppose $b_k^j \rightarrow_c b$. Since $b_k^j \rightarrow_c b$, we have $b_k^j \rightsquigarrow b$ and $postdom(b_c)$ after b_k^j hence after $postdom(b_k^{j+1})$. According to Lemma 6.6, $bc(b)$

is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. This means that the branch condition of b is actually one of the ψ_m, where $m \leq k-1$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 6.7, b will be evaluated to the same direction in $\pi(t)$ and $\pi(t')$. So b_k^j is also executed in $\pi(t')$.

For any j, $1 \leq j < i$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t'))$ is also in $rs(b_k^j, \pi(t))$. We prove this by contradiction. Assume to the contrary that there is an s in $\pi(t')$, where $s \in rs(b_k^j, \pi(t'))$ and s is between b_k^j and $postdom(b_k^{j+1})$, but s is not in $rs(b_k^j, \pi(t))$. There must exist two nodes $s1$ and $s2$ in $b_k^j \rightsquigarrow s$ in $\pi(t')$ such that $b_k^j \rightsquigarrow s1$ in $\pi(t)$, but $s1 \not\rightarrow s2$ in $\pi(t)$. If it is not the case, then in $\pi(t)$ we have $b_k^j \rightsquigarrow s$. We prove $s1 \rightarrow s2$ to draw the contradiction in two steps: (i) $s2$ appears in $\pi(t)$. (ii) $s1 \rightarrow s2$.

We first prove that $s2$ appears in $\pi(t)$. According to the dependence type from $s1$ to $s2$ in $\pi(t')$, we have the following two cases:(i) $s1 \rightarrow_c s2$. the existence of $s1$ in $\pi(t)$ shows that $s2$ also exists in $\pi(t)$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Assume to the contrary that $s2$ does not appear in $\pi(t)$. We find the last control ancestor $s3$ of $s2$ that is in both $\pi(t)$ and $\pi(t')$. According to Lemma 6.1, $s3$ is evaluated to different directions in $\pi(t)$ and $\pi(t')$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ is caused by the use of variable v at $s1$(in case of multiple such variables, we choose one randomly). There should be no definition of v between $postdom(s3)$ and $s1$ in $\pi(t)$. Otherwise, that definition would be kept in $\pi(t')$, making $s1 \rightarrow s2$ impossible in $\pi(t')$. According to the definition of $\rightsquigarrow_s$, $s1 \rightsquigarrow_s s3$ in $\pi(t)$. Therefore, we have $b_k^j \rightsquigarrow s1 \rightsquigarrow s3$ in $\pi(t)$. Because $s2 \rightsquigarrow_c s3$, $postdom(s3)$ is after $s2$ and hence after $postdom(b_k^{j+1})$ in $\pi(t')$. Since the relative time order of any two statement instances does not change across different paths, the existence of both $s3$ and b_k^{j+1} in $\pi(t)$ indicates that $postdom(s_3)$ is after $postdom(b_k^{j+1})$ in $\pi(t)$. According to Lemma 6.6, $bc(s3)$ is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 6.7, $s3$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. However, this contradicts that $s3$ is evaluated to different directions in $\pi(t)$ and $\pi(t')$.

Then, we show $s1 \rightarrow s2$ in $\pi(t)$. According to the dependence type from $s1$ to $s2$ in $\pi(t')$, we have the following two cases: (i) $s1 \rightarrow_c s2$. The existence of $s1$ and $s2$ already shows that $s1 \rightarrow_c s2$ in $\pi(t)$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Assume to the contrary that $s1 \not\rightarrow s2$ in $\pi(t)$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ is caused by the use of variable v at $s1$(in case of multiple such variables, we just choose one randomly). Therefore, $s1 \not\rightarrow s2$ in $\pi(t)$ could only be v is redefined by some statement instance between $s1$ and $s2$ in $\pi(t)$. We denote this statement instance as $s4$. Suppose $s4$ is control dependent on $s5$ in $\pi(t)$, we have $b_k^j \rightsquigarrow s4 \rightsquigarrow s5$ and $postdom(s5)$ after $s4$ hence after $postdom(b_k^{j+1})$. According to Lemma 6.6, $bc(s5)$ is reordered before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 6.7, $s5$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Therefore, $s4$ will be executed in $\pi(t')$. This contradicts that $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t')$.

For any j, $1 \leq j < i$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t))$ is also in $rs(b_k^j, \pi(t'))$. For a statement s that is in $rs(b_k^j)$ and is between b_k^j and $postdom(b_k^{j+1})$, We first prove that s exists in $\pi(t')$. For any branch instance b_c such that $s \rightsquigarrow_c b_c$, we have $b_k^j \rightsquigarrow s \rightsquigarrow b_c$ and $postdom(b_c)$ after s hence after $postdom(b_k^{j+1})$. According to Lemma 6.6, $bc(b_c)$ is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 6.7, b_c is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Therefore, s will be executed in $\pi(t')$.

Then, we prove that $b_k^j \rightsquigarrow s$ in $\pi(t')$. Assume to the contrary that this is not the case. There must exist two nodes $s1$ and $s2$ in $b_k^j \rightsquigarrow s$ in $\pi(t)$ such that $b_k^j \rightsquigarrow s1$ in $\pi(t)$, but $s1 \not\rightarrow s2$ in $\pi(t')$. If it is not the case, then in $\pi(t')$ we have $b_k^j \rightsquigarrow s$. According the proof in the last paragraph, $s1$ and $s2$ are both executed in $\pi(t')$. According to the dependence type from $s1$ to $s2$ in $\pi(t)$, we have the following two cases:(i) $s1 \rightarrow_c s2$. The existence of $s1$ and $s2$ already shows that $s1 \rightarrow_c s2$ in $\pi(t')$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t)$ is caused by the use of variable v at $s1$(in case of multiple such variables, we just choose one randomly). Therefore, $s1 \not\rightarrow s2$ in $\pi(t')$ could only be v is redefined by some statement instance between $s1$ and $s2$ in $\pi(t')$. We denote this statement instance as $s4$. According to the above proof, $s4$ also exists in $\pi(t)$. Therefore, v should also be redefined by $s4$ in $\pi(t)$, contradicting that $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t)$. □

Lemma 6.9. *Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ is a prefix of $reordered\_rsc(C, \pi(t))$. If $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, then sp$(b(\psi_k))$ is the same in $\pi(t)$ and $\pi(t')$.*

Proof. Let $b(\psi_k)$ be b_k. Suppose the shortened priority sequence for b_k in $\pi(t)$ is $sp(b_k) = [b_k^1, b_k^2, \ldots, b_k^i]$, where $b_k^i == b_k$.

We prove $sp(b(\psi_k))$ is also $[b_k^1, b_k^2, \ldots, b_k^i]$ in $\pi(t')$ in the following steps:(i) We prove that $b_k^j \rightsquigarrow b_k^{j+1}$ for each j, $1 \leq j < i$, in $\pi(t')$. (ii) We prove that each b_k^j is in $p(b_k)$ in $\pi(t')$. (iii) We prove that for any branch instance in $p(b_k)$ in $\pi(t')$, either it is transitively control dependent on some b_k^j, where b_k^j is in $sp(b_k)$ in $\pi(t)$ or it is some b_k^j. (iv) We show that for each b_k^j in $\pi(t')$, if $b_k^j \rightsquigarrow_c b_c$ then b_c is not contained in $p(b_k)$ in $\pi(t')$.

We first show that $b_k^j \rightsquigarrow b_k^{j+1}$ in $\pi(t')$, where $1 \leq j < i$. Since $b_k^j \rightsquigarrow b_k^{j+1}$ and $b_k^j \not\rightsquigarrow_c b_k^{j+1}$ in $\pi(t)$, there must be a statement instance s between $postdom(b_k^{j+1})$ and b_k^j in $\pi(t)$ such that $b_k^j \rightsquigarrow s$ and $s \rightsquigarrow_s b_k^{j+1}$. Note that such an s could be the same as b_k^j. Suppose $s \rightsquigarrow_s b_k^{j+1}$ is caused by the use of variable v at s(in case of multiple such variables, we choose one randomly). As proved in Lemma 6.8, between $postdom(b_k^{j+1})$ and b_k^j, $rs(b_k^j, \pi(t'))$ is exactly the same as $rs(b_k^j, \pi(t))$. Therefore, $b_k^j \rightsquigarrow s$ in $\pi(t')$ and there is not definition of v between $postdom(b_k^{j+1})$ and s in $\pi(t')$. According to the definition of $\rightsquigarrow_s$, $s \rightsquigarrow_s b_k^{j+1}$ is irrespective of the direction of b_k^{j+1}. So in $\pi(t')$, we also have $s \rightsquigarrow_s b_k^{j+1}$. So we have $b_k^j \rightsquigarrow b_k^{j+1}$ in $\pi(t')$.

Next, we prove that each b_k^j is in $p(b_k)$ in $\pi(t')$, where $1 \leq j \leq k$. Assume to the contrary that this is not the case. Since ψ_k is in $reorderd\_rsc(C, \pi(t))$, $b(\psi_k)$(same as b_k) is in $rs(C, \pi(t))$. Then the first element in $sp(b_k)$ must be from C. According to the proof in last paragraph, $b_k^1 \rightsquigarrow b_k$ in $\pi(t')$. Therefore, the first element of $p(b_k)$ in $\pi(t')$ would still be from C, which is b_k^1. So b_k^1 is in $p(b_k)$ in $\pi(t')$. For $j > 1$, suppose b_k^j is the first one in $sp(b_k)$ in $\pi(t)$ that is not in $p(b_k)$ in $\pi(t')$. Therefore, we have b_k^{j-1} is in $p(b_k)$ in $\pi(t')$. According to the process of computing $p(b_k)$, there must be some "pivot" b (including b_k^j) between b_k^{j-1} and b_k^j, where $b_k^{j-1} \rightsquigarrow b$ and $b \not\rightsquigarrow b_k^j$ and $b \rightsquigarrow b_k$ in $\pi(t')$. According to the proof in last paragraph, we have $b_k^{j-1} \rightsquigarrow b_k^j$ and $b_k^j \rightsquigarrow b_k$ in $\pi(t')$. Therefore, b could not be the same as b_k^j, meaning b could only be after b_k^j and before b_k^{j-1}. According to the possible locations of b, we have the following two cases: (i) b is between b_k^j and $postdom(b_k^j)$. If b is between b_k^j and $postdom(b_k^j)$, then $b \rightsquigarrow_c b_k^j$, contradicting that $b \not\rightsquigarrow b_k^j$. (ii) b is between $postdom(b_k^j)$ and b_k^{j-1}. As shown in Lemma 6.8, $rs(b_k^{j-1}, \pi(t'))$ is exactly the same as $rs(b_k^{j-1}, \pi(t))$ between b_k^{j-1} and b_k^j. Since we have $b \not\rightsquigarrow b_k^j$ in $\pi(t)$, $b \not\rightsquigarrow b_k^j$ in $\pi(t')$ either. This contradicts that $b \rightsquigarrow b_k^j$ in $\pi(t')$. In each case, we get a contradiction showing that the assumption is wrong. So we have b_k^j is in $p(b_k)$ in $\pi(t')$.

Then, we prove that for any given branch instance in $p(b_k)$ in $\pi(t')$, either this branch instance is transitively control dependent on some b_k^j, where b_k^j is in $sp(b_k)$ in $\pi(t)$, or it is some b_k^j. This is the same as: for any branch b, if b is not between any pair of b_k^j and $postdom(b_k^j)$, $1 < j \leq i$, then b cannot be in $p(b_k)$ in $\pi(t')$. Note that $j > 1$ is because the range between b_k^1 and $postdom(b_k^1)$ is after the slicing criteria C(same as b_k^1) in time order. Assume to the contrary that such a b exists, then b must be between some b_k^j and $postdom(b_k^{j+1})$. According to Lemma 6.8, between $postdom(b_k^{j+1})$ and b_k^j, $rs(b_k^j, \pi(t'))$ is exactly the same as $rs(b_k^j, \pi(t))$, then such b is also in $\pi(t)$. According to the process of computing $p(b_k)$, b is contained in $p(b_k)$ in $\pi(t)$, contradicting that $p(b_k)$ does not contain any branches that are between $postdom(b_k^{j+1})$ and b_k^j. Therefore, we have proved that such b could not exist.

Finally, we show that for each b_k^j in $\pi(t')$, if $b_k^j \rightsquigarrow_c b_c$ then b_c is not contained in $p(b_k)$ in $\pi(t')$. Assume to the contrary that there exists a b_c, $b_k^j \rightsquigarrow_c b_c$ and b_c is contained in $p(b_k)$ in $\pi(t')$. According to proof in the last paragraph b_c must be either transitively dependent on some b_k^i or b_c is the same as b_k^i. In either case, we have $b_k^j \rightsquigarrow_c b_k^i$ in $\pi(t')$. Recall that control dependence between two statement instances are preserved across paths as long as the two statement instances both exist. Since b_k^j and b_k^i are also in $\pi(t)$, we have $b_k^j \rightsquigarrow_c b_k^i$ in $\pi(t)$. Therefore b_k^j can not be in $sp(b_k)$ in $\pi(t)$, contradicting that b_k^j is in $sp(b_k)$ in $\pi(t)$.

According to the process of computing shortened priority sequence, $sp(b_k)$ in $\pi(t')$ would be $[b_k^1, b_k^2, \ldots, b_k^i]$. □

Lemma 6.10. *Let t be an input and b_x be a branch instance in $rs(C, \pi(t))$. If the shortened priority sequence of b_x in $\pi(t)$ is $sp(b_x) = [b_x^1, \ldots, b_x^\alpha]$, then for any i, $1 \leq i < \alpha$, $b_x^i \rightsquigarrow b_x^{i+1}$. This essentially means that there is a dependence chain from slicing criteria to b_x, which means b_x will be included in $rs(C, \pi(t))$.*

Lemma 6.11. *Let π_1 and π_2 be two paths. Let f and g be $reordered\_rsc(C, \pi_1)$ $reordered\_rsc(C, \pi_2)$ respectively. Suppose f is $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and g is $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. If the first different branch condition between f and g is at location k, then $\varphi_k == \neg\psi_k$.*

Proof. We first show that $b(\varphi_k)$ and $b(\psi_k)$ must be the same. We prove this by contradiction. Assume to the contrary that $b(\varphi_k)$ and $b(\psi_k)$ are different. Let $b(\varphi_k)$ be b_x and $b(\psi_k)$ be b_y. Since the first different branch condition between f and g is at location k, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$ (same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$) is satisfied by the input of both paths. , Since the input of π_1 satisfy $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$, b_y is contained in $rs(C, \pi_1)$ according to Lemma 6.10. The branch condition $bc(b_y)$ should not be in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, which is the same as $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$. So $bc(b_y)$ could only be after $bc(b_x)$(same as φ_k) in $reordered\_rsc(C, \pi_1)$. Similarly, $bc(b_x)$ could only be after $bc(b_y)$ in $reordered\_rsc(C, \pi_2)$. According to Lemma 6.5, we have $sp(b_x) > sp(b_y)$ from π_1. Similarly we have $sp(b_y) > sp(b_x)$ from π_2. This contradicts that the shortened priority sequences are the same in both paths by Lemma 6.9. So $b(\varphi_k)$ and $b(\psi_k)$ must be the same.

According to Lemma 6.3 and 6.4, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$ can guarantee that the symbolic values of the variables used at $b(\varphi_k)$(same as $b(\psi_k)$) are the same in π_1 and π_2. So φ_k could only be different from ψ_k if the branch $b(\varphi_k)$ and $b(\psi_k)$ are evaluated to different directions in π_1 and π_2. So we have $\varphi_k == \neg\psi_k$. □

Lemma 6.12. *Let t and t' be two inputs. If $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reodered\_rsc(C, \pi(t))$, then $reordered\_rsc(C, \pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix.*

Proof. We will prove the following properties of $\pi(t')$:

- Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is executed in $\pi(t')$ and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$.
- Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is contained in $rs(\pi(t'))$.
- The order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ in $\pi(t')$.
- The first i branch conditions of $reodered\_rsc(C, \pi(t))$ must be from the branch instances $\{b(\psi_k) | 1 \leq k \leq i\}$.

Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is executed in $\pi(t')$ and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$. Since $k \leq i$, so $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$. According to Lemma 6.3 and 6.4, each $b(\psi_k)$ in $\pi(t)$ is executed and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$.

Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is contained in $rs(\pi(t'))$. Since $k \leq i$, so $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$. According to Lemma 6.10, each $b(\psi_k)$ in $\pi(t)$ is contained in $rs(\pi(t'))$.

The order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ in $\pi(t')$. Let ψ_j and ψ_k be any two branch conditions in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $1 \leq j < k \leq i$. According to Lemma 6.6, if ψ_j is before ψ_k, then $sp(b(\psi_j)) > sp(b(\psi_k))$. According to Lemma

6.9, the priority sequence of $b(\psi_j)$ and $b(\psi_k)$ in $\pi(t')$ are the same as those in $\pi(t)$ respectively. Therefore in $\pi(t')$, we also have $sp(b(\psi_j)) > sp(b(\psi_k))$. This shows that the relative order of any two branch conditions in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ are the same $\pi(t)$ and $\pi(t')$. Therefore, the order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. in $\pi(t')$.

The direction of each branch instance is restricted by the corresponding branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$.

The first i branch conditions of $reodered\_rsc(C, \pi(t))$ must be from the branch instances $\{b(\psi_k)|1 \leq k \leq i\}$. Assume to the contrary that this is not the case. Let φ be the first branch condition whose branch instance is not in $\{b(\psi_k)|1 \leq k \leq i\}$ and φ is one of the first i branch conditions in $reordered\_rsc(C, \pi(t'))$. According to the above proof, all the branch conditions before φ are satisfied by t. Therefore, according to Lemma 6.9, $b(\varphi)$ appears in $\pi(t)$. Since $sp(\varphi) > sp(\psi_i)$, this contradicts that $b(\varphi)$ does not appear in $\{b(\psi_k)|1 \leq k \leq i\}$.

According to the above proved properties, $reordered\_rsc(C, \pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. $\square$

We now prove the completeness of our path search method.

Theorem 6.2. *Given a program P and an execution trace $\pi(t)$ for input t in P, Algorithm 6.1 must explore an execution trace $\pi(t')$ for some input t' such that $\pi(t)$ and $\pi(t')$ share the same relevant-slice condition (irrespective of the initial test input with which Algorithm 6.1 is started) — provided the total number of relevant-slice conditions in P is bounded.*

Proof. Consider *any* input t in program P, its execution trace $\pi(t)$ and the associated reordered *relevant-slice condition* g. We use $dist(f, g)$ to denote the distance from f to g where f is also a reordered *relevant-slice condition* of some path. Suppose $f = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Let k be the number such that (i) for all $m \leq k$ we have $\varphi_m == \psi_m$, and (ii) either $k == min(i, j)$ or $\varphi_{k+1} \neq \psi_{k+1}$.

We first show that when $k == min(i, j)$, it must be that $f == g$. Without losing generality, let us assume to the contrary that $f \neq g$ and $k == j$, which means that $i > j$. If an input t_f satisfies f, then $t_f \models \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$, which is the same as $t_f \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{j-1} \wedge \psi_j$. According to the proof of Lemma 6.9, $b(\psi_{j+1})$ must appear in the trace $\pi(t_f)$, which contradicts that the first k branch conditions in f and g are the same and the length of f is only k.

We now define the distance on reordered *relevant-slice conditions*. Given two reordered *relevant-slice conditions* f and g, we define $dist(f, g) \equiv 1 - \frac{k}{i}$. When $dist(f, g) == 0$, f and g are the same. The definition of $dist$ is asymmetric, that is, $dist(f, g) \neq dist(g, f)$ is possible.

In Algorithm 6.1, we maintain a $f_{current}$ which has the closest distance to g among all the explored *relevant-slice conditions*. Suppose $f_{current} = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Suppose the first different branch condition between $f_{current}$ and g is at location $k+1$. When $f_{current}$ is explored, the partial *relevant-slice condition* $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ is pushed into the stack. This formula will be eventually processed by our path search algorithm, provided the total number of *relevant-slice conditions* is bounded in program P.

According to Lemma 6.11, $\neg\varphi_{k+1} == \psi_{k+1}$. It is clear that $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$. Note that $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_i$ is satisfiable, as g is the *relevant-slice condition* of a feasible path $\pi(t)$. Since $k < i$ ($f_{current}$ and g are same up to the first k conjuncts), $g \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$. Since g is satisfiable, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ (same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$) is also satisfiable. Let t_0 be an input which satisfies $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$, that is $t_0 \models \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$. Using Lemma 6.12 we get that $reordered\_rsc(C, \pi(t_0))$ contains $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ (which is same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$) as a prefix. By the definition of distance $dist$, the distance from $reordered\_rsc(C, \pi(t_0))$ to g should be

$$dist(reordered\_rsc(C, \pi(t_0)), g) \leq 1 - \frac{k+1}{i} < 1 - \frac{k}{i}$$

Replacing $f_{current}$ with $reordered\_rsc(C, \pi(t_0))$ will therefore decrease $dist(f_{current}, g)$. Thus, from $f_{current}$ our path search algorithm moves to the execution trace for input t_0 in one step. Since g contains only i conjuncts, we need at most i such steps to make $dist(f_{current}, g)$ to be 0. When $dist(f_{current}, g) == 0$, we have a path $\pi(t')$ that has the same reordered *relevant-slice condition* with g (such a t' can be found since in each step of replacing $f_{current}$ we obtain a feasible execution trace which is executed by at least one program input). Since the reordered *relevant-slice conditions* of $\pi(t)$ and $\pi(t')$ are identical, therefore the *relevant-slice conditions* of $\pi(t)$ and $\pi(t')$ must be identical. $\square$

6.3 Implementation

In this section, we discuss our combined infra-structure for symbolic execution and dependency analysis of Java programs.

Our implementation is based on JSlice[112][1]. JSlice is an open-source dynamic slicing tool working on Java bytecode. We have extended JSlice to compute *relevant-slice conditions*. The architecture of our extended JSlice is shown in Figure 6.9. The "Operand Stack" in Figure 6.9 stands for the stack of operands in a method activation frame. The recorded execution trace of JSlice does not contain operand stack information. This is a design choice for keeping the trace compact. During slicing, we need to recover the execution stack from a recorded trace to derive operation on stack variables. For example, given an `iload` instruction (loading integer to the top of current stack), we need the operand stack to know which stack variable is modified. Different from stack variables, operations of heap variables are directly recorded in execution trace.

JSlice keeps the collected trace in a compressed form to achieve scalability. The compression is online — as the trace is generated it is simultaneously compressed. The slicing algorithm then works directly on the compressed trace. We design our extension of JSlice to retain this feature (of analyzing compressed traces without decompression).

In Figure 6.9, relevant slicing and symbolic execution are separated for ease of understanding. However, we do not need the entire relevant slicing result to start computing *relevant-slice condition* in the implementation. The process of constructing the *relevant-slice condition* is done along with the backward relevant slicing to achieve efficiency. Since the relevant slicing process is backward, we also compute the relevant slice condition via a backward symbolic execution which starts from the slicing criteria and stops at the beginning of the trace.

For backward symbolic execution, we keep a set of symbolic values whose definitions have not been encountered and need to be explained later in the backward symbolic execution process. The symbolic value of a variable v is explained by either an assignment to v or by program input to v. Let us take the sample program in Figure 6.2 to show our backward symbolic execution on a relevant slice. Note that although we show this example at the source code level, our implementation is at the Java bytecode level. Suppose the input is $\langle x == 6, y == 5, z == 2\rangle$. The relevant slice for the

[1]`http://jslice.sourceforge.net/`

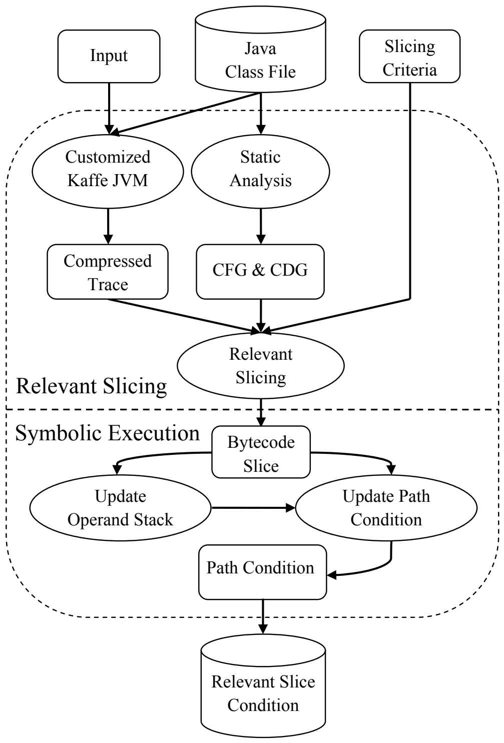


Figure 6.9: Architecture of *relevant-slice condition* computation

Table 6.3: Backward symbolic execution example

Relevant slice	Symbolic values	To be explained variables	Relevant slice condition
`16 return out;`	$\{\}$	$\{ out \}$	true
`15 out = b;`	$\{ out \rightarrow b \}$	$\{ b \}$	true
`10 b = a;`	$\{out \rightarrow a, b \rightarrow a \}$	$\{ a \}$	true
`9 if(x+y > 10)`	$\{out \rightarrow a, b \rightarrow a \}$	$\{a, x, y \}$	$x + y > 10$
`6 a = x;`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x + y > 10$
`5 if(x-y >0)`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x - y > 0 \wedge x + y > 10$

execution trace of this input is $[5, 6, 9, 10, 15, 16]$. Backward symbolic execution along this relevant slice is shown in Table 6.3. The set of to-be-explained variables are shown in the third column of Table 6.3.

To construct the *relevant-slice conditions*, we need to precisely represent the semantics of each bytecode type in the generated formulae. There are more than 200 different bytecode types in the Java Virtual Machine instruction set, and all of them are handled in our implementation. Our implementation also handles native method calls. However, due to the JSlice version that our implementation is based on, currently we cannot handle programs with multi-threading and reflection.

In the original implementation of JSlice, the concrete operand values of most executed instructions are not stored in the compressed trace as they are not needed in the slicing process. However, these values are needed when the semantics of some operations cannot be precisely modelled. In such cases, we have to under-approximate the generated path condition/*relevant-slice condition* by concretizing certain symbolic values in the *relevant-slice condition*. For example, Java allows a program to use libraries written in other languages through native method call. Since the native calls cannot be traced in Java Virtual Machine, the symbolic return values from native calls cannot be precisely modelled. In this case, we simply concretize the symbolic return value from a native call using the concrete return value of the native call (therefore, the concrete return value of native calls are traced in our implementation).

As mentioned in Section 6.2, we need to reorder the branch conditions in a *relevant-slice condition* in our path exploration process. Let $rs(C, \pi)$ be the relevant slice on trace π w.r.t. the slicing criteria C. Let $rsc(C, \pi)$ be the *relevant-slice condition* computed on $rs(C, \pi)$. To reorder the branch conditions in $rsc(C, \pi)$ using the *reorder* procedure shown in Algorithm 6.1, we need to compute a relevant slice using each branch instance in $rs(C, \pi)$ as the slicing criteria. Suppose there are m branch instances in $rs(C, \pi)$, our implementation traverses the trace π for m times to compute the m relevant slices. In future, we plan to speed up this process, by computing all m relevant slices at the same time of computing $rs(C, \pi)$. We also observe that there are a lot similarities among the slices w.r.t. different branch instances (used as slicing criteria) in the same trace. For example, if a branch instance b_i is in the relevant slice of branch instance b_j, then the relevant slice w.r.t. b_i is a subset of the relevant slice w.r.t. b_j. In future, we could exploit the similarities among these slices to further reduce the cost of our *reorder* procedure.

Our execution engine is a combined infra-structure for dynamic dependency analysis and dynamic symbolic execution. Thus, apart from computing *relevant-slice conditions*, we can simply disable the dependency analysis in our engine to compute path conditions. The path conditions and *relevant-slice conditions* generated from our tool are in the SMT-LIB format[2], which can be solved by various Satisfiability Modulo Theory or SMT solvers. In our implementation, we choose Z3 [41][3] as the SMT solver for our tool.

[2]`http://www.smt-lib.org/`

[3]`http://research.microsoft.com/en-us/um/redmond/projects/z3/`

6.4 Experiments

In the following, we first compare our *relevant-slice condition* based path exploration method with *path condition* based path exploration. We then present two applications of *relevant-slice conditions* in: i) the debugging of evolving programs and ii) test-suite augmentation.

Algorithm 6.3 PCExplore:path exploration using path condition

```
1:  **Input:**
2:  $P$ : The program to test
3:  $t$ : An initial test case for $P$
4:  **Output:**
5:  $T$: A test-suite for $P$
6:
7:  $Stack = null$ // The stack of partial PC to be explored
8:  $Execute(t, 0)$
9:  while $Stack$ is not empty do
10:     let $\langle f, j \rangle = pop(Stack)$
11:     if $f$ is satisfiable then
12:         let $\mu$ be one input that satisfies $f$
13:         put $\mu$ into $T$
14:         $Execute(\mu, j)$
15:     end if
16: end while
17: return $T$
18:
19: procedure $Execute(t, n)$
20:     execute $t$ in $P$ and compute path condition $pc$
21:     let $pc = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{m-1} \wedge \psi_m$
22:     for all i from n+1 to m do
23:         let $h = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i)$
24:         push $\langle h, i \rangle$ into $Stack$
25:     end for
26:     return
27: end procedure
```

6.4.1 Path exploration

We compare our path exploration algorithm RSCExplore in Algorithm 6.1 with the PCExplore shown in Algorithm 6.3. The PCExplore algorithm closely resembles our RSCExplore algorithm in Algorithm 6.1. The main difference is that PCExplore uses path condition instead of *relevant-slice condition*. Because of using path condition, neither slicing nor reordering takes place in PCExplore.

The subject programs shown in Table 6.4 are from SIR [43] repository. Tcas and Schedule are originally written in C language, we manually translate them into Java language. For Tcas and Schedule, the slicing criteria are set as the final program outputs. For the other three data-structure

Table 6.4: Experiments in full program exploration

Subject prog.	Size (LOC)	RSC coverage	Time		#Testcases		Avg. formula size		#Solver calls	
			RSC	PC	RSC	PC	RSC	PC	RSC	PC
Tcas	113	100%	6.3s	13.1s	29	88	5744	64810	412	939
BinarySearchTree	175	75%	6.1s	58.6s	64	453	3836	49266	163	3188
OrdSet	211	79%	2.1s	7.4s	12	59	6444	55461	96	293
Schedule	257	100%	0.3s	15.4s	3	75	1808	13728	13	932
DisjointSet	102	100%	20.8s	64.8s	69	278	7643	170533	1192	3855

```
1  int foo(int x, int y){ //input
2    int out; //output
3    int a[2] = {0,1};
4    if(x > 0)
5      System.out.println("x is greater than zero");
6    if(a[x]>0){
7      if(y > 0)
8        out = 1;
9     else
10       out = -1;
11   }else{
12     out = 0;
13   }
14   return out;//slicing criteria
15 }
```

Figure 6.10: Example of imprecise array modelling

programs, the slicing criteria are set as the outputs of the test drivers[4]. The lines of code (LOC) in each program are also shown in Table 6.4.

The completeness of exploration is difficult to achieve in practice for several reasons. Two of the main reasons are (i) the limited power of current SMT solvers and (ii) imprecise modeling of program semantics for symbolic execution. Because of these two reasons, our technique may miss a certain *relevant-slice condition* rsc_i when PCExplore can explore a path whose *relevant-slice condition* is rsc_i. More discussion of the incompleteness of SMT solvers is provided in Section 6.5. In the next paragraph, we explain how imprecise modeling of array can cause our implementation to be not as complete as PCExplore in terms of *relevant-slice condition* coverage.

Consider the example program in Figure 6.10. The branch in line 6 uses `a[x]` in its branch condition. In our current implementation, we concretize symbolic array index using the value observed at execution time. Suppose the initial input for the program in Figure 6.10 in both our

[4]These test drivers are written by us.

method RSCExplore and PCExplore is $\langle x == 0, y == 0\rangle$. When computing the branch condition of line 6 for input $\langle x == 0, y == 0\rangle$, we concretize the symbolic value of x using 0, which is the value of x in line 6 when executing the program with input $\langle x == 0, y == 0\rangle$. After concretization, the branch condition of line 6 is $\neg(a[0] > 0)$, which is reduced to $true$. Thus, due to the concretization of symbolic array index, the branch at line 6 cannot contribute a branch condition to either path condition or *relevant-slice condition*. The path condition and *relevant-slice condition* for $\langle x == 0, y == 0\rangle$ are $\neg(x > 0)$ and $true$ respectively. Since the *relevant-slice condition* for the initial input $\langle x == 0, y == 0\rangle$ is $true$ (containing no branch condition), our technique terminates. However, some *relevant-slice conditions* are missed by our technique. In particular, the *relevant-slice conditions* of paths that evaluate branch at line 6 to false are missed. In contrast, PCExplore could explore all feasible paths (hence all *relevant-slice conditions*) of the program in Figure 6.10. If arrays are modelled precisely, this problem will disappear.

The "RSC coverage" column in Table 6.4 measures how much incompleteness in *relevant-slice condition* coverage is introduced by the imprecise modelling of program semantics in our implementation. The numbers in the "RSC coverage" column are computed as follows. Let the program being explored be P. We employ PCExplore on P to explore program paths and construct a test-suite $T_{\text{PCExplore}}$ which covers the set of all paths in P covered by PCExplore. For each test case t in $T_{\text{PCExplore}}$, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set $S_{\text{PCExplore}}$. Similarly, we generate a test-suite $T_{\text{RSCExplore}}$ for program P using our path exploration method RSCExplore. For each test case t in $T_{\text{RSCExplore}}$, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set $S_{\text{RSCExplore}}$. Then the "RSC coverage" column in Table 6.4 is $\frac{|S_{\text{RSCExplore}}|}{|S_{\text{PCExplore}}|}$. As shown in Table 6.4, our method cannot always achieve 100 percent relevant slice coverage as compared to PCExplore due to the imprecise modelling of program semantics in our implementation.

In columns 4-11 of Table 6.4, we compare the time, number of generated test cases, formula size and number of solver calls between our method RSCExplore and PCExplore. The formula size is measured by the number of bytes in the SMT-LIB formula file. For getting these numbers, both our method (RSC) and the PCExplore method are *run to completion*, and the running time is recorded. Note that the time reported in Table 6.4 includes the time taken in every steps of our method and PCExplore. For example, the time taken by our method includes the time for program execution, relevant slicing, *relevant-slice condition* computation, branch condition reordering, formula solving, etc. As shown in Table 6.4, our technique takes much less time than PCExplore. The efficiency comes from several sources. First, since we use *relevant-slice condition* instead of path condition, the formula size of our approach is much smaller than that of PCExplore. This reduces the time taken by the solver. Second, the number of different *relevant-slice conditions* is considerably smaller than the number of path conditions. This reduces both the number of executions and the number of solver calls.

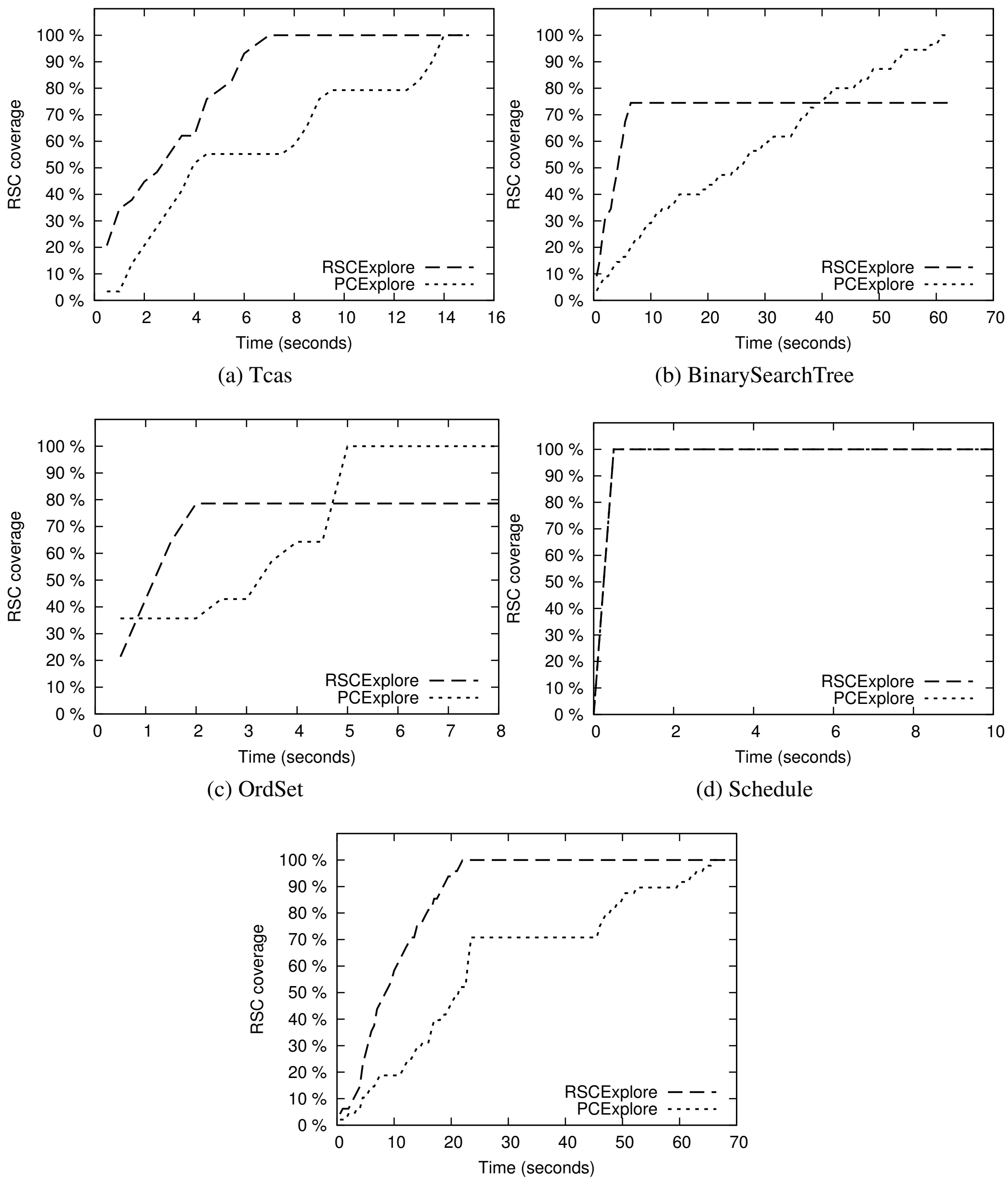
Figure 6.11 compares the *relevant-slice condition* coverage of our Algorithm 6.1 with PCExplore *under the same time limit*. Note that PCExplore intends to achieve path coverage. However, as we have observed - several paths may have the same input-output relationship, and testing is always done by checking outputs. We check the number of *relevant-slice conditions* that are covered by the paths explored in PCExplore. As shown in Figure 6.11, our technique gets higher *relevant-slice condition* coverage then PCExplore when the given time is short.

6.4.2 Debugging of evolving programs

The obvious application of *relevant-slice conditions* is in software testing - it groups program paths and can be used to efficiently generate a concise test-suite. We now show another application of *relevant-slice conditions* namely in our DARWIN debugging technique as presented in Chapter V.

Recall that DARWIN compares the path conditions pc and pc' to reason about the root causes of regression errors. We observe that the path conditions pc and pc' in the above method can be replaced by *relevant-slice conditions*. Path condition is not “goal-directed” — it contains the constraints of branches which are not “related” to the observable error. In particular, a path condition will typically contain constraints for branches which are not in the dynamic or relevant slice of the observable error.

Thus, due to the inherent parallelism in sequential programs, path conditions contain constraints for branches which are not in the slice of the observed error. Composing these path conditions for debugging then allows for such “unrelated” branches to be incorporated into the bug report (which is output by the debugging method). Indeed including these “unrelated” branch constraints increases the burden on the SMT solvers invoked by the DARWIN method, both in terms of the size of the



(a) Tcas (b) BinarySearchTree

(c) OrdSet (d) Schedule

(e) DisjointSet

Figure 6.11: Relevant-slice condition coverage comparison

Table 6.5: DARWIN debugging results (LOC stands for Lines of Code)

Subject prog.	Stable version	Buggy version	Diff	Time		Debugging results	
				PC	RSC	PC	RSC
JLex	1.2.1 (7290 LOC)	1.1.1 (6984 LOC)	518 LOC	543 min	15 min	50 LOC	3 LOC
JTopas	0.8 (4514 LOC)	0.7 (5754 LOC)	2489 LOC	81 min	5 min	4 LOC	4 LOC
NanoXML	2.1(4947 LOC)	2.2 (5244 LOC)	2496 LOC	2m56s	43s	8 LOC	6 LOC

formulae and the number of the formulae to solve. In addition, these "unrelated" branch constraints also introduce some false positives into the bug report produced by the DARWIN method.

Replacing path condition with *relevant-slice condition* in the DARWIN method resolves these issues. Thus, given a test case t that passes in the old version program P but fails in the new version program P' — we now compute rsc and rsc', the *relevant-slice conditions* of t in P and P' respectively. We then solve $rsc \wedge \neg rsc'$ in a manner similar to the solving of $pc \wedge \neg pc'$ in DARWIN (where pc, pc' were the path conditions of t in programs P, P').

We compare the debugging result of DARWIN using *relevant-slice conditions* with the original DARWIN method (which uses path conditions) in Table 6.5.

Both methods are *fully* automated. We did not use the same SIR programs as used in Section 6.4.1 because debugging regression errors for SIR programs is usually trivial. This is because the difference between two SIR program versions is usually small. The first subject program being used is `JLex`[5]. `JLex` is a lexical analyzer generator written in Java. We use version 1.2.1 of `JLex` as the stable version, and version 1.1.1 as the buggy version. There are 6984 and 7290 lines of code in version 1.1.1 and version 1.2.2 respectively. The changes across version 1.1.1 and version 1.2.1 consist of 518 lines of code. In particular, the version 1.1.1 of `JLex` cannot recognize '\r' as the newline symbol, while in version 1.2.1 this bug is fixed. We use an input file manifesting this bug.

The experimental results from DARWIN using *relevant-slice conditions* vs. the original DARWIN method appears in Table 6.5. The original DARWIN method, which uses path conditions, takes 543 minutes (or 9 hours) to perform the debugging. The result of DARWIN is a bug report containing 50 lines of code, which are highlighted to the programmer as potential root-causes of the observable error. In contrast, DARWIN using *relevant-slice condition* takes only 15 minutes. The result is a bug report containing only 3 lines of code — potential root causes of the observed error. Indeed,

[5]`http://www.cs.princeton.edu/~appel/modern/java/JLex/`

the actual error root-cause lies in one of these three lines of code. Thus, by using *relevant-slice conditions* inside our DARWIN debugging method - we could avoid 47 false positives among the potential error causes which are reported to the programmer. Moreover, there is a huge savings in the debugging time (15 minutes vs 9 hours) which comes from the *relevant-slice conditions* being much smaller than path conditions.

We also conducted experiments using `JTopas` [6] as the subject program. `JTopas` is a Java library for parsing arbitrary text data. We use version 0.8 of `JTopas` as the stable version, and version 0.7 as the buggy version. There are 5754 and 4514 lines of code in version 0.7 and version 0.8 respectively. `JTopas` allows users to customize whitespace characters (i.e. characters that are considered as whitespace characters) by using function *setWhitespaces*. `JTopas` also uses a boolean field *_defaultWhitespaces* to control whether the default whitespace characters are used or the user-customized whitespace characters are used. To use the customized whitespace characters, *_defaultWhitespaces* has to be set to *false*. Unfortunately, the buggy `JTopas-0.7` does not reset the member *_defaultWhitespaces* leading to the default whitespace characters still being used instead of the customized ones although the user has specified the custom whitespace characters. In our experiment, we customize whitespace characters to {‘ ’, ‘\r’,‘\t’} ({‘ ’, ‘\n’, ‘\r’,‘\t’} by default) and use an input file manifesting the aforementioned bug. The debugging results of DARWIN using path condition and DARWIN using *relevant-slice condition* are shown in Table 6.5. The results from the original DARWIN method (using path condition) and DARWIN using *relevant-slice condition* are both four lines of code. They both contain the location where ‘\n’ is treated differently between the two versions. The pinpointed location shows that the stable version does not consider ‘\n’ as a whitespace. In contrast, the buggy version still treats ‘\n’ as a whitespace because *_defaultWhitespaces* is *true* (even though whitespace characters have already been customized). From this clue, the programmer could easily infer that the member *_defaultWhitespaces* was not assigned to the correct value. Although using *relevant-slice condition* does not eliminate any false positives in the debugging result, it does reduce the time taken by DARWIN from 81 minutes to 5 minutes.

Lastly, we applied DARWIN technique on a regression bug in NanoXML [7]. NanoXML is a simple XML file parser. A regression bug happened when NanoXML was changed from version 2.1

[6] `http://jtopas.sourceforge.net/jtopas/index.html`
[7] `http://devkix.com/nanoxml.php`

```
public int getProperty(String key, int defaultValue)
{
  String val=(String)attributes.get(key);
  if (val == null) {
    return defaultValue;
  } else {
    try {
      return parseInt(val);
    } catch (NumberFormatException e) {
       throw invalidValue(key, val);
    }
  }
}
```

(a) getProperty in NanoXML-2.1

```
public int getProperty(String name, int defaultValue)
{
  return getIntAttribute(name, defaultValue);
}
public int getIntAttribute(String name, int defaultValue)
{
  String value=(String)attributes.get(name);
  if (value == null) {
    return defaultValue;
  } else {
    try {
      return parseInt(name); //bug
    } catch (NumberFormatException e) {
      throw invalidValue(name, value);
    }
  }
}
```

(b) getProperty in NanoXML-2.2

Figure 6.12: Regression bug in NanoXML-2.2

to version 2.2. The simplified source code of the bug is shown in Figure 6.12. Given a property name of an XML element as specified in the parameter, the `getProperty` method is used to get the integer-typed value of the property. The implementation of `getProperty` method was changed from version 2.1 to 2.2. In particular, in version 2.2, the code was restructured and `getProperty` method was implemented by simply calling another method `getIntAttribute`. Unfortunately, the implementation of method `getIntAttribute` contains a bug. The bug lies in the second return statement in Figure 6.12b. Instead of `return parseInt(name)`, it should be `return parseInt(value)`. We applied our DARWIN debugging technique with version 2.1 as the reference version and version 2.2 as the buggy version. Version 2.1 and version 2.2 of NanoXML have 4947 and 5244 lines of code respectively, and there are 2496 different lines of code between these two versions. The original path-condition-based DARWIN technique took 2 minutes and 56 seconds to generate a bug report with 8 lines of source code. In contrast, DARWIN technique using *relevant-slice condition* only took 43 seconds to generate a bug report with 6 lines. Both the debugging time and the debugging result were improved.

6.4.3 Test-suite augmentation

Test-suite needs to be augmented when old test-suite no longer meets the test requirement due to program changes [93, 120]. Suppose a program P is changed to program P'. We can employ PCExplore to generate a set of change-exposing test cases to augment the existing test suite. More specifically, we can first apply PCExplore to get signatures of program P and P' separately. Suppose the signature for program P is $Sig(P)$, which is a set of path condition and symbolic output value pairs. For each path π explored by PCExplore, we add $\langle pc(\pi), symout(\pi)\rangle$ into $Sig(P)$, where $pc(\pi)$ denotes the path condition along path π and $symout(\pi)$ denotes the symbolic value of output computed on π. Similarly, we compute the signature for program P' as $Sig(P')$. We could try to generate a test case for each formula $f(\pi_i, \pi_j) \stackrel{\text{def}}{=} (pc(\pi_i) \wedge pc(\pi_j) \wedge symout(\pi_i) \neq symout(\pi_j))$, where $\langle pc(\pi_i), symout(\pi_i)\rangle \in Sig(P)$ and $\langle pc(\pi_j), symout(\pi_j)\rangle \in Sig(P')$. If $f(\pi_i, \pi_j)$ is satisfiable, its solution is guaranteed to have different output in program P and P' according to the definition of $f(\pi_i, \pi_j)$. We then enumerate all possible values of i and j which make $f(\pi_i, \pi_j)$ satisfiable. Whenever $f(\pi_i, \pi_j)$ is satisfiable, its solution is put into a set $T_{\text{PCExplore}}$. Through this process, we get a set of test cases in $T_{\text{PCExplore}}$. These test cases in $T_{\text{PCExplore}}$ expose the semantic changes between the two programs.

As we have seen in this chapter, we could use the *relevant-slice condition* to efficiently generate a more concise signature for a program. When using *relevant-slice condition* to generate the signature, each element in $Sig(P)$ becomes $\langle rsc(\pi), symout(\pi)\rangle$, where $rsc(\pi)$ is the relevant slice condition along path π. We also need to change the definition of $f(\pi_i, \pi_j)$ accordingly. For signatures generated using *relevant-slice condition*, we define $f(\pi_i, \pi_j)$ as $f(\pi_i, \pi_j) \stackrel{\text{def}}{=} (rsc(\pi_i) \wedge rsc(\pi_j) \wedge symout(\pi_i) \neq symout(\pi_j))$. By solving all the possible instances of $f(\pi_i, \pi_j)$, we get a set of test cases $T_{\text{RSCExplore}}$. As path exploration based on *relevant-slice condition* does not lose any precision when generating the signature, $T_{\text{RSCExplore}}$ has the same change-exposing ability than $T_{\text{PCExplore}}$.

On the other hand, path exploration based on *relevant-slice condition* is much more efficient than PCExplore, which makes computing $T_{\text{RSCExplore}}$ much less costly than computing $T_{\text{PCExplore}}$.

```
1 int foo(int x, int y, int z){
2   int out; // output variable
3   int a;
4   int b = 2;
5   if(x - y > 0) //b1
6     a = x;
7   else
8     a = y;
9   if(x + y > 10) //b2
10    b = a;
11  if(z*z > 3)  //b3
12    System.out.println("square(z)>3");
13  else
14    System.out.println("square(z)<=3");
15  out = b;
16  return out; //slicing criteria
17}
```

(a) Original Program

```
1 int foo(int x, int y, int z){
2   int out; // output variable
3   int a;
4   int b = 2;
5   if(x - y > 2) //b1, changed
6     a = x;
7   else
8     a = y;
9   if(x + y > 10) //b2
10    b = a;
11  if(z*z > 3)  //b3
12    System.out.println("square(z)>3");
13  else
14    System.out.println("square(z)<=3");
15  out = b;
16  return out; //slicing criteria
17}
```

(b) Changed Program

Figure 6.13: Sample programs for test-suite augmentation

We now show the process of computing $T_{\text{RSCExplore}}$ in action using the programs in Figure 6.13 as an example. The program in Figure 6.13a is the same as the program in Figure 6.2. Figure 6.13b contains a changed version of the program in Figure 6.13a. The branch at line 5 is changed from `if(x-y>0)` to `if(x-y >2)`. As mentioned earlier, the program in Figure 6.13a has the following signature

- $(x - y > 0) \wedge (x + y > 10) \Rightarrow out == x$
- $(x - y \leq 0) \wedge (x + y > 10) \Rightarrow out == y$
- $(x + y \leq 10) \Rightarrow out == 2$

The signature of the changed program in Figure 6.13b is as follows,

- $(x - y > 2) \wedge (x + y > 10) \Rightarrow out == x$
- $(x - y \leq 2) \wedge (x + y > 10) \Rightarrow out == y$
- $(x + y \leq 10) \Rightarrow out == 2$

Following the aforementioned process of generating $T_{\text{RSCExplore}}$, after removing unsatisfiable formulae, only the following formula is satisfiable

$$((x - y > 0) \wedge (x + y > 10)) \wedge ((x - y \leq 2) \wedge (x + y > 10)) \wedge (x \neq y)$$

By generating a test input from this formula, we get $T_{\text{RSCExplore}}$ as $\{\langle x == 6, y == 5\rangle\}$ (The input variable z is not bounded and can be any integer value). Programs P in Figure 6.13a and P' in Figure 6.13b produce different output when given this input.

We compared the change-exposing ability of $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$ using `tcas` from SIR as the benchmark. There are 41 versions of `tcas` with seeded bugs. Each one of these 41 versions has only one change from the original program. Among them, two versions always crash with array out of bound exceptions after being translated from C to Java. These two versions are not considered in our experiment. For each version i, we use the original program as P and version i as P'. Then we compute the two set of test cases $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$ based on P and P'. We compare the change-exposing ability of $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$. We observe that whenever the number of change-exposing inputs in $T_{\text{PCExplore}}$ is not zero, the number of change-exposing inputs in $T_{\text{RSCExplore}}$ is not zero. This shows that using *relevant-slice condition* to explore the program does not change the ability of generating change-exposing inputs. On the other hand, redundant path exploration is avoided, which improves the efficiency of the approach. Overall, the time taken to generate $T_{\text{RSCExplore}}$ is 20.8% of the time taken to generate $T_{\text{PCExplore}}$ in the experiment with `tcas`.

We have also discussed a test-suite augmentation technique in Chapter IV where a PIE (Propagate-Infect-Execute) based approach was discussed. We now compare the two techniques. First, the technique in Chapter IV could only handle one program change at a time where the technique we discuss here could function irrespective to the number of program changes. Second, in Chapter IV, we try to generate only one test case for one change where our technique based on *relevant-slice conditions* tries to generate many test cases that could expose the changes in different manners. Last, on the scalability aspect, the technique in Chapter IV is more scalable as various heuristics are used in trying to generate only one test case.

6.5 Threats to Validity

Internal threats An internal threat to validity comes from potential bugs in our implementation. We note that the slicing functionality of JSlice is thoroughly tested and has been widely used in both academia and industry. Another threat to validity comes from the subject program selection in the evaluation. Studies on more subject programs could help better assess the effectiveness of our technique.

Program crashes Our path exploration does not try to cover all paths. Instead, we try to group paths based on symbolic outputs. This is done with the goal of test-suite construction, where testing will expose possible failures in the program. However, failure of a test case does not only come from unexpected outputs - it can also come from program crashes. Thus, for the paths which we do not explore if they contain program crashes - these will not be exposed by the test-suite computed by our technique. For example, given `if(x>0)`{`p[i] = 0;`}, it is possible that the branch is never evaluated to true in our exploration process, hence any possible `ArrayIndexOutOfBoundsException` in the access to `p[i]` will not be spotted by the generated test-suite. For branches that are not in the relevant slice of any trace, our technique do not guarantee that both directions of the branch will be explored. Realistically, our test-suite construction could be supplemented by techniques to statically detect possible program crashes, such as memory errors [116].

Approximation of relevant slice Due to the conservative nature of static analysis used in computing relevant slice, our technique may over-approximate the potential dependencies and hence the relevant slice. Since more branches will appear in the over-approximate relevant slice (than what should appear in the actual relevant slice), therefore the computed *relevant-slice conditions* from the over-approximated relevant slice will be stronger than the *relevant-slice conditions* that would have been computed from the actual relevant slice. In that case, we may explore more than one paths that have the same *relevant-slice condition*. Consider the following program.

```
101 if(x > 0){
102     p.num = 0;
103 }
104 out = q.num;
```

Suppose p and q never alias each other. If the static analysis cannot determine the non-alias between p and q, line 104 is potential dependent on line 101 when the branch at line 101 is evaluated to false. Therefore, the branch at line 101 is included in relevant slice and our technique will try to explore both directions of the branch at line 101, which is unnecessary. Note that this strengthening of *relevant-slice condition* only causes duplicated exploration of some *relevant-slice conditions*, it does not affect the completeness claim of our technique.

Different output types and multiple outputs Programs produce outputs in various ways including return value, side-effect on heap variables, direct interaction with files, etc. Our technique naturally considers all these output types of outputs on users' demand. When multiple outputs exist, users simply need to include all these outputs in the slicing criteria. Consequently, the computed relevant slices contain all statements affecting at least one of the outputs. However, including more outputs in slicing criteria increases the size of relevant slices and relevant slice conditions, which reduces the effectiveness of our technique. Let us use the example in Figure 6.2 to illustrate this issue. In our earlier discussion, we have been considering the return statement in line 16 as the slicing criteria. Suppose now we also want to consider the two printing statements in line 12 and 14 as outputs. We first need to transform the program into the program in Figure 6.14, so that the slicing criteria post-dominate the program entry. After the transformation, both line 17 and line 18 in Figure 6.14 are set as slicing criteria. Applying our technique, our tool generates the following summary with 6 entries as opposed to the 3-entry summary shown earlier when only the return statement is set as the slicing criteria.

- If $x - y > 0$ and $x + y > 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == x$

```
int foo(int x, int y, int z){//input variables
  int out; // output variable
  String s = null;
  int a;
  int b = 2;
  if(x - y > 0) //b1
    a = x;
  else
    a = y;
   if(x + y > 10) //b2
    b = a;
  if(z*z > 3)  //b3
    s = "square(z) > 3";
  else
    s = "square(z) <= 3";
  out = b;
  System.out.println(s); //slicing criteria
  return out; //slicing criteria
}
```

Figure 6.14: Sample program with multiple outputs

- If $x - y > 0$ and $x + y > 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == x$
- If $x - y \leq 0$ and $x + y > 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == y$
- If $x - y \leq 0$ and $x + y > 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == y$
- If $x + y \leq 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == 2$
- If $x + y \leq 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == 2$

With more outputs, our technique is less efficient. However, for the above program, our technique is still more efficient than full path exploration, which will exercise all eight paths.

Scalability issues Apart from considering more outputs, lack of inherent parallelism in a program also reduces the effectiveness of our technique. If a program contains little inherent parallelism, the relevant slice of an input t may contain the majority of the execution trace of input t. In such case, the improvement of our technique over path exploration based on path condition is limited. Although our technique considerably improves the efficiency of the path exploration, the path explosion problem still exists. In the worst case, the number of *relevant-slice conditions* grows exponentially with the number of branches in the program.

```
int foo(int i){ //input variable
  int a[2];
  int out = 0;//output variable
  a[0] = 0;
  a[1] = 1;
  if(a[i] > 0){
    out = 2;
  }
  return out;
}
```

Figure 6.15: Example program of imprecise array modelling

SMT solver support The proofs presented in Section 6.2.2 assume that the underlying SMT solver is both sound and complete. The SMT solvers we have used are sound. That is, if the solver declares that a formula is satisfiable (unsatisfiable), then the formula is indeed satisfiable (unsatisfiable). We now examine the completeness aspect of the assumption. In general, off-the-shelf SMT solvers are not complete for *relevant-slice conditions* generated from real programs. However, SMT solvers could be complete for formulae within certain theories. For example, the STP [48] solver is sound and complete for quantifier-free formulae in the theory of bit-vectors and arrays. Therefore, if a subject program only uses fixed-size integer variables and the fixed-size integers are modeled as bitvector arrays, the STP solver then acts as a decision procedure for such formulae. In our experiment, Z3 is used as the underlying SMT solver. Z3 is not complete for non-liner integer operations. Although being incomplete, various heuristics incorporated into Z3 have been shown to be effective for solving formulae with non-liner integer operations in practice. Z3 has three types of output: `sat`, `unsat` and `unknown`. The `sat` output indicates that the formula is satisfiable and the `unsat` output indicates that the formula is unsatisfiable. The `unknown` output suggests that Z3 fails due to incompleteness. However, we did not observe any `unknown` output from Z3 in our evaluation presented in Section 6.4. If incompleteness of the underlying SMT solver occurs, our path exploration could be incomplete.

Modeling of bytecode semantics in implementation Finally, we note that the completeness proof of Algorithm 6.1 in Section 6.2.2 also assumes that the semantics of the different program statement executed in a trace is precisely modeled in the computed *relevant-slice condition* of that execution trace. However, in our implementation, certain program features are not precisely modeled, which causes our path exploration to be incomplete. In particular, polymorphism and arrays are not precisely modeled in our current implementation. Let us consider the example program in Figure 6.15. Suppose the slicing criteria is at line 9. The precise *relevant-slice condition* for the execution trace of input $i == 0$ is $i \geq 0 \wedge i < 2 \wedge a[i] > 0 \wedge ((i == 0 \wedge a[i] == 0) \vee (i == 1 \wedge a[i] == 1))$. To get this precise *relevant-slice condition*, we need to trace all the possible assignments to any element of array `a[]`. However, in our implementation, we compute an approximated *relevant-slice condition* by concretizing the array index of any array reference as is done in other dynamic symbolic execution engine, such as BitBlaze [99]. Therefore, the approximated *relevant-slice condition* we get is $a[1] > 0$, which could be reduced to true in this example. Because of this approximation, exploration then misses the case that the branch at line 6 could be evaluated to false.

6.6 Summary

In this chapter, we have presented a novel path exploration method based on symbolic program outputs. Our path exploration dynamically groups paths on-the-fly, where two paths that have the same symbolic output are grouped together. Given such a path partitioning, we can generate a test case from each partition. This enables us to efficiently obtain a concise test-suite which stresses all possible input-output relationships in the program.

We experimentally compare the efficiency and coverage of our method with respect to path search method based on symbolic execution. The path partitioning computed by our method can be exploited in various other software engineering activities. We have shown its use in the debugging of errors introduced by program changes, that is, in root-causing observable software regressions. By comparing the path partitioning in two program versions, we infer the semantic differences across the versions, leading to precise root cause identification. We have also shown the use of our method in test-suite augmentation. We generate test-cases to augment existing test-suite by focusing on the different partitions across two program versions.

CHAPTER VII

Other Work

In this chapter, we present two more techniques to further extend our solutions on detecting and localizing software regression errors.

First, we present a language for intended semantics of program changes – software change contracts. In our test generation and DARWIN debugging work, one important piece of missing information is the intended semantics of program changes. Without knowing the intended semantics of program changes, our proposed techniques are restricted. For test generation, we are unable to tell whether the generated test cases signal regression errors or they actually witness the intended changes of programmers. For our DARWIN debugging technique, we had to assume that the program requirements vis-a-vis existing features of the program do not change when the intended requirement changes are unavailable. If the program requirements are indeed changed, false positives will be generated by DARWIN. Our proposed change contracts come into rescue for the aforementioned issues. The intended semantics of program changes can be expressed formally in change contracts and used to check the correctness of real code changes as we will illustrate in this chapter.

Second, we present a technique to localize failure-inducing changes in execution environments. We have mainly focused on software regression errors in the application programs in previous chapters. However, software may also fail due to problems in the underlying execution environments (e.g., operating systems). In this work, we try to localize the root causes of software failure caused by evolving execution environments.

7.1 Software Change Contracts

In this section, we propose the notion of "change contracts" to deal with incorrect program changes. Change contracts specify the *intended* semantic change corresponding to changes in program code. When the *actual* program changes break what is documented in the change contract, an inconsistency can be detected. If the change contract is properly written, such an inconsistency points out incorrect program changes. Therefore, with the help of change contracts, an incorrect program change can be detected and corrected - prior to checking in such incorrect changes into the code repository.

The concept of change contract is inspired by Design by Contract programming [77, 25]. In Design by Contract programming, programs are checked against contracts to enable early error detection. Contracts typically appear in the form of pre- and post-condition of methods, as well as invariant properties whose correctness is preserved by the method execution. However, this early error detection comes at the cost of manually written contracts. This is probably the main reason for the lack of adoption of "design by contract" : programmers are reluctant to write non-trivial specifications.

Compared to program contracts which are recommended in design by contract programming, our change contracts are designed to specify only the intention of change in program behavior rather than the full program behavior. In fact, to detect regression errors, *no change contract* is required at all; we can simply have a default contract which says that the program output after the change should be same as the output before the change. As our experiments show, checking such default change contracts (which does not involve *any* effort from the programmer) can help reveal many subtle program errors.

The fact that change contracts are easier to write than program contracts comes from the intrinsic nature of change contracts. Program contracts are often specified as pre- and post-conditions of methods. Thus, they specify *what a program method does*, about which the programmer may not always have deep understanding (unfortunately!) in real-life. In contrast, a change contract specifies *how the functionality of a program method is changed* with respect to the old program. The common behavior between two programs, which is usually dominant, does not need to be specified in the change contract. Besides, we allow users to write change contracts at multiple levels of precision.

The more precise a change contract is, the more checking is done by our system. The users can choose the level of precision at will. Finally, we note that there exists a large body of code today which completely lacks any formal specification. The concept of change contracts also provides a pragmatic way of adding specifications of intended behavior on top of this huge code base lacking formal specifications.

The rest of this section is organized as follows. We first propose the notion of change contracts to prevent incorrect program changes. Then, we design a change contract language for Java programs. Our language extends the Java Modeling Language or JML with specific keywords to relate behaviors of program versions. We present the formal semantics of our change contract language. We also show via user studies how various kinds of real life program changes can be specified using our change contract language. Real-life changes from three large Java open source programs (Ant, JMeter, log4j) were investigated in the user studies. In total, users wrote 52 change contracts in our user study. We did not meet any change that cannot be expressed using our change contract language. Last but not the least, we design and implement a system for dynamically checking change contracts by building on top of the Run-time Assertion Checker of JML. We found 10 real-life incorrect changes from the same programs (Ant, JMeter, log4j) and wrote change contracts for them. All 10 incorrect changes are detected by the Run-time Assertion Checker via the change contracts.

7.1.1 Overview

We now show a series of code changes made on a file of a well-known build automation software, Apache Ant [9], and explain how our change contract language and its supporting tool can help with the development and maintenance of programs that change over time.

Figure 7.1 shows in reverse chronological order how a method `checkIncludePatterns` in file DirectoryScanner.java of Ant was changed over time. The program in Figure 7.1a is a bug-fixed version of the program in Figure 7.1b. The problem was that `null` could be assigned to variable `f` at line 3 when method `findFile` failed to find file name `c` in the base directory `b` in a case-insensitive way as indicated by the last boolean value. As a result, an NPE (i.e., NullPointerException) was raised at line 4 before. While the fix for NPE is usually as simple as adding a conditional guard as shown in the figure, NPE is pervasive in most Java programs as one of the most common causes of errors. Interestingly, this particular bug was reported by developer Curt while the fix was made by

```
1 void checkIncludePatterns(){
2  ...
3  File f=findFile(b,c,false);
4  if(f!=null && f.exists()){
5  ...
6 }
```

(a) Current version, bug is fixed

```
1 void checkIncludePatterns(){
2  ...
3  File f=findFile(b,c,false);
4  if(f.exists()){
5  ...
6 }
```

(b) Buggy version

```
1 void checkIncludePatterns(){
2   ...
3   File f=findFileCaseInsensitive(b,c);
4   if(f.exists()){
5   ...
6 }
```

(c) Original version

Figure 7.1: Reverse chronological change history; the topmost one is the latest one

another developer Stefan. Indeed, it is common to see that problems missed by the original developer or a maintainer are found by other developers or even end-users.

In fact, the above NPE is a regression error resulted from a previous change; the same problem did not occur until that previous change was made by yet another developer Matthew. The program version in Figure 7.1c shows what the same method looked like before an NPE-causing change had been made. Notice that a different method `findFileCaseInsensitive` was called then instead of `findFile`. In Figure 7.1c, method `findFile` is used only in a case-sensitive way, and a case-insensitive search is performed by `findFileCaseInsensitive`. A regression-error-causing change was made when these two methods were merged into a new method `findFile` where its last boolean parameter is used to choose a case-sensitivity mode.

Now notice that the conditional guard at line 4 in Figure 7.1c does not yet check whether `f` is null. Nevertheless, an NPE did not occur in the original version in Figure 7.1c. The reason of this difference is that when there is no file name `c` in base directory `b`, the merged method `findFile` in Figure 7.1b

```
1  //@ changed_behavior
2  //@  when_signaled (NullPointerException)
3  //@          findFile(b,c,false)==null;
4  //@  signals (NullPointerException) false;
5  void checkIncludePatterns() {
6   ...
7   File f=findFile(b,c,false);
8   if(f!=null && f.exists()){
9   ...
10 }
```

Figure 7.2: An annotated change contract for the latest change

returns `null` whereas `findFileCaseInsensitive` in the original version as shown in Figure 7.1c creates a fresh dummy object of type `File`.[1] Apparently, it seems that the developer mistakenly assumed that the merged method `findFile`, when its boolean parameter is set `false` to indicate case-insensitivity, always behaves in the same way as the removed `findFileCaseInsensitive` did in the previous version. It is, however, difficult to put the entire blame on the developer because without proper tool support most developers are likely to make similar mistakes.

We now show how change contract can help deal with program changes described above in various ways. A change contract is essentially a formal specification about intended program changes. Like other formal specifications, change contracts can be used as *unambiguous documentation*. For example, Curt who found the aforementioned unexpected NPE could have reported the NPE problem with the change contract shown in Figure 7.2. The given annotation describes the following two things. (i) First, line 2˜3 describes that there exists a certain input that caused the previous version to signal an NPE; when that NPE was signaled, the boolean expression at the end of the `when_ensured` clause indicates that the NPE is caused by the `null` return value from the `findFile` method. (ii) Second, line 4 describes that the current version cannot signal an NPE as indicated by `signals (NullPointerException) false` when given the same input as signaled the NPE before. Once such a change contract is written, Stefan, who is in charge of maintaining this part of code, should be able to understand the NPE problem.

Our change contract is not only unambiguously understandable but also *automatically checkable*. With the help of a supporting tool provided by us, developers can check whether their code changes indeed match intended changes expressed as change contracts. Thus, change contracts function as

[1]It is created by `new File(b,c)`.

method-specification ::= *spec-case* [`also` *spec-case*]$*$
spec-case ::= **changed_behavior** [*spec-clause*]$*$
spec-clause ::= `requires` *pred* | `ensures` *pred* | `signals` *Exception-Type pred*
| **when_ensured** *pred* | **when_signaled** *pred*
exp ::= ... | *quantified-exp* | `\result` | `\old(exp)` | \prev`(exp)`

Figure 7.3: Core change contract language

test oracles for intended changes.

Our supporting tool automatically checks not only whether intended changes are made to the updated version, but also whether unintended changes are mistakenly made. Note that unintended changes can cause regression errors. When an input does not match any given change contracts, previous method and current method are assumed to behave exactly the same for that input by default. Exploiting our by-default-equal assumption, the regression error of our running example could have been detected earlier. When merging two methods that find a file, the changes made to method `checkIncludePatterns` are merely auxiliary, and its behavior was supposed to remain the same. This is the case where our by-default-equal assumption can be exploited to the extreme; no change contract needs to be provided and any behavioral changes are reported as unexpected changes. Our tool can detect such unexpected changes caused by programmer's mistakes.

7.1.2 Change contract language and semantics

We now present the change contract language and its semantics. Our change contract language is built on top of JML (Java Modeling Language) [29], which is the de facto standard program contract language for Java. JML, being a full-fledged specification language, has various language features. We will only discuss the core language of JML that is essential for understanding change contract. Following the JML convention, change contract for a method should be written immediately before the method definition and enclosed in special comments `//@ ...` or `/*@ ... */`.

Figure 7.3 shows the core language of our change contract. The keywords in bold are added for change contracts. The change contract for a method starts with a *method-specification*. Each *method-specification* could have several specification cases (*spec-case*). Under each specification case, there are different specification clauses (*spec-clause*) describing different aspects of the intended program changes. We now illustrate the meaning of each specification clauses.

The `requires`, `signals`, `ensures` clauses are all inherited from JML. The clause `requires` *pred* states that the given change contract only concerns program behavior when the *pred* is true. Note that the meaning of `requires` clause is different when used in program contract, `requires` *pred* means that the caller of the method is responsible to guarantee that *pred* is true.

In general, a Java method can terminate by either throwing an exception or returning to the caller normally. The `signals` and `ensures` clauses allow the users to specify the condition that has to be satisfied in each case of method termination respectively. The clause `signals` *Exception-Type pred* dictates that the method must terminate by throwing an exception of type *Exception-Type* and satisfy *pred* at the same time. On the other hand, `ensures` *pred* dictates that the method must terminate normally (without throwing any exception) and the program state must satisfy *pred* when the method terminates.

Similar to `ensures` and `signals`, we introduce `when_ensured` and `when_signaled` in change contract to specify the termination condition of the previous version method. Apart from specifying the termination condition, these two clauses also serve similar goals as the `requires` clause. That is, the change contract only concerns program behavior change when the condition specified in `when_ensured` clause or `when_signaled` clause is satisfied.

In change contract, we need to express the program state at different time. Either the program state before executing the method or the program state after executing the method can be expressed. In `ensures` and `signals` clauses, without any modifier, an expression refers to the program state after executing the method. To specify the value of an expression before executing the method, the modifier \old(...) has to be used. Since the `requires` clause states the pre-condition, \old(...) can be omitted for any expression in the `requires` clause without any ambiguity.

Since change contract expresses changes across two program versions, when an expression is used in change contract, we need to declare which version it refers to. To this end, we introduce the modifier \prev(...) in change contract. By default, if \prev(...) is not used, an expression refers to its state in the current version. To refer to its value in the previous version, the modifier \prev(...) has to be used.

Combining the time modifier \old(...) and the version modifier \prev(...), for any given expression E, we can have the following expressions referring the value of E in different time and different versions: (i) E without any modifier refers to the value of E in current version after the program is executed; (ii) \prev(E) refers to the value of E in previous program version after the program is executed; (iii) \old(E) refers to the value of E in the current version before the program is executed; (iv) \old(\\prev(E)) or \prev(\\old(E)) refers to the value of E in the previous version before the program is executed. Recall that we always assume the two program versions start with the same program state. Therefore, the value of E before the program is executed is the same in the previous version and current version. That is, \old(\\prev(E)) == \\prev(\\old(E)) == \\old(E).

We use the example in Figure 7.4 to illustrate the value of an expression under different modifiers. Suppose the value of `num` before executing method `foo` is 1. In change contract, `num` without any modifier is 3 as it refers to the value of `num` in the current version after method `foo` is executed. The expression \old(num) refers to the value of `num` before the method `foo` is executed (the version is unimportant as both versions have the same program state before the method is executed). In this example, the value of \old(num) is 1. The expression \prev(num) refers to the value of `num` after the method `foo` is executed in the previous version. Hence, the value of \prev(num) is 2 in this example.

```
int num = 1;
void foo(){
  num = num + 1;
}
```

(a) Previous version

```
int num = 1;
void foo(){
  num = num + 2;
}
```

(b) Current version

Figure 7.4: The value of an expression under different modifiers

7.1.2.1 Semantics of a change-specification case

We now present the semantics of a change-specification case consisting of method-specification clauses. Note that not all clauses need to be present in a change-specification case. When a certain type of clause is omitted, a default clause is used. For example, if there is no `requires` clause in a given specification case, we insert the default clause `requires true;`. The default clause of each clause type is shown in Table 7.1. We treat the empty contract separately as will be explained in Section 7.1.2.3.

For the sake of explanation, consider the following complete change-specification case for method m. In the below, Greek letters denote predicates, and two subscripted Ts represent exception

Table 7.1: Default clauses of a change contract

Clause Type	Default Clause
`requires`	`requires true;`
`when_ensured`	`when_ensured true;`
`when_signaled`	`when_signaled (Exception) true;`
`ensures`	`ensures true;`
`signals`	`signals (Exception) true;`

types (i.e., subtypes of `java.lang.Exception`). Lastly, variables `x1` and `x2` are scoped to θ and θ', respectively.

```
/*@ changed_behavior
  @  requires φ;
  @  when_ensured ψ;
  @  when_signaled (T₁ x1) θ;
  @  ensures ψ';
  @  signals (T₂ x2) θ';
  @*/
```

The above specification should be read as follows: when started with a pre-state satisfying φ, if the previous version of m satisfies ψ at its normal termination and θ at its abnormal termination raising an exception of type T_1, respectively, then the current version of m should satisfy ψ' at its normal termination and θ' at its abnormal termination raising an exception of type T_2, respectively. The following verification condition provides the meaning of the given change specification more formally:

$$\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta)) \Rightarrow wp(m_2, \psi') \vee \widehat{wp}(m_2, T_2, \theta')$$

In the above, the previous and the current versions of method m are distinguished as m_1 and m_2. We use two weakest-precondition notations $wp(m, \psi)$ for method m and its normal post-condition ψ, and $\widehat{wp}(m, T, \theta)$ for m's abnormal post-condition θ and exception type T. The latter makes it sure that a raised exception is of type T before asserting θ; i.e., $\widehat{wp}(m, T, \theta) \Leftrightarrow ((\texttt{x instanceof } T) \Rightarrow wp(m, \theta))$, where `x` refers to a raised exception. For the sake of simplicity, we assume methods m_1 and m_2 have only two exit points, one for normal termination and the other one for abnormal termination. As usual, all free variables appearing in the verification condition are assumed to be universally quantified. Our change contract checker disallows the verification condition to be vacuously true by issuing a warning when the left-hand side of the verification condition, $\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta))$, becomes false.

For the convenience of users, multiple instances of the same type clause are allowed to be included in a change specification case. When multiple instances of the same type clause exist, we reduce them to its semantically equivalent form with a single clause in a standard way by basically conjoining predicates of the same type. For example, writing `requires` φ_1`;requires` φ_2`;when_ensured` ψ_1`;when_ensured` ψ_2`;ensures` ψ'_1`;ensures` ψ'_2`;` is equivalent to writing `requires` $\varphi_1 \wedge \varphi_2$`;when_ensured` $\psi_1 \wedge \psi_2$`;ensures` $\psi'_1 \wedge \psi'_2$`;`. Similarly, `signals (`T_1 `x1)` θ'_1`;signals (`T_2 `x2)` θ'_2`;` is equivalent to a single clause `signals (Exception x) ((x instanceof` T_1`)` $\Rightarrow \theta'_1$`)`$\wedge$`((x instanceof` T_2`)`$\Rightarrow \theta'_2$`);`. Multiple instances of `when_signaled` are reduced in the same way.

7.1.2.2 By-default-equal rule

Program development is done gradually through many changes and bug-fixes only alter a small buggy portion of the program. Therefore, when a change is made, some program behavior is changed with most of the existing program behavior unchanged. Making sure that no undesired change is introduced is as important for assuring ourselves that all the changes are performed correctly. To this end, our change contract assumes that all program behavior is unchanged unless explicitly specified. Given the following change contract,

```
/*@ changed_behavior
  @  requires φ;
  @  when_ensured ψ;
  @  when_signaled (T₁ x1) θ;
  @  ensures ψ';
  @  signals (T₂ x2) θ';
  @*/
```

the previous program and current program should have the same behavior when $\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta))$ is false. That is, the above change contract only specifies how the program should change when $\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta))$ is true. When the behavior of an input is unspecified in change contract (this is true iff. the input could make $\varphi \wedge (wp(m_1, \psi) \vee \widehat{wp}(m_1, T_1, \theta))$ false), our by-default-equal rule requires the program behavior to be unchanged from previous to current version.

7.1.2.3 Empty change contract

When no change contract is provided, nothing about the program change is specified. Following the discussion about by-default-equal rule, we assume that the program behavior is not changed for any program inputs.

Our by-default-equal rule and treatment of the empty change contract allows users to focus only on the changed program behavior when writing change contract. On the other hand, undesired program behavior changes causing regression errors are detected automatically by the default rules without requiring any manual effort.

7.1.3 Writing change contracts

Change contracts mainly concern two aspects of program behavior – (i) under what conditions the program behavior changes (the pre-condition for the change), (ii) exactly how the program behavior changes (the post-condition after the change). We now discuss how these aspects of program changes are covered by writing of change contracts.

Specifying pre-conditions Usually when a method is changed, the behavior change is limited to only a sub-domain of its input space. We provide two ways to specify the input domains that contains behavior changes.

We can directly specify the set of inputs whose behavior is changed by specifying constraints on the inputs (including parameters, fields and global variables). The change contract then only concerns the inputs that satisfy the specified constraints. The keyword `requires` in JML serves this purpose. A change contract specification with `requires E` where `E` is an boolean expression on the method inputs means that we only focus on inputs that satisfy `E`. As mentioned in Section 7.1.2.1, a change contract comes equipped with pre-condition captured by a `requires` clause.

Apart from directly using `requires`, we can also specify the change contract pre-condition indirectly via `when_ensured` and `when_signaled` (the post-condition of the previous method). Suppose we have `when_ensured E1` in a change contract. It is equivalent to specifying `requires E2` where `E2` is the weakest pre-condition computed on the previous method with respect to `E1`. Thus, `when_ensured` and `when_signaled` can indirectly specify method pre-conditions. Figure 7.5 shows an example in which using `when_ensured` is more convenient (than using a `requires` clause) for specifying the pre-condition. In this example, if `requires` clause is used to specify the pre-condition, a complicated condition under which the `else` branch is executed has to be used. Instead, the complicated pre-condition can be simply specified using `when_ensured` clause on the post-state of previous method as shown in Figure 7.5c.

```
Set m(String s){
 if(/*complex predicate on s*/)
  return new HashSet();
 else
  return new TreeSet();
}
```

(a) Previous program

```
Set m(String s){
 if(/*complex predicate on s*/)
  return new HashSet();
 else
  return new TreeSet().add(s);
}
```

(b) Current program

```
/*@changed_behavior
  @ when_ensured \result instanceof TreeSet;
  @ ensures \result.size() ==   \prev(\result).
     size() + 1;
  @*/
```

(c) Change contract

Figure 7.5: An example of using previous result in pre-condition

Specifying behavior changes In general, there are two different styles to specify the behavioral changes of a method. We can either specify the behavior of previous and current methods separately or the relation of their behaviors can be specified.

We first discuss the approach that the behavior of previous and current methods are separately specified. The behavior of the current method can be specified using `ensures` or `signals` JML clauses. Similarly, the behavior of previous method can be specified using `when_ensured` clause and `when_signaled` clauses. Apart from specifying the behavior separately, a relation of the current program behavior and previous program behavior can be used to specify how the current program behavior is different with respect to previous program behavior. This can be achieved in change contract language through a combination of `ensures` (or `signals`) and `\prev`. Let us take the example in Figure 7.5. The change contract in Figure 7.5 is specified as a relation between the method return value of previous version and that of current version. Alternatively, the current method return value and previous method return value can be separately specified as follows.

```
/*@ changed_behavior
  @ when_ensured \result instanceof TreeSet;
  @ when_ensured \result.size() == 0;
  @ ensures \result.size() == 1;
  @*/
```

Behavior-preserving changes Some program changes actually preserve program behavior. One common type of behavior-preserving change is code refactoring, which can used to increase program's manageability and extensibility. Programmers also make changes with functionality preserving goals such as increasing program performance, reducing memory consumption and so on. As far as the program's functional behavior is not changed, we consider it as behavior-preserving change in this section. If a program's behavior is incidentally changed when behavior-preserving change is intended, a regression bug is introduced. Regression testing has been widely adopted to prevent regression errors. However, without knowing programmers' intention, it is difficult to classify a behavior change as a regression bug or an intended feature. Our notion of change contracts seeks to fill this gap. When behavior-preserving changes are made, no change contract needs to be written. By not providing any change contract, the default-equal assumption is activated. Any behavior change is clearly a regression bug, and it can be detected automatically by our checker - as evidenced by our experiments.

Another commonly occurring situation is that new fields are added in a class and only operations on the new fields are added. In this case, regression errors can also be prevented using the default-equal assumption. When new fields are added/removed, the default-equal assumption guarantees that the common fields and method result have the same value in the previous version and current version.

7.1.4 Change contract checking

Change contracts are checkable. As practiced in program contracts, various levels of checking are possible from lightweight runtime assertion checking (RAC) to heavyweight full static program verification (FSPV) and extended static checking (ESC) in between. Each level of checking has its own strength and weakness. In general, the degree of completeness of checking increases toward the FSPV side while the degree of easiness in usage and automation increases toward the RAC side. Currently, our tool supports RAC because RAC has been recognized as the most essential support for many well-known Design-by-Contract languages such as Eiffel [77] and JML [29]. As will be shown in Section 7.1.6, our change contract checker was used to detect incorrect changes that caused regression errors in various Apache software.

Recall that by default we assume an empty change contract. Even when no explicit change contract is given, we can check behavioral equivalence between the previous and the current versions. Thus, non-trivial behaviorial equivalence checking between program versions can be achieved with an empty change contract (whereas no checking can be achieved with empty program contracts!).

To automatically check a given change contract, we reduce the problem of change contract checking to the well-established problem of program contract checking. More specifically, our change contract checking is performed in three steps. (i) We first run the previous and the current versions of a program, and log program states at a few checkpoints. These checkpoints consist of the entry of the method under investigation, and exits of the previous and the current versions of that method. (ii) We then generate a helper program annotated with program contracts (*i.e.*, ordinary JML specifications) translated from a given change contract. In Figure 7.6, we use an example to illustrate the helper program generation. Figure 7.6b shows a helper class `A` we generate to check behavioral changes occurring when previous-version class `C` shown in the left of Figure 7.6a is changed to the current one shown in the right of Figure 7.6a. Notice that method `test` of the generated helper class `A` is annotated with an ordinary JML specification resembling the given change contract shown in Figure 7.6a. A prominent difference between the original change contract and the generated JML specification is that the latter uses generated fields instead of the original expressions (*e.g.*, `mod__res` instead of `\result`). These generated fields represent the program states logged in the previous step. Detailed explanation about generated fields and rules for helper generation will be provided shortly. Such helper program generation is performed in a way that the helper program passes the checking for the translated program contract iff. code changes occurred in the target program passes the checking of a given change contract. (iii) Finally, in the last step, we perform run-time assertion checking (RAC) on the generated helper program using a RAC facility of OpenJML [11], a JML tool-suite built on Oracle's OpenJDK.

```
// previous class     // current class
class C {             class C {
 int f;                int f;





                       //@changed_behavior
                       //@ requires \prev(f) > 0;
                       //@ ensures \result == \prev(\result)+x;
                       //@ ensures \latest(x);
 public int m(){       public int m(int x){
  return f;             return f+x;
 }                     }
}                     }
```

(a) Previous class and current class

```
// helper class
class A {
  int old_f, prev_f, mod_f;
  int old_x, prev_x, mod_x;
  int prev__res, mod__res;
  C old_this, prev_this, mod_this;
  Exception prev__expt, mode__expt;

  //@normal_behavior
  //@ requires prev_f > 0;
  //@ ensures mod__res == prev__res+mod_x;
  //@ ensures true;

  public static void main(String[] a)
  { /* First, init fields */ test(); }
}
```

(b) Generated helper class

Figure 7.6: An example of helper class generation. A generated helper class shown in Figure 7.6b; its fields and JML specification are translated from the user programs shown Figure 7.6a.

Our tool automatically performs the above three steps. First, to log program states, we use

AspectJ [64], a popular tool supporting aspect-oriented programming for Java. Using call pointcuts of AspectJ, our tool logs program states at the aforementioned designated checkpoints before and after method calls. These program states include the states of receiver object, method arguments, and method results. Next, our extension of OpenJML parses a given change contract and generates a helper class following our generation rules described below. In the following description, we assume that method `m` of class `C` is annotated with a change contract, and a helper class `A` is generated.

- If `C` has a field `f` with type `T`, helper class `A` has three fields `old_f`, `prev_f` and `mod_f` of type `T` to respectively represent the value of `f` before `m` enters, after the previous version `m` exits, and after the current version `m` exists.
- Similarly, if method `m` of `C` has a parameter `p` with type `T`, `A` has three fields `old_p`, `prev_p` and `mod_p` of type `T`.
- To represent receiver states, `A` also has three fields `old_this`, `prev_this`, and `mod_this` of type `C`.
- If method `m` has a non-void return type `T`, `A` has two fields `prev__res` and `mod__res` of type `T`. Note that we need only two fields in this case because there is no return value at the entry of a method.
- Similarly, `A` has two fields `prev__expt` and `mod__expt` of type `Exception` to represent the exceptions thrown.
- Lastly, `A` has only two methods `main` and `test`.

Method `main` of the last item of the above list performs two tasks. It (1) first initializes all the fields described above using the program states logged at the previous step, and then (2) calls method `test`. Meanwhile, method `test` is annotated with a translated program contract made up of the fields described in the preceding. The only purpose of adding `test` is to execute its annotated program contract through a RAC facility, and hence its body is empty. To obtain such translated program contract of `test`, we use the translation rules of Table 7.2. In the table, the notation $\varphi[x \mapsto x']$ is used to denote that free variables x appearing in φ are replaced with x'. While most of translation rules are obvious, `\latest` and `\deprecated` expressions are transformed to either

Table 7.2: Translation rules for change contract runtime checking

change contract	program contract
`changed_behavior`	`normal_behavior`
`\old(`x`)`	`old_`x
`\prev(`x`)`	`prev_`x
x	`mod_`x
`\prev(\result)`	`prev__res`
`\result`	`mod__res`
`\latest(`x`)`	`true` or `false`
`\deprecated(`x`)`	`true` or `false`
`when_ensured` φ	`requires` φ
`when_signaled (`T `x)` φ	`requires (prev__expt instanceof` T`) &&` φ[x $\mapsto$ prev__expt]
`signals (`T `x)` φ	`ensures (mod__expt instanceof` T`) &&` φ[x $\mapsto$ mod__expt]

`true` or `false` depending on the comparison result of abstract syntax trees for the previous and the current versions of the method.

Finally, our tool compiles a generated helper class with the RAC option of OpenJML turned on. Running the compiled code effectively checks the translated JML specification in the helper program, and its failure amounts to detecting a mismatch between actual code changes and the intended change expressed via the change contract. When the test input does not match any of the specification case in change contracts, the default-equal assumption is checked by checking whether all post-states of fields and method return values are the same in the previous version and current version.

7.1.5 Implementation of change contract runtime checking

We have implemented runtime assertion checking for change contact following the design in Section 7.1.4. The architecture of our tool is shown in Figure 7.7. Recall that to check the change contract of a method `m`, we generate a helper program. The checking of change contract of method `m` is done by checking the program contract of the `test` method in the helper program. OpenJML

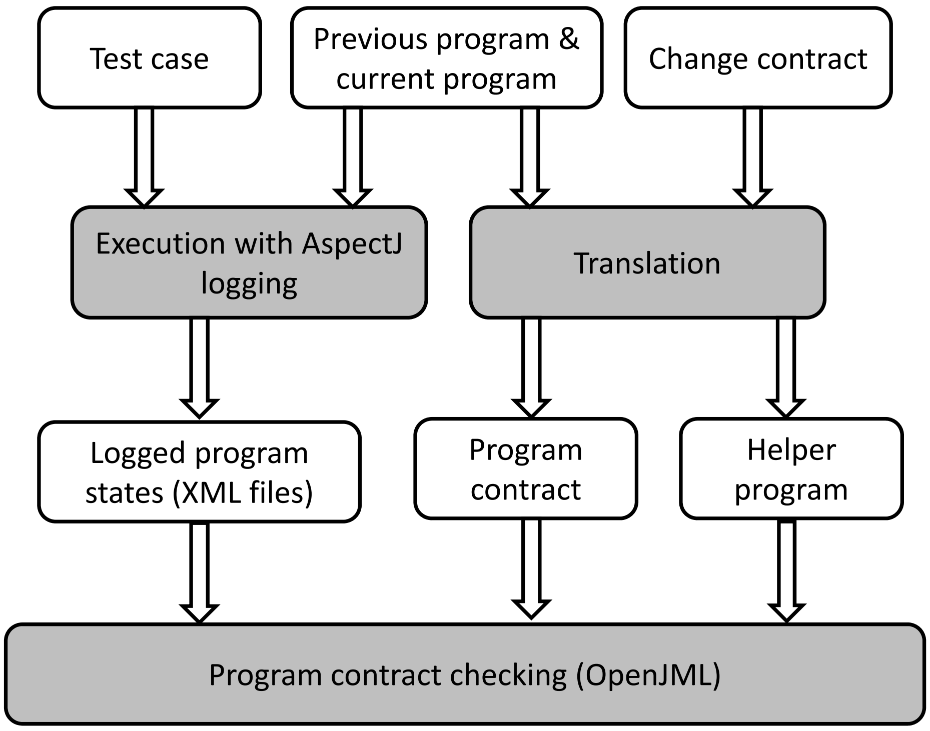

Test case
Previous program & current program
Change contract
Execution with AspectJ logging
Translation
Logged program states (XML files)
Program contract
Helper program
Program contract checking (OpenJML)

Figure 7.7: The architecture of change contract runtime checker

[11] is used in both the generation of helper program and program contract checking of the helper program. OpenJML is a set of tools that support both runtime assertion checking and static checking of Java programs annotated with JML. Based on OpenJDK, OpenJML supports the latest version of Java and JML standard.

To generate the helper program, we parse both the reference program version and the changed program version into ASTs using OpenJML. The fields and method body of the helper program are derived from the ASTs. Apart from the helper program itself, we also need to generate the program contract for its `test` method. The program contract is translated from the change contract of the changed method under study. We extended OpenJML to parse the extensions of JML introduced by change contract. The translation of change contract to program contract follows the rules mentioned in Section 7.1.4. Finally, we use OpenJML to check the standard JML-format program contract of the helper program.

When the unchanged and changed methods are executed, we need to log the program state that will be passed to the helper program. The logging of program state is done through Aspect programming using AspectJ. We instrument at both the beginning and end of the method to collect the states of surrounding object, arguments and method results. The collected states are stored into XML files and are later read by the helper program. We also log any program state modification to check the default-equivalent assumption. When the test input does not match any of the specification case in change contracts, we check whether the modified program states are the same.

7.1.6 Evaluation

We evaluate change contract and our change contract language in two different aspects. We first focus on the expressiveness and usability of change contract language. Following this, we evaluate the efficacy of change contract in detecting incorrect program changes.

Three open source Java programs — Ant, JMeter and log4j are used in our evaluation. All of these are widely used large-scale java programs (Ant and JMeter have more than 100,000 lines of code each, and log4j has around 13,000 lines of code). Ant is the de facto standard Java build automation tool that helps manage the build process. JMeter is used to test the behavior and performance of various servers, such as HTTP and POP3. Log4j is a Java library that eases the logging process in Java.

We evaluate the following two research questions (RQ).

RQ1: Can change contracts describe real-life changes? We have conducted user studies to answer the above research questions. Two users participated in this user study. Both users are second-year Master's students majoring in computer science. Before the user study, they both have no knowledge on program contract and JML. The users are asked to first understand the programs as well as the changes across different versions. Based on their full understanding of the changes, they wrote change contracts. Note that the change contracts are written based on real changes rather than the intention of these changes. Another experiment in which change contracts are written based on the programmer's intention to prevent incorrect changes is presented later.

We select changes from the Bugzilla database of each Java project. Only entries with patch files are selected. Each selected entry contains a set of discussions and some patch files containing the program changes. Note that other types of changes also exist in the Bugzilla database apart from bug-fixes. For example, new feature requests constantly appear in the Bugzilla database. We select changes in this way because the developers' comments and discussions in Bugzilla provide great help

Table 7.3: User study results on expressiveness and useability of change contract language

Subject prog.	Changes	Applicable Changes				Not Applicable	
		Refactoring	Behavior diff	Add/Delete	Not understood	Not supported	Non-code
Ant	43	4	13	15	3	3	5
JMeter	17	1	5	6	1	4	0
log4j	20	2	6	7	1	0	4

for understanding the changes. As these programs and changes are not written by the users, these detailed discussion logs are indeed very important for the users to understand the changes correctly.

The user study results are summarized in Table 7.3. The "Add/Delete" column denotes changes that involve adding or deleting fields, methods or parameters. There are some changes involving library calls that are not open-source. Without the source code of the libraries, the users are not able to fully understand the effect of the changes. The amount of these changes is given in the "Not understood" column. The last two columns show the changes that are not applicable for this user study. The "Not supported" column contains changes that are currently not supported by change contract, such as changes in synchronization in multi-threaded programs. The "Non-code" column shows changes that are not inside Java source code files. For example, a change in XML file is considered as non-code change. In total, 52 change contracts were written for the changes in column "Behavior diff" and "Add/Delete". No change contract needs to be written for re-factoring changes as it is covered by the default equivalence assumption. In the process of writing change contracts, except the changes in synchronization of multi-threaded programs, the users did not observe any case where changes cannot be expressed using change contracts.

Feedback from users We got the following feedback from users in this study.

- It is difficult to write change contract when the change happens on local variables that have long dependence chain from inputs and outputs. The users have to manually follow the program dependence chain to figure out under which condition the change is executed and how the change affects output. The users also suggested that program dependence tracking (such as the dependency analysis performed by a slicing tool) could reduce this manual effort.

- More time is spent on understanding the programs and changes than writing change contracts. This is however partly because the users did not write these programs themselves.

- Common changes seen in the user study are bug-fixes (typically fixing unexpected exceptions) and adding new features by adding new fields and methods. Changes in method signature and deletion of fields/methods are infrequent.

RQ 2: How effective are change contracts in terms of detecting incorrect changes? In the previous study, users write change contracts based on their understanding of the real program changes. Thus, if users do not make any mistake either in understanding the program changes or in witting change contracts, the programs should always be consistent with the written change contracts. However, change contracts, as designed, should reflect the intention of program changes. Only when change contracts contain the intention of program changes, incorrect changes are possible to be detected by change contracts.

We use the following approach to find incorrect program changes and the intended changes from real-life software repositories. Similar approach has been used in existing research to find incorrect bug-fixes in operating systems [122]. We start with a bug-fix in the repository. Let `v3` be the version where a bug is fixed. We search backward in the repository to find where the bug fixed in `v3` is introduced. If the bug resides in method `m`, we only need to focus on the changes that touched method `m`. Suppose we find that a change from version `v1` to version `v2` introduced the bug. We select the cases where the change from `v1` to `v2` clearly an incorrect change (By doing this, we avoid the cases where the change is correct but it reveals a latent bug). When the programmer made changes in `v1`, the intended resultant program should be the bug-free program `v3`. The programmer's intention when changing version `v1` to `v2` is then captured by the differences between version `v1` and `v3`.

Incorrect changes and their corresponding intended changes are found in the three open source Java programs using the aforementioned method. Change contracts are then written based on the intention of changes instead of the real program changes. Original incorrect program changes are checked against the written change contracts. A test case stressing the incorrect change is required in the Runtime checking method mentioned in Section 7.1.4. We write unit tests with the goal of stressing these incorrect changes.

Results from using change contracts to detect incorrect changes are shown in Table 7.4. We studied 10 incorrect changes in the repositories of Ant, JMeter and log4j. All incorrect changes are detected by the written change contracts.

Table 7.4: Checking of change contracts on incorrect changes

Subject prog.	Changes	Detected	Undetected
Ant	5	5	0
JMeter	3	3	0
log4j	2	2	0

7.1.7 Threats to validity

We had only two participants in our user study, and both users had similar background (Masters students with no prior knowledge on program contracts). More participants with different background would allow us to better assess the usability of our change contract language.

When evaluating the effectiveness of change contracts in detecting incorrect changes, we wrote the change contract and test cases on our own. In general, change contract written by programmers may contain mistakes which will reduce the effectiveness of change contracts. In our experiments, the test cases are written intentionally to stress the incorrect change. The effectiveness of change contracts will also be reduced when ideal test cases are not available during dynamic checking.

Another threat to validity in our evaluation is the subject programs selected. Our selected programs are mainly mainstream large open-source projects. Our technique could be better evaluated with more subject programs having different characteristics.

7.1.8 Summary

In this section, we propose the notion of ”change contracts” as the specification of intended program changes. Incorrect changes can be easily detected when checked with respect to their change contracts. Since change contracts only focus on behavior differences across program versions, they can be easier to write than program contracts. In particular, owing to the default-equal assumption, regression errors can be detected without the need for writing any change contracts. Based on JML, we have designed a full annotation language for specifying intended changes in Java programs. We present the precise formal semantics of our annotation language for specifying change contracts. Several concrete examples are given to illustrate the usage of change contracts. We have also proposed a runtime checking method for change contract and implemented it based on the runtime assertion checker of JML. Through a user study on three Java open source projects (Ant, JMeter, log4j), we find that our change contract language is expressive and usable. In addition, all 10 incorrect changes found in our experiments are detected by their change contracts.

Over and above our technical contributions, we believe that the concept of change contract takes us one-step closer to the overarching goal of writing quality software. We conjecture that change contracts can be used in (at least) the following scenarios.

- *Early detection of incorrect program changes*. We have discussed this scenario in this section. Either the programmer or the tester writes change contracts to make sure that program changes are correct.
- *Serving as program change requirement*. Change contracts can be written (potentially by programmers) prior to making changes in code. In this case, the change contracts serve as formal requirements for program changes.
- *Providing formal change logs*. Programmers often maintain change logs in natural language, to document the changes being made to programs. Sometimes, the change logs are inconsistent with real program changes. The inconsistency is hard to discover and causes serious confusion for other colleague programmers. This problem can be solved if checkable change contracts are used in change logs.
- *Change contract and previous program version jointly form the oracle for testing the current program*. In case the intended program behavior is not changed, we can use the output from previous program version for the purpose of testing the current program version. However, if the intended program behavior changes, the expected output of the current program can be found from the change contract as well as the previous program version's output.
- *Change contracts can help in test suite augmentation.* Current test suite augmentation approaches often focus on syntactic changes (*e.g.* [86]) to generate a test case which executes a syntactic change and propagates it to the output. However, there may exist many possible dependency chains across which a change's effect can be propagated only some of which violate the intended change. Capturing the intended change as change contracts can thus help in more accurate test suite augmentation.

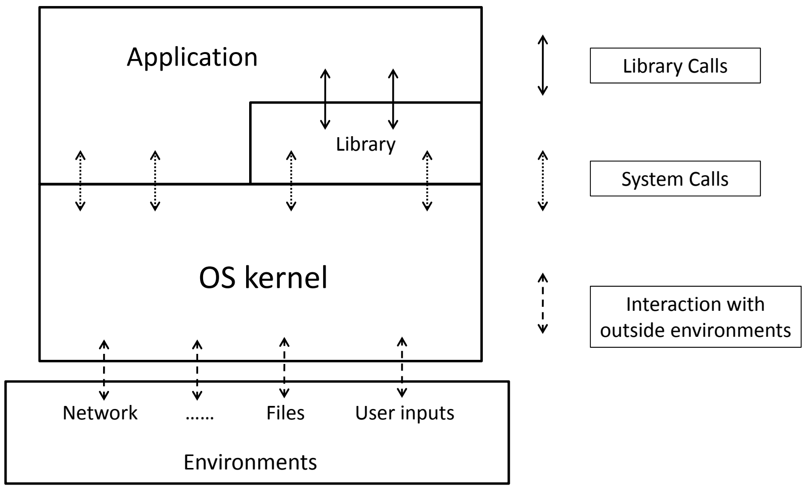


Figure 7.8: Relations between application, library, OS kernel and outside environments

7.2 Locating Failure-Inducing Environment Changes

In this section, we propose a technique to localize failure-inducing environment changes. Programs could fail not only due to bugs in themselves, but also due to their execution environment. According to a study by Chandra et al. [32], around 56% of faults in Apache depend on execution environment. Browsing through the Ubuntu bug list [8] reveals that many bugs are also environmental-related. More specifically, these bugs occur due to some changes in the Operating System (OS) environments. It is quite common that an application works perfectly in one OS environment but fails in another OS environment. As operating systems become increasingly complicated, debugging those operating system induced errors is a challenging task.

Figure 7.8 shows the relationship between applications and the operating system. An application has access to services provided by the OS kernel through system calls. System calls are defined in the Application Binary Interface (ABI) of an OS. The system calls allow the application to invoke kernel services to perform privileged tasks on behalf of the application, such as read or write a file, issue a control command to a device, create a new process, allocate memory and so on. Generally, systems also provide libraries that sit between the OS kernel and a normal application to increase portability, for example the `glibc` library. In this way, the application is less dependent on the OS kernel. The application communicates with the library functions through library calls. Figure 7.8 also shows that the OS kernel acts as an interface between an application and the outside environment to enable an application to interact with the user (or the network or the file system) to perform its intended function. As such, the operating system plays an important part in defining an application's behavior. Even when the input and the program is fixed, the behavior of a program P with an input t can still be affected by a lot of factors in the operating system. For example, dynamic libraries can be implemented differently which leads to implicit semantic difference. A more common case is that the content of some configuration file might be different from one environment to another.

The problem we tackle can be formalized as follows.

Problem Statement: Suppose we have an application program P, an input t, and two OS environments E and E'. The execution of P with input t succeeds in E but fails in E'. In this section, we are trying to explain the failure of P in E' by discovering a subset δ of the changes in $\Delta E = E' \backslash E$. Of course, we try to minimize δ to make our result precise and meaningful.

Assumptions We assume that the application is statically linked. This assumption allows us to focus on the interaction between the application and operating systems without worrying about the changes in dynamic libraries. We also assume that there is only one critical system call that is responsible for the failure of the application. Finally, we assume that the program execution is deterministic.

Building on the assumptions that P does not change, only the OS environment changes, the different behaviors of P in E and E' are completely determined by communications between the program and the underlying OS environment through system call interface. One intuitive solution to the above problem would be to compare the interface communication through the system calls. For each executed system call, we can record the return value and the side effect of the system calls. For each system call $syscall$, we record it as $syscall = \langle num, paras, ret, side\_effect \rangle$. The num is the system call number. Parameters and the return values are recorded in $paras$ and ret respectively. The side-effect of the system call (if any) is recorded in $side\_effect$. In this section, side-effect of a system call refers to the side-effect in the user space unless otherwise specified. The communications through system calls can be represented using a sequence $seq \ \langle syscall_1$, $syscall_2$, $\ldots$, $syscall_n \rangle$. Two sequences of system calls can be compared. There are several difficulties in employing this approach. First, there can be a large number of differences between the two sequences(of system calls) if the two environments are very much different. However, only a very small subset of the differences could be the root cause of the failure. In this case, finding the root cause is a tedious task. Secondly, aligning the two sequences is another error-prone task. Moreover, as the execution environments are different, there might be some system calls in one sequence with no matching system calls in the other. This makes comparing the two sequences more complicated.

We propose a record-replay technique to solve the aforementioned problem. There have been a lot of existing record-replay research in the literature. However, none of the existing techniques is suitable for our task. Most of the existing record-replay techniques replay the execution using the entire recorded data in the same OS environment. On the other hand, we are trying to identify a small subset of the data that constitutes the reason of the failure. Hence, we want to selectively change part of a failing environment and test whether the program succeed. More specifically, we need a technique which enables us to partially replay (for successful environment) and partially execute (for the failing environment) the application. When a system call is replayed, the effect of the system call on the running application is as if the system call is executed in the environment where the system call is recorded. Unfortunately, none of the existing techniques handles such partial replay with recorded data from a different environment. Moreover, there are a few challenges in such a partial replay. One of these arises from the dependency between system calls. Figure 7.9 gives a concrete example: the `read` system call is dependent on the file descriptor returned by the `open` system call. Consequently, in selective replay they have to be replayed or executed together. If only the `open` is replayed and the `read` is executed, the reading from undefined file descriptor will result in errors. Another problem is caused by the complex trace of system calls which makes it difficult to locate the root cause. Therefore, we need an efficient selective replay strategy to enable fast localization of a system call which contributes to the root of the failure. Our selective replay approach is designed to overcome these challenges. The proposed record-replay technique tracks system call dependencies to avoid inconsistent system state. Our technique uses binary-search to fast localize the failure-inducing environment change in a complex system call sequence.

The contributions of this section are as follows:

- We propose a *Semi-replay* technique which allows partial replaying and partial executing an application. The proposed technique enables efficient fault localization in the context of changing OS environment.

```
1   #include <unistd.h>
2   int main(int argc, char *argv[]){
3     int fd;
4     char[128] data;
5     char* config_file = "/path/to/config_file";
6     fd = open(config_file,O_RDONLY);
7     read(fd,data,128);
8       if(check_format(data)){
9         close(fd);
10        exit(1);//error
11    }else{
12       close(fd);
13       exit(0);
14    }
15  }
```

Figure 7.9: An example of defects in execution environment

- We implemented our technique for Linux based on Valgrind.
- We conducted case studies on three real life bugs to evaluate the effectiveness of the proposed technique. In all three cases, our technique is able to locate the change in OS environment that causes the bug.

7.2.1 Overview

We now give an overview of our approach through a motivating example. Let us consider the program P in Figure 7.9. This program opens a configuration file (line 6), reads data from the file (line 7) and checks the format of the data (line 8). The `check_format()` function in P checks whether the data read from the file satisfies some pre-defined format. Suppose the program in Figure 7.9 succeeds in environment E but fails in environment E'. We assume that the failed execution of P in E' is caused by the configuration file in E' which does not follow the pre-defined format. Our debugging method works as follows.

We first record the system call sequences of P in E and use it to identify the problematic system call of P in E'. Suppose the sequence $seq = \langle open_E, read_E \rangle$ has been recorded when executing P in E, where $open_E$ and $read_E$ denote the system calls `open` and `read` in execution environment E respectively.

To identify the root cause of the failed execution of P in E we “selectively” execute seq in E'.

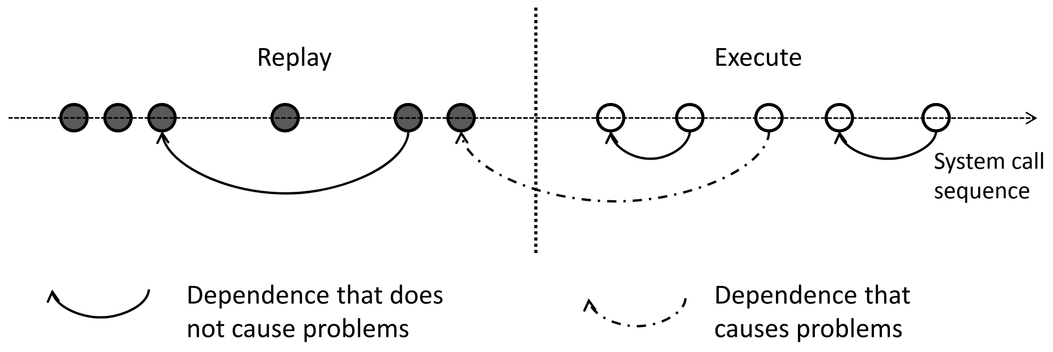

Replay
Execute
System call sequence
Dependence that does not cause problems
Dependence that causes problems

Figure 7.10: Dependencies among system calls. Each circle represents one system call. Arrows denote dependencies among system calls.

First we replay the first half of seq in E' and execute the remaining half in E', which means the system call $open_E$ will be replayed in E' and the system call `read` will be executed in E'. To replay a system call, we instrument the program executable file to intercept all the system calls and their return values and resulting side effects. Therefore, when executing P in E', whenever the system call `open` is invoked, we will replace its return values and side effects with the one recorded for $open_E$. However, when the system call $open_E$ is replayed and the system call `read` is executed in E', the program execution would become inconsistent because of the dependency between $open_E$ and $read_E$. More specifically, the `open` in line 6 is not executed since it is only replayed in E'. As a result, the `read` in line 7 would fail when being executed in E' because the file descriptor `fd` is not initialized. This program failure is caused by the dependency between the `read` in line 7 and the `open` in line 6.

To avoid inconsistent program state caused by system call dependencies, we track the effects of system calls on the kernel state, such as initializing file descriptor in an `open` system call. By doing this, we are able to detect that when we replay the `open` in E', we also need to execute it in E' so as to generate the desired kernel state and more importantly, to enable the execution of `read` in E'. However, when executing P in E', the recorded value for $open_E$ is still used as the return value for `open` in line 6.

Replaying $open_E$, and executing `read` for the execution of P in E', the execution fails. Consequently, we are able to deduce that the root cause of the failed execution lies in the second half of seq which is being executed in E'. As the second half contains only $read_E$, we are able to locate

$read_E$ as the system call which causes the failed execution of P in E'. We then further analyze from here. A simple comparison of the return values and side-effects recorded between the two executions reveals that when the system call `read` is executed, the contents of the file are different in two executions. This comparison enables us to conclude that the root cause of the failed execution of P in E' is caused by the wrong file format of the file provided in E'.

7.2.2 Our approach

Given a single-threaded deterministic program, the expected behavior of the program execution can be determined by three factors: (i) the program input (ii) the executed code and (iii) the environment. In this section, we assume that the program is compiled statically, which means all the libraries must be included during compile time. This guarantees that the entire executable is fixed. Therefore, the behavior of the program is only determined by (i) the program input and (ii) the interface communications through system calls. We also assume that same program input is used in the two executions. If for the two executions, one fails and one passes, then the problem lies in the interface communication with the underlying OS. Based on this, we devise a selective replay technique called *Semi-replay* to locate the changes in the underlying OS which causes the failed execution.

We are able to monitor almost all the interactions between a program and the execution environment by monitoring the system calls made by a program. For example, all file access, network access and even access to system time are done though system calls. Building our technique on system call, not only can we detect semantic changes in system call implementations, but also detect other environment changes reflected by system calls.

Given a program P and an input t, let E' be an environment where P fails to execute with t . Let the executed system call sequence of P in E' be

$$\langle syscall'_1, syscall'_2, \ldots, syscall'_{n-1}, syscall'_n \rangle$$

Algorithm 7.1 returns a number r such that $syscall'_r$ causes the failure of P in E'. The following properties are satisfied by $syscall'_r$:

1. Program P fails when $\langle syscall'_1, syscall'_2, \ldots, syscall'_{r-1} \rangle$ is replayed in *Semi-replay*.

2. Program P passes when $\langle syscall'_1, syscall'_2, \ldots, syscall'_r \rangle$ is replayed in *Semi-replay*.

In *Semi-replay*, system call dependencies cause unexpected program failure if they are not handled properly. In particular, suppose an executed system call $syscall_j$ is dependent on a replayed system call $syscall_i$. System call $syscall_j$ definitely fails if $syscall_i$ is only replayed but not executed. In this case, we need to handle the replay of $syscall_i$ differently. The following sub-sections defines and illustrates system call dependencies and then presents the $Semi\text{-}replay$ technique through the $Locate\_cause$ algorithm in Algorithm 7.1.

7.2.2.1 System call dependencies

We say that a system call $syscall_j$ is dynamically dependent on $syscall_i$ when information flow from $syscall_i$ to $syscall_j$ and denote as $syscall_j \rightarrow syscall_i$. More formally, $syscall_j \rightarrow syscall_i$ iff. $syscall_j$ is transitively dependent on $syscall_i$ via a chain of dynamic data dependencies. For example `read` $\rightarrow$ `open`, which means the `read` system call is dependent on the `open` system call as the file descriptor returned by the `read` system call is used by the `open` system call.

System call dependencies lead to unexpected program behavior during *Semi-replay* if dependent system calls are not being replayed or executed together. Figure 7.10 illustrates dependencies among system calls. Suppose we want to replay the first k system calls and execute the rest in E'. Obviously, dependent system calls that all belong to the "replay" or "execute" half do not cause any unexpected program behavior. However, if there is a system call $syscall_j$ in the "execute" half which is dependent on a system call $syscall_i$ in the "replay" half, then $syscall_j$ will fail because $syscall_i$ is only replayed but not executed. The problem is caused by the inconsistent assumption on system state by $syscall_j$. The system call $syscall_j$ assumes the existence of some kernel state modified by $syscall_i$. However, the modification of kernel state by $syscall_i$ is not regenerated when $syscall_i$ is replayed.

Algorithm 7.1 *Locate_cause*

```
1: INPUT:
2: $R$ // recorded syscall sequence in a successful run
3: $P$ //program being debugged
4: $E'$ // an environment that $P$ fails in
5:
6: $start = 0$
7: $end = n$
8: while $end - start > 1$ do
9:     $ret = Semi\text{-}replay(P, E', R, (end + start)/2)$
10:    if $ret$ then
11:        $end = (end + start)/2$
12:    else
13:        $start = (end + start)/2$
14:    end if
15: end while
16: return $end$
17:
18: procedure $Semi\text{-}replay(P, E', R, k)$
19:    let $R$ be $\langle syscall_1, syscall_2, ...syscall_n\rangle$
20:    $i = 0$
21:    execute $P$ in $E'$
22:    while a syscall is encountered in the execution do
23:        $i = i + 1$
24:        if $i \leq k$ then
25:            if $check\_dep(i, k, R)$ then
26:                execute the syscall
27:                overwrite syscall results using $syscall_i$
28:            else
29:                replay the syscall using $syscall_i$
30:            end if
31:        else
32:            execute the $syscall$ in $E'$
33:        end if
34:    end while
35:    if execution passes then
36:        return $true$
37:    else
38:        return $false$
39:    end if
40: end procedure
41:
42: procedure $check\_dep(i, k, R)$
43:    let $R$ be $\langle syscall_1, syscall_2, ...syscall_n\rangle$
44:    for all $j$ from $k + 1$ to $n$ do
45:        if $syscall_j$ is dependent on $syscall_i$ then
46:            return $true$
47:        end if
48:    end for
49:    return $false$
50: end procedure
```

7.2.2.2 Algorithms

We now explain in details the $Locate\_cause$ algorithm. Given a recorded system call sequence generated by the successful execution of P in E, the algorithm returns the index of a system call in the sequence which is the root cause of the failed execution of P in E'. The $Locate\_cause$ algorithm resembles a binary search algorithm. The main part of the algorithm is the $Semi\text{-}replay$ procedure. The basic idea of the algorithm is as follows. The algorithm first takes the mid-point of the sequence and tries to replay the first half of the system call sequence (line 25-30) and execute the remaining half (line 32). We refer to the first half as the *R-half* and the second half as the *E-half* and the index defining these two halves as k. There are two possibilities:

1. *The execution passes* (line 36): The $Semi\text{-}replay$ procedure returns $true$, which means that the root cause of the failure of P in E' must be located in the *R-half*. This is because executing the *E-half* does not cause the failure. In this case, the $Locate\_cause$ algorithm minimizes the the search for the problematic system call in the *R-half* by dividing this half further into two halves, moving the index k upwards to the mid-point of the *R-half* and iteratively calling the $Semi\text{-}replay$ procedure. As k is now moved to the mid-point of the *R-half*, the $Semi\text{-}replay$ procedure will replay only the first half of *R-half* and execute the second half of *R-half* plus *E-half*.

2. *The execution fails* (line 38): The $Semi\text{-}replay$ procedure returns $false$ which means that the root cause of the failure of P in E' must be located in the *E-half*. This is because when we execute the *E-half* of system calls in E', the execution fails. In this case, the $Locate\_cause$ algorithm minimizes the search for the problematic system call in the *E-half* by dividing the *E-half* further into two halves, moving the index k downwards to the mid-point of the *E-half* and then iteratively calling the $Semi\text{-}replay$ procedure. As k is now moved to the mid-point of the *E-half*, the $Semi\text{-}replay$ procedure will replay the *R-half* plus the first half of *E-half* and execute the second half of *E-half*.

This process continues until a single system call is located which causes a failed execution. One key procedure in this algorithm is $check\_dep(i, k, R)$. This is very important as executing a sequence of system calls involves handling dependency between system calls. For each system call

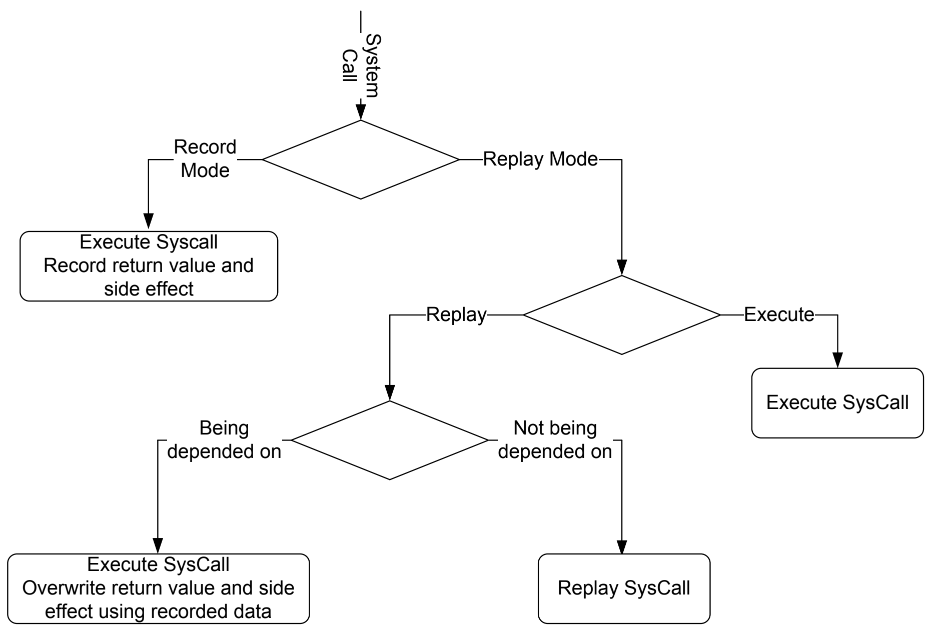

System Call
Record Mode
Replay Mode
Execute Syscall Record return value and side effect
Replay
Execute
Execute SysCall
Being depended on
Not being depended on
Execute SysCall Overwrite return value and side effect using recorded data
Replay SysCall

Figure 7.11: Flow of handling system calls in our implementation

$syscall_i$ being replayed, which means in the *R-half*, this procedure searches in the *E-half* for all the system calls that are dependent on the $syscall_i$. If there are some system calls that are dependent on $syscall_i$, then $syscall_i$ need to be executed as well to regenerate the side-effects and returned values to be used later by the dependent system calls.

Another important feature of the *Semi-replay* algorithm is that a prefix of system call sequence is always replayed and the suffix is executed. This feature is based on the fact that it is almost impossible to execute a prefix of a system call sequence and replay the suffix. This is because executing a prefix of a system call sequence could possibly drive the program execution to a totally different path from the recorded execution. Consequently, the recorded suffix does not match the executed prefix; thus making replaying of the suffix impossible.

7.2.3 Implementation

The proposed approach is implemented as a plug-in of Valgrind[6]. Valgrind[6] is a widely used dynamic binary analysis tool. To capture the communication between an application and the underlying execution environment, we leverage the Valgrind API to instrument the application's binaries. The instrumentation is designed to capture all system calls between an application and the

OS. For each system call, the return values and the side-effects are recorded.

The process of handling systems call is shown in Figure 7.11. Our plug-in provides two modes namely record mode and replay mode. In record mode, whenever a system call is encountered, the system call is executed directly by the OS kernel. When the execution of the system is completed, we record $\langle num, retval, side\_effect\rangle$ into our record file. The num and $retval$ are the system call number and the return value of this system call respectively. The $side\_effect$ contains other changes to the memory space of the executed program. For example, a `read` stores the read data into the buffer pointed to by the parameter of the system call. In this case, the content in the buffer is recorded after the system call's execution completes.

In the the replay mode, the plug-in takes as input a sequence of recorded system calls and only replays a prefix of the system call sequence as mentioned in Algorithm 7.1. When a system call $syscall_i$ is replayed, we first check whether $syscall_i$ is being dependent on by any other system calls that are to be *executed*. If yes, then the system call $syscall_i$ is both executed and replayed; that is we first execute the system to make the kernel state consistent and then use the recorded data to overwrite the return values and side-effects of the system call. If the system call $syscall_i$ is not being dependent on by any system calls that are to be *executed*, then the system call $syscall_i$ is only replayed with the recorded data without getting executed in the kernel.

There are some system calls that we did not fully implement. For these un-handled system calls, we simply execute them in both record mode and replay mode. Our techniques cannot deal with failures caused by these un-handled system calls.

7.2.4 Experiments

We report our experience in using the proposed technique to locate failure-inducing environment changes in real-life case studies. We present the results of our experiments in evaluating the effectiveness of our method.

Given a recorded system call sequence with length N, our technique only takes $log(N)$ executions to finish. In each following case study, our technique takes less than one minute to find the problematic system call. The record files are less than 10MB in all three cases.

7.2.4.1 Experience with MPD

Music Player Daemon (MPD) [7] is a server-side application which allows remote access for playing music. MPD comes with a client program, which is a console based jukebox commander. Clients may communicate with the server remotely over an intranet or over the Internet. To start MPD, a folder named “mpd” must be created under */var/run/*. In this folder, files including the pid file will be stored.

We run MPD in two different OS environments: (E) Ubuntu 9.04 with X service and (E') Ubuntu 8.04 in shell mode without X service. When running MPD in E, we were able to start and use the application. However, when running in E', we were unable to start the application and encountered the following error message:

```
Starting Music Player Daemon: could not open pid_file
"/var/run/mpd/pid" for writing: No such file or directory failed.
```

This program failure can be manually fixed by creating the folder and setting proper permission so that the pid_file can be created and MPD can be started. However, this does not fix the problem permanently because the bug occurs everytime the system is rebooted and it is unclear what is the root cause of this bug. Without using any debugging tool, it is difficult for the user to diagnose the root cause of this problem.

We applied the $Locate\_cause$ algorithm in Section 3.2 to locate the root cause of this problem. We collected the system call sequence when executing MPD in E. The sequence consists of 1038 system calls. We then run $Locate\_cause$ on the sequence of system calls, it returns the 7th system call as the root cause of this problem, which is: $open$. This system call suggests that MPD assumes the existence of the path */var/run/* so the application straight away creates a pid file under the directory */var/run/mpd/* everytime the system is rebooted. However, under the execution environment E', the path */var/run/* is not automatically mounted which caused the failed execution of MPD.

7.2.4.2 Experience with VSFTPD

VSFTPD is an FTP server daemon that runs on most current Unix-based operating systems. To allow anonymous user access in VSFTPD, one needs to create an anonymous FTP user and set appropriate permissions. According to the `VSFTPD` document, there are two ways to enable anonymous user access in the `VSFTPD` configuration file *vsftp.conf*:

1. set `anonymous_enable=YES` in vsftp.conf
2. create an user list file named `vsftpd.user_list` containing the “anonymous”, and then set `user_list_enable=YES` in vsftp.conf

However, to use the second way, the `local_enable` has to be set to `YES` in vsftp.conf.

In both environments E and E', Ubuntu version 8.04 is used. In the reference environment E, `anonymous_enable=YES` is set in vsftp.conf to allow anonymous users. In the environment E' where `VSFTPD` fails, `anonymous_enable` is set to `NO` and `user_list_enable` is set to `YES` in vsftp.conf. However, since `local_enable` is not set to `YES` in vsftp.conf, anonymous users are not allowed in environment E'. An anonymous access attempt in E' gets the following error message:

```
530 Permission denied. Login failed
```

This error message is too general and does not help much in figuring out the real cause of this problem.

We use our technique to locate the cause of this denied anonymous user access in E'. In this case study, the OS versions are the same in the two environments. The environment changes lie in the configuration used to enable anonymous user. We run the application in the successful environment E and used our tool to record the executed sequence of system calls. There were 26 system calls altogether that have been recorded. We then applied the $Locate\_cause$ algorithm in Section 3.2 and located the 2nd system call $read$, which caused the failed login of anonymous users. We further investigated from this system call. We found that it was the content of the file read by the system call $read$ that caused the failed execution of the program. We checked the file read by the system call and found out that it was the configuration file $vsftpd.conf$ which was read. By comparing the content of the two configuration files, one in E and one in E', we were able to figure out the difference in the configuration file that caused of the failed login with the anonymous user.

7.2.4.3 Experience with `Miniweb`

`Miniweb` [57] is an efficient light-weight web sever. `Miniweb` listens to a certain port of the OS for incoming HTTP requests. Either the port is specified by a command line option to `Miniweb` or the default port 80 is used. When the port used by `Miniweb` is already occupied, `Miniweb` fails to start with the following error message, which does not provide sufficient clue to locate the root cause of this problem:

```
Error starting instance #0 Failed to launch miniweb Shutting down
instance 0
```

In our experiment, we run `Miniweb` in two different environments: (E) Ubuntu 8.04 with the port used by `Miniweb` not occupied. (E') Ubuntu 8.04 with the port used by `Miniweb` occupied.

Using our technique, we record the system call sequences when `Miniweb` is executed in E. The successful execution of `Miniweb` in E returns a sequence of 16 system calls. Note that as `Miniweb` is a web-server, it will keep on running and generating a lot more system calls. Therefore, to successfully conduct this case study, we have to manually kill the process after it has started successfully. We then used this recorded sequence to locate the cause of failing to start `Miniweb` in E'. To tell whether `Miniweb` runs successfully, we check whether `Miniweb` is still running after it is started for a certain time (say 5 seconds). If `Miniweb` is still running after 5 seconds, we terminate `Miniweb` and deem the execution to be successful. Otherwise, `Miniweb` must have failed to start. Applying the Algorithm 7.1 in Section 3.2, we located a system call `socketcall` as the cause of the failure of `Miniweb` in E'. This `socketcall` system call is at the 7th place in the recorded sequence. The format of the `socketcall` system call is as follows:

```
int socketcall(int call, unsigned long *args);
```

where `call` determines which socket function to invoke, `args` is a pointer pointing to a block containing the actual arguments which are passed through to the appropriate `call`. For the socketcall we located, parameter `call` indicated that it is a *bind* operation. The `args` contained information about which port to bind and some other information. In the recorded (successful) execution, the `socketcall` system call returns 0 indicating that the operation is successful. However, in the failed execution, `socketcall` returned the error code 98, which corresponds to `EADDRINUSE` in `error.h`. `EADDRINUSE` means that the "address is already in used", which is root cause of the failed execution of `Miniweb` in environment E'.

7.2.5 Limitations

In this section, we assume that the program only use static libraries. The applicability of our technique is greatly reduced by this restriction. Allowing dynamic libraries give rise to a lot of challenges especially when different implementations of a dynamic libraries are used in different environments.

We only focused on single-thread deterministic program in this section. Therefore, if a bug is caused by non-deterministic signal and interrupt, our technique is not able to handle it.

The result of our technique is dependent on the closeness between the faulty environment and the reference environment. If the faulty environment is intended to be configured differently from the reference environment, the result from our technique may not be very useful.

Currently, only explicit system call dependency (e.g., the dependency between `open` and `read`) is handled. Handling of implicit system call dependency is not supported. In addition, our technique only functions when there is one system call that is responsible for the failure of the application.

Our debugging technique works at the system call interface layer. Therefore, we can only provide some suspicious system calls as the result of our technique. In some situations, the located system call does not provide enough detailed information to help debugging. For example, if a program uses one *read* system call to read a large faulty configuration file, our technique can only pinpoint this *read* system call. In this case, the user needs to look into the large configuration file to figure out the root cause of the program failure.

7.2.6 Summary

In this section, we have presented the *Semi-replay* method for locating the failure-inducing environment changes. Our approach takes in a program and two different OS environments where the execution of the program fails in one environment and passes in another. The proposed approach then locates a system call which is able to explain the failed execution of the program. Our approach captures the system call interface between an application and the underlying OS environment generated during the successful execution. The recorded system call sequence is then used to debug the failed execution of the application under another faulty OS environment. We have applied the proposed approach in three real-life case studies which give evidences to show the utility of our technique in debugging real bugs. The system call located by our approach can be used to easily identify the root cause of an error.

CHAPTER VIII

Conclusion

We conclude the thesis in this chapter. First, we summarize the contribution of the thesis. We then discuss some research directions that can be studied in the future.

8.1 Concluding Remarks

Software is always undergoing changes. Software is changed to adapt to new environment, meet new requirement and fix defects. In this process, maintaining the correctness of software is vitally important and extremely challenging. In this thesis, we target at reducing software errors in the context of software evolution. To this end, we build a series of semantic analysis based techniques ranging from requirement specification, to test-case generation and to regression debugging. Our techniques take full advantage of having previous program versions in analyzing evolving software, thereby achieving greater effectiveness and efficiency.

In particular, we have made the following contributions in this thesis.

- a test suite augmentation technique that generates test cases to expose the effects of program changes. It efficiently generates these test cases by gradually modifying test inputs to first reach program changes and then propagate the effects of program changes to program outputs.
- a regression debugging technique — DARWIN that semantically reasons about the root causes of regression errors in software evolution. Taking advantage of dynamic symbolic execution, DARWIN generates precise debugging results on real-life bugs.
- a novel symbolic execution technique at the level of program slice that naturally partitions

program paths based on their input-output relationship. We have also studied its application in dynamic path exploration, test-suite augmentation and regression debugging.

- a change contract language that allows the users to specify the intended effect of program changes. Change contracts containing the intention of users can provide great help in detecting and removing incorrect program changes.
- a debugging technique to localize defects in evolving software execution environment.

By fully utilizing the information from previous program versions, our proposed semantic analysis based techniques have advanced state-of-the-art software quality maintenance process in software evolution and have brought us closer to the ultimate goal of continuously providing quality software.

8.2 Future Work

One possible avenue of future work is to study efficient test-suite augmentation techniques for multiple program changes. We have studied the test-suite augmentation problem for a single change. Our technique could also be applied when multiple changes exist but do not interact with each other. When change interaction exists among multiple changes, test-suite augmentation becomes challenging. The effect of one change might be counteracted by another change. One change may take effects only when another change is exercised. Therefore, change interaction has to be considered for test-suite augmentation with multiple changes. However, the number of possible change interactions grows exponentially with the number of changes, without even considering different instances of changes due to loops. One possible solution to the combinatorial explosion would be to identify and only focus on "representative" change interactions.

Our proposed DARWIN debugging technique assumes that program requirement regarding the buggy input does not change. However, program requirements do change over time to adapt to new environment, meet new user need and so on. Therefore, a direct extension of DARWIN is to take evolving program requirements into consideration. Debugging evolving programs in presence of requirement evolution is a more challenging problem. When the program requirement is changed, the previous program version cannot serve as the perfect reference implementation anymore. Different program behavior between previous version and current version can be either buggy or intended for requirement changes. An unexpected behavior in the current program might be caused by code changes, or requirement changes, or even both. Different causes may lead to different debugging techniques. Therefore, figuring out the cause in terms of code change and/or requirement change is critical for debugging evolving programs with requirement changes. It is interesting to investigate the degree of help that our change contract language can provide in this regard.

Finally, there exist opportunities to improve the run-time checking of our change contract language. In our study, a unit test case has to be provided by the user during runtime checking. Ideally, these test cases could be automatically generated. However, existing test generation tools are not specifically designed to generate test cases for checking change contracts. We believe that these tools could be augmented by considering change contracts, thereby generating useful test cases for runtime checking. The users can then be relieved from the manual effort of creating test cases for runtime checking. Apart from runtime checking, change contract can also be checked statically. Similar to static checking of program contracts, previous and current program versions as well as change contracts can be translated into verification condition, which can be checked using SMT solvers. The efficiency of static checking is heavily dependent on SMT solvers. One way to ease the burden on the underlying SMT solver is to represent the unchanged program portions using un-interpreted functions as in differential symbolic execution [84]. In this way, static checking could be more efficient by focusing on changed program portions.

BIBLIOGRAPHY

[1] Apache webserver. `http://httpd.apache.org/`, 2009.

[2] ERESI. `http://www.eresi-project.org/`, 2009.

[3] LibPNG library. `http://www.libpng.org`, 2009.

[4] QEMU emulator. `http://www.qemu.org`, 2009.

[5] Savant webserver. `http://savant.sourceforge.net/info.html`, 2009.

[6] Valgrind. `http://valgrind.org/`, 2010.

[7] Music player daemon. `http://mpd.wikia.com/wiki/Music_Player_Daemon_Wiki`, 2011.

[8] Ubuntu bugs. `https://bugs.launchpad.net/ubuntu`, 2011.

[9] Apache Ant. `http://ant.apache.org/`, 2012.

[10] Apache software foundation. `http://www.apache.org`, 2012.

[11] OpenJML. `http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`, 2012.

[12] Software errors cost U.S. economy $59.5 billion annually. `http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm`, 2012.

[13] ABREU, R., ZOETEWEIJ, P., GOLSTEIJN, R., AND VAN GEMUND, A. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software 82*, 11 (2009).

[14] ACHARYA, M., AND ROBINSON, B. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the International Conference on Software Engineering* (2011).

[15] AGRAWAL, H., DEMILLO, R. A., AND SPAFFORD, E. H. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience 23*, 6 (1993).

[16] AGRAWAL, H., HORGAN, J., KRAUSER, E., AND LONDON, S. Incremental regression testing. In *Proceedings of International Conference on Software Maintenance* (1993).

[17] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation* (1990).

[18] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1996.

[19] ALLEN, F. E. Control flow analysis. In *Proceedings of a symposium on Compiler optimization* (1970).

[20] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. A differencing algorithm for object-oriented programs. In *Proceedings of the IEEE international conference on Automated software engineering* (2004).

[21] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering 14*, 1 (2007).

[22] ASHOK, B., JOY, J., LIANG, H., RAJAMANI, S. K., SRINIVASA, G., AND VANGALA, V. DebugAdvisor: a recommender system for debugging. In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (2009).

[23] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the USENIX conference on Operating systems design and implementation* (2010).

[24] BARNETT, M., CHANG, B.-Y. E., DELINE, R., JACOBS, B., AND LEINO, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the international conference on Formal Methods for Components and Objects* (2005).

[25] BARNETT, M., LEINO, K., AND SCHULTE, W. The Spec# programming system: An overview. *Proceedings of the international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (2005).

[26] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. Rwset: attacking path explosion in constraint-based test generation. In *Proceedings of the international conference on Tools and algorithms for the construction and analysis of systems* (2008).

[27] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium* (2007).

[28] BRUMMAYER, R., AND BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the international conference on Tools and algorithms for the construction and analysis of systems* (2009).

[29] BURDY, L., CHEON, Y., COK, D. R., ERNST, M. D., KINIRY, J. R., LEAVENS, G. T., LEINO, K. R. M., AND POLL, E. An overview of JML tools and applications. *Proceedings of the International Journal on Software Tools for Technology Transfer 7*, 3 (2005).

[30] BURGER, M., AND ZELLER, A. Replaying and isolating failing multi-object interactions. In *Proceedings of the international workshop on dynamic analysis* (2008).

[31] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX conference on Operating systems design and implementation* (2008).

[32] CHANDRA, S., AND CHEN, P. M. Whither generic recovery from application faults? a fault study using open-source software. In *Proceedings of the International Conference on Dependable Systems and Networks* (2000).

[33] CHANDRA, S., TORLAK, E., BARMAN, S., AND BODIK, R. Angelic debugging. In *Proceedings of the International Conference on Software Engineering* (2011).

[34] CHEN, Y.-F., ROSENBLUM, D. S., AND VO, K.-P. Testtube: a system for selective regression testing. In *Proceedings of the international conference on Software engineering* (1994).

[35] CHEON, Y., AND LEAVENS, G. T. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice* (2002).

[36] CHOI, J.-D., AND ZELLER, A. Isolating failure-inducing thread schedules. In *Proceedings of International Symposium on Software Testing and Analysis* (2002).

[37] CIUPA, I., AND LEITNER, A. Automatic testing based on design by contract. In *Proceedings of the Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World* (2005).

[38] CLAUSE, J., AND ORSO, A. A technique for enabling and supporting debugging of field failures. In *Proceedings of the international conference on Software Engineering* (2007).

[39] COK, D., AND KINIRY, J. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of Internationl Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (2005).

[40] CSALLNER, C., TILLMANN, N., AND SMARAGDAKIS, Y. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the international conference on Software engineering* (2008).

[41] DE MOURA, L., AND BJORNER, N. Z3: An efficient SMT solver. In *Proceedings of the international conference on tools and algorithms for the construction and analysis of systems* (2008).

[42] DETLEFS, D., LEINO, K., NELSON, G., AND SAXE, J. Extended static checking. Tech. rep., Compaq Systems Research Center, 1998.

[43] DO, H., ELBAUM, S., AND ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10*, 4 (2005).

[44] EASTWOOD, A. Firm fires shots at legacy systems. *Computing Canada 19*, 2 (1993).

[45] ELBAUM, S., MALISHEVSKY, A., AND ROTHERMEL, G. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis* (2000).

[46] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (1987).

[47] FLANAGAN, C., LEINO, K., LILLIBRIDGE, M., NELSON, G., SAXE, J., AND STATA, R. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2002).

[48] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the international conference on Computer aided verification* (2007).

[49] GODEFROID, P. Compositional dynamic test generation. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2007).

[50] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (2008).

[51] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).

[52] GODEFROID, P., LEVIN, M., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium* (2008).

[53] GYIMÓTHY, T., BESZÉDES, A., AND FORGÁCS, I. An efficient relevant slicing method for debugging. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (1999).

[54] HARROLD, M. J., GUPTA, R., AND SOFFA, M. L. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology 2*, 3 (1993).

[55] HOLMES, R., AND NOTKIN, D. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the International Conference on Software Engineering* (2011).

[56] HORWITZ, S. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1990).

[57] HUANG, S. Miniweb webserver. `http://miniweb.sourceforge.net/`, 2009.

[58] JACKSON, D., AND LADD, D. A. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings of the IEEE International Conference on Software Maintenance* (1994).

[59] JEFFREY, D., GUPTA, N., AND GUPTA, R. Fault localization using value replacement. In *Proceedings of the international symposium on Software testing and analysis* (2008).

[60] JIN, W., ORSO, A., AND XIE, T. Automated behavioral regression testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation* (2010).

[61] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM international Conference on Automated software engineering* (2005).

[62] JONES, J. A., HARROLD, M. J., AND STASKO, J. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering* (2002).

[63] JOSE, M., AND MAJUMDAR, R. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (2011).

[64] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming* (2001).

[65] KIM, M., AND NOTKIN, D. Discovering and representing systematic code changes. In *Proceedings of the ACM/IEEE International Conference on Software Engineering* (2009).

[66] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976).

[67] KOREL, B., AND LASKI, J. W. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988).

[68] LEAVENS, G., BAKER, A., AND RUBY, C. JML: A notation for detailed design. *Kluwer International Series in Engineering and Computer Science* (1999).

[69] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J. R., AND CHALIN, P. *JML Reference Manual*, 2012. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`.

[70] LEVESON, N. G., AND TURNER, C. S. An investigation of the therac-25 accidents. *IEEE Computer 26*, 7 (1993).

[71] LIBLIT, B. *Cooperative Bug Isolation*. PhD thesis, UC Berkeley, 2005.

[72] LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A., AND JORDAN, M. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).

[73] LIU, C., YAN, X., FEI, L., HAN, J., AND MIDKIFF, S. P. SOBER: statistical model-based bug localization. In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (2005).

[74] LOH, A., AND KIM, M. LSdiff: a program differencing tool to identify systematic structural differences. In *Proceedings of the ACM/IEEE International Conference on Software Engineering* (2010).

[75] MARINESCU, P. D., AND CADAR, C. make test-zesti: a symbolic execution solution for improving regression testing. In *Proceedings of the International Conference on Software Engineering* (2012).

[76] MCMILLAN, K. Lazy annotation for program testing and verification. In *Proceedings of the international conference on Computer aided verification* (2010).

[77] MEYER, B. *EIFFEL: The language and environment.* Prentice hall press, 1991.

[78] MEYER, B. Applying "Design by Contract". *IEEE Computer 25*, 10 (1992).

[79] MISHERGHI, G., AND SU, Z. HDD: hierarchical delta debugging. In *Proceedings of the IEEE international conference on Software engineering* (2006).

[80] NGUYEN, H. A., NGUYEN, T. T., NGUYEN, H. V., AND NGUYEN, T. N. iDiff: Interaction-based program differencing tool. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (2011).

[81] ORSO, A., APIWATTANAPONG, T., AND HARROLD, M. J. Leveraging field data for impact analysis and regression testing. In *Proceedings of the ACM SIGSOFT symposium and the European conference on Foundations of software engineering* (2003).

[82] ORSO, A., JOSHI, S., BURGER, M., AND ZELLER, A. Isolating relevant component interactions with jinsi. In *Proceedings of the international workshop on Dynamic systems analysis* (2006).

[83] PEARL, J. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann Publishers Inc., 1988.

[84] PERSON, S., DWYER, M., ELBAUM, S., AND PASAREANU, C. Differential symbolic execution. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2008).

[85] PERSON, S., YANG, G., RUNGTA, N., AND KHURSHID, S. Directed incremental symbolic execution. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (2011).

[86] QI, D., ROYCHOUDHURY, A., AND LIANG, Z. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010).

[87] RANISE, S., AND TINELLI, C. The SMT-LIB format: An initial proposal. In *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning* (2003).

[88] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2004).

[89] RENIERIS, M., AND REISS, S. P. Fault localization with nearest neighbor queries. In *Proceedings of IEEE International Conference Automated Software Engineering* (2003).

[90] RICHARDSON, D., AND THOMPSON, M. The relay model of error detection and its application. In *Proceedings of the workshop Software Testing, Verification, and Analysis* (1988).

[91] ROTHERMEL, G., AND HARROLD, M. J. A safe efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology 6*, 2 (1997).

[92] SANTELICES, R., CHITTIMALLI, P., APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. Test-suite augmentation for evolving software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (2008).

[93] SANTELICES, R., CHITTIMALLI, P. K., APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. Test-suite augmentation for evolving software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (2008).

[94] SANTELICES, R., AND HARROLD, M. J. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the international symposium on Software testing and analysis* (2010).

[95] SANTELICES, R., AND HARROLD, M. J. Applying aggressive propagation-based strategies for testing changes. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation* (2011).

[96] SANTELICES, R., JONES, J. A., YANBING, Y., AND HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In *Proceedings of the IEEE International Conference on Software Engineering* (2009).

[97] SEN, K., MARINOV, D., AND AGHA, G. CUTE: a concolic unit testing engine for C. In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (2005).

[98] SILLITO, J., MURPHY, G., AND DE VOLDER, K. Questions programmers ask during software evolution tasks. In *Proceedings of the ACM SIGSOFT international symposium on Foundations of software engineering* (2006).

[99] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security.* (2008).

[100] SRIDHARAN, M., FINK, S. J., AND BODIK, R. Thin slicing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (2007).

[101] SRIVASTAVA, A., AND THIAGARAJAN, J. Effectively prioritizing tests in development environment. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis* (2002).

[102] SU, Y., AND FLINN, J. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of the USENIX Annual technical conference* (2009).

[103] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of ACM SIGOPS symposium on Operating systems principles* (2007).

[104] SUMNER, W. N., BAO, T., AND ZHANG, X. Selecting peers for execution comparison. In *Proceedings of the International Symposium on Software Testing and Analysis* (2011).

[105] TANEJA, K., XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Guided path exploration for regression test generation. In *Proceedings of the International Conference on Software Engineering, New Ideas and Emerging Results* (2009).

[106] TSCHANNEN, J., FURIA, C. A., NORDIO, M., AND MEYER, B. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Proceedings of the International Conference on Software Engineering and Formal Methods* (2011).

[107] VAN SCHAIK, W. Pngsuite. http://www.schaik.com/pngsuite/, 1998.

[108] VOAS, J. M. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering 18*, 8 (1992).

[109] WANG, T., AND ROYCHOUDHURY, A. JSlice: A dynamic slicing tool for Java programs. National University of Singapore, `http://jslice.sourceforge.net`.

[110] WANG, T., AND ROYCHOUDHURY, A. Automated path generation for software fault localization. In *Proceedings of IEEE International Conference Automated Software Engineering* (2005).

[111] WANG, T., AND ROYCHOUDHURY, A. Hierarchical dynamic slicing. In *Proceedings of the International Symposium on Software Testing and Analysis* (2007).

[112] WANG, T., AND ROYCHOUDHURY, A. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems 30*, 2 (2008).

[113] WEI, Y., PEI, Y., FURIA, C. A., SILVA, L. S., BUCHHOLZ, S., MEYER, B., AND ZELLER, A. Automated fixing of programs with contracts. In *Proceedings of the international symposium on Software testing and analysis* (2010).

[114] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984).

[115] WONG, W. E., HORGAN, J. R., LONDON, S., AND MATHUR, A. P. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the international conference on Software engineering* (1995).

[116] XIE, Y., CHOU, A., AND ENGLER, D. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (2003).

[117] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation* (2008).

[118] XIN, B., AND ZHANG, X. Memory slicing. In *Proceedings of the international symposium on Software testing and analysis* (2009).

[119] XU, Z., COHEN, M. B., AND ROTHERMEL, G. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proceedings of the annual conference on Genetic and evolutionary computation* (2010).

[120] XU, Z., KIM, Y., KIM, M., ROTHERMEL, G., AND COHEN, M. B. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT international symposium on Foundations of software engineering* (2010).

[121] XU, Z., AND ROTHERMEL, G. Directed test suite augmentation. In *Proceedings of the Asia-Pacific Software Engineering Conference* (2009).

[122] YIN, Z., YUAN, D., ZHOU, Y., PASUPATHY, S., AND BAIRAVASUNDARAM, L. N. How do fixes become bugs? In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (2011).

[123] ZELLER, A. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the European software engineering conference and ACM SIGSOFT international symposium on Foundations of software engineering* (1999).

[124] ZELLER, A. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT symposium on Foundations of software engineering* (2002).

[125] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002).

[126] Zhang, X., Gupta, N., and Gupta, R. Locating faults through automated predicate switching. In *Proceedings of the ACM/IEEE International Conference on Software Engineering* (2006).

[127] Zhang, X., Gupta, N., and Gupta, R. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2006).