POST-MORTEM DYNAMIC ANALYSIS FOR SOFTWARE DEBUGGING

WANG TAO

(B.Science, Fudan University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2007

ACKNOWLEDGEMENTS

There are lots of people whom I would like to thank for a variety of reasons. I sincerely acknowledge all those whom I mention, and apology to anybody whom I might have forgotten.

First of all, I am deeply grateful to my supervisor, Dr. Abhik Roychoudhury, for his valuable advice and guidance. I sincerely thank him for introducing me to the exciting area of automated software debugging. During the five years of my graduate study, Dr. Abhik Roychoudhury has given me immense support both in academics and life, and has helped me stay on the track of doing research.

I express my sincere thanks to Dr. Chin Wei Ngan and Dr. Dong Jin Song for their valuable suggestions and comments on my research works. I would also like to thank Dr. Satish Chandra for taking time out of his schedule and agreeing to be my external examiner.

I have special thanks to my parents and family for their love and encouragement. They have been very supportive and encouraging throughout my graduate studies.

I really appreciate the support and friendship from my fiends inside and outside the university. I thank my friends Jing Cui, Liang Guo, Lei Ju, Yu Pan, Andrew Santosa, Mihail Asavoae, Xianfeng Li, Shanshan Liu, Xiaoyan Yang, Dan Lin, Yunyan Wang and Zhi Zhou to name a few.

I would like to thank the National University of Singapore for funding me with research scholarship. My thanks also go to administrative staffs in School of Computing, National University of Singapore for their supports during my study. This work presented in this thesis was partially supported by a research grant from the Agency of Science, Technology and Research (A*STAR) under Public Sector Funding.

TABLE OF CONTENTS

SUMMARY

With the development of computer hardware, modern software becomes more and more complex, and it becomes more and more difficult to debug software. One reason for this is that debugging usually involves too much programmers' labor and wisdom. Consequently, it is important to develop debugging approaches and tools which can help programmers locate errors in software. In this thesis, we study the state-of-art debugging techniques, and address the challenge to make these techniques applicable for debugging realistic applications.

First, we study dynamic slicing, a well-known technique for program analysis, debugging and understanding. Given a program P and input I, dynamic slicing finds all program statements which directly/indirectly affect the values of some variables' occurrences when P is executed with I. In this thesis, we develop a dynamic slicing method for Java programs, and implement a slicing tool which has been publicly released. Our technique proceeds by backwards traversal of the bytecode trace produced by an input I in a given program P. Since such traces can be huge, we use results from data compression to compactly represent bytecode traces. We show how dynamic slicing algorithms can directly traverse our compact bytecode traces without resorting to costly decompression. We also extend our dynamic slicing algorithm to perform "relevant slicing". The resultant slices can be used to explain omission errors that is, why some events did not happen during program execution.

Dynamic slicing reports the slice to the programmer. However, the reported slice is often too large to be inspected by the programmer. We address this deficiency by hierarchically applying dynamic slicing at various levels of granularity. The basic observation is to divide a program execution trace into "phases", with data/control

dependencies inside each phase being suppressed. Only the inter-phase dependencies are presented to the programmer. The programmer then zooms into one of these phases which is further divided into sub-phases and analyzed.

Apart from dynamic slicing, we also study test based fault localization techniques, which proceed by comparing a "failing" execution run (*i.e.* a run which exhibits an unexpected behavior) with a "successful" run (*i.e.* a run which does not exhibit the unexpected behavior). An issue here is how to generate or choose a "suitable" successful run; this task is often left to the programmer. In this thesis, we propose a control flow based difference metric for automating this step. The difference metric takes into account the sequence of statement instances (and not just the set of these instances) executed in the two runs, by locating branch instances with similar contexts but different outcomes in the failing and the successful runs. Our method automatically returns a successful program run which is close to the failing run in terms of the difference metric, by either (a) constructing a feasible successful run, or (b) choosing a successful run from a pool of available successful runs.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the last decades, computer software become more and more complex, and software development becomes increasingly difficult. Many innovative concepts and techniques, such as Object Oriented Programming (OOP), the Integrated Development Environment (IDE), design pattern [35], have been proposed and used to ease the tasks of software design and implementation. Unfortunately, almost any software module of moderate size will contain bugs. This is not because programmers are careless or irresponsible, but because humans have only limited ability to manage the complexity of modern software.

1.1 Problem Definition

The task of software debugging is an extremely time-consuming and laborious phase of software development. An introspective survey [41] on this topic mentions the following: "Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem.. over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools." So, we need automated tools to detect the root cause from the observable error! Currently such tools are missing for real-life programming languages like C, C++, Java.

In this thesis, we have tried to address both of these issues - (i) bridging the chasm between software error cause and observable errors, and (ii) building automated debugging tools to do so.

1.2 Methods Developed

Traditionally, when a programmer tries to locate the error in a program, he/she typically repeats the following two steps until the error is found:

1. get clues and hypothesize a location in the program as the error.

2. confirm that the location is indeed the error.

Traditionally, the programmer has to *manually* perform both steps based on his/her experience and understanding of the program, with little help from existing debugging tools. This makes debugging difficult and time consuming. As a result, it is important to develop new tools which can increase the degree of automation in the task of debugging.

Over the last few decades, the research community has proposed many program analysis techniques such as type systems, model checking and program slicing [59, 33, 43, 18, 107, 106, 49, 32, 60, 81] for the purpose of debugging. These techniques *automatically* analyze the program behaviors and identify some potentially erroneous statements. Instead of blindly searching through the program or the execution run, the programmer can start debugging from these reported statements, which are likely to be related to the real error. In other words, we could develop novel tools to help programmers in the first step of debugging, *i.e.* identifying potential erroneous locations in the program.

The second step of debugging, confirming the error, is typically left as a manual step to the programmer. This is because, an important challenge in automating the second step is to characterize the desired program behaviors. The typical approach for describing correct program behaviors requires programmers to write specifications. Unfortunately, many programmers are reluctant to provide such specifications.

Consequently, the core of (semi-) automatic debugging is to apply program analysis techniques to automatically identify potential erroneous statements of a buggy

program, so that we can Program analysis techniques are divided into two categories: static and dynamic. Static analysis is usually performed on the *source code* without actually executing programs; dynamic analysis is performed on the *execution runs* by executing programs. In general, dynamic analysis is more useful for software debugging than static analysis, because of the following three reasons:

- Static analysis considers all inputs of the program, but dynamic analysis only considers one or a few inputs. Clearly, dynamic analysis naturally supports the task of debugging via running the program with selected inputs.

- Due to the conservative nature of the auxiliary program analysis methods used for debugging (such as points-to analysis), the bug-reports constructed by static analysis based debugging methods are often very large. Most importantly, these results often contain false positives, *i.e.* wrongly identify some program statements as faulty.

- Some static analysis methods (such as model checking) proceed by constructing the evidence (such as an execution run) which violates some given properties. However, this contrasts with the typical debugging process, where the programmer has an execution run and tries to find the properties which the execution violates.

In recent years, a number of dynamic analysis approaches [6, 57, 86, 20, 68, 42, 112, 105, 114, 118, 113, 16] have been proposed in order to ease the task of software debugging. Among existing techniques, dynamic slicing [6, 57] is a well-known one for software debugging and comprehension.

Dynamic slicing analyzes the execution run with unexpected behaviors, and returns a *dynamic slice*. The dynamic slice includes the closure of dynamic control and data dependencies from an "observable error". Such a slice may capture the faulty

statements, with the explanation of the cause-effect relations between the faulty statements and the "observable error" through dependencies. Roughly speaking, dynamic slicing works as follows. Given a program P, an input I and an "observable error", dynamic slicing can be used to find out statements of P executed under input I which can potentially be responsible for the error (via control or data flow). Typically, the "observable error" is specified as a *slicing criterion* (l, v) — a variable v and the location l of a statement instance in the execution. Thus, if the value of variable v at location l is "unexpected", we perform slicing w.r.t. the criterion (l, v). The resultant slice can be inspected to explain the reason for the unexpected value.

Dynamic slicing has been studied for about two decades, and a lot of research has been conducted in this area [4, 6, 7, 57, 58, 71, 51, 98, 102, 105, 104, 109]. In this thesis, we present an infrastructure for dynamic slicing of Java programs. Our method operates on bytecode traces. First, the bytecode stream corresponding to an execution trace of a Java program for a given input is collected. We then perform a backward traversal of the bytecode trace to compute dynamic data and control dependencies on-the-fly. The slice is the closure of the dynamic control and data dependencies detected.

Our dynamic slicing method/tool operates at the Java bytecode level, since the slice computation may involve looking inside library methods and the source code of libraries may not always be available. In addition, dynamic slicing always requires run-time information of the execution run. It is easy to collect such information by modifying a Java Virtual Machine, which operates at the bytecode level. The resultant slice at the bytecode level can be easily translated back to the source code level with the help of information available in Java class files.

The dynamic slicing technique is presented w.r.t. bytecodes for Java in this thesis. However, the general principles and methodology can also be applied to the

Common Language Infrastructure (CLI) for the Microsoft .NET Framework. During compilation of .NET programming languages, the source code is translated into *Common Intermediate Language*(CIL) code, and the CIL is then executed by a virtual machine. Because of the similarity between the bytecode for Java and the CIL for .NET, the approaches in this thesis can be implemented in the CLI, and support debugging multiple .NET programming languages, such as C#, Visual Basic .NET and C++/CLI.

Based on the dynamic slicing infrastructure, we conduct research on dynamic slicing. In particular, we find that previous research mainly focuses on the accuracy of the slicing algorithm and the application of dynamic slicing. However, there remain the following problems which have not been thoroughly studied.

- **Trace Representation**. Dynamic slicing methods typically involve traversal of the execution trace. This traversal may be used to pre-compute a dynamic dependence graph or the dynamic dependencies can be computed on demand during trace traversal. The trace traversal can be performed either forwards or backwards. Forward traversal based dynamic slicing method does not involve storage of the trace, but it is not goal-directed (w.r.t. the slicing criterion). On the other hand, backward traversal based dynamic slicing method is goal-directed. However, the traces tend to be huge in practice; [116] reports experiences in dynamic slicing programs like `gcc` and `perl` where the execution trace runs into *several hundred million instructions*. It might be inefficient to perform post-mortem analysis over such huge traces. Consequently, the representation of execution traces is important for dynamic slicing. It is useful to develop a compact representation for execution traces which capture both control flow and memory reference information. This compact trace should be generated *on-the-fly* during program execution. Other researchers have also conducted research on the topic of lossless trace compression [27, 114]. We compare our

approach with these works in Chapter 7.

- **Execution Omission Errors**. Dynamic slicing tries to capture the faulty statements by analyzing *actual* control/data dependencies between executed statements. However, it does not consider "Execution Omission" errors, where the execution of certain statements is wrongly omitted. Consequently, the dynamic slice may not include all statements which are responsible for the error, and the slice may mislead the programmer. To fill this caveat, relevant slicing was introduced in [7, 40]. However, previous relevant slicing algorithms may either wrongly ignore some useful statements or include some unnecessary statements, and they were not experimentally evaluated for real programs.

- **Slice Comprehension**. Traditionally, the dynamic slice, *i.e.* the result of dynamic slicing, is reported as a flat set of statements to a programmer for debugging and comprehension. Unfortunately, for most real programs, the dynamic slice is often too large for humans to inspect and comprehend. So, it is important to either prune the dynamic slice or develop innovative tools to help a programmer understand a large dynamic slice.

Dynamic slicing is a powerful debugging technique, by guiding a programmer to systematically explore important dependencies to locate the error. However, dynamic slicing is believed to be an expensive technique, because it requires collecting the entire control flow and data flow of an execution. This has been validated in several research reports [27, 104, 114, 115].

Recently, researchers have proposed *test based fault localization* techniques [22, 38, 51, 83, 86, 87, 39, 103, 110] for software debugging. These techniques often provide cheap ways to analyze the program execution runs, and discover potentially erroneous statements. Such heuristics sometimes work very well for debugging, by pinpointing the error. When the heuristics are not useful, we can then turn to the general purpose

methods like dynamic slicing.

Test based fault localization techniques consider certain execution traces of the buggy program itself as representative correct behaviors. These techniques proceed by comparing the failing execution run with some successful run (a run which does not demonstrate the error). The difference between the failing and successful execution runs is likely to be related to the error. This is because, if we change all the differences from the failing run, the failing run will become a successful run, and the observable error will disappear.

A lot of research has been conducted in this topic. However, the following problem has not been thoroughly studied.

- **Availability of the Successful Run**. Most of the research in this line of work has focused on how to compare the successful and failing execution runs. They exploit the successful run to find out points in the failing run which may be responsible for the error and for each of those points which variables may be responsible for the error. However, an issue here is the generation or selection of a "suitable" successful run. This task is often left to the programmer. Clearly, this will increase the programmer's burden, and should be automated.

1.3 Summary of Contributions

In this thesis, we study dynamic analysis techniques for software debugging. Our goal is to improve debugging tools with a higher degree of usability and automation. The contributions of this thesis can be summarized as follows:

- In this thesis, we present an infrastructure for dynamic slicing of Java programs. We have built a dynamic slicing tool *JSlice* based on this infrastructure, and released this tool as open source software at `http://jslice.sourceforge.net/`. To the best of our knowledge, ours is the first dynamic slicing tool for

Java programs. It supports Java program debugging via testing. Test cases which fail can be further analyzed via dynamic slicing in JSlice, thereby aiding the programmer to locate the error cause. Since October 2006, more than 80 users from more than 20 different countries have registered and used our tool. Note that our software is open-source and not locked to a particular machine. So, typically only one person from an organization might be registering with us to obtain the open-source software. Our user-base includes (1) university researchers (e.g. from CMU, King's College, NTU), and (2) developers (e.g. from Nokia, Agitar Software), and (3) industrial researchers (e.g. from IBM Watson, NEC Research).

- This thesis presents a space efficient representation of the trace for a Java program execution. This compressed trace is constructed on-the-fly during program execution. The dynamic slicer then performs backward traversal of this compressed trace *directly* to retrieve data/control dependencies. That is, slicing does not involve costly trace decompression. In addition, the compressed trace representation can be used to represent program traces for other post-mortem analysis.

- We enhance our dynamic slicing algorithm to capture "Execution Omission" errors via "*relevant slicing*" [7, 40], so that the resultant slice has less chance to mislead the programmer for debugging. We show that our definition of *relevant slice* is more accurate that previous ones [7, 40]. Our relevant slicing algorithm also operates *directly* on the compressed bytecode traces, as our dynamic slicing algorithm.

- We propose *hierarchical dynamic slicing* to help a programmer understand a large dynamic slice. The human programmer is gradually exposed to a slice in a hierarchical fashion, rather than having to inspect the large slice after it is

computed. The basic observation is to divide a program execution trace into "phases", with data/control dependencies inside each phase being suppressed. Only the inter-phase dependencies are presented to the programmer. The programmer examines these inter-phase dependencies to find out the phase which is responsible for the error. This phase is then further divided into sub-phases and analyzed.

- We propose a control-flow based difference metric to compare execution runs (i.e. data flow in the runs is not taken into account). We take the view that the difference between two runs can be summarized by the sequence of comparable branch statement instances which are evaluated differently in the two runs. This difference metric is used to (a) generate a feasible successful run, or (b) choose a suitable successful run from a pool of successful runs. The generated/chosen successful run is close to, that is, has *little difference* with, the failing run. We return the sequence of branch instances evaluated differently in the failing run and the successful run as bug report.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. The next chapter presents an overview of the approach taken in this thesis. Chapter 3 presents a dynamic slicing infrastructure which works on a compact trace representation. Chapter 4 discusses the relevant slicing, which extends dynamic slicing to capture execution omission errors. Chapter 5 explains how to guide a programmer hierarchically explore and understand a large slice. Chapter 6 presents our test based fault localization technique which detects the bug by comparing execution runs. Chapter 7 discusses the related works. The conclusion and future work appear in Chapter 8.

CHAPTER 2

OVERVIEW

In this chapter, we provide the background in the area of software debugging, and present an overview of the approaches taken in this thesis. First, we describe the typical steps in the task of debugging, and show how these steps can be automated by using dynamic slicing and test based fault localization techniques. Next, we present an infrastructure for dynamic slicing of Java programs. Then we describe an overview of the approaches which address existing challenges of dynamic slicing, and show how these approaches are incorporated in the slicing infrastructure. Finally, we briefly introduce the test based fault localization technique proposed in this thesis.

2.1 Background

Debugging is a difficult and time consuming task, because the erroneous statements are usually far away from the location where some unexpected behavior is exhibited and observed. That is, the erroneous statements often *indirectly* affect the observable error. Now, let us assume that a program P is executed with a test input I, and the program does not behave as it is supposed to. How does a developer identify the erroneous statements in the program code?

Traditionally, the developer debugs a program by examining a series of program states, where these states are generated by executing program P with input I. The examination process continues until the developer finds a location l of the execution, where the program state before l is correct but the program state after l is wrong. The statements at the location l are indeed the buggy statements which should be fixed.

However, the program execution typically generates a large number of states, and each state consists of a lot of variables. It is impossible to manually examine all the states for debugging. In practice, developers hypothesize some locations which are likely to be the error, and only examine program states around these locations. In general, the debugging process can be summarized as:

1. hypothesize a location l which is likely to be the error, according the developer's understanding of the program,

2. examine the states before/after l to determine whether the location l is indeed the error.

During the debugging process, the two steps are repeated until the developer detects the erroneous statements. The standard debugging tools (such as GDB) provide breakpoints, traces and other facilities, so that the developer can easily examine the program state (*i.e.* the second step in debugging). However, the developer has to manually perform the first step.

Automated debugging techniques are proposed to increase the degree of automation in the first step of debugging, by automatically providing suspicious locations to the developer. In this thesis, we discuss dynamic slicing and test based fault localization techniques in this area.

Dynamic slicing detects the suspicious locations by analyzing the dependency chains between the erroneous statements and the observable error. This is because, the erroneous statements affect the observable error via control flow and/or data flow. This is captured by dynamic control and/or data dependencies. In fact, when a developer manually debugs, he/she will (manually) analyze the dependencies to understand how the observable error is produced, thereby locating the real error. Dynamic slicing automates this analysis, by computing the closure of the dynamic control/data dependencies from the observable error. Statements which do not appear

in the dependency chains do not (transitively) affect the observable error. These statements are unlikely to be responsible for the observable error, and the dynamic slicing technique ignores these statements for inspection.

Static slicing can also be used for software debugging, by analyzing static control/data dependencies inside the program. However, we believe that dynamic slicing is more suitable for the purpose of debugging. This is because, static slicing considers all possible program inputs, and relies on auxiliary program analysis methods (such as points-to analysis). Thus, static slices often contains more false positives than dynamic slices. Additionally, dynamic slicing focuses on a particular execution run (the one in which an error is observed). This naturally supports the task of debugging via running the program with selected inputs.

Test based fault localization techniques take another approach to detect the suspicious locations. That is, these techniques compare the behaviors between failing runs (*i.e.* execution runs with unexpected behaviors) and successful runs (*i.e.* execution runs without unexpected behaviors). The difference *diff* is reported to the developer as suspicious. This is because, through the comparison, we can deduce that the appearance of the behavior *diff* is correlated with the observable error, *i.e.* the behavior *diff* appears/disappears at the same time with the observable error. Because of this correlation, the difference *diff* might be helpful to locate the error.

Now, we use some real examples to explain how dynamic slicing and test based fault localization techniques work.

2.1.1 Background on Dynamic Slicing

We first use an example to explain how dynamic slicing works for debugging. Figure 2.1 shows a simplified program fragment from the Apache JMeter utility [1]. There is an error at line 7 of Figure 2.1, which should be $savedValue$ `= "null"`. With the input $runningVersion = false$, the execution trace of the program fragment is

$1^1, 2^2, 3^3, 7^4, 9^5$. The trace is given as a sequence of line numbers. The superscript here is used to differentiate multiple executions of the same line, although it is not meaningful in this example.

```
1.  void setRunningVersion (boolean runningVersion) {
2.    this.runningVersion = runningVersion;
3.    if ( runningVersion ) {
4.      savedValue = value;
5.    }
6.    else {
7.      savedValue = "";
8.    }
9.    System.out.println(savedValue);
10. }
```

Figure 2.1: Example: A fragment from the Apache JMeter utility to explain dynamic slicing.

When the execution finishes, the programmer finds that the output of this program is the empty string, and deems this as an error. He/She can then specify $< 9^5, savedValue >$ as the slicing criterion, and perform dynamic slicing for debugging. The resultant dynamic slice includes lines 1, 3, 7 and 9. Line 7 is included in the dynamic slice, because its occurrence 7^4 defines the variable $savedValue$, and directly affects the slicing criterion. Additionally, line 3 is included in the dynamic slice, because its occurrence 3^3 decides whether 7^4 will be executed, and indirectly affects the slicing criterion. Line 1 is also be included, since 3^3 is both dynamically control and data dependent on 1^1. 3^3 is dynamically control dependent on 1^1 because line 1 represents the method head, and 3^3 is dynamically data dependent on 1^1 because of the variable $runningVersion$.

The statements in the dynamic slice explain how the incorrect value of variable $savedValue$ is produced. Other statements, such as lines 2 and 4 in Figure 2.1, are irrelevant to the computation of the observable error. So, the programmer can focus on the dynamic slice to locate the error, instead of inspecting the code of the whole program.

In general, the dynamic slice includes the closure of dynamic control and data dependencies from the slicing criterion. The dynamic control and data dependencies are defined as follows, where β represents an occurrence of the statement $stmt(\beta)$.

Definition 2.1. Dynamic Control Dependency *The statement occurrence β is dynamically control dependent on an earlier statement occurrence β' iff.*

1. *$stmt(\beta)$ is statically control dependent*[1] *on $stmt(\beta')$, and*

2. *$\nexists \beta''$ between β and β' where $stmt(\beta)$ is statically control dependent on $stmt(\beta'')$.*

Definition 2.2. Dynamic Data Dependency *The statement occurrence β is dynamically data dependent on an earlier statement occurrence β' iff.*

1. *β uses a variable v, and*

2. *β' defines the same variable v, and*

3. *the variable v is not defined by any statement occurrence between β and β'.*

Formally, a dynamic slice can be defined using the *Dynamic Dependence Graph* (DDG) [6]. The DDG captures dynamic control and data dependencies between statement occurrences during program execution. Each node of the DDG represents one particular occurrence of a statement; edges represent dynamic data and control dependencies. As an example, Figure 2.2 shows the Dynamic Dependence Graph (DDG) for the program in Figure 2.1 with input $runningVersion = false$.

The dynamic slice is then defined as follows.

Definition 2.3. Dynamic Slice *for a slicing criterion consists of all statements whose occurrence nodes can be reached from the node(s) representing the slicing criterion in the DDG.*

[1] *The static control dependence is defined in [31] according to the post-dominators over the control flow graph.*

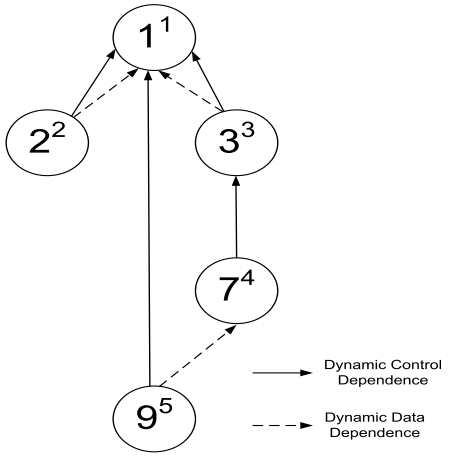


Figure 2.2: The Dynamic Dependence Graph (DDG) for the program in Figure 2.1 with input $runningVersion = false$.

The Dynamic Dependence Graph and the Dynamic Slice here are defined at the level of statement. These definitions can be easily generalized to other forms of program representation, such as Java bytecode.

2.1.2 Background on Test Based Fault Localization

We now use an example to explain how test based fault localization technique works for debugging. The literature has proposed many approaches to compare different characteristics of failing runs against successful runs.

In this thesis, we have proposed a difference metric to measure the "similarity" between execution runs of a program for the purpose of debugging. The metric considers branch instances with similar contexts but different outcomes in two execution runs, because these branch instances may be related to the cause of error. When these branch instances are evaluated differently from the failing run, certain faulty statements may not be executed — leading to disappearance of the observable error in the successful run.

Figure 2.3 shows a program fragment from a faulty version of `replace` program in the Siemens benchmark suite [47, 89] — simplified here for illustration. There is a bug

in this program fragment, where the bug fix lies in strengthening the condition in line 3 to `if ((m >= 0) && (lastm != m))`. This piece of code changes all substrings s_1 in string `lin` matching a pattern to another substring s_2, where variable `i` represents the index to the first un-processed character in string `lin`, variable `m` represents the index to the end of a matched substring s_1 in string `lin`, and variable `lastm` records variable `m` in last loop iterations. At the ith iteration, if variable `m` is not changed at line 2, line 3 is wrongly evaluated to true, and substring s_2 is wrongly returned as output, deemed by programmer as an observable "error". The execution of the ith iteration of this failing run π_f could follow path $1^1, 2^2, 3^3, 4^4, 5^5, 7^6, 8^7, 9^8$. In this case, a successful run π_s whose ith iteration follows path $1^1, 2^2, 3^3, 7^4, 8^5, 9^6$ can be useful for error localization. By comparing π_f with π_s, we see that only the branch at line 3 is evaluated differently. Indeed this is the erroneous statement in this example, and was pinpointed by our method in the experiment. For programs whose erroneous statement is not a branch, we will report the nearest branch for locating the error.

```
1.  while (lin[i] != ENDSTR) {
2.    m = ......
3.    if (m >= 0) {
4.      ......
5.      lastm = m;
6.    }
7.    if ((m == -1) || (m == i)) {
8.      ......
9.      i = i + 1;
10.   }
11.   else
12.     i = m;
13. }
14. ......
```

Figure 2.3: An example program fragment to explain test based fault localization.

2.2 Dynamic Slicing

Dynamic slicing helps the developer systematically explore the dynamic dependencies which are related to the observable error. In this section, we discuss a dynamic slicing

framework for Java programs, and briefly present the approaches taken in this thesis to address three deficiencies of dynamic slicing.

Figure 2.4 presents our infrastructure for dynamic slicing of Java programs. The infrastructure consists of two parts:

- a front end, which is the user interface,
- a back end, which collects traces and performs dynamic slicing.

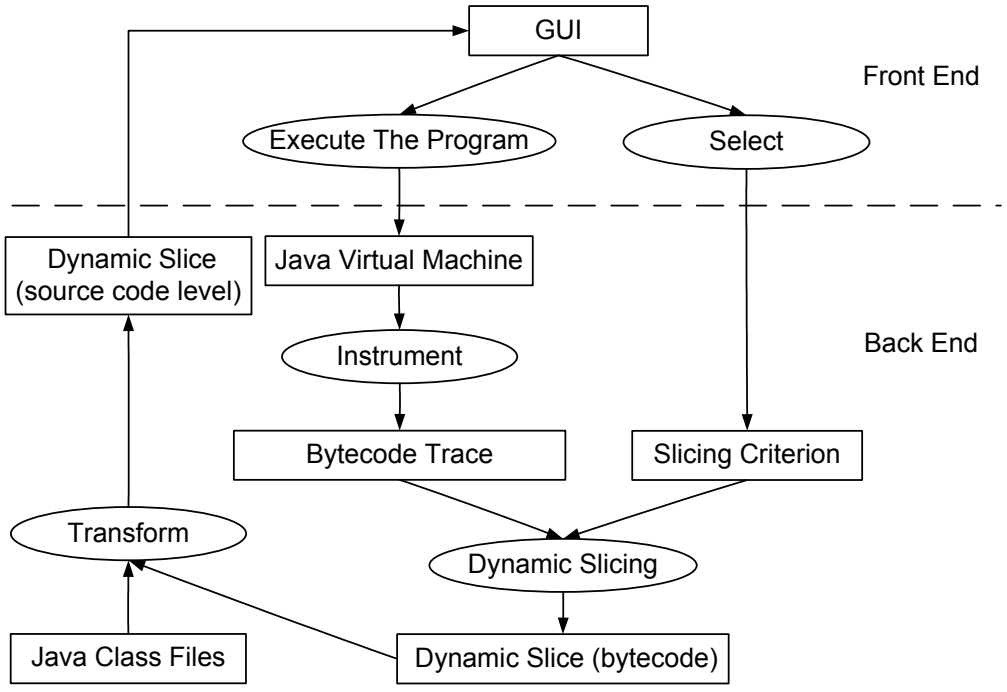


Figure 2.4: An infrastructure for dynamic slicing of Java programs.

The programmer specifies the program input via the front end, and executes the program on a Java Virtual Machine (JVM). The JVM instruments the execution of the program, and collects the bytecode stream corresponding to the execution trace. The programmer also specifies the observable error as the slicing criterion via the front end. The criterion, together with the bytecode trace, is fed to the dynamic slicing algorithm. The slicing algorithm then returns a dynamic slice at the level of

bytecode. Finally, the resultant slice is transformed to the source code level with the help of information available in Java class files, and is reported to the programmer via the GUI for comprehension and debugging.

Traditionally, dynamic slicing is performed w.r.t. a slicing criterion (l, v), where l represents the location of a bytecode instance in the execution trace, and v is a program variable. A dynamic slicing algorithm can proceed by forward or backward exploration of an execution trace. Here we summarize a backwards slicing algorithm. This algorithm is goal-directed (w.r.t. the slicing criterion), and relies on efficient storage/traversal of the trace. During the trace traversal which starts from the bytecode occurrence in the slicing criterion, a dynamic slicing algorithm maintains the following quantities: (a) the dynamic slice φ, (b) a set of variables δ whose dynamic data dependencies need to be explained, and (c) a set of bytecode instances γ whose dynamic control dependencies need to be explained. Initially, we set the following $\varphi = \gamma =$ the bytecode instance at location l in trace, and $\delta = \{v\}$.

Since a dynamic slice includes the closure of dynamic control and data dependencies from the criterion, the algorithm performs the following two checks, for each bytecode instance β encountered during the backward traversal. The algorithm terminates when we reach the beginning of the trace.

check dynamic control dependencies. If any bytecode instance in γ is dynamically control dependent on β, all statement instances which are dynamically control dependent on β are removed from γ. Variables used by β are inserted into δ, and β is inserted into φ and γ.

check dynamic data dependencies. Let v^{β}_{def} be the variable defined by β. If $v^{\beta}_{def} \in \delta$, it means that we have found the definition of v^{β}_{def} which the slicing algorithm was looking for. So, v^{β}_{def} is removed from δ, and variables used by β are inserted into δ. In addition, β is inserted into φ and γ.

Computing the dynamic data dependencies on bytecode traces is complicated due

to Java's stack based architecture. The main problem is that partial results of a computation are often stored in the Java Virtual Machine's operand stack. This results in implicit data dependencies between bytecodes involving data transfer via the operand stack. For this reason, our backwards dynamic slicing algorithm performs a "reverse" stack simulation while traversing the bytecode trace from the end.

When the dynamic slicing algorithm terminates, the resultant dynamic slice, *i.e.* statements whose bytecode occurrences are included in the set φ, is reported back to the programmer for inspection.

Dynamic slicing has been studied for about two decades, and it has been shown that dynamic slicing is quite useful in debugging. However, there are still three challenges which have not been thoroughly studied. They are:

1. Space efficient trace representation.

2. Enhance dynamic slicing to capture execution omission errors.

3. Guide the developer to effectively explore the dynamic slice.

In this thesis, we have proposed three approaches to address the above challenges, as briefly discussed in the following.

2.2.1 Compact Trace Representation for Dynamic Slicing

In the dynamic slicing infrastructure presented in Figure 2.4, the bytecode trace is very important, since it is the foundation of the dynamic slicing algorithm. However, the bytecode trace tends to be huge for real programs. So, it is important to develop space efficient representation of the trace.

Our method proceeds by on-the-fly construction of a compact bytecode trace during program execution. The compactness of our trace representation is due to several factors. First, bytecodes which do not correspond to memory access (*i.e.* data transfer to and from the heap) or control transfer are not stored in the trace.

Operands used by these bytecodes are fixed and can be discovered from Java class files. Secondly, the sequence of addresses used by each memory reference bytecode or control transfer bytecode is stored separately. Since these sequences typically have high repetition of patterns, we exploit such repetition to save space. We modify a well-known lossless data compression algorithm called SEQUITUR [78] for this purpose. This algorithm identifies repeated patterns in the sequence on-the-fly and stores them hierarchically.

Generating compact bytecode traces during program execution constitutes the first phase of our dynamic slicer. Furthermore, we want to traverse the compact execution trace to retrieve control and data dependencies for slicing. This traversal should be done without decompressing the trace. In other words, the program trace should be collected, stored and analyzed for slicing – all in its compressed form. This is achieved in our dynamic slicer which traverses the compact bytecode trace and computes the data/control dependencies in compression domain. Since we store the sequence of addresses used by each memory-reference/control-transfer bytecode in compressed format, this involves marking the "visited" part of such an address sequence without decompressing its representation.

2.2.2 From Dynamic Slicing to Relevant Slicing

The dynamic slicing algorithm is the core in the slicing infrastructure presented in Figure 2.4. The slicing algorithm analyzes the bytecode trace, and returns a dynamic slice. The dynamic slice may help debugging by focusing the programmer's attention on a part of the program. However, due to the limitation of the dynamic slicing algorithm, there are certain difficulties in using dynamic slices for program debugging.

Traditionally, a dynamic slice only includes statements which have *actual* dynamic control/data dependencies w.r.t. the observable error. Unfortunately, the erroneous statement is not always included in the dynamic slice. Let us look at the example in

Figure 2.5 which is taken from the NanoXML utility [92]. There is an error at line 3, which should be `if (ch == '&')`.

```
1.  ch = reader.read();
2.  buf.append(ch);
3.  if (ch == ' ') {
4.    while (ch != ';') {
5.      ch = reader.read();
6.      buf.append(ch);
7.    }
8.  }
9.  return buf;
```

Figure 2.5: A fragment from the NanoXML utility to explain relevant slicing.

When the input *reader* is the string "&abc;", the trace of the program fragment follows lines $1^1, 2^2, 3^3, 9^4$, where 1^1 means statement 1 is executed as the *first* statement and so on. The resultant *buf* is "&", which is deemed as error by the programmer. If the programmer wants to use dynamic slicing to explain the error, the dynamic slice only contains lines 1, 2 and 9, by considering the dynamic control and data dependencies. Unfortunately, line 3, the actual bug, is excluded from the dynamic slice.

In this example, the observable error arises from the execution of lines 4-6 being wrongly omitted, which is caused by the incorrect condition at line 3. In fact, if we change line 3, this may cause the predicate at line 3 to be evaluated differently; then lines 4-6 will be executed and the value of *buf* at line 9 might be different. In other words, dynamic slicing does not consider the effect of the unexecuted statements at lines 4-6.

The notion of *relevant slicing*, an extension of dynamic slicing, fills this caveat. Relevant slicing was introduced in [7, 40]. Besides dynamic control and data dependencies, relevant slicing considers *potential dependencies* which capture the potential effects of unexecuted paths of branch and method invocation statements. The relevant slice includes more statements which, if changed, may change the "wrong" behaviors

w.r.t. the slicing criterion. In the example of Figure 2.5 with input *reader* ="&abc;", statement instance 9^4 is potentially dependent on execution of the branch at line 3 (3^3), because if the predicate at 3^3 is evaluated differently, the variable *buf* may be re-defined and then used by 9^4. Thus, line 3 is included into the resultant relevant slice.

Like dynamic slices, the relevant slices are also computed w.r.t. a particular program execution (*i.e.* it only includes executed statements). In general, *Dynamic Slice* $\subseteq$ *Relevant Slice* $\subseteq$ *Static Slice*.

In this thesis, we propose a relevant slicing algorithm, which operates on our compact bytecode traces without the costly decompression. We compare our definition of relevant slice against previous ones [7, 40], and show that ours is more accurate. Additionally, we experimentally evaluate the performance of relevant slicing with realistic programs. In our experiments, we show that the sizes of the relevant slices are close to the sizes of the corresponding dynamic slices.

2.2.3 Hierarchical Exploration of the Dynamic Slice

Traditionally, the dynamic slice is reported to a programmer as a flat set of statements, as shown in Figure 2.4. According to the experimental evaluation in literature [100, 118] and our own experience, the dynamic slices of real programs are often too large for humans to inspect and comprehend. So, we either need to prune dynamic slices, or need tools to help a programmer understand a large dynamic slice.

In this thesis, we take the second route. However, our method can be combined with techniques for pruning a dynamic slice (such as [113]). We build a dynamic slicing method where the human programmer is gradually exposed to a slice in a hierarchical fashion, rather than having to inspect a very large slice after it is computed. The key idea is simple — we systematically interleave the slice computation and comprehension steps. Conventional works on slicing have only concentrated on

the computation of the slice, comprehension of the slice being left as a post-mortem activity. In this thesis, we integrate the two activities in a synergistic fashion:

- Computation of the slice is guided (to a limited extent) by the human programmer so that very few control/data dependencies in a large slice need to be explored and inspected.

- The programmer's comprehension of the slice is greatly enhanced by the nature of our slice computation which proceeds hierarchically. Thus, for programs with long dependence chains, this allows the programmer to gradually zoom in to selected dynamic dependencies.

To understand the potential benefits one can gain from our method, let us examine the reasons which make the comprehension of dynamic slices difficult.

- Many programs have long dependence chains spanning across loops and function boundaries. These dependence chains are captured in the slice. However, the slice being a (flat) set of statements, much of the program structure (loops/functions) is lost. This makes the slice hard to comprehend.

- Programs often also have a lot of inherent parallelism. So, a slice may capture many different dependence chains.

We now discuss how hierarchical computation/exploration of slices can help programmers to comprehend large slices containing these two features — (a) long dependence chains, and (b) many different dependence chains. Figure 2.6(a) shows an example program with a long dependence chain. Consider an execution trace of the program $...3^1, 4^2, 5^3, 6^4$ — where lines 3,4,5,6 of Figure 2.6(a) are executed. Slicing this execution trace w.r.t. the criterion $(6^4, y)$ (*i.e.*, the value of y at the occurrence of line 6) yields a slice which contains lines $3, 4, 5, 6$ as well as lines *inside* the body of the functions $f1, f2, f3$. In other words, since the slice is a (flat) set of statements,

the program structure is lost in the slice. This structure is explicitly manifested in Figure 2.6(b), where we show the dependence chain in a *hierarchical fashion* as dashed arrows. In other words, the dependencies inside the functions $f1, f2, f3$ are not shown. Here, a hierarchical exploration of the dependence chains will clearly be less burdensome to the programmer. Thus, in Figure 2.6(b), by inspecting the dependencies hierarchically, the programmer may find it necessary to inspect the dependencies inside a specific function (say $f2$). As a result, we can avoid inspecting the dependence chain(s) inside the other functions (in this case $f1, f3$).

Now, let us consider programs with many different dependence chains. Figure 2.7(a) shows a schematic program with several dependence chains, and hence substantial inherent parallelism. If the slicing criterion involves the value of y in line 6 — we need to consider the dependencies between y and $x3$, y and $x2$, as well as, y and $x1$. These three dependencies are shown via broken arrows in Figure 2.7(b). Again, with the programmer's intervention, we can rule out some of these dependencies for exploration and inspection.

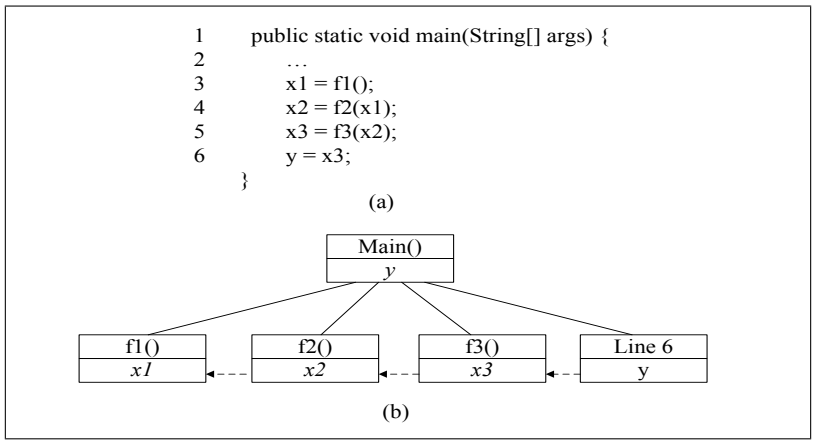


Figure 2.6: Example: A program with a long dynamic dependence chain.

In summary, our method works as follows. Given an execution trace (corresponding to a program input) containing an observable behavior which is deemed as an

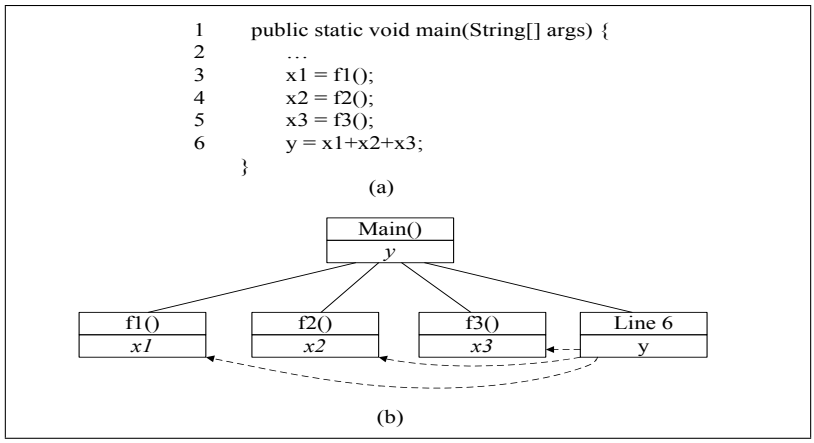


Figure 2.7: Example: A program with inherent parallelism (several dynamic dependence chains).

"error" by the programmer, we divide the trace into **phases**. This division is typically done along loop/procedure/loop-iteration boundaries so that each phase corresponds to a logical unit of program behavior. Only the inter-phase data and control dependencies are presented to the programmer; the intra-phase dependencies are completely suppressed. The programmer then identifies a likely suspicious phase which is then subjected to further investigation in a similar manner (dividing the phase into sub-phases, computing dependencies across these sub-phases and so on). This process continues until the error is identified. Of course, an underlying assumption here is that the programmer will be able to identify the erroneous statement once this statement is pinpointed to him/her.[2]

One may comment that such a hierarchical exploration of dynamic dependencies involves programmer's intervention, whereas conventional dynamic slicing is fully automatic. Here we should note that, the process of error detection by using/exploring a dynamic slice involves a *huge* manual effort; the manual effort in exploring the slice simply happens *after* the computation of the slice. In our hierarchical method, we

[2]This assumption is rather standard in existing works on debugging (*e.g.*, see the score computation by Renieris and Reiss [86], which forms the basis of experimentation in many fault localization techniques [22, 39, 86]).

are interleaving the computation and comprehension of dynamic dependencies. As in dynamic slicing, the computation of the dynamic dependencies is automatic in our method; only the comprehension involves the programmer. Moreover, we are *gradually* exposing the programmer to the complex chain(s) of program dependencies, rather than all at once — thereby allowing better program comprehension.

2.3 Test Based Fault Localization

Dynamic slicing is believed to be a useful but heavy technique. In the past few years, substantial research has been conducted on novel debugging techniques [22, 51, 83, 86, 87, 110]. These approaches compare the failing execution run (*i.e.* an execution run with observable errors) with the successful execution run (*i.e.* an execution run without observable errors). The difference may be related with the error, and is reported to the programmer for inspecting. These fault localization techniques do not require the entire control and data dependence information of the execution, and are often cheaper than slicing.

Most of the research in this topic has focused on how to compare the successful and failing execution runs. In this thesis, we present a control flow based difference metric, and we show how to use this difference metric to (a) generate a successful run, or (b) choose a successful run from a pool of successful runs.

Our approach for automatically generating a feasible successful run[3] is based on the notion of the difference metric. Given a failing run π_f of program P, our approach attempts to find a feasible successful run of P which is "similar" to π_f. We feel that, the successful executions which are "similar" to the failing execution run can be more useful for fault localization, since the programmer may locate the error by investigating the "small" difference between the failing run and the successful run.

[3]A feasible successful run is an execution run which is exercised by some program input and does not exhibit the bug being localized.

The successful run is constructed from the failing run by toggling the outcomes of some of the conditional branch instances in the failing run.

Another way to get the successful run is to *choose* a successful run from a given pool for the comparison. Given a failing run π_f and a pool of successful runs S, we select the most similar successful run $\pi_s \in S$ in terms of the difference metric, and generate a bug report by returning the difference between π_f and π_s. Because our difference metric is based on the control flow, we only need to collect the path of every execution run. This kind of tracing often incurs little overheads. For example, [11] reports that the time overhead of their path collection approach is only 31% on average for the SPEC95 benchmarks; while our experiments show that it often takes 200%-1000% of the execution time to trace both the control flow and the data flow information.

2.4 Remarks

In this chapter, we describe the basic principles and approaches of software debugging, and introduce existing techniques to automate the debugging process, *i.e.* dynamic slicing and test based fault localization. We then illustrate the problems of existing approaches and present our proposals to improve dynamic slicing and test based fault localization.

CHAPTER 3

DYNAMIC SLICING ON JAVA BYTECODE TRACES

In this chapter, we describe a dynamic slicing technique for Java programs. Our technique operates on compact bytecode traces. First, the bytecode trace corresponding to an execution is collected. Since such traces can be huge, we use results from data compression to compress the bytecode traces on-the-fly during the program execution. The major space savings in our method come from the optimized representation of (a) data addresses used as operands by memory reference bytecodes, and (b) instruction addresses used as operands by control transfer bytecodes. We then present a dynamic slicing algorithm. The slicing algorithm performs a backwards traversal of the compressed program trace to compute data/control dependencies on-the-fly, without resorting to costly decompression. The dynamic slice is updated as these dependencies are encountered during trace traversal.

The rest of this chapter is organized as follows. The next section describes our compressed representation of a Java bytecode stream. Section 3.2 presents our slicing algorithm which proceeds by traversing the compact bytecode traces. Section 3.3 reports the space efficiency and time overheads of our compressed trace representation. Section 3.4 concludes this chapter.

3.1 Compressed Bytecode Trace

We now discuss how to collect compact bytecode traces of Java programs *on the fly*. This involves a discussion of the compaction scheme as well as the necessary instrumentation. The compaction scheme used by us is exact, lossless and on-the-fly.

3.1.1 Overall representation

The simplest way to define a program trace is to treat it as a sequence of "instructions". For Java programs, we view the trace as the sequence of executed bytecodes, instead of program statements. This is because only bytecodes are available for Java libraries, which are used by Java programs. Furthermore, collecting traces at the level of bytecode has the flexibility in tracing/not tracing certain bytecodes. For example, the `getstatic` bytecode loads the value of a static field. This bytecode does not need tracing, because which static field to access is decided at compile-time, and can be discovered from class files during post-mortem analysis.

However, representing a Java program trace as a bytecode sequence has its own share of problems. In particular, it does not allow us to capture many of the repetitions in the trace. Representation of the program trace as a single string loses structure in several ways.

- The individual methods executed are not separated in the trace representation.
- Sequences of target addresses accessed by individual control transfer bytecodes are not separated out. These sequences capture control flow and exhibit high regularity (*e.g.* a loop branch repeats the same target many times).
- Similarly, sequences of addresses accessed by individual memory load/store bytecodes are not separated out. Again these sequences show fair amount of repetition (*e.g.* a read bytecode sweeping through an array).

In our representation, the compact trace of the whole program consists of trace tables; one trace table is stored for each method. Method invocations are captured by tracing bytecodes which invoke methods. The last executed bytecode w.r.t. the entire execution is clearly marked. Within the trace table for a method, each row maintains traces of a specific bytecode or of the exit of the method. Monitoring and

tracing every bytecode may incur too much time and space overheads. We monitor only the following five kinds of bytecodes to collect the trace, where the first two are necessary to capture data flow of the execution, and the last three are necessary to capture control flow of the execution.

- *Memory allocation bytecodes.* Memory allocation bytecodes record the identities of created objects.

- *Memory access bytecodes.* The bytecodes to access local variables and static fields are not traced since the addresses accessed by these bytecodes can be obtained from the class file. For bytecodes accessing object fields / array elements, we trace the addresses (or identities since an address may be used by different variables in the lifetime of a program execution) corresponding to the bytecode operands.

- *Method invocation bytecodes.* Java programs use four kinds of bytecodes to invoke methods. Two of them, `invokevirtual` and `invokeinterface`, may invoke different methods on different execution instances. These invoked methods have the same method name and parameter descriptor (which can be discovered in class files), but they belong to different classes. So, for every `invokevirtual` and `invokeinterface` bytecode, we record the classes which the invoked methods belong to.

- *Bytecodes with multiple predecessors.* Some bytecodes have multiple predecessors in the control flow graph. For such a bytecode, we record which bytecodes are executed immediately before itself.

- *Method return bytecodes.* If a method has multiple `return` bytecodes, the trace of the method-exit records which `return` bytecodes are executed.

Monitoring the last two kinds of bytecodes (bytecodes with multiple predecessors and method return bytecodes) and marking the last executed bytecode are required due to backward traversal of the trace during post-mortem analysis. On the other hand, if slicing proceeds by forward traversal of the trace, it is not necessary to monitor bytecodes with multiple predecessors and method return bytecodes. Instead, for each conditional branch bytecode we can record which bytecodes are executed immediately after the branch bytecode (*i.e.*, the target addresses).

As mentioned earlier, our trace representation captures each method's execution in the trace as a trace table. Each row of the trace table for a method m represents the execution of one of the bytecodes of m (in fact it has to be a bytecode which we trace). A row of a trace table thus captures all execution instances of a specific bytecode. The row corresponding to a bytecode b in method m stores the sequence of values taken by each operand of b during execution; if b has multiple predecessors, we also maintain a sequence of the predecessor bytecode of b. Thus, in each row of a trace table we store several sequences in general; *these sequences are stored in a compressed format*. Separating the sequence of values for each bytecode operand allows a compression algorithm to capture and exploit regularity and repetition in the values taken by an operand. This can be due to regularity of control or data flow (*e.g.*, a read bytecode sweeping through an array or a loop iterating many times). Before presenting how to compress trace sequences, let us look at an example to understand the trace table representation. Note that sequences are not compressed in this example for ease of understanding.

Example The left part of Figure 3.1 presents a simple Java program, and the right part shows the corresponding bytecode stream. Table 3.1 shows the trace tables for methods `main` and `foo`, respectively. The constructor method `Demo` has no trace table, because no bytecode of this method is traced. Each row in the trace table consists of:

```
1:  class Demo{
2:
3:    public int foo(int j){
4:        int ret;
5:        if ( j % 2 == 1 )
6:            ret= 2;
7:        else
8:            ret= 5;
9:        return ret;
10:   }
11:
12:   static public void main (String argvs[]){
13:       int i, k, a, b;
14:       Demo obj= new Demo();
15:       int arr[]= new int[4];
16:
17:       a=2;
18:       b=1;
19:       k=1;
20:       if (a>1){
21:          if (b>1){
22:                k=2;
23:          }
24:       }
25
26:       for (i=0; i < 4; i++){
27:             arr[i]=k;
28:             k= k + obj.foo(i);
29:       }
30:
31:       System.out.println(k);
32:   }
33: }
```

```
public static void main(String[]);

  1:  new Class Demo
  2:  dup
  3:  invokespecial  Demo()
  4:  astore  5
  5:  iconst_4
  6:  newarray int
  7:  astore  6
  8:  iconst_2
  9:  istore_3
 10:  iconst_1
 11:  istore  4
 12:  iconst_1
 13:  istore_2
 14:  iload_3
 15:  iconst_1
 16:  if_icmple      22
 17:  iload  4
 18:  iconst_1
 19:  if_icmple      22
 20:  iconst_2
 21:  istore_2
 22:  iconst_0
 23:  istore_1
 24:  iload_1
 25:  iconst_4
 26:  if_icmpge      39
 27:  aload  6
 28:  iload_1
 29:  iload_2
 30:  iastore
 31:  iload_2
 32:  aload  5
 33:  iload_1
 34:  invokevirtual  foo:(int)
 35:  iadd
 36:  istore_2
 37:  iinc  1, 1
 38:  goto  24
 39:  getstatic
 40:  iload_2
 41:  invokevirtual  println:(int)
 42:  return
```

```
Demo();

  43:  aload_0
  44:  invokespecial  Object()
  45:  return


public int foo(int);

  46:  iload_1
  47:  iconst_2
  48:  irem
  49:  iconst_1
  50:  if_icmpne      54
  51:  iconst_2
  52:  istore_2
  53:  goto  56
  54:  iconst_5
  55:  istore_2
  56:  iload_2
  57:  ireturn
```

Figure 3.1: Example: A simple Java program, and its corresponding bytecodes.

(a) the id/address for a bytecode (in the *Bytecode* column), and (b) collected traces for that bytecode (in the *Sequences* column).

For our example Java program, there are 57 bytecodes altogether, and only 8 of them are traced, as shown in Table 3.1. Bytecodes 1 and 6 (*i.e.* two `new` statements at lines 14 and 15 of the source program) allocate memory for objects, and their traces include o_1 and o_2, which represent identities of the objects allocated by these bytecodes. Bytecode 30 defines an element of an array (*i.e.* define `arr[i]` at line 27 of the source program). Note that for this `iastore` bytecode, two sequences are stored. These sequences correspond to the two operands of the bytecode, namely: identities of accessed array objects (*i.e.* $\langle o_2, o_2, o_2, o_2 \rangle$) and indices of accessed array element (*i.e.* $\langle 0, 1, 2, 3 \rangle$). Both sequences consist of four elements, because bytecode 30 is executed four times and accesses $o_2[0], o_2[1], o_2[2], o_2[3]$ respectively; each element in a sequence records one operand for one execution of bytecode 30. Bytecodes 34 and 41

Bytecode	Sequences
1	$\langle o_1 \rangle$
6	$\langle o_2 \rangle$
22	$\langle 19 \rangle$
24	$\langle 23, 38, 38, 38, 38 \rangle$
30	$\langle o_2, o_2, o_2, o_2 \rangle$ $\langle 0, 1, 2, 3 \rangle$
34	$\langle C_{Demo}, C_{Demo}, C_{Demo}, C_{Demo} \rangle$
41	$\langle C_{out} \rangle$

(a)

Bytecode	Sequences
56	$\langle 55, 53, 55, 53 \rangle$

(b)

Table 3.1: Example: Trace tables for (a) method `main()` and (b) method `foo()` of Figure 3.1

invoke virtual methods; the operand sequences record classes which invoked methods belong to, where C_{Demo} represents class `Demo` and C_{out} represents the standard output stream class. Bytecodes 22, 24 and 56 have multiple predecessors in the control flow graph. For example, bytecode 56 (*i.e.* `return ret` at line 9 of the source program) has two predecessors: bytecode 53 (*i.e.* after `ret=2` at line 6 of the source program) and bytecode 55 (*i.e.* `ret=5` at line 8 of the source program). The sequence recorded for bytecode 56 (see Table 3.1(b)) captures bytecodes executed immediately before bytecode 56, which consists of bytecodes 55 and 53 in this example. Note that every method in our example program has only one `return` bytecode, so no `return` bytecode is monitored and no trace of the method-exit is collected.

Clearly, different invocations of a method within a program execution can result in different traces. The difference in two executions of a method results from different operands of bytecodes within the method. These different traces are all stored implicitly via the sequences of operands used by the traced bytecodes. As an example, consider the trace table of method `foo` shown in Table 3.1(b). The different traces of `foo` result from the different outcomes of its only conditional branch, which is captured by the trace sequence for predecessors of bytecode 56 in Figure 3.1, as shown in Table 3.1(b).

3.1.2 Overview of SEQUITUR

So far, we have described how the bytecode operand sequences representing control flow, data flow, or dynamic call graph are separated in an execution trace. We now employ a lossless compression scheme to exploit the regularity and repetition of these sequences. Our technique is an extension of the SEQUITUR, a lossless data compression algorithm [78] which has been used to represent control flow information in program traces [63]. First we briefly describe SEQUITUR.

The SEQUITUR algorithm represents a finite sequence σ as a context free grammar whose language is the singleton set $\{\sigma\}$. It reads symbols one-by-one from the input sequence and restructures the rules of the grammar to maintain the following invariants: (A) no pair of adjacent symbols appear more than once in the grammar, and (B) every rule (except the rule defining the start symbol) is used more than once. To intuitively understand the algorithm, we briefly describe how it works on a sequence 123123. As usual, we use capital letters to denote non-terminal symbols.

After reading the first four symbols of the sequence 123123, the grammar consists of the single production rule

$$S \rightarrow 1, 2, 3, 1$$

where S is the start symbol. On reading the fifth symbol, it becomes

$$S \rightarrow 1, 2, 3, 1, 2$$

Since the adjacent symbols $1, 2$ appear twice in this rule (violating the first invariant), SEQUITUR introduces a non-terminal A to get

$$S \rightarrow A, 3, A \quad A \rightarrow 1, 2$$

Note that here the rule defining non-terminal A is used twice. Finally, on reading the last symbol of the sequence 123123 the above grammar becomes

$$S \rightarrow A, 3, A, 3 \quad A \rightarrow 1, 2$$

This grammar needs to be restructured since the symbols $A, 3$ appear twice. SEQUITUR introduces another non-terminal to solve the problem. We get the rules

$$S \rightarrow B, B \quad B \rightarrow A, 3 \quad A \rightarrow 1, 2$$

However, now the rule defining non-terminal A is used only once. So, this rule is eliminated to produce the final result.

$$S \rightarrow B, B \quad B \rightarrow 1, 2, 3$$

Note that the above grammar accepts only the sequence 123123.

3.1.3 Capturing Contiguous Repeated Symbols in SEQUITUR

One drawback of SEQUITUR is that it cannot efficiently represent contiguous repeated symbols, including both terminal and non-terminal symbols. However, contiguous repeated symbols are not uncommon in program traces. Consider the example in Figure 3.1. Bytecode 24 (*i.e.* `i<4` at line 26 of the source program in Figure 3.1) has two predecessors: bytecode 23 (*i.e.* `i=0` at line 26 of the source program in Figure 3.1) and bytecode 38 (after `i++` at line 26 of the source program in Figure 3.1). The `for` loop is iterated four times, so the predecessor sequence for bytecode 24 is: $\langle$ 23, 38, 38, 38, 38 $\rangle$ as shown in Table 3.1(a). To represent this sequence, SEQUITUR will produce the following rules:

$$S \rightarrow 23, A, A \quad A \rightarrow 38, 38$$

In general, if the `for` loop is iterated k times, SEQUITUR needs $O(lgk)$ rules in this fashion. To exploit such contiguous occurrences in the sequence representation, we propose the Run-Length Encoded SEQUITUR (RLESe).

RLESe constructs a context free grammar to represent a sequence *on the fly*; this contrasts with the work of [85] which modifies the SEQUITUR grammar post-mortem. The right side of each rule is a sequence of "**nodes**". Each node $\langle sym : n \rangle$

consists of a symbol *sym* and a counter n (*i.e.* run length), representing n contiguous occurrences of *sym*. RLESe can exploit contiguous repeated symbols, and represent the above trace sequence ⟨ 23, 38, 38, 38, 38 ⟩ of bytecode 24 using the following one rule:

$$S \rightarrow 23:1, 38:4$$

The RLESe algorithm constructs a context free grammar by reading from the input sequence symbol by symbol. On reading a symbol *sym*, a node $\langle sym:1 \rangle$ is appended to the end of the start rule, and grammar rules are re-structured by preserving following three properties. The first property is unique to RLESe, resulting from its maintenance of contiguous occurrences of grammar nodes. The second and third properties are taken (and modified) from SEQUITUR.

1. *No contiguous repeated symbols property.* This property states that each pair of adjacent nodes contains different symbols. Continuous repeated symbols will be encoded within the run-length.

2. *Digram uniqueness property.* This property means that no *similar* digrams appear in resulting grammar rules. Here a digram refers to two consecutive nodes on the right side of a grammar rule. Two digrams are *similar* if their nodes contain the same pair of symbols *e.g.* $\langle a:2, X:2 \rangle$ is similar to $\langle a:3, X:4 \rangle$, but $\langle a:3, X:2 \rangle$ is not similar to $\langle X:2, a:3 \rangle$.

3. *Rule utility property.* This rule states that every rule (except the start rule S) is referenced more than once. When a rule is referenced by only one node and the run length n of that node equals 1, the reference will be replaced with the right hand side of this rule.

To maintain the digram uniqueness property in RLESe, we might need to split nodes during grammar construction. This split operation allows the algorithm to

obtain duplicated *identical* digrams, and represent them by one grammar rule for potential space saving. Two digrams are *identical* if they have the same pairs of symbols and counters. For example, digram $\langle a:2, X:2\rangle$ is identical to $\langle a:2, X:2\rangle$, but digram $\langle a:2, X:2\rangle$ is not identical to $\langle a:3, X:4\rangle$.

Given two similar digrams $\langle sym_1 : n_1, sym_2 : n_2\rangle$, and $\langle sym_1 : n'_1, sym_2 : n'_2\rangle$, we can split at most two nodes to obtain two occurrences of $\langle sym_1 : min(n_1, n'_1), sym_2 : min(n_2, n'_2)\rangle$, where $min(n_1, n'_1)$ denotes the minimum of n_1 and n'_1. Consider bytecode 24 of Figure 3.1, which corresponds to the termination condition `i<4` of loop at line 26 of the source program. Assume that in some execution, such a loop is executed twice, one time with 6 iterations, and another time with 8 iteration. Recall that bytecode 24 has two predecessors: bytecode 23 (corresponding to `i=0` of the source program in Figure 3.1) and bytecode 38 (after `i++` of the source program in Figure 3.1). The predecessor sequence for bytecode 24 is:

$$S \rightarrow 23:1, 38:6, 23:1, 38:8$$

To ensure digram uniqueness property, we will split the node $\langle 38:8\rangle$ to a digram $\langle 38:6, 38:2\rangle$. This is to remove duplicate occurrences of similar digrams as:

$$S \rightarrow A:2, 38:2 \qquad A \rightarrow 23:1, 38:6$$

The split operation introduces more nodes (at most two) into the grammar, but may save space when the identical digram appears frequently in the sequence.

In addition to the run-length encoding performed in RLESe, we also need to modify the terminal symbols fed into RLESe algorithm. In particular, we need to employ "difference representations" in memory reference sequences. For example, the sequence $\langle 0, 1, 2, 3\rangle$ in Table 3.1(a), which represents the indices of the array elements defined by bytecode 30 in Figure 3.1, cannot be compressed. By converting it into its difference representation as $\langle 0, 1, 1, 1\rangle$, we can can represent the sequence as

$$S \rightarrow 0:1, 1:3$$

As with SEQUITUR [78], the RLESe compression algorithm is linear in both space and time, assuming that it takes constant time to find similar digrams. Detailed space/time complexity analysis of the RLESe compression scheme is presented in Appendix A.1. Experiments comparing RLESe with SEQUITUR (refer Section 3.3) show that RLESe can often achieve competitive compression ratio in less time. This is because RLESe can postpone re-constructing the grammar so the grammar rules are re-constructed less frequently. That is, on reading a symbol sym from input, instead of appending node $\langle sym : 1\rangle$ and re-constructing the grammar immediately, the RLESe algorithm first compares sym against the last node $\langle sym' : n\rangle$ of the start rule. If sym is the same as sym', the node $\langle sym' : n\rangle$ is updated to $\langle sym' : n+1\rangle$ and the grammar is not further re-constructed. If not, node $\langle sym : 1\rangle$ is appended to the end of the start rule, and the grammar is re-structured so as to preserve the three properties of RLESe.

3.2 Techniques for Dynamic Slicing

In this section, we focus on how to perform dynamic slicing of Java programs. Our dynamic slicing algorithm operates on the compact bytecode traces described in the last section. Dynamic slicing is performed w.r.t. a slicing criterion (H, α, V), where H is an execution trace, α represents some bytecodes the programmer is interested in, and V is a set of variables referenced at these bytecodes. The dynamic slice contains all bytecodes which have affected values of variables in V referenced at last occurrences of α in the execution trace H.

Often, the user understands a Java program at the statement level. Thus, the user-defined criterion is often of the form (I, l, V), where I is an input, and l is a line number of the source program; the user is interested in statements (instead of bytecodes) which have affected values of variables in V referenced at last occurrences of statements at l during the execution with input I. This form is a little different

from our bytecode based slicing criterion (H, α, V). In this case, program execution with input I produces the trace H, and α represents the bytecodes corresponding to statements at l. The user is interested in statements corresponding to bytecodes included in the dynamic slice. In order to map bytecodes to a line number of the source file and vice versa, we use the *LineNumberTable* attribute in a Java's class file [70] which describes such a map.

The dynamic slice includes the closure of dynamic control and data dependencies from the slicing criterion. A dynamic slice can be defined over *Dynamic Dependence Graph* (DDG) [6], and dynamic slice consists of all bytecodes whose occurrence nodes can be reached from the node(s) representing the slicing criterion in the DDG, as defined in Definition 2.3. We can construct the DDG as well as the dynamic slice during a backwards traversal of the execution trace.

3.2.1 Core Algorithm

Figure 3.2 presents an inter-procedural dynamic slicing algorithm, which returns the dynamic slice defined in Definition 2.3. Before slicing, we pre-compute the static *control flow graph* for the program. In addition, we pre-compute the *control dependence graph* [31], where each node in the graph represents one bytecode, and an edge from node v to v' represents that bytecode of v' decides whether bytecode of v will be executed. This static control dependence graph is used at lines 22 and 23 of the algorithm in Figure 3.2 to detect dynamic control dependencies.

Lines 1-5 of Figure 3.2 introduce five global variables for the slicing algorithm, including the slicing criterion. During dynamic slicing, we maintain δ, a list of variables whose values need to be explained, φ, the set of bytecode occurrences which have affected the slicing criterion, *op_stack*, a operand stack for simulation (see Section 3.2.3), and *fram*, a stack of frames for method invocations. The dynamic slice includes all bytecodes whose occurrences appear in φ at the end of the algorithm. For

every method invocation during trace collection, we create a frame for this invocation during slicing. Each frame contains the method name and a γ set; the γ set includes bytecode occurrences β, where (a) β belongs to this method invocation, and (2) the dynamic control dependencies w.r.t. β need to be explained.

Initially we will set δ, φ, *op_stack*, and *fram* to empty. However, if the program had been aborted in the middle of an execution, the call stack *fram* is initialized differently. In this case, our tracing will record the call stack at the point of abort, call it stk_{abort}. Our dynamic slicing algorithm then initializes *fram* to stk_{abort} and proceeds by backward traversal of the execution trace.

Our slicing algorithm traverses the program's execution trace backwards, starting from the last executed bytecode recorded in the trace H. For each occurrence β of bytecode b^{β} (*i.e.* β represents one execution of the bytecode b^{β}) encountered during the backward traversal for slicing, a frame is created and pushed to *fram* whenever b^{β} is a return bytecode (lines 9-12 of Figure 3.2). The γ set of the new frame is initialized to empty (line 11 of Figure 3.2), since no bytecode occurrence for this method invocation has been traversed. If b^{β} is a method invocation bytecode, a frame is popped from *fram* (lines 13-16 of Figure 3.2). The dynamic slicing algorithm checks whether the encountered bytecode occurrence β has affected the slicing criterion during trace collection, at lines 19-31 of Figure 3.2. In particular, line 19 of Figure 3.2 checks if β is the slicing criterion. Line 22 of Figure 3.2 checks dynamic control dependencies when b^{β} is a control transfer bytecode. The method `computeControlDependence`(b^{β}, *curr_fram*, *last_fram*) returns true iff. any bytecode occurrence included in the dynamic slice is dynamically control dependent on β, where *curr_fram* is the top of the stack *fram*, and *last_fram* captures the frame popped from *fram* (whenever b^{β} is a method invocation bytecode). More specifically, the `computeControlDependence` method returns true iff.

```
1   (H, α, V)= the slicing criterion
2   δ= ∅, a set of variables whose values need to be explained
3   φ= ∅, the set of bytecode occurrences which have affected the slicing criterion
4   op_stack= empty, the operand stack for simulation
5   fram= empty, the frames of the program execution
6   dynamicSlicing()
7      b^β = get last executed bytecode from H;
8      while (b^β is defined)
9         if (b^β is a return bytecode)
10           new_fram= createFrame();
11           new_fram.γ= ∅;
12           push(fram, new_fram);
13        if (b^β is a method invocation bytecode)
14           last_fram = pop(fram);
15        else
16           last_fram = null;
17        β = current occurrence of bytecode b^β;
18        curr_fram = the top of fram;
19        if (β is the last occurrence of b^β in H, and b^β ∈ α)
20           use_vars = V ∩ variables used at β;
21           φ = φ ∪ {β};
22        if (computeControlDependence(b^β, curr_fram, last_fram) )
23           BC= {β' | β' ∈ curr_fram.γ and β' is dynamically control dependent on β};
24           curr_fram.γ = curr_fram.γ − BC ;
25           use_vars = variables used at β;
26           φ = φ ∪ {β};
27        if (computeDataDependence(β, b^β))
28           def_vars = variables defined at β;
29           δ = δ − def_vars;
30           use_vars = variables used at β;
31           φ = φ ∪ {β};
32        if (β ∈ φ)
33           curr_fram.γ = curr_fram.γ ∪ {β};
34           δ = δ ∪ use_vars;
35        updateOpStack(β, b^β);
36        b^β = getPrevBytecode(β, b^β);
37     return bytecodes whose occurrences appear in φ;
```

Figure 3.2: The dynamic slicing algorithm

- b^{β} is a conditional branch bytecode, and some bytecode whose occurrence appears in $curr\_fram.\gamma$ is statically control dependent on b^{β} (intra-procedural control dependence check), or

- b^{β} is method invocation bytecode, and the $last\_fram.\gamma$ set is not empty, that is some of the bytecode occurrences in the method body are included in the slice (inter-procedural control dependence check).

Bytecode occurrences which are dynamically control dependent on β are then removed from $curr\_fram.\gamma$ at line 24 of Figure 3.2, because their dynamic control dependencies have just been explained. Line 27 of Figure 3.2 checks dynamic data dependencies. When the algorithm finds that the bytecode occurrence β has affected the slicing criterion (*i.e.* any of the three checks in lines 19, 22 and 27 of the algorithm in Figure 3.2 succeeds), β is included into $curr\_fram.\gamma$ and used variables are included into δ (at lines 32-34 of Figure 3.2), in order to find bytecode occurrences which have affected β and hence the slicing criterion. The simulation operand stack *op_stack* is also properly updated at line 35 for further check of data dependencies (this is explained in Section 3.2.3). The dynamic control and data dependencies checks (lines 22 and 27 of Figure 3.2) can be reordered, since they are independent of each other.

In the rest of this section, we elaborate on the underlying subtle issues in using the slicing framework of Figure 3.2. Section 3.2.2 presents how to traverse the execution backwards without decompressing the compact bytecode trace. Section 3.2.3 explains the intricacies of our dynamic data dependence computation in presence of Java's stack based execution. Section 3.2.4 illustrates the dynamic slicing algorithm with an example and Section 3.2.5 shows the correctness and cost of our dynamic slicing algorithm.

```
1   getPrevBytecode (β: bytecode occurrence, b^β: bytecode)
2     if (b^β has exactly one predecessor in the control flow graph)
3       b^last= the predecessor bytecode;
4     else
5       G= compressed control flow operand sequence for b^β in the compact bytecode trace H
6       π= a root-to-leaf path for G;
7       b^last= getLast(G, π);
8     if (b^last is a method invocation bytecode)
9       meth= the method invoked by β;
10      if (meth has exactly one return bytecode)
11        return the return bytecode;
12      else
13        G'= compressed operand sequence for exit of meth in trace H
14        π'= a root-to-leaf path for G';
15        return getLast(G', π');
16    if (b^last represents the start of a method)
17      return the bytecode which invokes current method;
18    return b^last;
```

Figure 3.3: The algorithm to get the previous executed bytecode during backward traversal of the execution trace.

3.2.2 Backward Traversal of Trace without decompression

The dynamic slicing algorithm in Figure 3.2 traverses the program execution backwards, starting from the last executed bytecode recorded in the trace H (line 7 of Figure 3.2). The algorithm proceeds by iteratively invoking the `getPrevBytecode` method to obtain the bytecode executed prior to current occurrence β of bytecode b^β during trace collection. Figure 3.3 presents the `getPrevBytecode` method. The algorithm first retrieves the last executed bytecode within the same method invocation of b^β into b^{last}. It returns b^{last} if b^{last} does not cross method boundaries (lines 2-7 of Figure 3.3). If b^{last} invokes a method $meth$, the last executed return bytecode of method $meth$ is returned (lines 8-15 of Figure 3.3). If b^{last} represents the start of a method, the bytecode which invokes current method is returned (lines 16-17 of Figure 3.3).

The `getPrevBytecode` method has to retrieve last executed bytecode from the compact bytecode trace H. For this purpose, it needs to traverse the predecessor sequence of a bytecode with multiple predecessors. Since such sequences are compactly

stored as RLESe grammars, we need to efficiently traverse RLESe grammars; this is accomplished by the method `getLast`. The `getLast` method gets the last executed predecessor from a RLESe grammar G without decompression, using a root-to-leaf path π in G. The slicing algorithm maintains such a path π for each compressed RLESe sequence G, where the path π clearly marks which portion of G has been already visited. We now explain in details the mechanics of efficient traversal over RLESe representation.

In our trace compression scheme, all operand sequences are compressed using RLESe. The dynamic slicing algorithm traverses these sequences from the end to extract predecessors for computation of control flow, and to extract identifies of accessed variables for computation of data flow. For example, consider the operand sequence of array indices for bytecode 56 in Table 3.1(b), which is $\langle 55, 53, 55, 53\rangle$. During dynamic slicing on the program of Figure 3.1, we traverse this sequence backwards, that is, from the end. At any point during the traversal, we mark the last visited operand (say $\langle 55, 53, 55, \overline{53}\rangle$) during slicing. The sequence beginning with the marked operand (*i.e.* $\langle\overline{53}\rangle$) has been visited. When the slicing algorithm tries to extract next operand from the operand sequence, we use this mark to find last unvisited element (*i.e.* value 55 in this example). We now describe how such markers can be maintained and updated in the RLESe grammar representation.

The RLESe grammar of a sequence σ can simply be represented as a directed acyclic graph (DAG). The RLESe grammar consists of run-length annotated symbols of the form $\langle sym : n\rangle$, where sym is a terminal or non-terminal symbol and n denotes a run-length. The grammar node $\langle sym : n\rangle$ represents n contiguous occurrences of symbol sym. Let us consider an example where the operand sequence is $\langle abbabbcabb\rangle$. The RLESe compression algorithm will produce the following rules to represent this operand sequence.

$$S \rightarrow A : 2, c : 1, A : 1 \quad A \rightarrow a : 1, b : 2$$

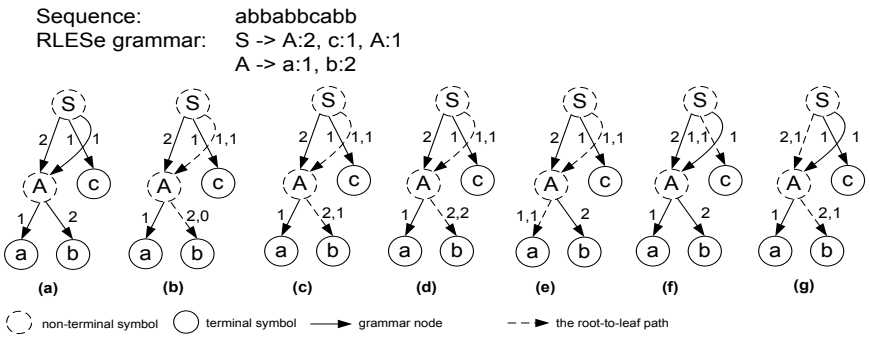


Figure 3.4: Example: Extract operand sequence over RLESe representation without decompression

Small letters denote terminal symbols in the operand sequence, and capital letters denote non-terminal symbols. Figure 3.4(a) shows corresponding DAG representation, where dashed circles represent non-terminal symbols, circles represent terminal symbols, and edges are annotated by run-lengths. For example, the run-length annotated symbol $\langle A : 1 \rangle$ at the end of the start rule is captured by the node A and the incoming edge $S \xrightarrow{1} A$ for node A. We then use a root-to-leaf path π over the DAG representation to mark the symbol in σ that was last visited during backward traversal. For every edge of the path π, we maintain both the run length n of corresponding grammar node $X = \langle sym : n \rangle$, and a visitation counter $k \leq n$, where k denotes the number of times that node X has been visited so far. For example, in the edge $A \xrightarrow{2,1} b$ in Figure 3.4(c), 2 represents the run length of grammar node $\langle b : 2 \rangle$, and 1 represents that this node has been visited once.

The dynamic slicing algorithm maintains one root-to-leaf path π for every compressed operand sequence. The path π is initialized from the root to the rightmost leaf node in the DAG, and the visitation counter annotated for the last edge in π is set to 0, since no symbol has been visited. The slicing algorithm then uses the path π to find the last unvisited terminal symbol of the RLESe grammar by invoking the `getLast` method of Figure 3.5. In the `getLast` method, G is the DAG representation of the RLESe grammar for a sequence σ and π is the root-to-leaf path for the symbol

```
1   getLast(G: Grammar, π: path in G)
2     e = the last edge of π;
3     while (e is defined)
4       let e = sym_1 --n1,k1--> sym'_1;
5       if ( k_1 < n_1)
6         break;
7       else
8         remove edge e from π;
9         change the annotation of edge e from (n_1, k_1) to (n_1);
10        Sib_e= immediate left sibling of e in G;
11        if ( such a sibling Sib_e exists)
12          e = Sib_e;
13          break;
14        else
15          e = last edge of π;
16    let e = sym_2 --n2,k2--> sym'_2;
17    change the annotation of edge e from (n_2, k_2) to (n_2, k_2 + 1);
18    G_X = DAG rooted at node sym'_2 within G;
19    for (each edge e' from node sym'_2 to rightmost leaf node of G_X)
20      insert edge e' into π;
21      let e' = sym_3 --n3--> sym'_3;
22      change the annotation of edge e' from (n_3) to (n_3, 1);
23    return symbol in rightmost leaf node of G_X;
```

Figure 3.5: One step in the backward traversal of a RLESe sequence (represented as DAG) without decompressing the sequence.

in σ that was last visited. The `getLast` method returns the last unvisited symbol and updates the path π. Note that the "immediate left sibling" of an edge $e = x \rightarrow y$ is the edge $e' = x \rightarrow z$ where node z is the immediate left sibling of node y in the graph; this notion is used in lines 10 and 11 of Figure 3.5.

For the example operand sequence $\langle abbabbcabb \rangle$, Figure 3.4(b) shows the initialized root-to-leaf path π for the RLESe grammar. Figure 3.4(c-g) present the resultant path π by calling the `getLast` method of Figure 3.5 each time, and the symbol of the leaf node pointed by the path π is returned as the last unvisited symbol. For example, Figure 3.4(c) shows the path π after the first calling the `getLast` method. The path π includes two edges (*i.e.* $S \overset{1,1}{\rightarrow} A$ and $A \overset{2,1}{\rightarrow} b$), representing both edges have been visited once. The leaf node b is referenced by π. Thus, b is returned by the `getLast` method, which represents the last symbol b in the original sequence $\langle abbabbcabb \rangle$.

With the `getLast` algorithm in Figure 3.5, we can extract an operand sequence efficiently. Given the grammar for a sequence with length N, the `getLast` method will be invoked N times to extract the entire sequence. The overall space overhead is $O(N)$, and the overall time overhead is $O(N)$. The space overhead is caused by maintaining the root-to-leaf path π, which is used by all invocations of the `getLast` method to extract a sequence. The length of path π is linear in the number of grammar rules (the grammar has no recursive rules). There are fewer grammar rules than grammar nodes, and the number of grammar nodes is bounded by the length of the original sequence. Thus, the space overhead to extract the entire sequence is $O(N)$.

The time overhead to extract the entire sequence comes from the time to access edges and nodes in the DAG. Whenever a node with terminal symbol is accessed, the `getLast` method immediately returns the terminal symbol. So, the total number of times to access node with terminal symbol is $O(N)$ in order to extract a sequence with length N. The total number of accesses to non-terminal nodes is $O(\sum_i \frac{N}{2^i})$= $O(N)$. Consequently, the time overhead to extract the entire sequence from a RLESe grammar (by repeatedly invoking `getLast` to get the last symbol which has not been visited) is $O(N)$.

3.2.3 Computing Data Dependencies

The typical way to detect dynamic data dependencies is to compare addresses of variables defined/used by bytecode occurrences (line 27 of Figure 3.2). However, this is complicated by Java's stack based architecture. During execution, the Java virtual machine uses an operand stack to hold partial results of execution. Thus, dynamic data dependence exists between bytecode occurrences β and β', when a value is pushed into the operand stack by β and is popped by β'. Consider the program in Figure 3.1 as an example. Assume that statement 27 of the source program is executed, and the

```
1    updateOpStack (β: bytecode occurrence, b^β: bytecode)
2        for (i = 0; i < def_op(b^β); i = i + 1)
3            pop(op_stack);
4        for (i = 0; i < use_op(b^β); i = i + 1)
5            push(op_stack, β);
```

Figure 3.6: The algorithm to maintain the simulation stack *op_stack*.

corresponding trace at the level of bytecode is $\langle 27^1, 28^2, 29^3, 30^4 \rangle$ where 27^1 means that the first element of the trace is bytecode 27 and so on. Bytecode occurrence 30^4 (which defines the array element `arr[i]` at line 27 of the source program) is dynamically data dependent on bytecode occurrences 27^1, 28^2 and 29^3 (which load local variables `k` and `i`, array object reference `arr` at line 27 of the source program, respectively). The three bytecode occurrences push three values into the operand stack, all of which are popped and used by bytecode occurrence 30^4.

Clearly, dynamic data dependencies w.r.t. local variables and fields can be easily detected by comparing the addresses (or identities) of accessed variables. However, detecting data dependencies w.r.t. the operand stack requires *reverse* simulating the operand stack (since we traverse the trace backwards). Figure 3.6 presents how to maintain the stack *op_stack* for simulation, which is used by the dynamic slicing algorithm at line 35 of Figure 3.2. We pop the simulation stack for defined operands, and push used operands into the simulation stack. The function $def\_op(b^\beta)$ ($use\_op(b^\beta)$) at line 2 (4) of Figure 3.6 returns the number of operands defined (used) by bytecode b^β. Note that the stack simulated during slicing does not contain actual values of computation. Instead, each entry of the stack stores the bytecode occurrence which pushed the entry into the stack.

Figure 3.7 shows the method to determine whether a bytecode occurrence has affected the slicing criterion via dynamic data dependencies. This method is used by the dynamic slicing algorithm at line 27 of Figure 3.2. If a bytecode occurrence β defines a variable which needs explanation (lines 2-10 of Figure 3.7), or β defines a

```
1   computeDataDependence (β: bytecode occurrence, b^β: bytecode)
2     if (β defines a variable)
3       if (β defines a static field or local variable)
4         def_loc= get address of the defined static field or local variable from class files;
5       if (β defines an object field or an array element)
6         G= compressed operand sequence for b^β in the compact bytecode trace H
7         π= a root-to-leaf path for G;
8         def_loc= getLast(G, π);
9       if ( def_loc ∈ δ )
10        return true;
11    ω= the set of bytecode occurrences in top def_op(b^β) entries of op_stack;
12    if ( ω ∩ φ ≠ ∅)
13      return true;
14    return false;
```

Figure 3.7: The algorithm to detect dynamic data dependencies for dynamic slicing

partial result which needs explanation (lines 11-13 of Figure 3.7), the method returns *true* to indicate that β has affected the slicing criterion. A partial result needs explanation if the bytecode occurrence which pushes corresponding entry into the stack has already been included in φ. The function $def\_op(b^\beta)$ at line 11 of Figure 3.7 returns the number of operands defined by bytecode b^β. Our dynamic slicing algorithm needs the addresses of variables accessed by each bytecode occurrence for detecting data dependencies (lines 20, 25, 28 and 30 of Figure 3.2 and lines 3-8 of Figure 3.7). If a local variable or a static field is accessed, the address can be found from the class files. If an object field or an array element is accessed, the address can be found from operand sequences of corresponding bytecode in the compact bytecode trace.

When the algorithm detects data dependencies via reverse stack simulation, it requires analyzing some bytecodes whose operands are not traced. Consider the program in Figure 3.1 as an example. The statement `if (j%2==1)` at line 5 of the source program corresponds to bytecode sequence $\langle 46^1, 47^2, 48^3, 49^4, 50^5 \rangle$. Figure 3.8 shows the *op_stack* after processing each bytecode occurrence during backward traversal. Note that the operands of bytecode 48 (*i.e.* bytecode `irem` which stands for the mathematical computation "%") are not traced. When bytecode occurrence 48^3 is encountered, `computeDataDependence` method in Figure 3.7 can detect that 50^5

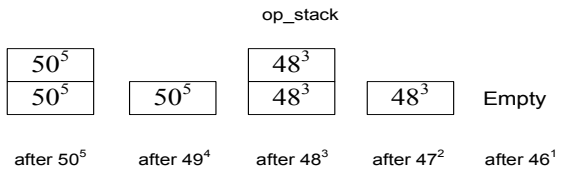


Figure 3.8: Example: Illustrate the *op_stack* after each bytecode occurrence encountered during backward traversal

is dynamically data dependent on 48^3. In addition, the bytecode 48 will also update the *op_stack*, as shown in Figure 3.8. As we can see from the example, in order to detect implicit data dependencies involving data transfer via the operand stack, it is important to know which bytecode occurrence pushes/pops an entry from the *op_stack*. The actual values computed by the bytecode execution are not important. This highlights difference between our method and the work on Abstract Execution [64]. In Abstract Execution, a small set of events are recorded and these are used as guide to execute a modified program. In our work, we record some of the executed bytecodes in our compressed trace representation. However, the untraced bytecodes are not re-executed during the analysis of the compressed trace.

3.2.4 Example

Consider the example program in Figure 3.1, and corresponding compressed trace in Table 3.1. Assume that the programmer wants to find which bytecodes have affected the value of `k` at line 31 of the source program. Table 3.2 shows each stage using the dynamic slicing algorithm in Figure 3.2 w.r.t. the `k`. For simplicity, we do not illustrate slicing over the entire execution, but over last executed eight statements – $\langle 26, 27, 28, 5, 6, 9, 26, 31 \rangle$. The corresponding bytecode sequence is a sequence of thirty-one bytecode occurrences shown in the first column of Table 3.2. For each bytecode occurrence, the position number $1, \ldots, 31$ of the bytecode occurrence in the sequence is marked as superscript for the sake of clarity.

β	δ	*fram*		*op_stack*	$\in \varphi$
		method	γ		
41^{31}	$\{\}$	*main*	$\{\}$	$\langle 41^{31}, 41^{31} \rangle$	
40^{30}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle 41^{31} \rangle$	$\star$
39^{29}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle\rangle$	
26^{28}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle 26^{28}, 26^{28} \rangle$	
25^{27}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle 26^{28} \rangle$	
24^{26}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle\rangle$	
38^{25}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle\rangle$	
37^{24}	$\{k\}$	*main*	$\{40^{30}\}$	$\langle\rangle$	
36^{23}	$\{\}$	*main*	$\{40^{30}, 36^{23}\}$	$\langle 36^{23} \rangle$	$\star$
35^{22}	$\{\}$	*main*	$\{40^{30}, 36^{23}\}$	$\langle 35^{22}, 35^{22} \rangle$	$\star$
57^{21}	$\{\}$	*foo*	$\{57^{21}\}$	$\langle 35^{22}, 57^{21} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
56^{20}	$\{ret\}$	*foo*	$\{57^{21}, 56^{20}\}$	$\langle 35^{22} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
53^{19}	$\{ret\}$	*foo*	$\{57^{21}, 56^{20}\}$	$\langle 35^{22} \rangle$	
		main	$\{40^{30}, 36^{23}\}$		
52^{18}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 52^{18}\}$	$\langle 35^{22}, 52^{18} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
51^{17}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 52^{18}, 51^{17}\}$	$\langle 35^{22} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
50^{16}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 50^{16}\}$	$\langle 35^{22}, 50^{16}, 50^{16} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
49^{15}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}\}$	$\langle 35^{22}, 50^{16} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
48^{14}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}\}$	$\langle 35^{22}, 48^{14}, 48^{14} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
47^{13}	$\{\}$	*foo*	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}, 47^{13}\}$	$\langle 35^{22}, 48^{14} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
46^{12}	$\{j\}$	*foo*	$\{57^{21}, 56^{20}, 50^{16}, 49^{15}, 48^{14}, 47^{13}, 46^{12}\}$	$\langle 35^{22} \rangle$	$\star$
		main	$\{40^{30}, 36^{23}\}$		
34^{11}	$\{\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}\}$	$\langle 35^{22}, 34^{11}, 34^{11} \rangle$	$\star$
33^{10}	$\{i\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}\}$	$\langle 35^{22}, 34^{11} \rangle$	$\star$
32^{9}	$\{i, obj\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}\}$	$\langle 35^{22} \rangle$	$\star$
31^{8}	$\{i, obj, k\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle\rangle$	$\star$
30^{7}	$\{i, obj, k\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7}, 30^{7}, 30^{7} \rangle$	
29^{6}	$\{i, obj, k\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7} \rangle$	
28^{5}	$\{i, obj, k\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle 30^{7} \rangle$	
27^{4}	$\{i, obj, k\}$	*main*	$\{40^{30}, 36^{23}, 34^{11}, 33^{10}, 32^{9}, 31^{8}\}$	$\langle\rangle$	
26^{3}	$\{i, obj, k\}$	*main*	$\{40^{30}, 26^{3}\}$	$\langle 26^{3}, 26^{3} \rangle$	$\star$
25^{2}	$\{i, obj, k\}$	*main*	$\{40^{30}, 26^{3}, 25^{2}\}$	$\langle 26^{3} \rangle$	$\star$
24^{1}	$\{i, obj, k\}$	*main*	$\{40^{30}, 26^{3}, 25^{2}, 24^{1}\}$	$\langle\rangle$	$\star$

Table 3.2: Example: Illustrate each stage of the dynamic slicing algorithm in Figure 3.2. The column β shows bytecode occurrences in the trace being analyzed.

For each bytecode occurrence β encountered during backward traversal, one row of Table 3.2 shows resultant δ, *fram*, *op_stack* and φ after analyzing β by our dynamic slicing algorithm. The $\star$ in the last column indicates that the corresponding bytecode occurrence has affected the slicing criterion and is included into φ.

When bytecode occurrence 40^{30} is encountered, it is found to be the slicing criterion. The used variable `k` is inserted to δ to find which bytecode occurrence defines `k`; the bytecode occurrence 40^{30} is inserted to γ for control dependency check; we pop 41^{31} from the operand stack *op_stack* because 40^{30} loads one value to the operand stack during trace collection. It should be noted that in this simple example, we refer to a variable with its name (*e.g.* `k`), since both methods are invoked once, and every variable has a distinct name in this example. This is for simplicity of illustration. In the implementation, we use identifiers to distinguish between variables from same/different method invocation.

After 40^{30}, bytecode occurrence 39^{29} is encountered during backward traversal and so on. We omit the details of the entire traversal but highlight some representative bytecode occurrences.

- After analyzing bytecode occurrence 56^{20}, the slicing algorithm finds that bytecode 56 has two predecessors, and retrieves last unvisited value from operand sequence of bytecode 56 in Table 3.1(b). Therefore, bytecode occurrence 53^{19} is next analyzed.

- When bytecode occurrence 30^{7} is encountered, o_2 and 3 are retrieved from operand sequences of bytecode 30 in Table 3.1(a), representing an assignment to array element $o_2[3]$. However, $o_2[3]$ is irrelevant to the slicing criterion, so neither δ nor γ is updated.

3.2.5 Proof of Correctness and Complexity Analysis

In this section we discuss the correctness proof and complexity analysis of our dynamic slicing algorithm.

Theorem 3.1. *Given a slicing criterion, the dynamic slicing algorithm in Figure 3.2 returns dynamic slice defined in Definition 2.3.*

Proof Sketch: We only present the proof sketch here. The full proof appears in Appendix A.2.

Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 3.2, φ_* be the resultant φ set when the algorithm finishes, and β be the bytecode occurrence encountered at the ith loop iteration.

We prove the soundness of the algorithm by induction on loop iterations of the slicing algorithm, *i.e.* for any $\beta' \in \varphi_*$ we show that β' is reachable from the slicing criterion in the dynamic dependence graph (DDG).

We prove the completeness of the slicing algorithm, *i.e.* $\forall \beta'$ reachable from slicing criterion in the Dynamic Dependence Graph (DDG) $\Rightarrow \beta' \in \varphi_*$. Note that there is no cycle in the DDG, so we prove the completeness by induction on structure of the DDG. □

Now we analyze the cost of the dynamic slicing algorithm in Figure 3.2. Given the compressed trace for an execution which executes N bytecodes, the space overhead of the slicing algorithm is $O(N)$, and the time overhead is $O(N^2)$.

The space overhead of the algorithm is caused by the maintenance of δ, φ, *op_stack*, *fram*, compressed operand sequences and root-to-leaf paths for every compressed sequence. The sizes of δ, φ, *op_stack* and *fram* are all $O(N)$. For δ, this is because one execution of a bytecode can use a constant number of variables; for *op_stack*, this is because the number of operands popped from and pushed to the operand stack by one bytecode is bound by a constant; for *fram*, this is because the

γ set of each method invocation in *fram* only contains bytecode occurrences for this invocation, and does not overlap with each other. Assume that each bytecode b_i has executed $\eta(b_i)$ times, so $\sum_{b_i} \eta(b_i) = N$. The size of all compressed operand sequences is $\sum_{b_i} O(\eta(b_i)) = O(N)$, because every bytecode has a fixed number of operand sequences, and the size of the compact representation is linear in the length of original operand sequence; proof of this claim appears in Appendix A.1. The size of each root-to-leaf path is bound by the size of corresponding compressed operand sequence. Consequently, the overall space cost of the slicing algorithm is $O(N)$.

During dynamic slicing, the algorithm performs the following four actions for each occurrence β of bytecode b^β encountered during backward traversal of the execution trace.

1. extract operand sequences of bytecode b^β from the compressed trace for backward traversal,
2. perform slicing criterion, dynamic control/data dependency checks,
3. update δ, φ, *op_stack*, and *fram*,
4. get previous executed bytecode.

According to the complexity analysis of the `getLast` method which is presented in Section 3.2.2, it needs $O(\eta(b_i))$ time to extract an operand sequence of bytecode b_i which is executed $\eta(b_i)$ times during trace collection. Note that $\sum_i \eta(b_i) = N$. Consequently, the overall time overhead to perform action (1) to extract all operand sequences is $\sum_i O(\eta(b_i)) = O(N)$, that is, linear in the length of the original execution. After extracting operand sequences, the overall time to perform action (2) and (3) is clearly $O(N^2)$, since they may update/enquire sets δ, φ, *op_stack* and *fram* whose sizes are bound by N. This complexity can be improved further by using efficient data structures for set representation.

To get previous executed bytecode (action (4) in the preceding), the method `getPrevBytecode` of Figure 3.3 may perform two actions: (a) get the predecessor bytecode from class files, or (b) extract last executed bytecode from compressed operand sequences. The overall time to perform get predecessor bytecode from class files is $O(N)$, since it needs constant time to get the predecessor bytecode from Java class files every time. The overall time to extract operand sequences is $O(N)$ as discussed earlier.

Consequently, the overall time cost of the dynamic slicing algorithm is $O(N^2)$.

3.3 Experimental evaluation

We have implemented a prototype slicing tool and applied it to several subject programs. The experiments report time and space efficiency of our trace collection technique. We compress traces using both SEQUITUR and RLESe, and compare the time overheads and effectiveness of the improved compression algorithm against the previous one, in order to investigate the cost effectiveness of the proposed compression algorithm. In addition, the experiments to perform dynamic slicing over our compact trace representation are reported in Chapter 4, in order to compare dynamic and relevant slicing.

To collect execution traces, we have modified the Kaffe virtual machine [3] to monitor interpreted bytecodes. In our traces, we use object identities instead of addresses to represent objects. Creation of structures such as multi-dimensional arrays, constant strings etc. may implicitly create objects. We trace and allocate identities to these objects as well. The virtual machine may invoke some methods automatically when "special" events occur (*e.g.* it may invoke the static initializer of a class automatically when a static field of the class is first accessed). These event are also stored, and used for backward traversal of the execution trace.

Subject	Description	Input
Crypt	IDEA encryption and decryption	200,000 bytes
SOR	Successive over-relaxation on a grid	100 × 100 grid
FFT	1-D fast Fourier transform	2^{15} complex numbers
HeapSort	Integer sorting	10000 integers
LUFact	LU factorisation	200 × 200 matrix
Series	Fourier coefficient analysis	200 Fourier coefficients
_201_compress	Modified Lempel-Ziv method (LZW)	228.tar in the SPECjvm suite
_202_jess	Java Expert Shell System	fullmab.clp in the SPECjvm suite
_209_db	Performs multiple database functions	db2 & scr2 in the SPECjvm suite
JLex	A Lexical Analyzer Generator for Java	sample.lex from the tool's web site

Table 3.3: Descriptions and input sizes of subject programs.

Subject	Total # of Bytecodes	Executed Bytecodes	# Bytecode Instances	# Branch Instances	# Method invocations
Crypt	2,939	1,828	103,708,780	1,700,544	48
SOR	1,656	740	59,283,663	1,990,324	26
FFT	2,327	1,216	72,602,818	2,097,204	37
HeapSort	1,682	679	11,627,522	743,677	15,025
LUFact	2,885	1,520	98,273,627	6,146,024	41,236
Series	1,800	795	16,367,637	1,196,656	399,425
_201_compress	8,797	5,764	166,537,472	7,474,589	1,999,317
_202_jess	34,019	14,845	5,162,548	375,128	171,251
_209_db	8,794	4,948	38,122,955	3,624,673	19,432
Jlex	22,077	14,737	13,083,864	1,343,372	180,317

Table 3.4: Execution characteristics of subject programs.

Most Java programs use libraries. Dynamic dependencies introduced by the execution of library methods are often necessary to compute correct slices. Our implementation does not distinguish between library methods and non-library methods during tracing and slicing. However, after the slicing finishes, we will filter statements inside the libraries from the slice. This is because libraries are often provided by other vendors, and programmers will not look into them.

3.3.1 Subject Programs

The subjects used in our experiments include six subjects from the Java Grande Forum benchmark suite [50], three subjects from the SPECjvm suite [96], and one medium sized Java utility program [15]. Descriptions and inputs of these subjects are

Subject	Orig. Trace	Trace Table	RLESe Sequences	All	All/Orig. (%)
Crypt	64.0M	8.9k	8.8k	17.8k	0.03
SOR	73.4M	7.6k	10.8k	18.5k	0.02
FFT	75.2M	8.2k	87.3k	95.5k	0.12
HeapSort	23.6M	7.7k	1.7M	1.7M	7.20
LUFact	113.1M	9.1k	179.7k	188.9k	0.16
Series	24.4M	7.7k	444.4k	452.2k	1.81
_201_compress	288.6M	23.7k	8.8M	8.8M	3.05
_202_jess	25.7M	79.2k	3.4M	3.4M	13.23
_209_db	194.5M	29.0k	39.2M	39.2M	20.15
Jlex	49.9M	62.6k	1.5M	1.5M	3.01

Table 3.5: Compression efficiency of our bytecode traces. All sizes are in bytes.

shown in Table 3.3. We ran and collected execution traces of each program with inputs shown in the third column in Table 3.3. Corresponding execution characteristics are shown in Table 3.4. The second column in Table 3.4 shows the number of bytecodes of these subjects. The column *Executed bytecodes* in Table 3.4 presents the number of distinct bytecodes executed during one execution, and the fourth column in Table 3.4 shows corresponding total number of bytecode occurrences executed. The last two columns present the number of branch bytecode occurrences and method invocations, respectively. Bytecodes from Java libraries are not counted in the table. However, bytecodes of user methods called from library methods are included in the counts shown.

3.3.2 Time and Space Efficiency of Trace Collection

We first study the efficiency of our compact trace representation. Table 3.5 shows the compression efficiency of our compact trace representation. The column *Orig. Trace* represents the space overheads of storing uncompressed execution traces on disk. To accurately measure the compression achieved by our RLESe grammars, we *leave out* the untraced bytecodes from the *Orig. Trace* as well as the compressed trace. Next two columns *Trace Table* and *RLESe Sequences* show the space overheads to maintain the trace tables and to store the compressed operand sequences of bytecodes on disk.

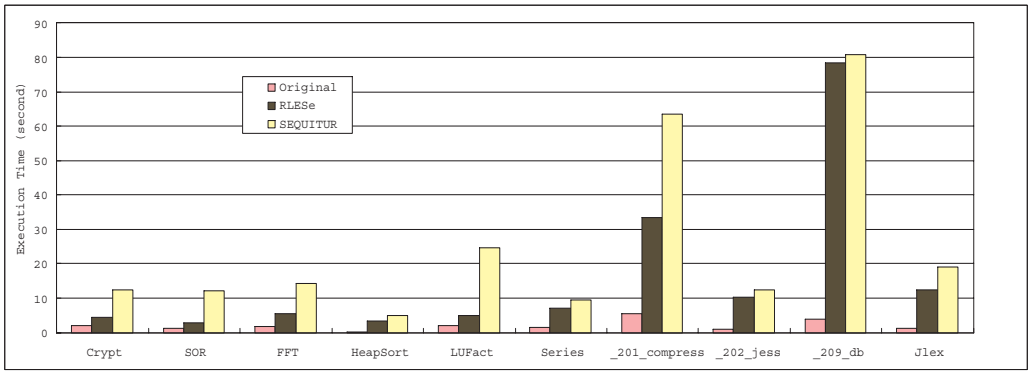


Figure 3.9: Time overheads of RLESe and SEQUITUR. The time unit is *second*.

The column *All* represents the overall space costs of our compact bytecode traces, *i.e.* sum of space overheads of *Trace Table* and *RLESe Sequences*. The % column indicates *All* as a percentage of *Orig. Trace.* For all our subject programs, we can achieve at least 5 times compression. Indeed for some programs we get more than two orders of magnitude (*i.e* 100 times) compression.

Comparison with Conventional Compression Schemes We also compare the space efficiency of our compact bytecode trace representation with a general purpose data compression scheme such as the *gzip* utility [122]. We found that the compression ratio obtained by RLESe is comparable or even significantly better than gzip. Again, for benchmarks with more regular data accesses (like Crypt, SOR) our compression scheme outperforms gzip by several orders of magnitude (in terms of compression ratio). For example, the compression ratios for the Crypt subject program are 0.03% and 5.74%, by using our secheme and gzip respectively. For subject programs with random data accesses (like HeapSort) the compression ratios of RLESe and gzip are comparable. For example, the compression ratios for the HeapSort subject program are 7.20% and 8.72%, by using our secheme and gzip respectively. Of course, the major benefit of our compressed trace representation comes from the ability to traverse it without decompression – which is not possible for gzipped traces.

Figure 3.9 presents the absolute running time of the subject programs without instrumentation, tracing with RLESe and tracing with SEQUITUR. All experiments were performed on a Pentium 4 3.0 GHz machine with 1 GB of memory. From this figure we can see that the slowdown for collecting traces using RLESe compared to the original program execution is $2 - 10$ times for most subject programs, and the slowdown is near 20 times for *_209_db* which randomly accesses a database. This shows that our method is suitable for program executions with many repetitions, but the time overhead may be not scalable to program executions with too many random accesses. Note that these additional time overheads are caused by compression using RLESe/SEQUITUR. The results reflect one limitation of RLESe (as well as SEQUITUR) — the time efficiency of the compression algorithm heavily depends on the efficiency of checking the digram uniqueness property. To speed up this check, a sophisticated index of digrams should be used. However, the choice of the index has to trade-off between time and space efficiency, since the index will introduce additional space overheads during trace collection. In our implementation, we use the B+ tree (instead of sparse hash) to index digrams during compressing, and the index is no longer needed after compression is completed.

Subject	RLESe %	SEQUITUR %
Crypt	0.03	0.07
SOR	0.02	0.04
FFT	0.12	0.34
HeapSort	7.20	7.20
LUFact	0.16	0.40
Series	1.81	2.50
_201_compress	3.05	3.12
_202_jess	13.23	13.62
_209_db	20.15	18.35
Jlex	3.01	4.01

Table 3.6: Comparing compression ratio of RLESe and SEQUITUR.

We now compare the space efficiency of RLESe against SEQUITUR. Both algorithms were performed on operand sequences of bytecodes. For a fair comparison, we

Subject	RLESe	SEQUITUR
Crypt	800,605	20,499,488
SOR	393,202	25,563,918
FFT	2,020,976	25,722,054
HeapSort	2,262,916	7,066,784
LUFact	561,233	42,586,136
Series	2,501,001	9,507,982
_201_compress	10,240,652	80,923,387
_202_jess	2,090,153	6,666,460
_209_db	24,386,746	54,033,314
Jlex	4,555,338	15,497,211

Table 3.7: The number of times to check digram uniqueness property by RLESe and SEQUITUR.

use the same index to search for identical/similar digrams in the implementation of both algorithms. Table 3.6 compares the space costs of both algorithms by presenting their compression ratio (in percentage). From the tables, we can see that the space consumption of compressed traces produced by both algorithms are somewhat comparable. RLESe outperforms SEQUITUR for eight subject programs, and SEQUITUR outperforms for one (where the _209_db program randomly accesses a database and does not have many contiguous repeated symbols in operand traces). Since RLESe employs run-length encoding of terminal and non-terminal symbols over and above the SEQUITUR algorithm, nodes in grammars produced by RLESe are usually less than those produced by SEQUITUR.

Figure 3.9 compares the time overheads of RLESe and SEQUITUR. Clearly RLESe outperforms SEQUITUR in time on studied programs. The time overheads of both algorithms are mainly caused by checking digram uniqueness property. RLESe usually produces less nodes in grammars, so that similar digrams can be found more efficiently. In addition, RLESe checks this property after contiguous repeated symbols have finished, whereas SEQUITUR does this on reading every symbol. Table 3.7 shows this by representing the frequency of checking the digram uniqueness property. When there are many contiguous repeated symbols in the execution traces (*e.g.*

the LUFact, SOR subjects), RLESe checks the property much less frequently than SEQUITUR. Consequently the tracing overheads of RLESe are also much less than those of SEQUITUR.

3.3.3 Summary and Threats to Validity

In summary, the RLESe compaction scheme achieves comparable or better compression ratio than SEQUITUR. The time overheads for the online compaction in RLESe is found to be less than the compaction time in SEQUITUR. Our compact trace representation can be used to perform dynamic slicing efficiently, both in time and space.

We note that there are various threats to validity of the conclusion from our experiments. Our conclusions on less time overheads of RLESe (as compared to SEQUITUR) can be invalidated if the execution trace has very few contiguous repeated symbols. In this case, RLESe checks the diagram uniqueness property almost as frequently as SEQUITUR. This can happen in a program with random data accesses (*e.g.*, the _209_db subject program used in our experiments).

3.4 Summary

In this chapter, we have presented a space efficient scheme for compactly representing bytecode traces of Java programs. The time overheads and compression efficiency of our representation are studied empirically. We use our compact traces for efficient dynamic slicing. Our method has been implemented on top of the open source Kaffe Virtual Machine. The traces are collected by monitoring Java bytecodes. The bytecode stream is compressed online and slicing is performed post-mortem by traversing the compressed trace without decompression.

CHAPTER 4

RELEVANT SLICING

Dynamic slicing has long been studied for software debugging and comprehension. Unfortunately, there are certain difficulties in using dynamic slicing for debugging, because dynamic slicing may miss some important statements which are responsible for the error. In particular, dynamic slicing does not consider 'Execution Omission" errors, *i.e.* the execution of certain statements is wrongly omitted. Consequently, all statements responsible for the error may not be included into the dynamic slice, and the slice may mislead the programmer.

In this chapter, we study relevant slices [7, 40]. These slices extend dynamic slices to include certain statements *stmt*, which if altered due to debugging, can alter the execution flow of P w.r.t. input I and affect the criterion. Such statements *stmt* can be used to explain that the execution of statements guarded by *stmt* is wrongly omitted.

Since the existing works on relevant slicing present a slice as a set of program statements, we first define a relevant slice as a set of statements. However, like our dynamic slicing algorithm presented in Chapter 3, our relevant slicing algorithm works at the level of bytecodes. Most importantly, the relevant slicing algorithm also operates *directly* on the compressed bytecode trace, without involving costly trace decompression.

The rest of this chapter is organized as follows. In Section 4.1, we recall *past work* on potential dependencies, a notion which is crucial for computing the relevant slice. Section 4.2 then presents our definition of relevant slice, and compares our definition with existing works. Section 4.3 presents our relevant slicing algorithm which operates

on the compressed bytecode trace. Section 4.4 reports our experimental studies and Section 4.5 concludes the chapter.

4.1 Background

In this section, we recapitulate the definition of potential dependence. The material in this subsection was mostly studied in [7, 40].

Relevant slicing extends dynamic slicing by considering statements which may affect the slicing criterion. These statements are executed and do not affect the criterion. However, if these statements are changed, branches may be evaluated differently, or alternative methods may be invoked. We consider the following two situations for the execution of alternative code fragment due to program changes.

1. *Branch*: If the value of a variable used by the predicate of a branch statement is changed, the predicate may be evaluated differently.

2. *Method invocation*: If the programmer changes the assignment w.r.t. the object which invokes a method, or the declaration of parameters, an alternative method may be called.

In relevant slicing, we use the notion of *potential dependence* to capture the effects of these branch and method invocation statements. If these statements are evaluated differently, variables may be re-defined and affect the slicing criterion. We define potential dependence as follows; for any statement occurrence β, the corresponding statement is written as S_β.

Definition 4.1. Potential Dependence *Given an execution trace H of a program P, for any statement instances β and β' appearing in H, β is* potentially dependent *on an earlier branch or method invocation statement instance β' if and only if all of the following conditions hold.*

1. *β is not (transitively) dynamically control/data dependent on β' in H.*

2. *there exists a variable v used by β s.t. v is not defined between β' and β in H and v may be defined along an outgoing edge l of statement $S_{\beta'}$ where $S_{\beta'}$ is the statement at statement occurrence β'. That is, there exists a statement X satisfying the following.*

   - *(a) X is (transitively) control dependent on $S_{\beta'}$ along the edge l, and*
   - *(b) X is an assignment statement which assigns to variable v or a variable u which may be aliased to v.*

3. *the edge l is not taken at β' in H.*

The purpose of potential dependence is to find those statements which will be missed by dynamic slicing. If a branch or method invocation statement has affected the slicing criterion, we do not consider that it has potential influence, since the dynamic slice will always include such a statement. We use condition (1) to exclude dynamic control and data dependencies from potential dependencies. Conditions (2) and (3) ensure that β' has potential influence on β. The statement X in condition 2(b) appears in program P, but its execution is prohibited by the evaluation of branch/method invocation in β'. However, if it is executed (possibly due to a change in the statement $S_{\beta'}$), the value of variable v used at β will be affected. We cannot guarantee that v must be re-defined if β' is evaluated to take edge l. This is because even if β' is evaluated differently, execution of the assignment to v may be guarded by other (nested) conditional control transfer statements. Furthermore, condition (2) requires computation of static data dependence. In the presence of arrays and dynamic memory allocations, we can only obtain conservative static data dependencies in general. We now proceed to define a relevant slice, and introduce our relevant slicing algorithm to compute such a slice.

4.2 The Relevant Slice

To define relevant slices, first we define the notion of an *Extended Dynamic Dependence Graph* (EDDG). The EDDG captures dynamic control dependencies, dynamic data dependencies and potential dependencies w.r.t. a program execution. It is an extension of the *Dynamic Dependence Graph* (DDG) described in [6]. Each node of the DDG represents one particular occurrence of a statement in the program execution; edges represent dynamic data and control dependencies. The EDDG extends the DDG with potential dependencies, by introducing a dummy node for each branch statement occurrence or a method invocation statement occurrence. For each statement occurrence β in the execution trace, a non-dummy node $nn(\beta)$ appears in the EDDG to represent this occurrence. In addition, if β is a branch statement or a method invocation statement, a dummy node $dn(\beta)$ also appears in the EDDG. As far as the edges are concerned, the following are the incoming edges of any arbitrary node $nn(\beta)$ or $dn(\beta)$ appearing in the EDDG.

- dynamic control dependence edges from non-dummy node $nn(\beta')$ to $nn(\beta)$, iff. β' is dynamically control dependent on β.
- dynamic data dependence edges from both non-dummy node $nn(\beta')$ and dummy node $dn(\beta')$ (if there is a dummy node for occurrence β') to $nn(\beta)$, iff. β' is dynamically data dependent on β.
- potential dependence edges from non-dummy node $nn(\beta')$ and dummy node $dn(\beta')$ (if there is a dummy node for β') to $dn(\beta)$, iff. β' is potentially dependent on β.

These dependencies can be detected during backwards traversal of the execution trace.

What is a Relevant Slice The *relevant slice* is then defined based on the extended dynamic dependence graph (EDDG) as follows.

```
1    b = 1;
2    k = 1;
3    if (a > 1) {
4        if (b > 1){
5            k = 2
         }
     }
6    ... = k
```

Figure 4.1: Example: A "buggy" program fragment.

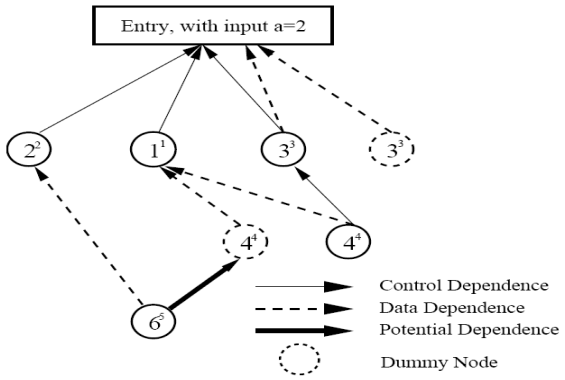


Figure 4.2: The EDDG for the program in Figure 4.1 with input a=2.

Definition 4.2. Relevant slice *for a slicing criterion consists of all statements whose occurrence nodes can be reached from the node(s) for the slicing criterion in the Extended Dynamic Dependence Graph (EDDG).*

In order to accurately capture the effects w.r.t. potential dependencies, we have introduced a dummy node into the EDDG for each occurrence of a branch or method invocation statement. The EDDG can then represent dynamic control dependence edges and potential dependence edges separately. This representation allows the reachability analysis for relevant slicing not to consider dynamic control dependence edges which are immediately after potential dependence edges, because such dynamic control dependencies cannot affect behaviors w.r.t. the slicing criterion. The following example shows how to compute a relevant slice, and the effect to introduce the dummy node.

```
1    x = 2;
2    y = 2;
3    if (x > 1)
4        y = 1;
5    if (y! = 1)
6        z = 3;
7    ... = z;
```

Figure 4.3: Example: compare our relevant slicing algorithm with Agrawal's algorithm.

Example Figure 4.1 shows an example program fragment, and Figure 4.2 shows the EDDG for the example program with input `a = 2`. The execution trace is $\langle 1^1, 2^2, 3^3, 4^4, 6^5 \rangle$. The resultant relevant slice w.r.t. variable `k` at line 6 consists of lines $\{1, 2, 4, 6\}$, because nodes $1^1, 2^2, 4^4$ and 6^5 can be reached from the criterion 6^5. Note that statement occurrence 6^5 is potentially dependent on 4^4, and 4^4 is dynamically control dependent on statement occurrence 3^3. However, changes related to line 3 will not affect the value of `k` at 6^5, nor will it decide whether line 6 will be executed. Therefore, line 3 should not be included into the slice. By using the dummy node to separate dynamic control dependencies and potential dependencies w.r.t. statement occurrence 4^4, we can easily exclude line 3 of Figure 4.1 from our relevant slice. Of course, if line 5 is executed when the program is executed with another input, both lines 3 and 4 will be included in the slice, because of dynamic control dependencies.

Comparison with previously proposed notions of relevant slices The notion of relevant slicing presented above is *not* completely new. Relevant slicing techniques (or other variants of it which try to extend dynamic slicing) have been studied [7, 40]. We now compare our notion of relevant slices against existing works, thereby pinpointing some salient features of our relevant slices.

Agrawal et al. first introduced relevant slicing, and applied it for regression testing [7]. Their relevant slice includes all nodes reachable from the slicing criterion in

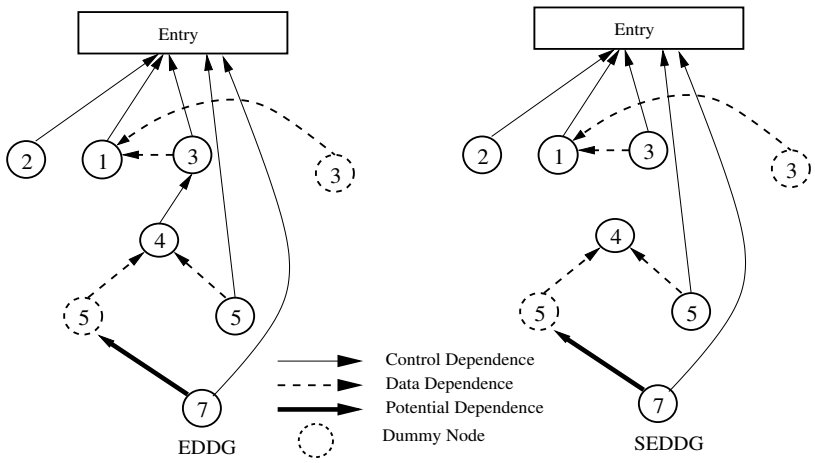


Figure 4.4: The EDDG and SEDDG for the program in Figure 4.3.

```
1    x = 1;
2    for (i = 0; i < 3; i++)
3        if (x % 2 == 0 )
4            z = 4;
5        x = i;
6    ... = z;
```

Figure 4.5: Example: compare our relevant slicing algorithm with Gyimóthy's algorithm.

a *Simplified Extended Dynamic Dependence Graph* (SEDDG). The SEDDG is constructed from the EDDG by removing dynamic control dependence edge w.r.t. a node β, if every path from the slicing criterion to β contains potential dependence edges. Because of the removed control dependence edges, some important statements may be excluded from the slice. Let us take the program in Figure 4.3 as an example. Figure 4.4 shows the corresponding EDDG and SEDDG. The relevant slice of [7] w.r.t. the criterion involving the value of z at line 7, will ignore line 1. However, if line 1 is changed to $x = 1$, the value of z at line 7 is affected.

Gyimóthy et al. [40] proposed a forward relevant slicing method for program debugging. Their relevant slice includes all nodes reachable from the slicing criterion in a *Accumulated Extended Dynamic Dependence Graph* (AEDDG). The AEDDG is constructed from the EDDG by adding an accumulated potential dependence edge

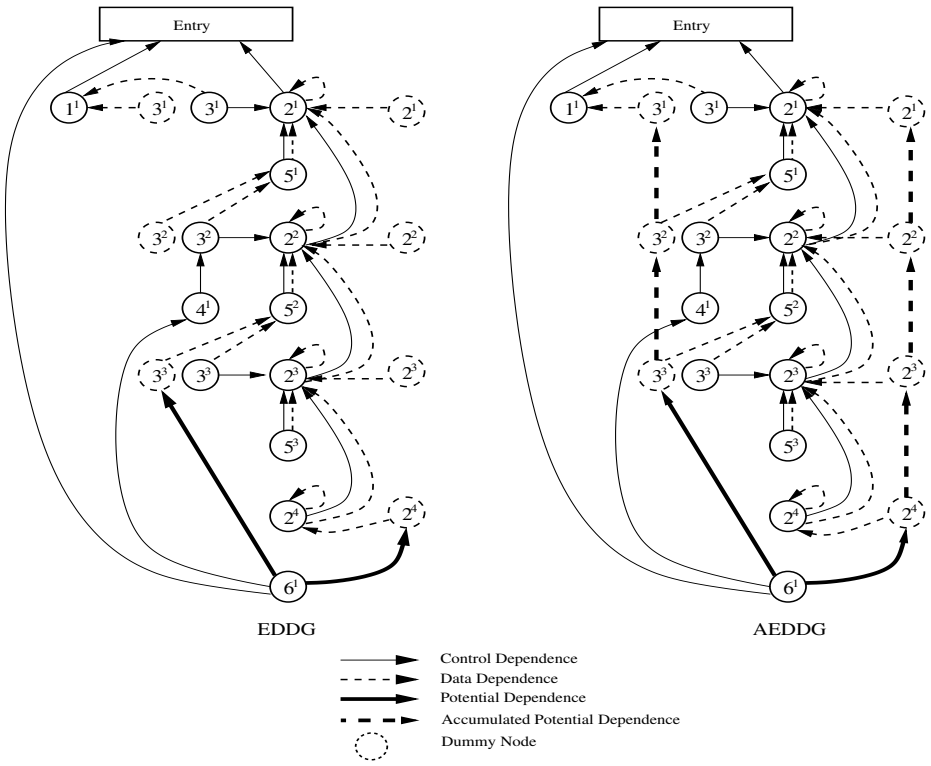


Figure 4.6: The EDDG and AEDDG for the program in Figure 4.5.

between two dummy nodes β and β' if and only if β and β' represent two contiguous execution instances of the same statement. The accumulated dependence edges in AEDDG may cause superfluous statements to be included into the relevant slice. Consider the example in Figure 4.5; figure 4.6 shows the corresponding EDDG and AEDDG. Line 1 will never affect z at line 6 of the program in Figure 4.5. However, this statement will be included into Gyimóthy's relevant slice.

4.3 The Relevant Slicing Algorithm

We now discuss how our dynamic slicing algorithm described in Section 3.2 (operating on compact Java bytecode traces) can be augmented to compute relevant slices. Thus, like the dynamic slicing algorithm, our relevant slicing algorithm operates on the compact bytecode traces described in Section 3.1.

As in dynamic slicing, relevant slicing is performed w.r.t. a slicing criterion

(H, α, V), where H is the execution history for a certain program input, α represents some bytecodes the programmer is interested in, and V is a set of variables referenced at these bytecodes. Again, the user-defined criterion is often of the form (I, l, V) where I is the input, and l is a line number of the source program. In this case, H represents execution history of the program P with input I, and α represents the bytecodes corresponding to statements at l.

Figure 4.7 presents a relevant slicing algorithm, which returns the relevant slice as defined in Definition 4.2. This algorithm is based on backward traversal of the compressed execution trace H (described in Section 3.1). The trace H contains execution flow and identities of accessed variables, so that we can detect various dependencies during the traversal. Although the slice can also be computed after constructing the whole EDDG, this approach is impractical because the entire EDDG may be too huge in practice. Before slicing, we compute the *control flow graph*, which is used to detect potential dependencies and to get last executed bytecode. We also pre-compute the static *control dependence graph* [31] which will be used at lines 21, 22 and 46 of Figure 4.7, and at line 9 of the `computePotentialDependence` method in Figure 4.8. Prior to running our relevant slicing algorithm we run static points-to analysis [9, 97]; the analysis results are used for determining potential dependencies.

In the relevant slicing algorithm, we introduce a global variable θ, to keep track of variables used by bytecode occurrences β, iff. β is included into the slice because of potential dependencies. In particular, each element in θ is of the form $\langle \beta, prop \rangle$, where β is an bytecode occurrence, and *prop* is a set of variables. Every variable in *prop* is used by a bytecode occurrence β', where

- β' is included into φ (the relevant slice) because of potential dependencies, and
- $\beta' = \beta$, or β' is (transitively) dynamically control dependent on β.

The purpose of the δ set (set of unexplained variables) is the same as in the

dynamic slicing algorithm. That is, the δ set includes variables used by bytecode occurrence β for explanation, where β is included into the slice φ because β belongs to the slicing criterion, or there is any bytecode occurrence in φ which is dynamically control/data dependent on β.

For each bytecode occurrence β of bytecode b^{β} encountered during the backward traversal for slicing, we first check if β has affected the slicing criterion via dynamic control/data dependencies as the in dynamic slicing algorithm of Figure 3.2. In particular, line 18 checks whether β belongs to the slicing criterion. Line 21 checks dynamic control dependencies if b^{β} is a conditional control transfer bytecode. Line 26 checks dynamic data dependencies. With the introduction of θ, variables in both δ and θ need explanation. Consequently, the `computeDataDependence` method has been slightly changed, as re-defined in Figure 4.9. If all the three checks fail, the algorithm proceeds to check the potential dependencies at line 37, by invoking the `computePotentialDependence` method in Figure 4.8.

For the computation of potential dependencies, we need to pre-compute the effects of various outcomes of a control transfer bytecode. Each such outcome triggers a different code fragment whose effect can be summarized by all possible assignment bytecodes executed. This summarization is used by the `computePotentialDependence` method in Figure 4.8. Note that δ is used to check dynamic data dependencies (line 26 of Figure 4.7) as well as potential dependencies (line 6 in Figure 4.8).

The `intersect(`MDS`,`δ`)` method used by the `computePotentialDependence` method in Figure 4.8 checks whether the execution of the alternative path of a bytecode b^{β} may define some variables which are used by bytecode occurrences in the slice. Here MDS includes variables which may be defined if b^{β} is evaluated differently, and δ includes variables which have affected the slicing criterion and need explanation. This check is non-trivial because MDS contains static information, while δ contains dynamic information. Let $meth$ be the method that the bytecode b^{β} belongs to, and

```
1   (H, α, V)= the slicing criterion
2   δ= ∅, a set of variables whose values need to be explained
3   φ= ∅, the set of bytecode occurrences which have affected the slicing criterion
4   op_stack= empty, the operand stack for simulation
5   fram= empty, the frames of the program execution
6   θ= ∅, (bytecode occurrences, used variables) included in slice due to potential dependencies
7   relevantSlicing()
8       b^β = get last executed bytecode from H;
9       while (b^β is defined)
10          if (b^β is a return bytecode)
11              new_fram= createFrame();new_fram.γ=∅;
12              push(fram, new_fram);
13          last_fram=null;
14          if (b^β is a method invocation bytecode)
15              last_fram=pop(fram);
16          β = current occurrence of bytecode b^β;
17          curr_fram = the top of fram;
18          if (β is the last occurrence of b^β and b^β ∈ α)
19              use_vars = V ∩ variables used at β;
20              φ = φ ∪ {β};
21          if (computeControlDependence(b^β, curr_fram, last_fram) )
22              BC= {β' | β' ∈ curr_fram.γ and β' is dynamically control dependent on β};
23              curr_fram.γ = curr_fram.γ − BC ;
24              use_vars = variables used at β;
25              φ = φ ∪ {β};
26          if (computeDataDependence(β, b^β))
27              def_vars = variables defined at β;
28              δ = δ − def_vars;
29              use_vars = variables used at β;
30              φ = φ ∪ {β};
31              for (each ⟨β', prop'⟩ in θ)
32                  prop' = prop' − def_vars;
33          if (β ∈ φ)
34              curr_fram.γ = curr_fram.γ ∪ {β};
35              δ = δ ∪ use_vars;
36          else
37              if (computePotentialDependence(β, b^β))
38                  φ = φ ∪ {β};
39                  if ( b^β is a branch bytecode)
40                      use_vars = variables used at β; θ = θ ∪ {⟨β, use_vars⟩};
41                  if ( b^β is a method invocation bytecode)
42                      o = the variable to invoke a method; θ = θ ∪ {⟨β, {o}⟩};
43          if (b^β is a branch bytecode or method invocation bytecode)
44              prop=∅;
45              for (each ⟨β', prop'⟩ in θ)
46                  if (β' = β or β' is control dependent on β)
47                      prop = prop ∪ prop'; θ = θ − {⟨β', prop'⟩};
48              θ = θ ∪ {⟨β, prop⟩};
49          updateOpStack(β, b^β);
50          β = getPrevBytecode(β, b^β);
51      return bytecodes whose occurrences appear in φ;
```

Figure 4.7: The relevant slicing algorithm.

```
1   computePotentialDependence (β: bytecode occurrence, b^β: bytecode)
2     if ( b^β is a branch or method invocation bytecode )
3       for (each possible outcome x of b^β)
4         if (outcome x of b^β did not occur at β )
5           MDS = the set of variables which may be defined
                    when outcome x occurs;
6           if (intersect(MDS, δ))
7             return true;
8           for (each ⟨β', prop'⟩ in θ)
9             if (β' is not control dependent on β and intersect(MDS, prop') )
10              return true;
11    return false;
```

Figure 4.8: Detect potential dependencies for relevant slicing.

curr_invo be the current invocation of *meth*. Note that both MDS and δ include local variables and fields. Every local variable in MDS is also a local variable of method *meth*, and is represented by its identity $\langle var \rangle$. Every local variable in δ for explanation is represented as $\langle invo, var \rangle$, where *invo* refers to the method invocation which uses the local variable *var*. Many points-to analysis algorithms [9, 97] represent abstract memory locations of objects using their possible allocation sites. Thus, we represent an object field in MDS as $\langle site, name \rangle$, where *site* refers to a possible allocation site of the object, and *name* refers to the name of the field. We also represent an object field in δ as $\langle site, \_timestamp, name \rangle$, where *site* refers to the allocation site of the object, *_timestamp* distinguishes between objects created at the same allocation site, and *name* refers to the name of the field. Note that *_timestamp* is only important for detecting dynamic data dependencies. The `intersect(`MDS`,`δ`)` method returns true iff:

- there is a local variable where $\langle var \rangle \in MDS$ and $\langle curr\_invo, var \rangle \in \delta$, or
- there is a field where $\langle site, name \rangle \in MDS$ and $\langle site, \_timestamp, name \rangle \in \delta$.

We do not consider partial results (*i.e.* operands in the operand stack) for potential dependencies because partial results will not be transferred between bytecodes of different statements.

```
1   computeDataDependence (β: bytecode occurrence, b^β: bytecode)
2     if (β defines a variable)
3       if (β defines a static field or local variable)
4         def_loc= get address of the defined static field or local variable from class files;
5       if (β defines an object field or an array element)
6         G= compressed operand sequence for b^β in the compact bytecode trace H
7         π= a root-to-leaf path for G;
8         def_loc= getLast(G, π);
9       if ( def_loc ∈ δ )
10        return true;
11      for (each ⟨β', prop'⟩ in θ)
12        if (def_loc ∈ prop')
13          return true;
14    ω= the set of bytecode occurrences in top def_op(b^β) entries of op_stack;
15    if ( ω ∩ φ ≠ ∅)
16      return true;
17    return false;
```

Figure 4.9: Detect dynamic data dependencies for relevant slicing.

The proof of correctness of the relevant slicing algorithm in Figure 4.7 is similar to that of the dynamic slicing algorithm in Figure 3.2. The detailed proof of correctness can be found in Appendix A.3.

Now we analyze the cost of the relevant slicing algorithm in Figure 4.7. The space overheads of the slicing algorithm are $O(N^2 + m^3)$, and the time overheads are $O(m^2 \cdot N^3)$, where N is the length of the execution, and m is the number of bytecodes of the program. Since the relevant slicing algorithm is similar with the dynamic slicing algorithm in Figure 3.2, the cost analysis is also similar except costs w.r.t. the θ and MDS.

The θ set contains at most N elements of the form $\langle \beta, prop \rangle$, because every bytecode occurrence β has at most one element $\langle \beta, prop \rangle$ in θ. The size of each $prop$ is $O(N)$, because at most N bytecode occurrences are (transitively) dynamically control dependent on β and every bytecode occurrence uses a constant number of variables. Consequently, the space overheads of θ are $O(N^2)$.

Each MDS includes the set of variables which may be defined when a specific outcome of a branch bytecode occurs. If m is the number of bytecodes in the program,

clearly there are at most m MDSs. How do we bound the size of each MDS? Each MDS may have at most m assignments and each of these assignments may affect at most m locations (provided we distinguish locations based on allocation sites as is common in points-to analysis methods). Thus, the size of each MDS is $O(m^2)$. Since there are at most m MDSs, the space overheads of maintaining the MDSs are $O(m^3)$. The other portions of the relevant slicing algorithm are taken from dynamic slicing; the space overheads of these portions of the relevant slicing algorithm are $O(N)$, as explained in Section 3.2. So, the overall space overheads of our relevant slicing algorithm are $O(N^2 + m^3)$.

We now calculate the time overheads of relevant slicing. First we estimate the time overheads for maintaining the θ set in the relevant slicing algorithm. Note that the θ set contains $O(N)$ elements of the form $\langle \beta, prop \rangle$. The set difference operation at line 32 of Figure 4.7 is executed $O(N^2)$ times. Since each $prop$ set contains $O(N)$ variables, the overall time overheads to perform the set difference operation at line 32 of Figure 4.7 are $O(N^3)$. The total time overheads to perform the set union operation at lines 40, 42 and 48 of Figure 4.7 are $O(N^2)$, because lines 40, 42 and 48 are executed $O(N)$ times and the size of the θ set is $O(N)$. Given a bytecode occurrence β and the θ set, there are a constant number of elements $\langle \beta', prop' \rangle \in \theta$, where β' is (directly) dynamically control dependent on β. This is because there is no explicit `goto` statement in Java programs. Different occurrences of the same bytecode are dynamically control dependent on different bytecode occurrences. Note that there are a constant number of bytecodes which are statically control dependent on the bytecode b^β of occurrence β. Thus, there are a constant number of bytecode occurrences which are *directly* dynamically control dependent on β. In addition, every bytecode occurrence β' has at most one $\langle \beta', prop' \rangle$ in θ. Consequently, line 47 is executed $O(N)$ times, and the overall time overheads to execute line 47 are $O(N^3)$. The total time overheads to perform check at line 12 of Figure 4.9 are $O(N^3)$, which

is similar to perform set difference operation at line 32 of Figure 4.7.

Now, we analyze the overall time overheads to perform the `intersect` operation by the `computePotentialDependence` method in Figure 4.8. The `intersect` operation at line 6 of Figure 4.8 can be executed at most N times. In each execution of the intersect operation we compare the contents of MDS and δ. Since the size of MDS is $O(m^2)$ and the size of the set δ is $O(N)$, therefore the time overheads of a single `intersect` operation are $O(N \cdot m^2)$. Thus, the total time overheads to execute all the `intersect` operations at line 6 of Figure 4.8 are $O(N \cdot N \cdot m^2)$. Similarly, the total time overheads to execute all the `intersect` operations at line 9 of Figure 4.8 are $O(N^3 \cdot m^2)$, since this `intersect` operation is executed $O(N^2)$ times.

The other portions of the relevant slicing algorithm are taken from dynamic slicing; the time complexity of these portions of the relevant slicing algorithm is $O(N^2)$, as discussed in Section 3.2. This leads to a time complexity of $O(N^3 \cdot m^2)$ for our relevant slicing algorithm.

4.4 Experimental evaluation

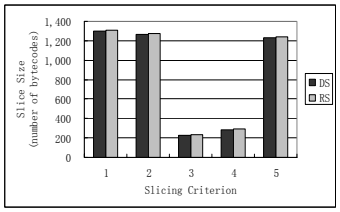
In order to evaluate the performance of our relevant slicing algorithm, we implemented a prototype based on the infrastructure which we presented in Chapter 3. We applied the prototype to the subject programs described in Section 3.3.1, *i.e.* the same subject programs for studying the compact trace representation in Chapter 3.

In the experiments, we measured the sizes of dynamic and relevant slices and the time overheads to compute these slices. Thus, apart from studying the absolute time/space overheads of dynamic and relevant slicing, these experiments also serve as comparison between dynamic and relevant slicing,

We did not evaluate the effectiveness of dynamic and relevant slicing for debugging, since this has been thoroughly studied in [100, 118]. For each subject from the Java Grande Forum benchmark suite, we randomly chose five distinct slicing criteria

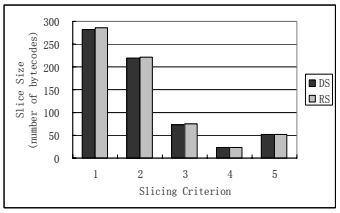
because of their relatively simple program structures; for other bigger subjects, we randomly chose fifteen distinct slicing criteria.

Note that static points-to analysis results are required in our relevant slicing algorithm (for computing potential dependencies). To compute points-to sets, we used the *spark* toolkit [66] which is integrated in the compiler optimization framework *soot* [99].



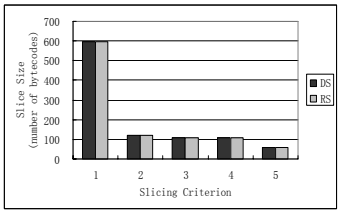
Pgm.	DS	RS	%
2,939	862	868	0.70

Crypt



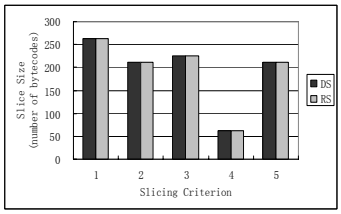
Pgm.	DS	RS	%
1,656	130	131	0.77

SOR



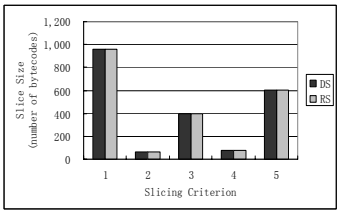
Pgm.	DS	RS	%
2,327	199	199	0.00

FFT



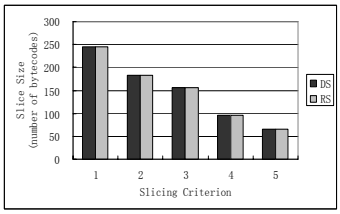
Pgm.	DS	RS	%
1,682	195	195	0.00

HeapSort



Pgm.	DS	RS	%
2,885	420	420	0.00

LUFact



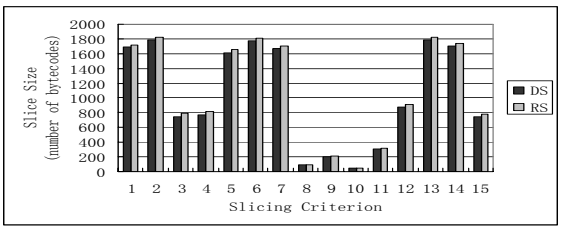
Pgm.	DS	RS	%
1,800	149	149	0.00

Series

Figure 4.10: Compare sizes of relevant slices with those of dynamic slices.

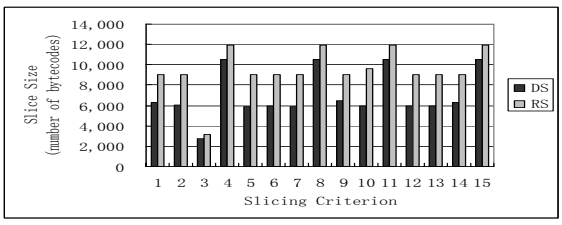
4.4.1 Sizes of Dynamic Slices and Relevant Slices

Figure 4.10 and 4.11 present the slice sizes. *Pgm.* represents sizes of the subject programs. *DS* and *RS* represent average sizes of dynamic slices and relevant slices, respectively. All sizes are reported as the number of bytecodes. The last % column represents the increased sizes of relevant slices (*i.e.* $\frac{RS-DS}{DS}$) in percentage. As we can see, most dynamic slices and relevant slices were relatively small, less than 30% of corresponding source programs on average in our experiments. This is because



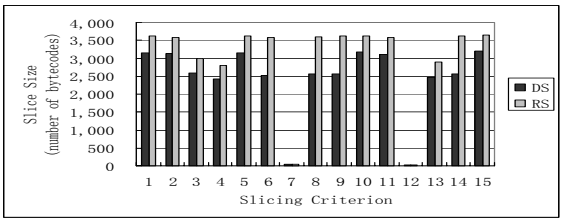
Pgm.	DS	RS	%
8,797	1,054	1,082	2.66

_201_compress



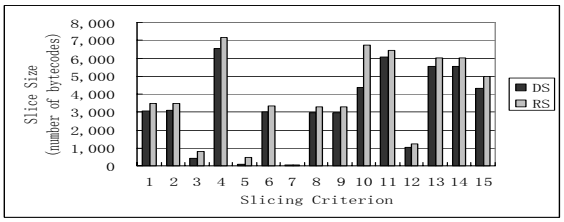
Pgm.	DS	RS	%
34,019	7,023	9,440	34.42

_202_jess



Pgm.	DS	RS	%
8,794	2,448	2,988	22.06

_209_db



Pgm.	DS	RS	%
22,077	3,283	3,784	15.26

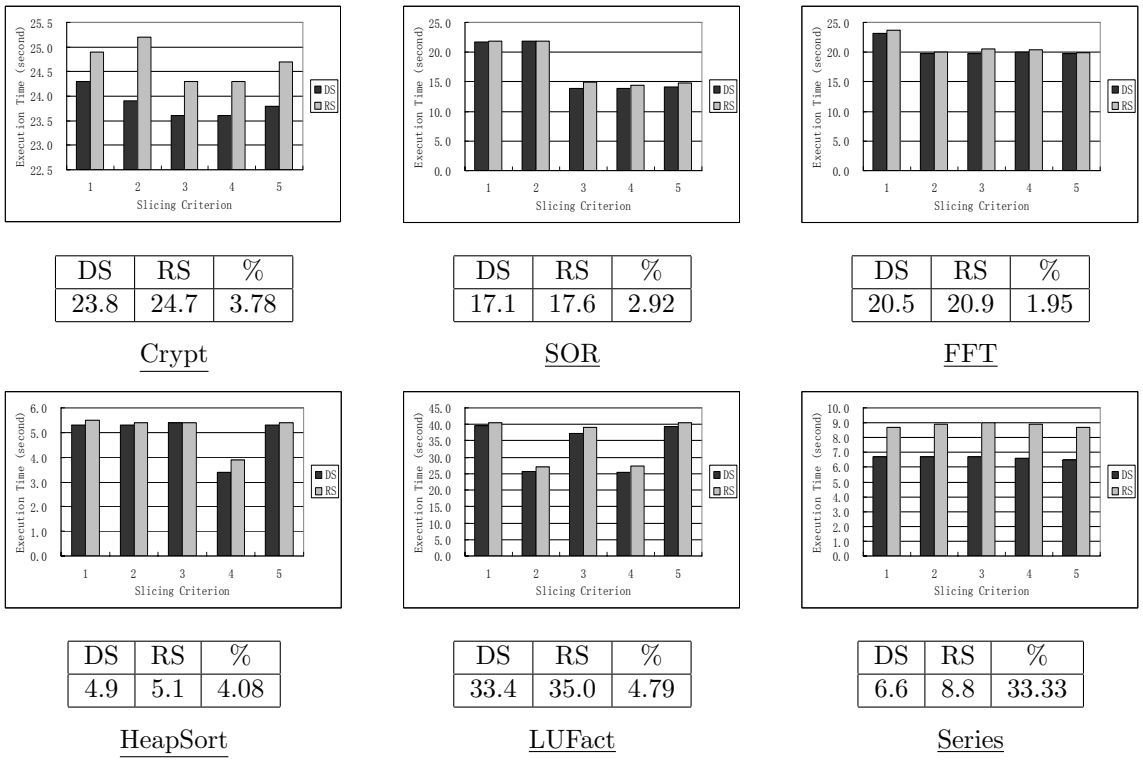
JLex

Figure 4.11: Compare sizes of relevant slices with those of dynamic slices.

programs often consist of several parts which work (almost) independently. The irrelevant portions are excluded from both dynamic slices and relevant slices.

In our experiments, control and data structures of subject programs from Java Grande Forum benchmark suite are quite simple. Furthermore, class hierarchies of these programs are simple, with only limited use of inheritance. So, there are not many candidate branches/method invocations for potential dependencies. The relevant slices are almost the same as corresponding dynamic slices for these programs, as shown in Figure 4.10.

On the other hand, other subject programs (*i.e.* _201_compress, _202_jess, _209_db, and JLex) are more complex. These programs have more sophisticated control structures. In addition, these programs use inheritance and method overloading and overriding, so that different methods may be called by the same method invocation bytecode. Both of the factors lead to potential dependencies between bytecode



DS	RS	%
23.8	24.7	3.78

Crypt

DS	RS	%
17.1	17.6	2.92

SOR

DS	RS	%
20.5	20.9	1.95

FFT

DS	RS	%
4.9	5.1	4.08

HeapSort

DS	RS	%
33.4	35.0	4.79

LUFact

DS	RS	%
6.6	8.8	33.33

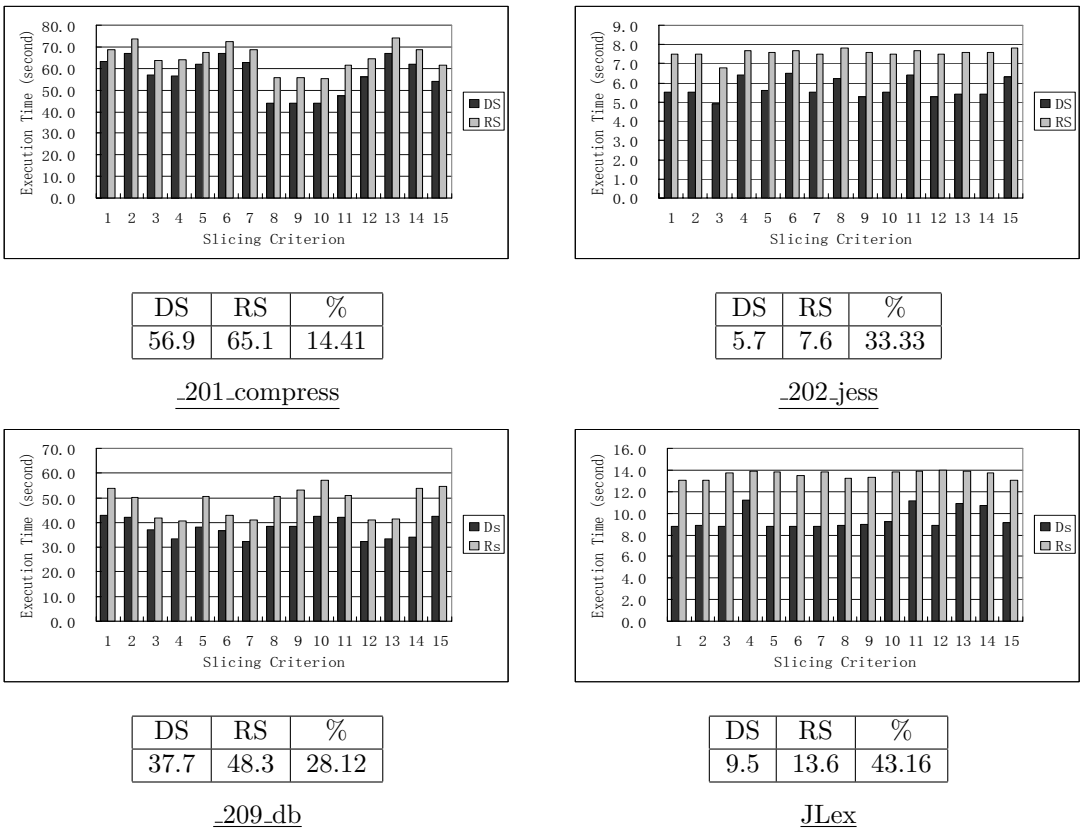
Series

Figure 4.12: Compare time overheads of relevant slicing with those of dynamic slicing.

occurrences. More importantly, these subject programs involve substantial use of Java libraries, such as collection classes and I/O classes. These library classes contribute a lot to relevant slices, because of their complex control structures and usage of object-oriented features like method overloading (which lead to potential dependencies). In fact, if we do not consider potential dependencies inside such library classes, the average sizes of relevant slices for _201_compress, _202_jess, _209_db, and JLex were 1072, 8393, 2523, and 3728, which were 1.71%, 19.51%, 3.06%, 13.55% bigger than corresponding dynamic slices, respectively.

4.4.2 Time overheads

Figure 4.12 and 4.13 show the time overheads to compute dynamic slices and relevant slices. *DS* and *RS* represent average time to perform dynamic slicing and relevant



DS	RS	%
56.9	65.1	14.41

_201_compress

DS	RS	%
5.7	7.6	33.33

_202_jess

DS	RS	%
37.7	48.3	28.12

_209_db

DS	RS	%
9.5	13.6	43.16

JLex

Figure 4.13: Compare time overheads of relevant slicing with those of dynamic slicing.

slicing, respectively. All time is reported in second. The last % column represents the increased time for relevant slicing (*i.e.* $\frac{RS-DS}{DS}$) in percentage. Clearly, the time overheads to compute dynamic slices and relevant slices are sensitive to choice of programs and slicing criteria. According to our slicing algorithms in Figure 3.2 and 4.7, the time overheads mainly come from: (1) extracting operand sequences of bytecodes and backwards traversal, and (2) updating/comparing various sets to detect dependencies. For a particular execution trace, the first task is common for slicing w.r.t. every slicing criterion, and the second task is sensitive to the sizes of the slices. For bigger slices, their sets to detect dependencies are also bigger during slicing, resulting in larger time overheads.

4.4.3 Effect of points-to analysis

As mentioned earlier, results from static points-to analysis are required in our relevant slicing algorithm for computing potential dependencies. To compute points-to sets, we used the *spark* toolkit [66] which is integrated in the compiler optimization framework *soot* [99]. Spark is a flexible framework for experimenting with points-to analysis for Java. It supports various points-to analysis with different trade-offs between precision and efficiency. One trade-off comes from the way to construct the call graph, where the call graph is essential for inter-procedural points-to analysis. Spark can either construct the call graph ahead of time using *class hierarchy analysis* (CHA), or *on the fly* (otf) as the analysis proceeds. Another trade-off comes from the way to represent the field deference expressions. The *field-based* (fb) analysis ignores the base objects in field deference expressions, considering only the fields; while the *field sensitive* (fs) analysis parameterizes each field deference expression by its base object for greater precision.

In our experiment, we used spark to generate points-to sets with different precision, and evaluated the impact of points-to analysis on relevant slicing. In particular, we used otf & fs setting to produce precise points-to sets, and CHA & fb setting to produce less precise sets, since [67] suggests that otf (fs) analysis is more precise than CHA (fb) analysis respectively. We computed the relevant slices (for each of the eight programs with three different criteria) twice: once using the less precise points-to information and once using the more precise information. In our experiments, the relevant slice sizes and time overheads for computing them did not change due to this variation of points-to information .

4.4.4 Summary and Threats to Validity

In our experiments, relevant slices were often bigger than dynamic slices (since they consider *potential dependencies*), but relevant slices were still relatively small compared against the entire program. Of course, relevant slicing for a given criterion takes more time than dynamic slicing for the same criterion. However, the increased time overheads, which depend on the choice of slicing criterion, the control structures and data flow structures of the program, were moderate (between 1.95% and 43.16%)in our experiments .

It is also possible to encounter a situation where the dynamic slice is quite small compared to the program, but the relevant slice is much bigger than the dynamic slice. Since relevant slice computation involves detecting potential dependencies and potential dependencies involve computing static data dependencies, a lot depends on the result of the points-to analysis used to detect static data dependencies. If the points-to analysis is very conservative, it may lead to a large relevant slice.

4.5 Summary

This chapter has studied relevant slices. Over and above dynamic slices, relevant slices capture those executed statements which if changed can change the execution flow and affect the slicing criterion. This chapter has presented a relevant slicing algorithm which proceeds by backwards traversal of the execution trace. We have compared the precision of our relevant slices to existing works [4, 12]. We have also conducted an experimental study to compare sizes/time overheads of relevant slicing and dynamic slicing.

CHAPTER 5

HIERARCHICAL EXPLORATION OF THE DYNAMIC SLICE

Traditionally, a dynamic slicing algorithm returns a flat set of statements as the dynamic slice to the programmer for inspection. However, the dynamic slice tends to be huge. Our experience shows that, the dynamic slices often contain more than 400 statements for realistic programs. It is difficult for the programmer to understand and employ such big slices for debugging and comprehension.

In this chapter, we present *hierarchical dynamic slicing* to help the programmer understand/explore large dynamic slices. Hierarchical dynamic slicing proceeds by systematically interleaving computation and comprehension of dynamic dependencies for program debugging. During this process, a program execution trace is divided into ***phases*** at various levels of granularity. We then perform dynamic slicing w.r.t. the execution of the whole program and the observable error. However, we only report dynamic data/control dependencies which (1) are reachable from the slicing criterion, and (2) span *across* phases. Detailed data and control dependence computation inside each phase is not exposed to the programmer, thereby reducing program understanding effort (as compared to inspecting the dynamic slice). The programmer then examines these inter-phase dependencies and identifies a likely suspicious phase. The suspicious phase is then subjected to further investigation in a similar manner, *i.e.* dividing the phase into *sub-phases*, performing dynamic slicing w.r.t. the execution of this suspicious phase and the suspicious inter-phase dependency, and reporting

dynamic data/control dependencies which are (1) reachable from the suspicious inter-phase dependency, and (2) across *sub-phases*. This process continues until the error is identified.

During hierarchical dynamic slicing, the generation of "*program phases*" is critical. In the next section, we present our notion of phases, which is then used in Section 5.2 to develop our slicing algorithm. We also present an algorithm for dividing an execution trace into phases in a hierarchical fashion; this hierarchy corresponds to the levels of hierarchy we will explore for uncovering the dynamic dependencies gradually. The phase division algorithm divides a trace along control structure boundaries such as procedure calls and loop boundaries. We compare our notion of program phases with previous works on *phase detection* (*e.g.*, see [28]). These works have defined phases based on aggregate performance metrics (*e.g.*, which basic blocks are executed may define a phase).

The remainder of this chapter is organized as follows. Section 5.1 introduces our notion of program phases, and compares our phase computation with previous phase detection algorithms which were proposed for program optimization. In Section 5.2, we present our *hierarchical dynamic slicing* algorithm. Section 5.3 summarizes the experimental results of our slicing algorithm, and Section 5.4 concludes the chapter.

5.1 Phases in an Execution Trace

Before giving our phase division method, we discuss past work on phase detection in the programming languages community (Section 5.1.1). These works detect phases based on program performance characteristics. The phases detected can be used/exploited for dynamically re-configuring the hardware on which the program runs, that is, the hardware is re-configured when the program enters a new phase. In Section 5.1.2, we present our phase division algorithm and compare it with these past works on phase detection.

5.1.1 Phase Detection for Improving Performance

Phase detection for program optimization has been a rich area of research [28, 77, 94]. Given an execution trace H, these techniques typically divide the trace H into *fixed-length intervals*. Program performance related information for each interval (such as basic block vectors, which, roughly speaking, capture the relative occurrences of various basic blocks in an interval) are collected. Finally, consecutive intervals with similar information are clustered into a single phase.

In order to study whether such a definition of phase is useful program understanding/debugging, we implemented a prototype which detects phases of an execution trace using basic block vectors (BBVs) [94]. The prototype is based on the open source Kaffe virtual machine.[1] It divides an execution trace into fixed length intervals; we set the length of these intervals to 100,000 bytecode instances in our experiments. For every fixed-length interval, we maintain a basic block vector (BBV) to collect the frequencies of basic blocks executed in this interval. Given an interval ρ, its basic block vector BBV_ρ is a vector of length n where n is the number of basic blocks in the program. For any basic block b s.t. $1 \leq b \leq n$

$$BBV_\rho[b] = Occur_{b,\rho} * Num\_bytecode_b$$

where $Occur_{b,\rho}$ is the number of occurrences of basic block b in the interval ρ and $Num\_bytecode_b$ is the number of bytecodes in basic block b. Thus, the basic block vector estimates the (relative) execution times spent in each basic block. Once the BBV for a fixed length interval is computed, we *normalize* it by dividing each element of the BBV by the sum of all the elements in the vector. The similarity between two consecutive intervals is measured by the Manhattan distance of their normalized BBVs. In particular, given normalized BBVs u and v of length n, their Manhattan distance can be computed as the sum of absolute differences between the elements of

[1] Available from `http://www.kaffe.org`

vectors u, v. Thus,

$$ManhattanDis(u, v) = \Sigma_{i=1}^{n} |u[i] - v[i]|$$

The Manhattan distance of two normalized BBVs lies between 0 and 2, where 0 indicates that the two BBVs are identical and 2 indicates that the two BBVs are completely different. If the Manhattan distance of two consecutive execution intervals is less than a certain threshold (say 1), existing methods for phase detection (such as [94]) cluster these intervals into a single phase.

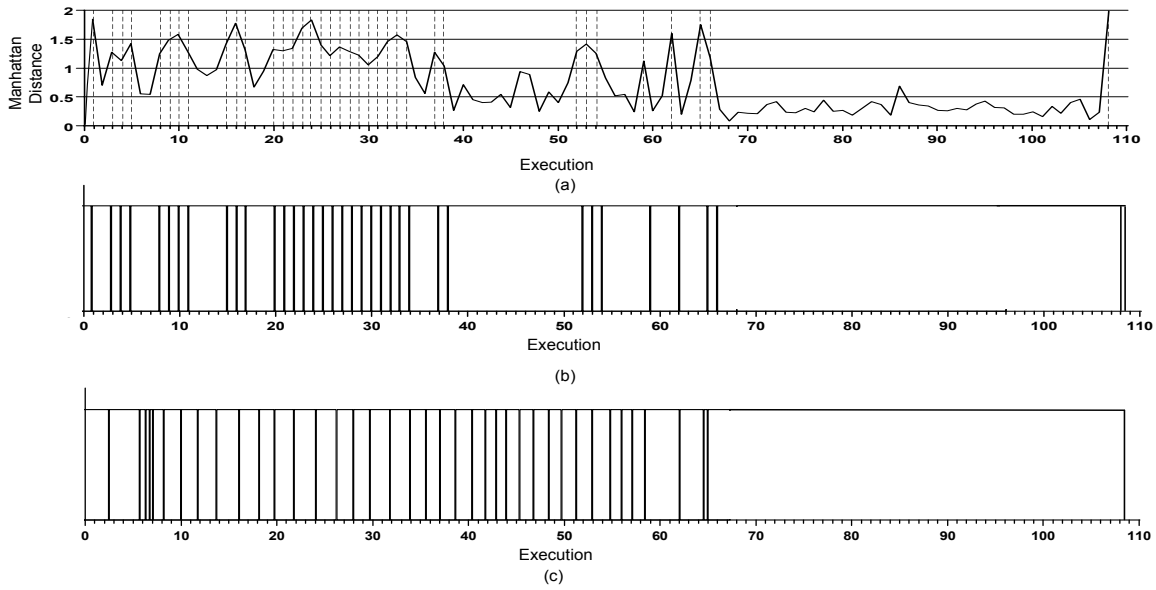


Figure 5.1: (a) Manhattan distances. (b) Phase boundaries w.r.t. manhattan distances. (c) Phase boundaries generated by hierarchical dynamic slicing

In Figure 5.1(a) and 5.2(a), we plot the Manhattan distances for consecutive intervals in the execution trace for `jess` and `db` programs drawn from the SPEC JVM benchmark suite [96]. The solid line represents Manhattan distances between consecutive execution intervals. Dashed lines represent phases with threshold=1.0. The program inputs used to generate the execution traces are taken from the standard inputs that come with these benchmarks in the SPEC JVM suite. `jess` is a Java

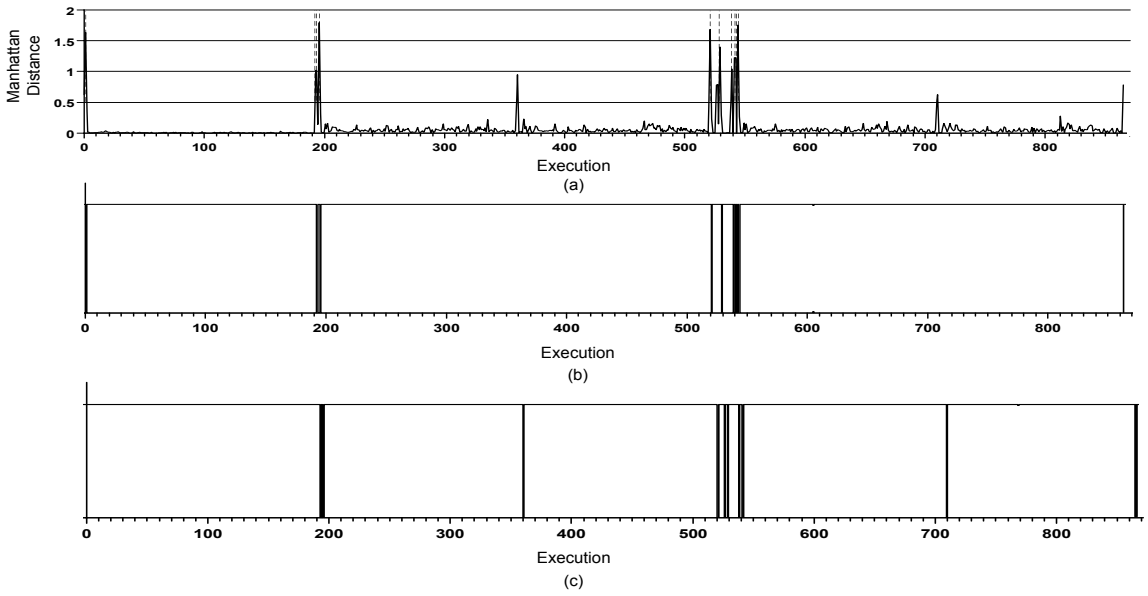


Figure 5.2: (a) Manhattan distances. (b) Phase boundaries w.r.t. manhattan distances. (c) Phase boundaries generated by hierarchical dynamic slicing

Expert Shell System based on NASA's CLIPS expert shell system. The system iteratively reads declarative rules and commands from the input; when the command "run" is read, the system tries to "reason" using the knowledge from those declarative rules. The `db` program performs certain operations on a database, such as insertion, deletion, and sorting. We set a threshold of 1 for the Manhattan distances, that is, consecutive intervals with a Manhattan distance greater than 1 are considered to be in different phases. Figure 5.1(a) and 5.2(a) show the phases which are thus detected, using dashed vertical lines. These phase boundaries are clarified for easy visualization in Figure 5.1(b) and 5.2(b), with threshold = 1.0.

From the phases calculated for the `jess` and `db` programs, we observe the following about existing phase detection techniques (which essentially aim to improve program performance on dynamically adaptable hardware). We note that existing phase detection methods chop an execution trace into fixed length intervals, and these interval boundaries may not closely correspond to the end of different logical

operations performed in a program execution trace. This is clearly the main difficulty in using the phases produced by existing phase detection methods for program understanding. However, this difficulty manifests itself in interesting and subtle ways as mentioned in the following. *Even though the following observations are made for phase detection using Basic Block Vectors, they generally apply to any existing phase detection method which calculates phases based on* **aggregate** *execution metrics* (such as cache miss, branch frequency [28, 94]).

- Existing phase detection methods start by dividing a trace into fixed-length intervals, and then combine some of these intervals into phases. Thus, they *cannot identify very short "phases" of program behavior* whose trace may be shorter than the length of an interval. As an example, consider interval number 521 in the `db` program in Figure 5.2(a). In this interval, the program performs nine different operations on a database. The execution trace for these operations fits into a single interval, thus it will be reported as a single phase (instead of nine phases). As we will see later, our phase detection method does not set an a-priori lower bound on the length of a phase and thus this problem is easily rectified.

- The manner of combining consecutive intervals into a single phase in existing methods may also be problematic.

  (a) First of all, intervals which are detected to be in the same phase by existing methods may correspond to very different logical operations. The reasoning is simple — even though the trace for two logical operations are different, the Basic Block Vectors (BBV is an aggregate quantity) may be similar.

  (b) Secondly, consecutive intervals within an execution trace may have very different execution traces as well as Basic Block Vectors. Such intervals will be placed in different phases by existing phase detection methods. However, the

intervals taken together may constitute a procedure which is accomplishing a specific task in the program. Thus, from a logical point of view, they should be treated as one single phase.

The difficulties mentioned in the preceding will be overcome by our phase division method, which we now elaborate.

5.1.2 Program Phases for Debugging

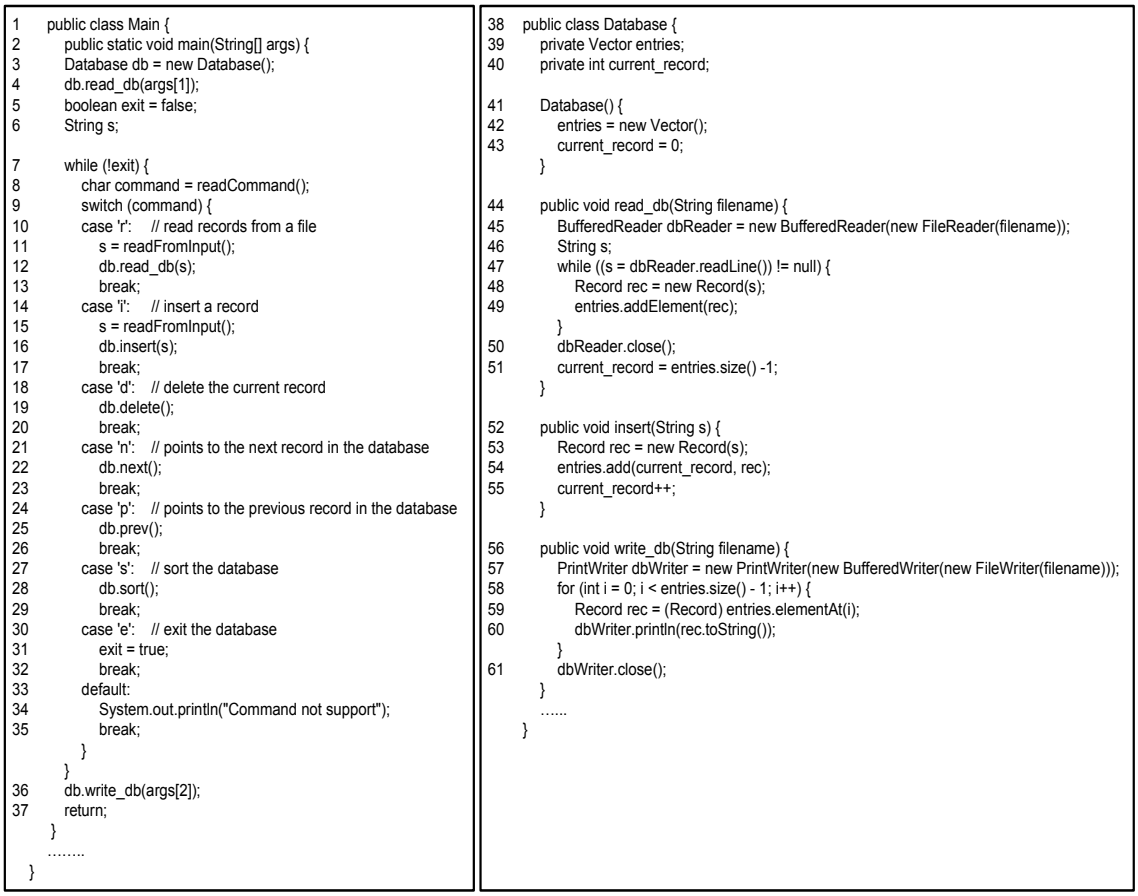
Our definition of phase is based on the syntax structure of a program. The intuition is that programmers often use loops and methods to implement specific tasks within a program. Furthermore, programs are constructed hierarchically. Thus, a task which is implemented by a procedure may contain sub-tasks which are implemented by other procedures and/or loops. Our phase division algorithm is based on (and exploits) these observations regarding program development.

We now present our notion of phases using an example. The example program appears in Figure 5.3; it simulates a database system, where a user can perform various operations such as insertion, deletion and sorting. The example is similar to the `db` program in the SPEC JVM benchmark suite. The `main()` method of the program initializes a database (lines 3-5), and then presents the user with seven options (lines 7-35). Based on the user's choice, the database system invokes one of six methods, defined in the `Database` class. Finally, the `main()` method writes the database to a file before the system terminates (lines 36-37).

Consider any execution trace of the database program (given as a sequence of line numbers).

$$3, 4, 5, 7-13, 7-9, 14-17, 7-9, 30-32, 36, 37$$

Using our phase division algorithm, we compute three phases at the top-level: (1) lines 3-5 of Figure 5.3, (2) lines 7-35 of Figure 5.3, and (3) lines 36-37 of Figure 5.3. These phases exactly correspond to the tasks of database initialization, data processing,



```
1    public class Main {
2       public static void main(String[] args) {
3       Database db = new Database();
4       db.read_db(args[1]);
5       boolean exit = false;
6       String s;

7       while (!exit) {
8          char command = readCommand();
9          switch (command) {
10         case 'r':    // read records from a file
11            s = readFromInput();
12            db.read_db(s);
13            break;
14         case 'i':    // insert a record
15            s = readFromInput();
16            db.insert(s);
17            break;
18         case 'd':    // delete the current record
19            db.delete();
20            break;
21         case 'n':    // points to the next record in the database
22            db.next();
23            break;
24         case 'p':    // points to the previous record in the database
25            db.prev();
26            break;
27         case 's':    // sort the database
28            db.sort();
29            break;
30         case 'e':    // exit the database
31            exit = true;
32            break;
33         default:
34            System.out.println("Command not support");
35            break;
           }
        }
36      db.write_db(args[2]);
37      return;
      }
      ……..
  }
```

```
38  public class Database {
39     private Vector entries;
40     private int current_record;

41     Database() {
42        entries = new Vector();
43        current_record = 0;
       }

44     public void read_db(String filename) {
45        BufferedReader dbReader = new BufferedReader(new FileReader(filename));
46        String s;
47        while ((s = dbReader.readLine()) != null) {
48           Record rec = new Record(s);
49           entries.addElement(rec);
          }
50        dbReader.close();
51        current_record = entries.size() -1;
       }

52     public void insert(String s) {
53        Record rec = new Record(s);
54        entries.add(current_record, rec);
55        current_record++;
       }

56     public void write_db(String filename) {
57        PrintWriter dbWriter = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
58        for (int i = 0; i < entries.size() - 1; i++) {
59           Record rec = (Record) entries.elementAt(i);
60           dbWriter.println(rec.toString());
          }
61        dbWriter.close();
       }
       ……
    }
```

Figure 5.3: Example: a program which simulates a database system.

and finalization. These phases are detected by our method (see the three phases at the top level in Figure 5.4). Since our phase division is employed hierarchically, each of these three phases can be further divided into sub-phases. Let us consider the second phase – the execution of lines 7-35. The sub-phases of this phase will be the different iterations of the loop in lines 7-35. The three sub-phases perform three different operations on the database — read, insert and exit. These sub-phases are also (hierarchically) shown in Figure 5.4.

The preceding example captures the two main features of our approach. First of all, instead of using aggregate flow metrics (such as total number of times a basic block is visited which is captured by Basic Block Vectors), we use the sequence of statements

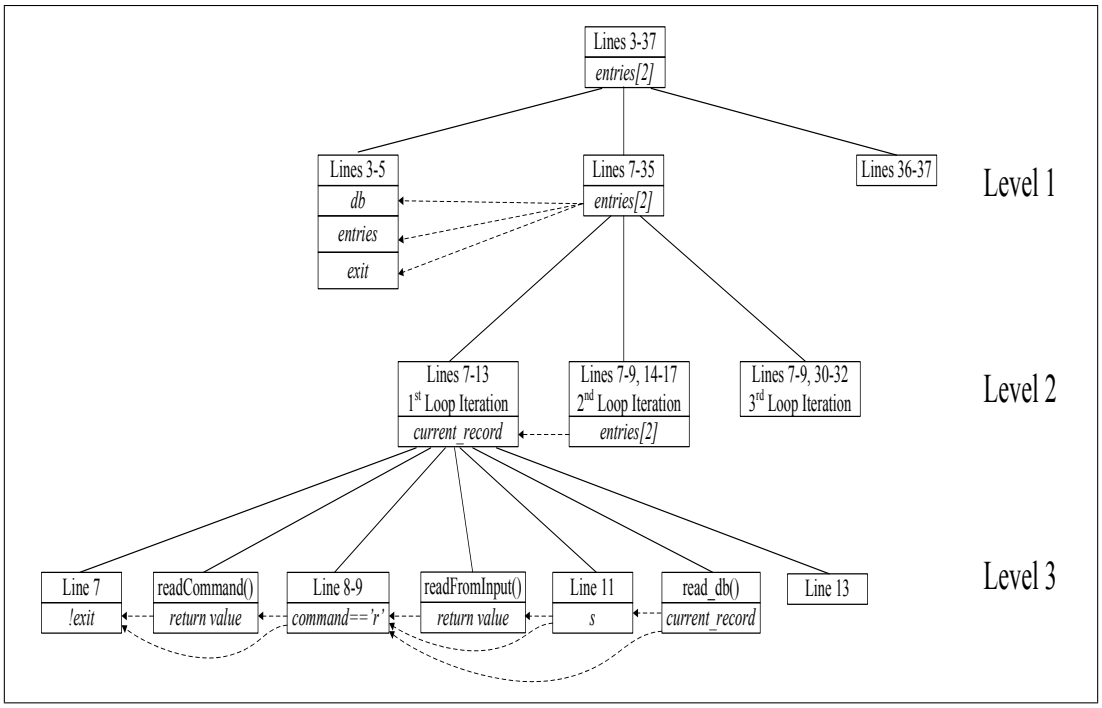


Figure 5.4: Phases for the running example in Figure 5.3. Rectangles represent phases. Dashed arrows represent inter-phase dynamic dependencies.

visited in an execution trace and chop off the phases based on some distinguished statements (such as loop/procedure boundaries). Secondly, the phase division is done hierarchically, that is, the execution trace is divided into some phases at the top-level and each of these phases can be further sub divided and so on.

Our algorithm for dividing an execution trace H into phases appears in Figure 5.5. We can understand the mechanics of the algorithm as follows. Suppose we visualize all loops as calls/return to dummy methods; thus, for each loop execution instance l contained in H suppose we insert a marker "*call to* m_l" ("*return from* m_l") at the beginning (end) of the loop. Here m_l is a dummy method name introduced by us (it does not appear in the program execution trace). Then, we first find the set of all method invocations appearing in H which are *not enclosed by any other method invocation*. We then use the entry and exit of such outermost method invocations to determine the phase boundaries. Clearly, these "outermost" method invocations may

```
1. dividePhase( H: an execution trace)
2. begin
3.   LOOPS = the set of loop entries which are not enclosed by any other loop or method in H;
4.   CALLS = the set of method calls which are not enclosed by any loop or method in H;
5.   if (LOOPS != Ø)
6.       for ( each loop in LOOPS)
7.           mark entry of loop as phase boundary; mark exit of loop (if it exists in H) as phase boundary;
8.   if (LOOPS = = Ø && CALLS != Ø)
9.       for (each method invocation call in CALLS)
10.          mark the entry of call as phase boundary; mark return from call (if it exists in H) as phase boundary;
11.  if (LOOPS = = Ø && CALLS == Ø)
12.      if ( H consists of only iterations of one loop)
13.          for ( iter = every Δ_loop iterations of this loop)
14.              mark the beginning of iter as a phase boundary;
15.      else
16.          for ( stmt = every Δ_stmt statement instances in H)
17.              mark the control location after stmt as a phase boundary;
18.  mark the beginning and end of H as phase boundaries;
19.  for (each marked phase boundary)
20.      ph[i] = the execution trace of H between the i-th and the (i+1)-th phases boundaries;
21.  return ph;
22. end
```

Figure 5.5: Divide an execution H into phases for debugging. Δ_{loop} (Δ_{stmt}) is a certain percentage of the number of loop iterations (statement instances).

correspond to either a call to a procedure in the program or the entry to a loop in the program (recall that we converted the loop entries to dummy method invocations). If H contains outermost procedure calls as well as outermost loops, we give priority to the loops for defining the phase boundaries of H, and use the procedure calls for defining the sub-phases of these phases. We feel that programmers use loops as a higher-level structuring mechanism whereas sub-tasks appearing as initialization or activities within a loop are often written up as procedures.

If no procedure call or loop exists in H, we check whether H contains iterations of a loop and if so we set phase boundaries after a certain number of iterations (given by the constant Δ_{loop}). If H does not even contain any loop iterations (*i.e.*, an acyclic fragment of a procedure body) we set phase boundaries after certain number of statement instances (given by the constant Δ_{stmt}) in Figure 5.5. The reason for

allowing phase division even in the absence of loops/procedures is to eventually focus at the level of statements, if the programmer chooses to do so for debugging.

The `dividePhase` method shown in Figure 5.5 itself is not hierarchical — given an execution trace it simply divides the trace into a finite number of phases. However, we will use it to achieve hierarchical division of a program execution trace H by invoking `dividePhase` on H, the phases of H returned by `dividePhase(`H`)`, and so on.

The phase boundaries generated by our phase detection method on the `jess` and `db` programs are shown in Figure 5.1(c) and Figure 5.2(c) respectively. The reader may wish to compare it with the output of the conventional phase detection methods shown in Figure 5.1(b) and 5.2(b) respectively. In Figure 5.1(c) and Figure 5.2(c), we have divided the execution traces of `jess` and `db` program up to the level of loop iterations (*i.e.*, one iteration of a loop is not fragmented further into smaller phases).

From our experiments, we notice that contrary to conventional phase detection methods, our algorithm can generate very short phases. In other words, our phase detection method is much more closely tied to the program behavior, rather than depending on artificial parameters (such as the minimum length of a phase). For example, the trace corresponding to the 521st interval of the `db` program which corresponds to nine different operations on a database will be divided by our algorithm to nine different phases as opposed to one phase produced by conventional phase detection methods (see Figure 5.2(b),(c)).

Furthermore, consecutive intervals which correspond to different logical operations and have very different execution traces will be identified to be in different phases by our method. Examples of such a situation are the intervals 42,43 in the `jess` program. These intervals are placed in the same phase by the BBV method (Figure 5.1(b)), but in different phases by our method (Figure 5.1(c)). The BBV method places these intervals in the same phase since the relative execution frequencies of basic blocks are similar in the two intervals (though the traces are different).

Finally, consecutive intervals which are part of one single logical operation but have very different execution traces will be identified to be in the same phase by our method (as long as the code executed in these intervals have been modularly placed in a loop or procedure by the programmer). Examples of such a situation can be seen for intervals 23,24 in the `jess` program, corresponding to the processing of a single rule by the expert shell represented by `jess`. They are placed in different phases by the BBV method (Figure 5.1(b)), but in the same phase by our method (Figure 5.1(c)).

5.2 Hierarchical Dynamic Slicing Algorithm

In this section, we present our slicing algorithm. Like dynamic slicing, hierarchical dynamic slicing explores dynamic data/control dependencies related to the observable error (also called the slicing criterion). The only difference lies in the manner in which these dependencies are presented and/or explored by the user. Section 2.2 summarizes a standard dynamic slicing algorithm, and Section 3.2.1 shows the details of such an algorithm.

As discussed earlier, a standard dynamic slice includes the closure of dynamic control and data dependencies from the slicing criterion. However, such a dynamic slice is often too big for human comprehension. Hierarchical dynamic slicing helps a human programmer explore and understand this large dynamic slice. Figure 5.6 shows our hierarchical dynamic slicing algorithm. The algorithm proceeds by employing a recursive procedure `hdslice()`. This procedure is invoked at the top level with the slicing criterion (H, l, v), where H represents an execution trace, l represents a location in H, and v is a program variable. The `hdslice()` procedure first divides the trace H into phases (line 3 of Figure 5.6) by employing `dividePhase`, our phase division algorithm presented in the last section. Inter-phase dependencies (dynamic data and control dependencies across phases) are then detected and collected in the

```
1. hdslice(H: an execution trace, l: a location in H, v: a variable)
2. begin
3.   ph = dividePhase(H); /* See Figure 5.5 for dividePhase algorithm */
4.   for ( each ph[i] of ph )
5.     ph[i].δ = ∅;
6.     ph[i].γ = ∅;
7.     ph[i].ipd = ∅;
8.   stmt_e = the statement instance at location l in trace H;
9.   let ph[e] = the phase which stmt_e belongs to;
10.  ph[e].δ = {v};
11.  ph[e].γ = {stmt_e};
12   stmt= stmt_e;
13.  while( stmt is defined )
14.    inInterPhaseDependence = false;
15.    let ph[s] = the phase which stmt belongs to;
16.    v_def = variable defined at stmt;
17.    V_use = the set of variables used at stmt;
18.    for ( each ph[i] of ph )
19.      if ( v_def ∈ ph[i].δ )
20.        ph[i].δ = ph[i].δ - {v_def};
21.        ph[s].δ = ph[s].δ ∪ V_use;
22.        ph[s].γ = ph[s].γ ∪ {stmt};
23.        if ( i != s ) /* ph[i] and ph[s] are not the same phase */
24.          inInterPhaseDependence = true;
25.      if ( ∃stmt' ∈ ph[i].γ where stmt' is dynamically control dependent on stmt )
26.        CD_s = {stmt' | stmt' ∈ ph[i].γ and stmt' is dynamically control dependent on stmt};
27.        ph[i].γ = ph[i].γ - CD_s;
28.        ph[s].δ = ph[s].δ ∪ V_use;
29.        ph[s].γ = ph[s].γ ∪ {stmt};
30.        if ( i != s ) /* ph[i] and ph[s] are not the same phase */
31.          inInterPhaseDependence = true;
32.    if ( inInterPhaseDependence )
33.      ph[s].ipd = ph[s].ipd ∪ {⟨stmt, v_def⟩};
34.    stmt = the statement instance before stmt;
35.  for ( each ph[i] of ph )
36.    report inter-phase dependencies ph[i].ipd to the programmer;
37.  ⟨stmt_err, v_err⟩= ProgrammerIntervention();
38.  l_err = the location of stmt_err in H;
39.  err_ph = the phase which stmt_err belongs to, i.e. the suspicious phase;
40.  H_err = the execution trace for the suspicious phase err_ph;
41.  hdslice(H_err, l_err, v_err);
42. end
```

Figure 5.6: The Hierarchical Dynamic Slicing algorithm.

`ipd` set for each phase. Finally, these dependencies are reported to a programmer. The programmer needs to identify "suspicious" ones and return the first (in order of occurrence) suspicious inter-phase dependency (at line 37). Note that the programmer

here is inspecting only dependencies *across phases*, not dependencies within a phase.

Each invocation of the `hdslice()` procedure detects inter-phase dependencies which are related to the observable error. This involves the following two steps.

1. determine which dynamic dependencies are (directly or indirectly) related to the observable error, and then

2. determine which of the dynamic dependencies identified in step 1 are inter-phase dependencies.

The first step is drawn from dynamic slicing, while the second step is novel to hierarchical dynamic slicing. Dynamic slicing algorithms (see Section 2.2 and 3.2.1) maintain sets δ (γ) to capture the variables (statements) whose data (control) dependencies are yet to be explained. In hierarchical dynamic slicing, we maintain several δ and γ sets, one for each phase. The splitting of δ and γ sets is to ease the task of determining which dynamic dependencies are inter-phase dependencies. For every statement instance *stmt* encountered during the backward traversal for slicing, let $ph[s]$ be the phase which *stmt* belongs to. If *stmt* defines an variable v_{def} which is included in $ph[i].\delta$ for some value of i, this means that v_{def} is used by statement instance in the ith phase $ph[i]$, and *stmt* is involved in a dynamic data dependence which is related to the observable error. We can then easily determine whether this data dependence spans phase boundaries by determining whether $ph[s]$ and $ph[i]$ are the same phase. Similarly, we could use the γ sets of the individual phases to determine whether a dynamic control dependence spans phase boundaries. The data and control dependencies which are thus identified to be inter-phase dependencies are captured in the `ipd` sets (Line 33 of Figure 5.6).

The programmer can use the values of the variables involved in inter-phase dynamic dependencies[2] to identify the "first" (in terms of order of occurrence in the

[2]The programmer may need to re-execute the program to obtain these values if the values are

trace) suspicious inter-phase dependency. We would now like to employ slicing to explain this suspicious dependency. However, slicing requires the slicing criterion to be set as a triple $\langle trace, location\ of\ a\ statement\ instance, variable \rangle$. Thus, we need to extract these parameters from an inter-phase dependency if it is deemed "suspicious" by the programmer. For an inter-phase dependency from phase p to phase p', we set the execution trace for phase p as the trace to be explored for further slicing. Thus, the phase p is marked as the error phase err_ph and its trace is the execution trace H_{err} to be further explored (see Lines 37-40 of Figure 5.6). Also, given any suspicious inter-phase dependency, we can associate a variable v_{err} with it. For data dependencies, v_{err} is the variable which is defined/used; for control dependencies, we can consider an auxiliary boolean variable corresponding to the guard involved in the control dependency. Finally, the location of the statement instance where v_{err} is defined in the error phase err_ph is marked as the suspected erroneous location l_{err}. We now recursively invoke the hierarchical dynamic slicing procedure `hdslice` (see Line 41 of Figure 5.6) with the new slicing criterion $(H_{err}, l_{err}, v_{err})$.

Example We use the program of Figure 5.3 as the example. This program simulates a database system where the variable `current_record` should always point to the last database record operated on. In this program, we introduced a bug in Line 51, which should be

```
current_record = entries.size()
```

instead of

```
current_record = entries.size() - 1
```

Let us consider the following execution trace — the program first reads one record into `entries` by executing lines 3-5, reads two additional records into `entries` by

not captured in the execution trace.

executing lines 7-13, and inserts one record into the database by executing line 7-9,14-17. Finally, the program exits the database by executing 7-9, 30-32, and writes the resultant database into a file by executing 36-37. Let us suppose the records read/inserted were "*Africa*", "*America*", "*Antarctica*" and "*Asia*" (in this order). Then, because of the faulty statement in line 51 of the program, the content of the database at the end of execution will be: "*Africa*", "*America*", "*Asia*", "*Antarctica*". In other words, the last and second last elements of the program array `entries` (*i.e.*, `entries[2]` and `entries[3]`) are reversed. This error can be observed from the file to which the database is written.

Figure 5.4 partially illustrates how the hierarchical dynamic slicing algorithm works to locate the faulty statement. Rectangles at the same horizontal level in Figure 5.4 are the phases generated in the same invocation of `hdslice()` procedure (our slicing algorithm). Dashed arrows in Figure 5.4 represent inter-phase dependencies. Variables involved in inter-phase dependencies for each phase appear in italics in Figure 5.4. We do not show the statement instances which define these variables, since they are clear from the program in Figure 5.3. As discussed earlier, the variable for an inter-phase control dependency may be captured by an auxiliary boolean variable representing the guard corresponding to the control dependency (see the guard *command* == *'r'* for the phase representing lines 8-9 in Figure 5.4). Note that the variable(s) mentioned for each phase in Figure 5.4 effectively serve as the "**outputs**" of the phase which are passed to the succeeding phases as "**inputs**". This input-output relationship constitutes the inter-phase dependencies which are shown using dashed arrows in Figure 5.4. Thus, given an invocation of `hdslice` on an execution trace H, the programmer can inspect these "outputs" of the phases corresponding to given "inputs" and check whether this matches his/her expectation of the input-output relationship supposed to be captured by the corresponding phase. *The programmer can avoid thinking about the computation inside any phase.*

For the example program given in Figure 5.3, the `hdslice()` procedure is first invoked with the execution of lines 3-37 of Figure 5.3, and the "incorrect" variable `entries[2]` (deemed incorrect since it is involved in the observable error in this example). This execution is divided into three phases, as shown in Figure 5.4. The second phase (execution of lines 7-35) defines the variable `entries[2]`. Although the first phase (execution of lines 3-5) defines several variables (including `entries`/`exit`) which are involved in inter-phase dependencies, the programmer in this case deems the initialization code in the first phase as "correct". Typically, the programmer will do this by inspecting the "outputs", that is the values of variables produced by execution of first phase. In this case, the programmer observes that at the end of the first phase `entries[2]` is not initialized (in fact, only `entries[0]` is initialized). So, the first phase is clearly unrelated to the error in `entries[2]`, and the programmer zooms into the second phase for further investigation. This results in a recursive invocation of the `hdslice` procedure on the second phase. The second phase is then further divided into three sub-phases. Again, the programmer observes from the inter-phase dependencies that the first sub-phase produces `current_record` as output which is fed as input to the second sub-phase (shown via dashed arrows in Figure 5.4). Furthermore, the value of the `current_record` variable is "unexpected"; this is based on the programmer's expectation that `current_record` should be an index to the current last record of the database. Consequently the programmer focuses on the value of `current_record` in the first sub-phase (lines 7-13 of Figure 5.3) via another invocation of `hdslice`.

5.3 Experimental evaluation

We have implemented hierarchical dynamic slicing on top of our JSlice dynamic slicing tool (see Chapter 3). The tool performs backwards dynamic slicing of sequential Java programs. Since backwards slicing requires storing of the execution trace, the tool

performs online compression during trace collection. The compressed trace representation is traversed without decompression during slicing. Our prototype implementation of hierarchical dynamic slicing also uses this compressed trace representation. In particular, the phase detection/representation/traversal in the execution trace are all done in compression domain.

Subject	Description	Size
NanoXML	a XML parser for Java	7646 LOC 24 classes
JTopas	a Java library for parsing text	5400 LOC, 50 classes
Apache JMeter	a performance testing tool	43400 LOC, 389 classes

Table 5.1: Descriptions of subject programs used to evaluate the effectiveness of our hierarchical dynamic slicing approach for debugging.

We applied our prototype implementation to subject programs written in Java available from the Software-artifact Infrastructure Repository (SIR) [29]. Since our slicing technique is applicable to sequential programs, we chose the *NanoXML*, *JTopas* and *JMeter* subjects. Note that *JMeter* is actually a multi-threaded Java program, but some test cases from [29] run only one thread of *JMeter* thereby making our slicing technique applicable. Descriptions and sizes of these subjects are shown in Table 5.1.

Each SIR subject comes with a pool of test inputs. SIR [29] also provides several buggy versions of each subject program, where each buggy version has exactly one injected bug. Some of the buggy versions are such that none of the given test inputs (for the corresponding subject program) exposes the bug. We did not include them in our experiments, since the failing input (*i.e.*, the input corresponding to the execution trace to slice on) did not come with the subject programs. Furthermore, some other buggy versions are such that the faulty statement is not included in the dynamic

slice[3]; we left out these buggy versions as well. Finally, we got three buggy versions for each of our three subject programs — resulting in a total of nine buggy programs.

Statement Instances Examined We first tried to evaluate the utility of hierarchical dynamic slicing for program debugging. Figure 5.7 compares (in a *log scale*) the number of statement instances which a programmer has to examine using the hierarchical dynamic slicing approach and the conventional dynamic slicing approach. For the hierarchical approach, a programmer has to examine only statement instances involved in inter-phase dependencies. We compare this against the size of the dynamic slice. In practice, a developer may prune some statements from the dynamic slice according to his understanding of the program. However, it is very difficult to quantitatively measure such human factors. In our experiments, we use the size of the dynamic slice to estimate what the programmer might examine in the conventional dynamic slicing approach. As we can see from Figure 5.7, our approach can significantly reduce (often by *orders of magnitude*) the number of statement instances which the programmer needs to examine for debugging. The improvement comes from the usage of phases in slicing. By dividing an execution into phases and reporting inter-phase dependencies, a programmer can quickly identify suspicious dependence chains. The inter-phase dependencies effectively expose the "inputs" and "outputs" of each phase. This allows the programmer to think of each phase in terms of the expected input-output relationship rather than worrying about the computations within each phase. Consequently, the number of statement instances to be investigated is significantly reduced.

[3]This is because dynamic slicing can only locate errors where the faulty statement is present in the program as well as in the execution trace (*e.g.*, it cannot capture "missing code" errors).

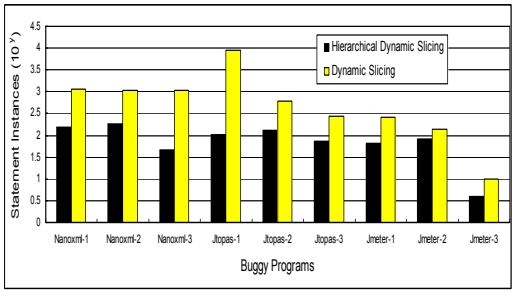


Figure 5.7: The number of statement instances that a programmer has to examine using the hierarchical dynamic slicing approach and the conventional dynamic slicing approach. *The figure is in log scale showing that our hierarchical approach is often orders of magnitude better.*

Subjects	# Interventions	# Hierarchy Levels
Nanoxml-1	17	29
Nanoxml-2	22	26
Nanoxml-3	2	7
Jtopas-1	4	5
Jtopas-2	10	10
Jtopas-3	7	8
Jmeter-1	3	3
Jmeter-2	8	10
Jmeter-3	2	2

Table 5.2: Number of Programmer Interventions & Hierarchy Levels in Hierarchical Dynamic Slicing.

User Interaction One of the key issues in hierarchical dynamic slicing is the interleaving of slice computation and comprehension steps. The aim is to aid program understanding by *gradually* exposing the programmer to complicated dependence chains. However, if the number of intervention steps required from the programmer is overwhelming, this can undermine the method's utility. For this reason, we experimentally evaluated the number of manual interventions required in the hierarchical dynamic slicing of our subject programs. The results appear in the column *# Interventions* of Table 5.2. In the experiments, we chose "simple" test cases which result in shorter length execution traces. We feel that this is natural, since programmers also favor a shorter execution trace demonstrating an error (over longer execution traces showing

the same error) for debugging purposes. In practice, the programmer can generate such "simple" test cases (which produce shorter execution traces) based on his/her intuition about the program, or (s)he can use automatic methods for simplifying test inputs [75, 111].

From our slicing algorithm (Figure 5.6) it seems that the number of programmer interventions is exactly equal to the number of hierarchy levels we explore (*i.e.*, the number of times we invoke the phase division algorithm). We were pleasantly surprised to find that *the number of manual interventions is often less than the number of hierarchies explored* (see the last two columns in Table 5.2). After dividing an execution trace into phases, we may find that dependence chains which are relevant to the observable error all lie in one phase, and dependence chains in other phases have no effect on the observable error. In other words, there is no inter-phase dependence which is relevant to the observable error. Then, our approach could proceed to the phase which is relevant to the observable error, without any user intervention.

Post-mortem pruning of slices We also tried out the following variation of our experiments on slicing. We first compute the entire dynamic slice, as in conventional slicing techniques. However, the slice is explored *post-mortem* along dependence chains (chains of length 1,2,...), starting from the slicing criterion until the error is found. Note that such an exploration is also not automatic since the programmer has to look through the dependence chains to check whether the error is found.

We compared the number of statement instances examined by such post-mortem exploration of the slice with our hierarchical dynamic slicing method (which performs exploration/comprehension as the slice is being computed). We found that the number of statement instances examined by this post-mortem guided exploration of the slice is still *substantially* higher than those examined by our hierarchical dynamic slicing method for most of the buggy programs. To be precise, hierarchical dynamic

slicing required substantially less statements to examine (as compared to the pruned slices) in 6 out of the 9 buggy programs. This is presumably because exploiting user-guidance *during* the slice computation (rather then *after* the slice computation) makes the exploration/comprehension of the slice more goal-directed.

5.4 Summary

This chapter presents hierarchical dynamic slicing to aid the comprehension of dynamic slices. The proposed application is in program debugging, where the programmer is gradually guided through complex program dependence chains. This is as opposed to the arduous task of understanding a full dynamic slice, where all of the comprehension is left to the programmer.

We have conducted detailed experiments on well-known subject programs written in Java drawn from the SIR repository [29] to evaluate the effectiveness of this approach. Our experiments show a substantial reduction in program understanding effort for our subject programs.

CHAPTER 6

TEST BASED FAULT LOCALIZATION

In the previous chapters, we address deficiencies of dynamic slicing and make it applicable for realistic programs. Because dynamic slicing requires the entire control flow and data flow of an execution, this technique can only be used to analyze the execution of up to a few seconds given the speed and storage capacity of modern workstations. This observation is confirmed in other research reports [27, 114, 115]. Zhang et al. proposed to reduce the execution run so that dynamic slicing can be applied to long execution runs [119], but their technique can be only used for applications which have many independent inputs.

Because it is believed that dynamic slicing is an expensive technique, we also study *test based fault localization* techniques. These techniques can be light, since they do not rely on the entire run time information of an execution. Test based fault localization techniques often proceed by comparing the failing program run with some "successful" run (a run which does not demonstrate the error). The difference may be related to the error, and is reported to the programmer as heuristics for debugging. Sometimes, the difference can pinpoint the error.

An issue of test based fault localization techniques is to generate or choose a "suitable" successful run; this task is often left to the programmer. In this chapter, we present a control flow based difference metric to (a) automatically generate a feasible successful run, or (b) automatically choose a successful run from a pool of successful runs for automated debugging.

Our difference metric summarizes the sequence of comparable branch statement instances which are evaluated differently in the two runs. Based on this difference

metric, we automatically generate a feasible successful run which is close to, that is, has little distance with, the failing run. In addition, when there is a pool S of successful runs available, we can *directly* use the difference metric to choose an appropriate successful run from the pool S.

The rest of this chapter is organized as follows. Section 6.1 presents an example. Our difference metric is presented in Section 6.2. Section 6.3 discuses how to choose or construct a successful run comparison. Section 6.4 presents the experimental setup. Section 6.5 discusses our experimental results. Section 6.6 concludes the chapter.

6.1 An Example

In this section, we use a sample segment of the TCAS program from the Siemens benchmark suite [47, 89], to introduce several important concepts which will be used throughout this chapter. The TCAS program is an altitude controller program. Figure 6.1 shows a program segment, with input variables *Climb* and *Up*. There is an bug in the program segment, where lines 2 and 4 are wrongly reversed. That is, line 2 should be `separation = Up+100` and line 4 should be `separation = Up`. Next, we illustrate the concepts of *failing run*, *successful run*, *bug report*, *feasible path* and *infeasible path* with some examples.

```
1.  if (Climb)
2.    separation = Up;
3.  else
4.    separation = Up + 100;
5.  if (separation > 150)
6.    upward= 1;
7.  else
8.    upward= 0;
9.  if (upward > 0)
10.   .....
11.   printf("upward");
12. else
13.   .....
14.   printf("downward");
```

Figure 6.1: A program segment from the TCAS program.

Failing run: When the program is executed with inputs `Climb = 1` and `Up = 100`, the output will be "downward". This execution run will be regarded as *failing run* π_f, because the developer would expect that the output is "upward".

Successful run: When the program is executed with inputs `Climb = 0` and `Up = 20`, the output will be "downward". Although this execution run is not perfect, *i.e.* some buggy statements are executed, we still consider this run as a *successful run* π_s, because the output matches the developer's expectation. A developer usually manually determines whether an execution run is a failing run or successful run.

Bug report: If we compare the above failing run π_f (along path $\langle 1, 2, 5, 8, 9, 13, 14\rangle$) and successful run π_s (along path $\langle 1, 4, 5, 8, 9, 13, 14\rangle$), we will find that branch 1 has similar context, but is evaluated differently in the two execution runs. According to our method in Section 6.2, we will generate a bug report which contains only line 1.

The bug report may not always pinpoint the buggy statements, as illustrated by this example. However, given a bug report with statements *stmt*, the developer can inspect those statements *stmt*$'$ which have a chain of control/data dependencies from/to *stmt*, until the error cause is found. A bug report is considered as high quality, if the total number of inspected statements (*i.e. stmt* and *stmt*$'$) is small.

Feasible path: When we construct a run π along the path $\langle 1, 2, 5, 8, 9, 13, 14\rangle$, we consider π as a *feasible run*, because there exist inputs `Climb = 1` and `Up = 100` which lead to this execution path.

Infeasible path: When we construct a run π along the path $\langle 1, 2, 5, 8, 9, 10, 11\rangle$, we consider π as an *infeasible run*. This is because lines $8, 9, 10$ conflict with each other, and there is no input which leads to such an execution path.

6.2 Measuring Difference between Execution Runs

We elaborate on the difference metric used for comparing execution runs in this section. We consider each execution run of a program to be a sequence of *events* $\langle e_0, e_1, ..., e_{n-1} \rangle$ where e_i refers to the ith event during execution. *Each event e_i represents an execution instance of a line number in the program*; the program statement corresponding to this line number is denoted as $stmt(e_i)$. To distinguish events from different execution runs, we denote the ith event in an execution run π as e_i^π, that is, the execution run appears as a superscript. We will drop the superscript when it is obvious from the context.

Our difference metric measures the difference between two execution runs π and π' of a program, by comparing behaviors of "corresponding" branch statement instances from π and π'. The branch statement instances with differing outcomes in π, π' are captured in $diff(\pi, \pi')$ – the difference between execution run π and execution run π'. In order to find out "corresponding" branch instances, we have defined a notion of *alignment* to relate statement instances of two execution runs. Our alignment is based on *dynamic control dependence*. Given an execution run π of a program, an event e_i^π is *dynamically control dependent* on another event e_j^π if e_j^π is the last event before e_i^π in π where $stmt(e_i^\pi)$ is statically control dependent [31] on $stmt(e_j^\pi)$. Note that any method entry event is dynamically control dependent on the corresponding method invocation event. We use the notation $dep(e_i^\pi, \pi)$ to denote the event on which e_i^π is dynamically control dependent in run π. We now present our definition of event alignment.

Definition 6.1 (Alignment)**.** *For any pair of events: e in run π and event e' in run π', we define $align(e, e') = true$ (e and e' are aligned) iff.*

1. *$stmt(e) = stmt(e')$, and*

2. *either e, e' are the first events appearing in π, π' or*

$align(dep(e, \pi), dep(e', \pi')) = true.$

When a branch event e_i^π cannot be aligned with any event from the execution π', this only affects alignments of events in π which are transitively dynamically control dependent on e_i^π. According to the alignment, the ith iteration of a loop in the execution π will be aligned with the ith iteration of the same loop in the execution π', in order to properly compare events from different loop iterations.

```
1.  if (a)
2.     i = i + 1;
3.  if (b)
4.     j = j + 1;
5.  if (c)
6.     if (d)
7.        k = k + 1;
8.     else
9.        k = k + 2;
10. printf("%d", k);
```

Figure 6.2: A program segment.

Figure 6.2 shows an example program fragment, and Figure 6.3 illustrates our definition of alignment. Here the first three columns show the event sequences of three execution runs π, π' and π'' of the program fragment; the 4th and 5th columns show alignments of (π, π') and (π, π''), where solid lines indicate aligned statement instances and dashed lines indicate unaligned statement instances. In other words, events along the same horizontal line are aligned.

According to the notion of alignment presented in Definition 6.1, for any event e in π, there exists *at most* one event e' in π' such that $align(e, e') = true$. The difference between π and π' (denoted as $diff(\pi, \pi')$) captures all branch event occurrences e in π which (i) e can be aligned to an event e' in π' and (ii) events e and e' have different outcomes in π and π'. Formally, the difference between two execution runs can be defined as follows.

Definition 6.2 (Difference Metric)**.** *Consider two execution runs π, π' of a program.*

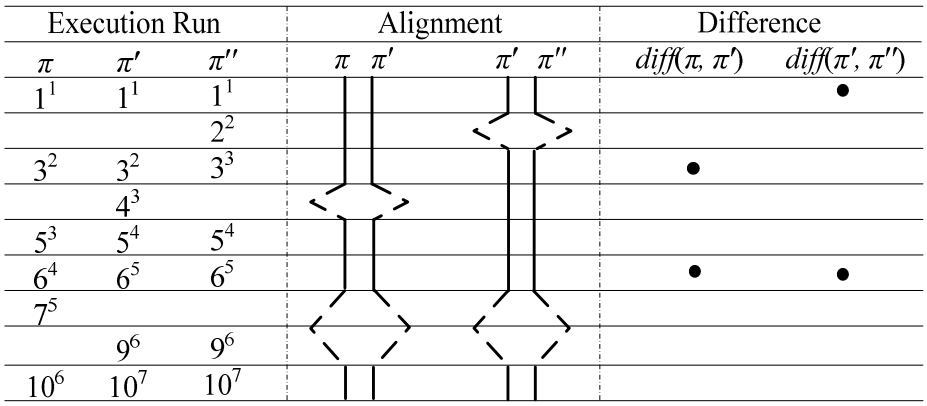


Figure 6.3: Example to illustrate alignments and difference metrics.

The difference between π, π', denoted $diff(\pi, \pi')$, is defined as:

$$diff(\pi, \pi') = \langle e^{\pi}_{i_1}, \ldots, e^{\pi}_{i_k} \rangle$$

such that

1. *each event e in $diff(\pi, \pi')$ is a branch event occurrence drawn from run π.*
2. *the events in $diff(\pi, \pi')$ appear in the same order as in π, that is, for all $1 \leq j < k$, $i_j < i_{j+1}$ (event $e^{\pi}_{i_j}$ appears before event $e^{\pi}_{i_{j+1}}$ in π).*
3. *for each e in $diff(\pi, \pi')$, there exists another branch occurrence e' in run π' such that $align(e, e')$=true (*i.e. *e and e' can be aligned). Furthermore, the outcome of e in π is different from the outcome of e' in π'* [1].
4. *all events in π satisfying criteria (1) and (2) are included in $diff(\pi, \pi')$.*

As a special case, if execution runs π and π' have the same control flow, then we define $diff(\pi, \pi') = \langle e^{\pi}_0 \rangle$.

Clearly we can see that in general $diff(\pi, \pi') \neq diff(\pi', \pi)$. The reason for making a special case for π and π' having the same control flow will be explained later in the section when we discuss comparison of differences.

[1] *Since e, e' can be aligned, they denote occurrences of the same branch statement.*

Consider the example in Figure 6.3. The difference between execution runs π and π' is: $\mathit{diff}(\pi, \pi') = \langle 3^2, 6^4 \rangle$, as indicated by the last two columns in Figure 6.3. This is because branch instances $3^2, 6^4$ are aligned in runs π and π' and their outcomes are different in π, π'. If the branches at lines $3^2, 6^4$ are evaluated differently, we get π' from π. Similarly, the difference between execution runs π and π'' is: $\mathit{diff}(\pi, \pi'') = \langle 1^1, 6^4 \rangle$.

Why do we capture branch event occurrences of π which evaluate differently in π' in the difference? Recall that we want to choose a successful run for purposes of fault localization. If π is the failing run and π' is a successful run, then $\mathit{diff}(\pi, \pi')$ tells us which branches in the failing run π need to be evaluated differently to produce the successful run π'. Clearly, if we have a choice of successful runs we would like to make minimal changes to the failing run to produce a successful run. Thus, given a failing run π and two successful runs π', π'', we choose π' over π'' if $\mathit{diff}(\pi, \pi') < \mathit{diff}(\pi, \pi'')$. This requires us to *compare* differences, as elaborated in the following.

Definition 6.3 (Comparison of Differences)**.** *Let π, π', π'' be three execution runs of a program. Let*

$$\mathit{diff}(\pi, \pi') = \langle e^{\pi}_{i_1}, e^{\pi}_{i_2}, \ldots, e^{\pi}_{i_n} \rangle \quad \text{and} \quad \mathit{diff}(\pi, \pi'') = \langle e^{\pi}_{j_1}, e^{\pi}_{j_2}, \ldots, e^{\pi}_{j_m} \rangle$$

We define $\mathit{diff}(\pi, \pi') < \mathit{diff}(\pi, \pi'')$ iff there exists an integer $K \geq 0$ s.t.

1. *$K \leq m$ and $K \leq n$*
2. *the last K events in $\mathit{diff}(\pi, \pi')$ and $\mathit{diff}(\pi, \pi'')$ are the same, that is, $\forall 0 \leq x < K \;\; i_{n-x} = j_{m-x}$.*
3. *one of the following two conditions holds*
   - *either $\mathit{diff}(\pi, \pi')$ is a suffix of $\mathit{diff}(\pi, \pi'')$, that is, $K = n < m$*
   - *or the $(K+1)$th event from the end in $\mathit{diff}(\pi, \pi')$ appears later in π as compared to the $(K+1)$th event from the end in $\mathit{diff}(\pi, \pi'')$, that is, $i_{n-K} > j_{m-K}$.*

Thus, given a failing run π and two successful runs π', π'' we say that $\mathit{diff}(\pi, \pi') < \mathit{diff}(\pi, \pi'')$ based on a combination of the following criteria.

- Fewer branches of π need to be evaluated differently to get π' as compared to the number of branches of π that need to be evaluated differently to get π''. This is reflected in the condition $K = n < m$ of Definition 6.3.

- The branches of π that need to be evaluated differently to get π' appear closer to the end of π (where the error is observed), as compared to the branches of π that need to be evaluated differently to get π'' . This is reflected in the condition $i_{n-K} > j_{m-K}$ of Definition 6.3.

To illustrate our comparison of differences, consider the example in Figure 6.3. Recall that $\mathit{diff}(\pi, \pi') = \langle 3^2, 6^4 \rangle$, and $\mathit{diff}(\pi, \pi'') = \langle 1^1, 6^4 \rangle$, as illustrated by the "•" in the last two columns of Figure 6.3. Comparing $\langle 3^2, 6^4 \rangle$ with $\langle 1^1, 6^4 \rangle$, we see that $\langle 3^2, 6^4 \rangle < \langle 1^1, 6^4 \rangle$ since statement instance 3^2 occurs after statement instance 1^1 in execution run π.

According to the comparison of differences in Definition 6.3, we favor late appearance of differing branch instances instead of early ones. This is because the early branch instances (where the two runs are different) are often not related to the error. For example, many programs check whether the input is legal in the beginning. If we favor early branch instances, we may get failing and successful runs which only differ in whether the input is legal for such programs. Comparing such runs is unlikely to produce a useful bug report.

Comparing runs with identical control flow Using Definitions 6.2 and 6.3 we can see that if π is the failing run, π_1 is a successful run with same control flow as that of π (*i.e.* same sequence of statements executed by a different input) and π_2 is a successful run with control flow different from π we will have $\mathit{diff}(\pi, \pi_2) < \mathit{diff}(\pi, \pi_1)$.

As a result, our method for choosing a successful run will avoid successful runs with same control flow as that of the failing run. This choice is deliberate; we want to find a successful run with minimal difference in control flow from the failing run, but not with zero difference. Recall here that we construct bug report by comparing the control-flow of the selected successful run with the failing run. If the two runs have the same control flow, the bug report is null and hence useless to the programmer. In our experiments, we encountered few cases where there were some successful runs with same control flow as the failing run; these were not chosen by our method of comparing differences between runs.

6.3 Obtain the Successful Run

In this section, we discuss how to generate or choose a "suitable" successful run for fault localization. The process is guided by the difference metric presented in Section 6.2. The selected successful run π_s is then compared against the given failing run π_f. The difference between π_f and π_s (see Definition 6.2) is constructed as a bug report.

Automatically Choosing Method When there are one failing run π_f and a pool S of successful runs available, we can simply choose a "suitable" successful run from S for the comparison based fault localization. The selection process is of course guided by our difference metric. That is, for each successful run $\pi_s \in S$, we compute the difference between π_f and π_s, *i.e.*, $\mathit{diff}(\pi_f, \pi_s)$ according to the Definition 6.2. Then the successful run with the "smallest" difference (as per Definition 6.3) is selected for debugging. Such a successful run has the minimal difference from the failing run, and may produce a bug report with high quality.

One question here is how we get the pool S of successful runs. There are two solutions to this problem. One possibility is to have a pre-defined large set of program inputs Inp; this set of test-cases might have been generated using some notion of

coverage. Now given the program, we find out which of the inputs in Inp produces successful runs. Another way of constructing the successful run pool is to use the input for the given failing run. We can slightly perturb this failing input to generate a set of program inputs; we then classify which of these perturbed inputs produce successful runs - thereby getting a pool of successful runs.

Automatic Path Generation Method When there is only the failing run π_f available, we can try to generate the closest successful run from the failing run π_f, according to the difference metric in Definition 6.2. In other words, given a failing run π_f, we seek to generate a successful run π_s such that there does not exist any other successful run π'_s with $\mathit{diff}(\pi_f, \pi_s) < \mathit{diff}(\pi_f, \pi'_s)$ (see Definition 6.3). The difference between π_f and π_s consists of the branches in the failing run π_f which need to be evaluated differently to get π_s. We now elaborate our algorithm for generating a feasible successful run from a failing run of a program.

6.3.1 Path Generation Algorithm

How do we construct the closest successful run from the failing run? Of course, we will not generate all the possible runs and then find out the closest. Instead, we seek to generate the "closest" program run from the failing run by exploiting our understanding of the difference metric. If this turns out to be a feasible successful run then our search stops; otherwise we try for the next closest run and so on.

Our notion of proximity (Definition 6.3) ensures that a run π is close to a failing run π_f (that is, difference between π_f, π is small) if the branches of π_f which need to be evaluated differently are near the end of π_f (where the error is observed). Furthermore, the number of branches of π_f that need to be evaluated differently should be small.

Thus, given a failing run π_f, we will first try to evaluate differently the last branch occurrence (call it b^{last}) in π_f to construct a run π_1. Among all the branch occurrences

Difference with failing run	*Execution run*
$\langle 6 \rangle$	$\langle 1, 3, 5, 6, 9, 10 \rangle$
$\langle 3, 6 \rangle$	$\langle 1, 3, 4, 5, 6, 9, 10 \rangle$
$\langle 1, 3, 6 \rangle$	$\langle 1, 2, 3, 4, 5, 6, 9, 10 \rangle$
$\langle 1, 6 \rangle$	$\langle 1, 2, 3, 5, 6, 9, 10 \rangle$
$\langle 5 \rangle$	$\langle 1, 3, 5, 10 \rangle$
$\langle 3, 5 \rangle$	$\langle 1, 3, 4, 5, 10 \rangle$
$\langle 1, 3, 5 \rangle$	$\langle 1, 2, 3, 4, 5, 10 \rangle$
$\langle 1, 5 \rangle$	$\langle 1, 2, 3, 5, 10 \rangle$
$\langle 3 \rangle$	$\langle 1, 3, 4, 5, 6, 7, 10 \rangle$
$\langle 1, 3 \rangle$	$\langle 1, 2, 3, 4, 5, 6, 7, 10 \rangle$
$\langle 1 \rangle$	$\langle 1, 2, 3, 5, 6, 7, 10 \rangle$

Table 6.1: Order in which candidate execution runs are tried out for the failing run $\langle 1, 3, 5, 6, 7, 10 \rangle$ in Figure 6.2.

in π_f, clearly b^{last} is nearest to the end of π_f. If π_1 is a successful and feasible run, we return π_1 as the closest successful run. Otherwise we successively construct other runs by evaluating b^{last} as well as other branch occurrences of π_f differently. If none of these runs is a feasible successful run, this indicates that the branch at b^{last} might have little relationship with the error cause. So, there is no point in evaluating b^{last} differently. Instead, we evaluate the second last branch occurrence in π_f differently and carry out the above steps again. This process goes on until a feasible successful run is obtained.

Example Let us take the program segment in Figure 6.2 as an example. Assume that the failing run $\pi_f = \langle 1, 3, 5, 6, 7, 10 \rangle$. The branch occurrences appearing in this run are at lines $1, 3, 5, 6$. Note that the execution run π_f does not contain multiple occurrences of any program statement; so we do not need to worry about distinguishing between occurrences of the same statement in a path as far as this example is concerned. Now, our method tries to evaluate some of the branches in lines $1, 3, 5, 6$ differently from the failing run π_f, thereby constructing new execution runs.

Table 6.1 shows the order in which the branches of failing run π_f will be evaluated differently leading to new execution runs. Let us assume that none of the new execution runs is a feasible successful run, so that we can elaborate all possible runs constructed by our algorithm. We first evaluate differently the branch at line 6, since this branch is the last one in the failing run. In the next step, the algorithm intends to evaluate differently a branch before line 6 as well as the branch at line 6. According to the difference metric, we should now choose the branch which is the closest to line 6. However, the algorithm cannot choose line 5 at this time, although line 5 is the closest. This is because line 6 is control dependent on line 5. If line 5 is evaluated differently, line 6 cannot be executed. Instead, line 3 is chosen, and the second run is constructed by evaluating differently branches at line 3 and 6. After this, the algorithm tries to evaluate differently a branch before line 3 as well as branches at line 3,6. Thus, line 1 is selected, and branches at line 1,3,6 are evaluated differently. Now all branches before line 3 and 6 have been considered, and no feasible successful run can be constructed. This means that line 3 and 6 might not be related to the error cause at the same time. The algorithm continues trying to evaluate differently branches before line 6 as well as the branch at line 6. After branches at lines 1, 6 have been evaluated differently, all branches before line 6 have been evaluated differently together with the branch at line 6. Corresponding runs have been shown in the first segment of the Table 6.1 (the segments are separated by horizontal lines). Thus, at this point the algorithm concludes that the branch at line 6 might have little bearing with the actual error cause. The algorithm gives up line 6, and evaluates differently the second last branch at line 5 as well as branches before line 5, as shown in the second segment of Table 6.1. After this, our algorithm considers the third last branch at line 3, and so on.

```
Global Variable:  sop, the program's initial event
                  eop, the program's event after which
                       the erroneous state is observable
                  π_f, the program's failing run to debug

generatePaths (paths: a set of execution runs, last: a branch event,
                  df:difference between π_f and runs in paths)
begin
1. br= branch event just prior to last in π_f;
2. while (br is defined)
3.   if ( no event in df is dynamically control dependent on br)
4.     newpaths= {}; /* empty set */
5.     for each π in paths do
6.       de = pde(br, π);
7.       subpaths = get_all(br, de, π);
8.       π_1 = sub-path of π from sop to br;
9.       π_2 = sub-path of π from de to eop;
10.      for each π' in subpaths do
11.        if (π' o π_2 is infeasible)
12.          continue;
13.        π_w = π_1 o π' o π_2;
14.        if (π_w is feasible and successful)
15.          return π_w;
16.        else
17.          insert π_w into newpaths;
18.    if (newpaths is not empty set)
19.      df'= ⟨br⟩ ∘ df;
20.      π_r = generatePaths(newpaths, br, df');
21.      if (π_r != Null)
22.        return π_r;
23.  else
24.    for each π in paths do
25.      π_3 = sub-path of π from br to eop;
26.      if (π_3 is infeasible)
27.        remove π from paths;
28.    if (paths is empty set)
29.      return Null;
30.  br= branch event just prior to br in π_f;
31return Null;
end
```

Figure 6.4: Algorithm to generate a successful run from the failing run.

Incremental Path generation So far, we have clarified the order in which the execution runs will be generated in our search for the closest successful run. In Table 6.1 we have shown the order of the generated execution runs for a given failing run and the differences of these runs from the failing run. However, our algorithm will *not* generate the differences and then find out the execution run(s) for each difference.

This would be inefficient since all execution runs will have to be generated from scratch by modifying the failing run. Let us consider the first two execution runs tried out in Table 6.1. They are

$$\pi_1 = \langle 1, 3, 5, 6, 9, 10\rangle \qquad \mathit{diff}(\pi_f, \pi_1) = \langle 6\rangle$$
$$\pi_2 = \langle 1, 3, 4, 5, 6, 9, 10\rangle \quad \mathit{diff}(\pi_f, \pi_2) = \langle 3, 6\rangle$$

Recall that the failing run is $\pi_f = \langle 1, 3, 5, 6, 7, 10\rangle$ and the buggy program is shown in Figure 6.2. The run π_2 shares a common suffix with run π_1 (the subpath $\langle 5, 6, 9, 10\rangle$); the runs also share a common prefix – the subpath $\langle 1, 3\rangle$. Run π_2 can be obtained by evaluating the branch at line 3 differently over and above π_1. Indeed, our algorithm generates the execution runs in this incremental fashion. Thus, run π_1 is constructed by modifying failing run π_f at line 6 (the last branch occurrence of π_f). Run π_2 is then constructed by incrementally modifying run π_1 in the branch at line 3, that is, we do not construct run π_2 from scratch by modifying the failing run π_f at lines 3 *and* 6. This incremental path construction is crucial for constructing our bug report efficiently.

Complications due to nested branch statements Note that when a branch in the failing run is evaluated differently, several execution runs may be obtained due to nested branch statements. For example, for the program in Figure 6.2 if the failing run is $\langle 1, 2, 3, 4, 5, 10\rangle$, our algorithm will first try to evaluate branch 5 differently since it is the last branch in the failing run. However, this produces two execution runs $\langle 1, 2, 3, 4, 5, 6, 7, 10\rangle$ and $\langle 1, 2, 3, 4, 5, 6, 9, 10\rangle$ due to the nested branch statement at line 6. Our algorithm will check whether *any* of these two runs is feasible and successful before proceeding to construct any other execution runs. This is part of our attempt to generate the closest successful run according to the difference metric of Definition 6.2. We now explain our path generation algorithm in details.

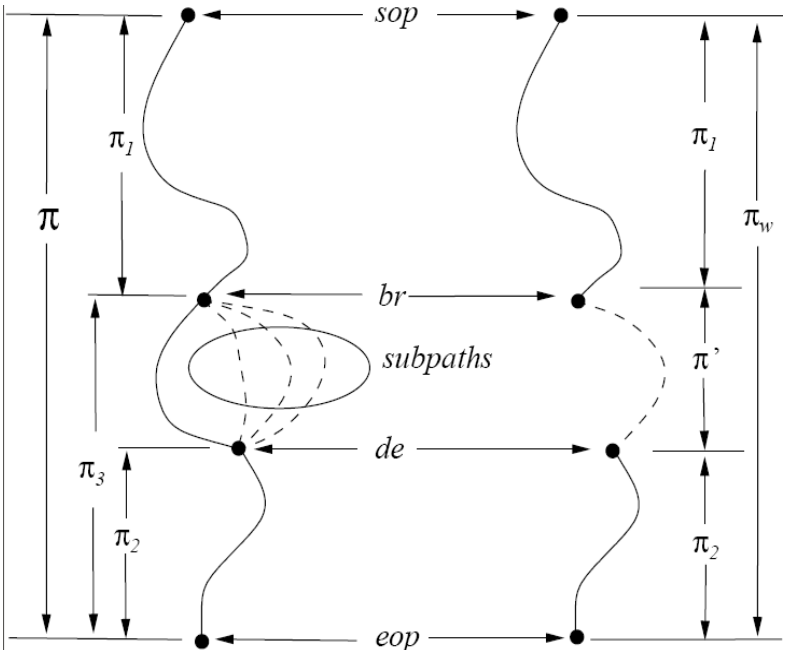


Figure 6.5: Explanation of algorithm in Figure 6.4.

Algorithm Description Our path generation algorithm is presented in Figure 6.4. Some of the variables used in the algorithm are pictorially explained in Figure 6.5, where solid line refers to subpath of the run π_f, and broken line refers to the subpaths constructed by evaluating the branch *br* differently from π_f. The algorithm proceeds by employing a recursive procedure `generatePaths`. This procedure is invoked at the top level with the parameters $\{\pi_f\}$, e_{last} and $\langle\rangle$, where π_f refers to the failing run, e_{last} refers to the last event in the π_f, and $\langle\rangle$ stands for the empty sequence.

As shown in Figure 6.4, the three parameters of `generatePaths` are *paths*, *last* and *df*. The `generatePaths` procedure constructs new execution runs from the runs captured in *paths* by evaluating branch events before the event *last* differently. All runs in *paths* have been constructed by evaluating differently events in the difference metric *df* w.r.t. the failing run π_f. These runs have the same difference w.r.t. the failing run. Let us re-visit the example in Table 6.1 which shows the order of path generation for the program in Figure 6.2 corresponding to the failing run $\pi_f = \langle 1, 3, 5, 6, 7, 10\rangle$.

The left column shows the *df* for all invocations of `generatePaths` procedure except the first (where *df* is the empty sequence). The right column shows the value of *paths* for each invocation of `generatePaths` except the first (where *paths* only contains the failing run). In this example, for every invocation of `generatePaths`, *paths* contains a single run.

The `while` loop in the `generatePaths` procedure iteratively retrieves a branch event prior to the event *last* in failing run π_f and assigns it to *br* (at line 30 of the algorithm). If there are no more branch events, *br* is undefined, and `generatePaths` returns Null (*i.e.* we cannot find a successful run). Each loop iteration of `generatePaths` tries to evaluate branch *br* differently along with other branch occurrences prior to *br* in failing run π_f.

In each iteration of the `while` loop of `generatePaths`, we first check whether any event in *df* is dynamically control dependent on *br*. If it is so, the branches in *br* as well as the branches in *df* cannot all be evaluated differently from the failing run π_f. To illustrate this point, let us look at the path generation example presented in Table 6.1; this table shows the order of path generation for the program in Figure 6.2 corresponding to the failing run $\pi_f = \langle 1, 3, 5, 6, 7, 10 \rangle$. Lines 5 and 6 of Figure 6.2 cannot be evaluated differently together w.r.t. the failing run.

If no event in *df* is dynamically control dependent on *br*, the algorithm generates new runs by evaluating differently the *br* event over and above the branches captured by *df*. Thus, *df* is updated to *df′* by adding *br* to *df*. Recall that the path-set *paths* captures the set of paths obtained by evaluating branches in *df* differently w.r.t. failing run π_f. Thus, to find the set of paths obtained by evaluating branches in *df′* differently w.r.t. failing run π_f, we exploit the relationship $df' = \langle br \rangle \circ df$ to simply evaluate *br* differently for all runs in *paths*. The resultant set of paths is captured in *newpaths*. Thus, our algorithm constructs *newpaths* by incrementally modifying *paths* instead of directly constructing it from the failing run π_f.

We now explain the functions used in the `generatePaths` procedure (lines 6 and 7 of Figure 6.4). The function $pde(br, \pi)$ called at line 6 returns de, the first event which is not (transitively) dynamically control dependent on br in the execution run π. The function $get\_all(br, de, \pi)$ called at line 7 of Figure 6.4 retrieves all acyclic paths where

- each acyclic path starts from $loc(br)$ (the control location of the branch event br) and ends at $loc(de)$ (the control location of the event de)
- br is evaluated differently from π_f in each acyclic path.

That is, we want to enumerate the unexecuted paths between $loc(br)$ and $loc(de)$, and use these paths to construct new execution runs. We choose to consider acyclic paths to avoid enumerating too many paths. However, this may cause us to miss the closest successful run since all possible program paths are not constructed by our algorithm.

In order to improve the performance of our algorithm, we have exploited the following property: if a path is infeasible, all extensions of the path are also infeasible. In particular, line 11 of the algorithm checks the feasibility of a subpath $\pi' \circ \pi_2$. If it is infeasible, all execution runs with $\pi' \circ \pi_2$ as suffix are also infeasible, and there is no need to check them. Similarly, when some event in df is dynamically control dependent on br, line 26 of the algorithm checks the feasibility of a subpath π_3 and prunes any execution runs with π_3 as suffix if π_3 is infeasible. Figure 6.5 shows the relation between various paths (*e.g.* π', π_2, π_3) used in Figure 6.4, the algorithm's pseudo-code.

Recall that we want the path generation algorithm in Figure 6.4 to construct execution runs monotonically w.r.t. the difference metric in Definition 6.2, so that it can return a successful run which is close to the failing run. This is stated in Theorem 6.1.

Lemma 6.1. *When the* `generatePaths` *method in Figure 6.4 is invoked with parameters paths, last and df, for* $\forall \pi \in paths, diff(\pi_f, \pi) = df$.

Proof. We use induction to prove this lemma.

Base : When the `generatePaths` method is initially invoked, $paths = \emptyset$. So, the lemma holds.

Induction : When the `generatePaths` method is recursively called at line 20 in Figure 6.4, all runs in *newpaths* are constructed by differently evaluating the branch *br* of runs in *paths*. Note that $\forall \pi \in paths, diff(\pi_f, \pi) = df$, and $df' = \langle br \rangle \circ df$. So, $\forall \pi \in newpaths, diff(\pi_f, \pi) = df'$. The lemma holds. □

Theorem 6.1. *[Proximity of Successful Run] Consider the failing execution run* π_f *for a program. If a run* π' *is constructed before another run* π'' *by the* `generatePaths` *method in Figure 6.4, then* $diff(\pi_f, \pi') < diff(\pi_f, \pi'')$ *(as per Definition 6.3) or* $diff(\pi_f, \pi') = diff(\pi_f, \pi'')$.

Proof. Note that execution run can only be constructed by line 13 of the algorithm in Figure 6.4 . When the run π' is constructed before another run π'', there are three possibilities:

(1) π' and π'' are constructed at the same iteration of the `while` loop (line 2 of Figure 6.4). Then, π' and π'' are constructed by differently evaluating the branch *br* of an execution run in *paths*. Note that $\forall \pi \in paths, diff(\pi_f, \pi) = df$, according to the Lemma 6.1. So, $diff(\pi_f, \pi') = diff(\pi_f, \pi'')$.

(2) π' and π'' are constructed at different iterations of the `while` loop (line 2 of Figure 6.4). Let π' is constructed by differently evaluating a branch *br* of an execution run in *paths*, and π'' is constructed by differently evaluating another branch br' of an execution run in *paths*. Since π' is constructed before π'', π' appears in an early iteration of the `while` loop, and *br* appears after br' in the π_f. Note that $\forall \pi \in paths, diff(\pi_f, \pi) = df$, according to the Lemma 6.1. So, $diff(\pi_f, \pi') < diff(\pi_f, \pi'')$, according to the comparison of differences in Definition 6.3.

(3) π' and π'' are constructed at different invocation of `generatePaths` method. Let π' is constructed by differently evaluating a branch br of an execution run in $paths$, and π'' is constructed by differently evaluating another branch br' of an execution run in $paths$. Since π' is constructed before π'', π' appears in an early invocation of the `generatePaths` method, and br appears after br' in the π_f. Note that $\forall \pi \in paths, diff(\pi_f, \pi) = df$, according to the Lemma 6.1. So, $diff(\pi_f, \pi') < diff(\pi_f, \pi'')$, according to the comparison of differences in Definition 6.3. □

Note that the above theorem does not claim that we generate the closest successful run from the failing run. *This reflects the reality, where we can, but choose not to generate the closest successful run for reasons of efficiency.* Our algorithm simply does not generate certain program paths and one of these can be the closest successful run.

Our path generation algorithm requires checking whether an execution run is feasible and successful (line 14 of Figure 6.4). We have used the automated theorem prover Simplify [26] to check for feasibility. This feasibility check returns the possible inputs under which the execution run is executed. We then check whether the execution run is successful (*i.e.* absence of the fault being localized) by checking the execution run for *any one* of these possible inputs. Clearly, for the same execution run, some inputs may lead to successful executions, while others lead to failing executions. Our implementation chooses any one of the feasible inputs and checks whether the corresponding execution run is successful. Checking whether the execution run for a specific input is successful, however, requires user intervention.

6.4 Experimental Setup

In order to experimentally validate our method for fault localization, we developed a prototype implementation of our path generation algorithm, and a prototype implementation of our algorithm to automatically choose a successful run from a pool of

successful runs. We have also implemented the Nearest Neighbor method with permutations spectrum, which performs best in [86], for a comparison with our method. We employed our prototypes on the Siemens benchmark suite [47, 89] and used the evaluation framework in [86] to quantitatively measure the quality of bug reports generated by all methods. The Siemens suite has been used by other recent works on fault localization [22, 86]. In this section, we introduce the subject programs (Section 6.4.1), the evaluation framework (Section 6.4.2), how our prototype implementation checks the feasibility of an execution run (Section 6.4.3), and the Nearest Neighbor method (Section 6.4.4).

6.4.1 Subject programs

Table 6.2 shows the subject programs from the Siemens suite [47, 89] which we used for our experimentation. There are 132 buggy programs in the Siemens suite, each of which is created from one of seven programs, by manually injecting defects. The seven programs range in size from 170 to 560 lines, including comments. The third column in Table 6.2 shows the number of buggy programs created from each of the seven programs. Various kinds of defects have been injected, including code omissions, relaxing or tightening conditions of branch statements, and wrong values for assignment statements.

In the experiments, we found that there was no input whose execution run observed the error, for two out of the 132 programs. Code inspection showed that, these two programs are syntactically different from, but semantically the same as correct programs. Actually, these two programs are not buggy programs, so we ruled out them from our experiments. We slightly changed some subject programs in our experiments. In particular, we rewrote all conditional expressions into *if* statements. We also rewrote the `schedule` and `schedule2` programs which read a floating point number and round it to an integer value to directly read an integer. This is because

Subject Pgm.	*Description*	*# Buggy versions*
schedule	priority scheduler	9
schedule2	priority scheduler	10
replace	pattern replacement	32
print_tokens	lexical analyzer	7
print_tokens2	lexical analyzer	10
tot_info	information measure	23
tcas	altitude separation	41

Table 6.2: Description of the Siemens suite.

our prototype uses the Simplify theorem-prover [26] to check the feasibility of an execution run, and Simplify does not work well with floating-point variables.

6.4.2 Evaluation framework

Renieris and Reiss have proposed an evaluation framework to evaluate the quality of a defect localizer [86]. Each error report is assigned a score to show the quality of this report. The score indicates the amount of code that an ideal programmer can ignore for debugging. Clearly, higher score indicates better quality bug report. We now discuss the score computation mechanism.

To compute the score of a bug report, [86] requires a correct version of the buggy program, where the defect has been fixed. Erroneous statements refer to the difference between the two programs (*i.e.* the statements which are fixed in the correct version). The score computation works on the program dependence graph (PDG) [44] for the buggy program. Nodes in a PDG represent statements in the program, and edges represent data or control dependencies between statements. We have used the Codesurfer [10] to construct the PDG. Erroneous statements are marked as "defect" in the PDG; statements included in the bug report are marked as "blamed" in the PDG. Let $DS(n)$ be the set of nodes that can reach or be reached from blamed nodes by traversing at most n directed edges in the PDG. For example, $DS(0)$ is the set of blamed nodes, and $DS(1)$ include blamed nodes and all nodes which have directed

edges to or from blamed nodes. We define that DS_* is the $DS(n)$ with the smallest n, which contains at least one erroneous statement. The score is then computed as:

$$score = 1 - \frac{|DS_*|}{|PDG|} \tag{6.1}$$

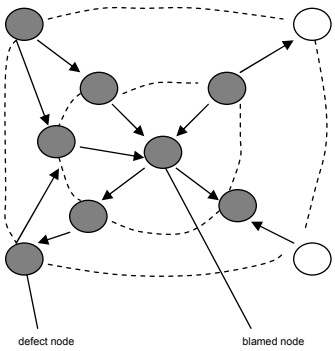


Figure 6.6: Example: illustrate the score computation

Figure 6.6 illustrates how the score is computed with an example. Each node represents a statement, and each edge represents static control or data dependence between statements. The "defect node" represents the erroneous statement which the developer wants to find, and the "blamed node" represents the statement in the bug report. In this example, the developer starts from the blamed node, and has to examines all the 8 dark nodes until the defect node is visited. That is, $DS_* = DS(2)$, where $DS(2)$ consists of the 8 dark nodes.

As a special case, we define the score to be zero for an empty report, since an empty report is useless to the programmer. This framework assumes that the programmer can find the error when he/she reads the erroneous statements, and he/she performs a breadth-first search for defect localization starting from statements in the error report. Thus, DS_* reflects the amount of code in the program that the programmer has to examine for defect localization using the bug report, and the score indicates the amount of code which can be ignored.

Note that the score only measures the utility of the bug report for debugging, *it does not necessarily correlate a good quality bug report with a lean bug report.* To address this weakness, we conducted separate experiments to measure bug report size. In addition, the edges in the PDG can also be considered as undirected; this should increase DS_* and hence decrease the score. When we ran experiments, we found that this change had little effect on the scores for bug reports generated by our approaches.

6.4.3 Feasibility check

One important problem in the prototype implementation is how to check the feasibility of an execution run or a subpath, as required by the path generation algorithm in Figure 6.4. Our prototype traversed the execution run from the beginning till the end to generate a constraint ϕ over program input and variables. We then invoked the Simplify theorem-prover [26] to prove the validity of the formula $\neg\phi$. If Simplify succeeds, we infer that the path is infeasible; otherwise we deem the path as feasible.

The use of the Simplify theorem prover requires us to consider the power of its decision procedures for inferencing. Simplify is sound but incomplete, that is, any formula it proves as valid is valid but it may fail to prove the validity of a valid formula. Thus, if ϕ_π is the constraint for a path π, $\neg\phi_\pi$ is valid (i.e. the path is actually infeasible) and Simplify fails to prove the validity of $\neg\phi_\pi$, we will actually treat π as feasible when it is not. However, this situation did not occur in our experiments with the Siemens suite.

Our prototype implementation models the heap as arrays in the constraint ϕ_π for a path π. It supports typical usage of the heap, *e.g.* reference/deference to pointers, dynamic memory allocation. However, it does not handle arithmetic operations over the pointers and type conversion.

6.4.4 The nearest neighbor method

Renieris and Reiss proposed a *Nearest Neighbor* (NN) method for fault localization in [86]. Their method assumes that there exists a failing run and a pool of successful runs, and then selects according to a *difference metric* the successful run that most resembles the failing run. The failing run and the successful run is then compared to produce a bug report of the program. The NN method is similar to our methods. However, they have defined and used another *difference metric*, which is based on the coverage of the program execution. In addition, the NN method can only be used to *choose* a successful run from a pool of successful run, and cannot be used to generate a successful run. We now elaborate the NN approach in the following.

For every execution run π, the NN method collects the numbers of times that each basic block is executed, and represents π as a sorted sequence s_π of basic blocks, where the sorting key is the collected basic block counts. The difference between two run π and π' is defined as the Ulam's distance [46] between sequences s_π and $s_{\pi'}$, where the Ulam's distance measures the minimal number of arbitrary moves to transform sequence s_π to $s_{\pi'}$. Given a failing run π_f and a pool of successful runs, the NN method selects the successful run π_s with the smallest Ulam's distance between s_{π_f} and s_{π_s}.

6.5 Experimental Evaluation

We employed the prototypes of our methods and the Nearest Neighbor method with permutations spectrum (NN method) [86] [2] to 130 buggy programs from the Siemens suite. The NN method compares code coverage between a failing run and the "nearest" successful run from a pool of successful runs. Through the experiments, we

[2]We used the accurate permutations spectrum for NN method and considered all failing runs which had *some* successful run with a different spectrum. So, we can study all the 130 programs compared to the 109 programs studied in [86] where certain programs were ruled out based on a coarser spectrum (coverage).

validate our two methods by answering the following four questions.

- Are our methods effective for fault localization?
- Is the size of generated bug report voluminous and overwhelming?
- How many successful runs are required available to make a "decent" choice, when we tries to automatically choose a successful run?
- Are our methods heavy in terms of time overheads?

In this section, we present experimental results for these questions.

6.5.1 Locating the Bug

In the Siemens benchmark suite, each buggy program P comes with a large pool of inputs, some of which result in successful runs, and others result in failing runs. We use the existing successful runs to evaluate our automatic choosing algorithm and the NN method.

However, for each failing run π_f, there may exist a set of successful runs $Closest(\pi_f)$ which are closest to π_f, in terms of our difference metric or that of the NN method. For our automatic choosing method and the NN method, the score for a failing run π_f averages scores of comparing π_f against each successful run π_s in $Closest(\pi_f)$, *i.e.*

$$score(\pi_f) = \frac{\sum_{\pi_s \in Closest(\pi_f)} score(\pi_f, \pi_s)}{|Closest(\pi_f)|}$$

where the quantity $score(\pi_f, \pi_s))$ is defined in Equation 6.1 in Section 6.4.2. For our automatic choosing method and the NN method, the score for a buggy program P averages scores of all failing run π_f of P, *i.e.*

$$pgm\_score(P) = \frac{\sum_{\pi_f \in Failing(P)} score(\pi_f)}{|Failing(P)|}$$

where $Failing(P)$ refers to the set of failing runs of program P. Our automatic choosing method differs from the NN method in which successful runs are selected for comparison, and (hence) which statements are reported in bug report.

Note that the complexity of our path generation algorithm is exponential with the length of the execution run. So, we always used simple failing runs, when we evaluated our path generation method. That is, we first chose a failing input which can observe the error for each buggy program from the Siemens test suite. We then used Zeller and Hildebrandt's approach [111] to further simplify the input for producing a failing run. This was particularly useful for the buggy versions of text-processing programs in the Siemens test suite (*e.g.*, `replace`, `print_tokens`). We then used such simplified failing run π_f to construct a successful run π_s.

Score	AC	APG	NN
0.9 - 1	30	38	14
0.8 - 0.89	18	9	26
0.7 - 0.79	11	9	27
0.6 - 0.69	14	6	18
0.5 - 0.59	19	5	14
0.4 - 0.49	12	8	13
0.3 - 0.39	8	5	3
0.2 - 0.29	12	10	4
0.1 - 0.19	3	2	1
0 - 0.09	3	38	10

Table 6.3: Distribution of scores.

Table 6.3 shows the distribution of *pgm_score* for three methods. Our automatic choosing method is shown as AC, and our automatic path generation method is shown as APG. As we can see, our two methods perform better than the NN method on the Siemens suite. Bug reports returned by either of our methods achieved a score of 0.9 or better for more than 30 buggy programs, while the NN method achieved a score of 0.9 or more for just 14 buggy programs. Note that a bug report with score of 0.9 or more indicates that programmer needs to inspect at most 10% of a buggy program

for fault localization using this bug report. This shows that our methods can often generate/choose a suitable successful run for fault localization.

Experience After generating the bug reports, we have studied these reports to understand what kind of errors can be easily localized by our methods. We found that both our methods are good at locating *Branch Faults*, where the error lies in some conditional branch statements. This is not surprising since the difference metric returned by our methods contains only branch statements with different outcomes in failing and successful runs.

For the same reason, our methods can also help effectively locate erroneous assignment statements, if branch statements which guard these assignments are included in the bug report. In addition, our methods may effectively locate errors due to code omission from the program text; these errors cannot be localized by conventional methods like dynamic or relevant slicing (see Chapter 3 and 4). In our methods, the successful run may differ from the failing run at the branch statement which the missing code is control dependent on. In the successful run, the missing code is not intended to be executed and hence we can locate the error.

Our methods cannot effectively locate errors due to wrong initialization of global variables. These statements are not guarded by any branch statements. In such cases, our methods often construct/choose a successful run which differs from the failing run at irrelevant branch statements. For example, several buggy programs of the `tcas` program in the Siemens benchmark suite contain such errors. For such buggy programs, our methods obtain a low score (less than 0.3).

6.5.2 Size of Bug Report

In the above experiments, we used scores to measure the quality of bug report according to the evaluation framework in Section 6.4.2. The reader should note that there is a fundamental difference between the bug report statements and the statements that

a programmer should inspect for debugging according to the evaluation framework. Clearly, measuring the amount of code to be inspected for debugging (captured by the bug report score) is important. However, we feel that measuring the bug report size is also important. If the programmer is overwhelmed with a voluminous bug report (*e.g.* 50 statements for a 500 line program), he/she may not even get to the stage of identifying which code to inspect using the bug report.

Figure 6.7 shows sizes of bug reports produced by our two methods and the NN method. We can see the bug reports produced by our methods are relatively small. For example, more than 90 bug reports contained less than 10 statements using either our methods, but the NN method only produced 39 small bug reports (*i.e.* contain less than 10 statements). Considering that programs in the Siemens suite are relatively small, reports with more than 10 statements may be too voluminous.

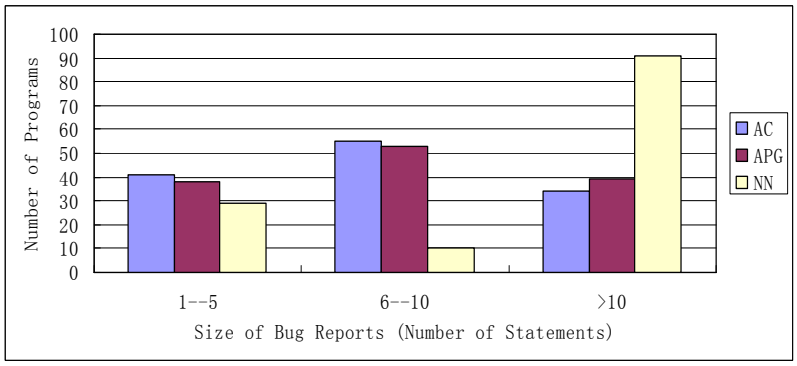


Figure 6.7: Size of bug reports.

6.5.3 Size of Successful Run Pool

In the Siemens suite, each faulty program has a large set of test inputs (1000 – 5000). The successful run pool is constructed out of these inputs. When we tries to automatically choose a successful run, how many successful runs are required for the programmer to make a decent choice? We study this in the following.

Given a program P, we selected the failing run π_f whose score $score(\pi_f)$ (using

both our method and NN method) is closest to the score of the program $pgm\_score(P)$) (again using both our and NN methods). The selected failing run π_f was used to study both our choosing method and the NN method. We did not conduct experiments w.r.t. all failing runs because it was too expensive.

Next, for every successful run π_s in the available pool of the Siemens suite, we computed the difference between π_f and π_s, generated a bug report by comparing π_f and π_s, and computed $score(\pi_f, \pi_s)$ (refer Equation 6.1). After all successful runs were processed, their differences were sorted in ascending order. Let π_i be the successful run with ith smallest difference w.r.t. π_f. The *parameterized mean score* of a faulty program P for a successful run pool-size of k is:

$$par\_score(P,k) = \sum_{i=1}^{n} score(\pi_f, \pi_i) \cdot p(i,k) \qquad p(i,k) = \frac{^{n-i}C_{k-1}}{^{n}C_k}$$

where π_f is the failing run chosen in P as mentioned above, n is the number of available successful runs in Siemens suite, and $p(i,k)$ is defined above. Here $^{n}C_k$ denotes a well-known quantity — the number of ways of choosing k items from n distinguishable items. Clearly, $p(i,k)$ denotes the probability that the ith-closest successful run of the failing run is chosen as the nearest successful run of a failing run from a pool of k different successful runs. Hence $par\_score(P,k)$ captures the statistical expectation of the score obtained for failing run π_f using any pool of k successful runs. Calculating the parameterized mean score $par\_score(P,k)$ allows us to avoid exhaustively enumerating the score of P for different successful run pools of size k.

Figure 6.8 presents the parameterized mean scores for different values of k, the successful run pool size. We see that both our automatic choosing method and the NN method made a decent choice of successful run from a pool of 5 runs and thereby achieved a score of at least 0.8 in 32 faulty programs. However, as the pool size increases to 40, our automatic choosing (AC) method achieved a score of 0.8 or more

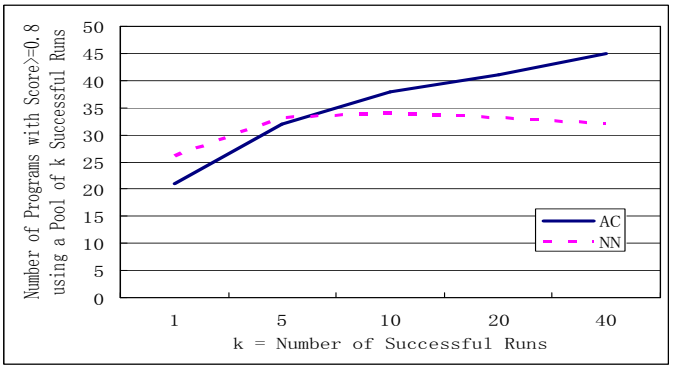


Figure 6.8: Impact of successful run pool-size.

for larger number of faulty programs (for 45 faulty programs). This is not the case for the NN method, which in fact needed even larger pool sizes.

6.5.4 Time Overheads

Figure 6.9 shows the time overheads for our automatic path generation (APG) approach. Our method found the successful run within 1 minutes for 80 buggy programs. Most of the time overheads for our method is due to the feasibility check by the external theorem prover Simplify. The feasibility check enables the following check to find whether a run is successful (since we cannot even observe the behavior of infeasible runs). Still the overall time overheads are tolerable for most programs in the Siemens suite.

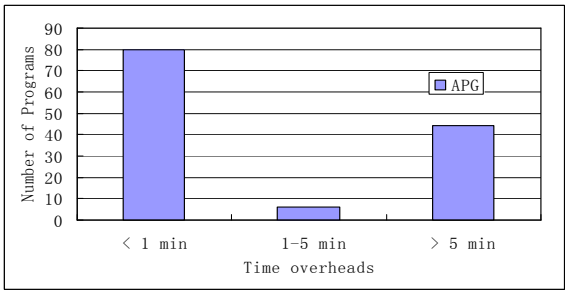


Figure 6.9: Time overheads for our path generation method.

Due to the limitation our prototype, we did not measure the time overheads of

our automatic choosing method. However, we believe that this method can be very fast. This is because the time overheads of our automatic choosing method is caused by the following two operations:

1. Collect the paths (*i.e.* sequences of executed statements) of the failing run and all successful runs, and

2. Compare successful runs against the failing run.

Note that there exist methods to efficiently collect the execution paths. [11] reports that the time overhead of their path collection approach is only 31% on average for the SPEC95 benchmarks. In addition, we only need a small number of successful runs to make a decent choice, according to our experiments. Consequently, the first operation can be performed with small time overheads. In our experiments, we found that the time for comparison (*i.e.* the second operation) is often very small and can be almost ignored.

6.5.5 Threats to Validity

In our experiments, we used the evaluation framework of Section 6.4.2 to measure the quality of bug report. However, the score computed by the framework of Section 6.4.2 may not accurately capture the human efforts for fault localization in practice. First, the framework assumes that the programmer can find the error when he/she reads the erroneous statements. This assumption may not hold for non-trivial bugs, where the programmer has to analyze program states. Secondly, the evaluation framework requires the programmer to perform pure breadth-first search for fault localization starting from statements in the bug report. However, the programmer usually has some understanding of the buggy program, and he/she can prune some irrelevant statements from bug report.

Our path generation algorithm generates execution runs close to the failing run and checks whether they are feasible and successful. As mentioned earlier, the check

for feasibility is done automatically by the Simplify theorem prover. Checking whether a run is successful is however done manually in our experiments. The time for this manual check is not included in the time overheads reported in Figure 6.9. In our experiments, the first feasible run constructed by the path generation algorithm was a successful one for most buggy programs. In the worst case, we had to manually examine 5 feasible failing runs for success before the algorithm found a feasible successful run. The reader should note that manual intervention in checking whether a run is successful is in some sense unavoidable. Otherwise the programmer has to precisely characterize the properties of a successful run, possibly as assertions; this eases our task of fault localization but places an additional burden on the programmer.

6.6 Summary

In this chapter, we have investigated the problem of test based fault localization, that is, localizing the error cause by comparing execution runes. We present a control flow based difference metric for this purpose. This difference metric can be used to:

- choose a successful run from a pool of program inputs, and
- generate a successful run close to the failing run.

The failing run and the successful run are then compared to discover the likely defects in the buggy program. Through this comparison, we highlight the sequence of branches in the failing run which are evaluated differently in the successful run.

CHAPTER 7

RELATED WORK

Software debugging has been an important topic of study for a long time. Various bugs may get introduced during software development, causing a program to deviate from the expected behaviors. The research on software debugging can be traced back to 1970s [73, 55] or even earlier. The early research focused on examining a program's execution states as the main debugging aid. These experiences and practices are still widely used in today's Debuggers, such as GDB [2]. However, the entire debugging process is done manually by the developer using various existing tools. The literature has investigated various ways to reduce the human intervention, and improve the state-of-art techniques for debugging. For example, the use of contracts can automate this process, by automatically checking the pre- and post-conditions of methods [16, 76]. Demsky et al. describe a tool Archie [24] which accepts specifications for consistency properties, and periodically checks these properties during execution.

There are also many methods and tools developed for (semi-) automated debugging, such as statistical analysis, model checking, theorem proving, type systems, and symbolic analysis [108, 33, 25, 43, 32, 18, 49, 59]. Actually, there is a rich body of work in this area. However, we discuss only those works which are relevant in the context of this thesis. In this thesis, we have presented an infrastructure for slicing of Java programs, and have addressed deficiencies in the existing slicing techniques. In addition, we have discussed test based fault localization technique which is not as expensive as slicing. In the rest of this section, we review the literature on the following two topics: program slicing and test based fault localization.

7.1 Program Slicing

The concept of program slicing was originally introduced by Weiser in [106]. Weiser's approach iteratively solves data-flow equations of a program to compute a program slice. The program slice consists of statements which could affect the behavior of the slicing criterion, *i.e.*, a variable referred at some interesting control location. Since then, various slightly different definitions of program slices have been proposed, as well as different algorithms to compute these slices. These different program slices are typically divided into two categories: static and dynamic, where a static slice is computed without any assumption about the program input, while a dynamic slice is computed corresponding to a specific program input. A survey of program slicing techniques developed in the eighties and early nineties appears in [98].

Static Slice Ottenstein was the first to use the Program Dependence Graph (PDG) to compute the static slice [80], where the nodes of the PDG are simple statements and the edges represent static data/control dependencies. The static slice is defined as all reachable nodes from the slicing criterion in the PDG.

Because Ottenstein's approach can only be applied to single-method programs, Horwitz et al. introduced the notion of System Dependence Graph (SDG) for static slicing of multi-method programs [45]. The SDG consists of dependence graphs for each method, as well as dependence edges to represent (1) control dependencies between the call statements and the callee methods, and (2) data dependencies caused by parameters and global variables. The static slicing is then defined as a reachability problem over the SDG, and the static slice can be computed by traversing the SDG twice.

Later, Larsen and Harrold extended Horwitz's algorithm to handle object-oriented programs [62]. The static slicing algorithm in [62] also operates on the System Dependence Graph (SDG), similar to the one used in [45]. However, Larsen and Harrold's

approach can incrementally construct the SDG, and they have shown how to compute slices for individual classes, groups of interacting classes and complete programs.

Dynamic Slice When the developer debugs a program, he/she usually focuses on a particular execution run. However, static slicing does not have any assumption about the program input. As a result, the static slice often contains many statements which are irrelevant to the observable error appearing in the selected execution. To solve this problem, dynamic slicing is proposed and studied.

The first dynamic slicing algorithm was introduced by Korel and Laski [57]. In particular, they exploited *dynamic flow concepts* to capture the dependencies between statement occurrences in the execution trace, and generated executable dynamic slices. Unfortunately, because of the conservative nature of their way to represent dependencies, the dynamic slice returned by [57] may contain some unnecessary statements.

Later, Agrawal and Horgan proposed to use the dynamic dependence graph (DDG) to precisely capture the dynamic dependencies between statement instances [6, 4], where each occurrence of a statement is represented as a distinct node in the DDG, and each dynamic control/data dependence is represented as an edge. The dynamic slice is defined as a reachable set over the DDG. The resulting slice is non-executable, but precise. Later, Xu et al. have shown how to perform dynamic slicing on object-oriented programs in [109].

Dynamic slicing is more suitable for the purpose of debugging, because dynamic slices are often much smaller and more precise than corresponding static slices. Additionally, dynamic slicing naturally supports the task of software debugging by analyzing a particular execution run. Agrawal et al. presented a systematic way to use dynamic slicing to (semi-) automate the debugging process.

The effectiveness of applying the dynamic slicing techniques for program debugging has been thoroughly evaluated experimentally in [100, 118]. These experiments show that, dynamic slices are typically much smaller than the original programs, and the real errors are often contained in these dynamic slices. This means that dynamic slicing is an effective method for software debugging, where the developer can locate the error by inspecting a small number of statements in the dynamic slice.

Besides software debugging, dynamic slicing has subsequently also been used for program comprehension and testing in many other innovative ways. In particular, dynamic slices (or their variants which also involve computing the closure of dependencies by trace traversal) have been used for studying causes of program performance degradation [121], identifying isomorphic instructions in terms of their run-time behaviors [91], and analyzing spurious counter-example traces produced by software model checking [72]. Even in the context of debugging, dynamic slices have been used in unconventional ways e.g. [8] studies reverse execution along a dynamic slice. Thus, dynamic slicing forms the core of many tasks in program development and it is useful to develop efficient methods for computing dynamic slices. Agrawal et al. showed how to use dynamic slicing to support the regression testing [7].

As far as slicing tools are concerned, several dynamic slicing tools have been developed in the research community [5, 79, 101, 58]. These slicing tools rely on the existing dynamic slicing algorithms to compute the dynamic slices, and highlight these dynamic slices in a source code browser. The developer then identifies the suspicious statement instances for debugging, by inspecting highlighted statements, that is, the dynamic slice in this case.

In this thesis, we have presented an infrastructure for dynamic slicing of Java programs, and have investigated the following three problems in the area of dynamic slicing:

1. Efficient tracing schemes for dynamic slicing,

2. Extend dynamic slicing to capture execution omission errors,

3. A better way to explore the dynamic slice.

These topics have been discussed and addressed in Chapters 3, 4, and 5 respectively. In the following three sub-sections, we discuss related works in the three topics.

7.1.1 Efficient Tracing Schemes

Dynamic slicing requires the entire control flow and data flow of an execution. It is well known that it is expensive to represent such a complete execution trace for realistic programs. The literature has proposed many efficient tracing schemes so that dynamic slicing can be applied to realistic programs. We now discuss them in the following.

Compact Trace Representations Because the execution trace can be viewed as a string, several researchers proposed to exploit the repetition among the string to reduce the high space overheads of storing and analyzing traces. Various compact trace representation schemes have been developed in [36, 63, 82, 120, 115] to compactly represent the execution trace.

Pleszkun presented a two-pass trace scheme, which recorded basic block's successors and data reference patterns [82]. The organization of his trace is similar to our compact trace representation (in Chapter 3). That is, the execution trace consists of trace tables. Each trace table stores the trace for one method m, and the table contains the trace sequences of each basic block of method m. However, Pleszkun's two-pass tracing technique does not allow traces to be collected on the fly. The space overhead is not indeed reduced. In addition, the trace is still large, because the techniques for exploiting repetitions in the trace sequences of basic blocks are limited.

The idea of separating out the data accesses of load/store instructions into a separate sequence (which is then compressed) is explored in [36] in the context of

parallel program executions. However, this work uses the SEQUITUR algorithm which is not suitable for representing contiguous repeated patterns. In our work, we have developed RLESe to improve SEQUITUR's space and time efficiency, by capturing contiguous repeated symbols and encoding them with their run-length. RLESe is different from the algorithm proposed by Reiss and Renieris [85], since it is an on-line compression algorithm, whereas Reiss and Renieris suggested modifying SEQUITUR grammar rules in a post processing step.

Recently, Larus proposed a compact and analyzable representation of a program's dynamic control flow via the on-line compression algorithm SEQUITUR [63]. The entire trace is treated as a single string during compression, but it becomes costly to access the trace of a specific method. Zhang and Gupta suggested breaking the traces into per-method traces [120]. However, it is not clear how to efficiently represent data flow in their traces. In a later work [115], Zhang and Gupta presented a unified representation of different types of program traces, including control flow, value, address, and dependence.

Zhang and Gupta present several heuristics to compactly represent the dynamic dependence graph, by exploiting repetitions of dependencies [114]. They also give a dynamic slicing algorithm which operates on the dependence graph. In contrast, our compression scheme is not related to the slicing criterion and exploits regularity/repetition of control/data flow in the trace. Our slicing algorithm operates directly on this compressed trace achieving substantial space savings at tolerable time overheads.

The approach in [117] uses a forward traversal based dynamic slicing algorithm, and computes the dynamic slice for every statement instance (*i.e.* regarding every statement instance as a distinct slicing criterion). Because there are many repetitions among these dynamic slices, [117] suggested to use reduced ordered binary decision diagrams (roBDDs) to represent the set of dynamic slices. Thus, the space and time

requirements of maintaining dynamic slices are greatly reduced. However, the forward traversal based dynamic slicing algorithm is not goal-directed w.r.t. the slicing criterion. It will compute many irrelevant dynamic dependencies, and construct useless dynamic slices.

Incomplete Trace Another approach for efficient tracing avoids tracing all bytecodes/instructions during trace collection.

The *abstract execution* technique [64] proposed by Larus falls in this category. Abstract execution technique executes a program P to record a small number of "significant events", thereby deriving a modified program P'. The program P' is then executed with the "significant events" as the guide; this amounts to re-executing parts of P for discovering information about instructions in P which were not traced. On the other hand, our method records certain bytecodes in an execution as a compressed representation. Although our method does not trace all bytecodes either, the post-mortem analysis of this compressed representation does not involve re-execution of the untraced bytecodes. To retrieve information about untraced bytecodes we detect dynamic dependencies via a lightweight flow analysis. This contrasts our approach from the abstract execution method.

In [27], Dhamdhere et al. presented an approach for dynamic slicing on compact execution traces. They do not employ any data compression algorithm on the execution trace. Instead, their technique classifies execution instances of statements as critical or non-critical, and store only the latest execution instances for non-critical statements. However, the classification of statements as critical/non-critical is sensitive to the slicing criterion.

7.1.2 Relevant Slicing

In the past, *relevant slicing* has been studied as an extension of dynamic slicing for the purpose of detecting the execution omission errors in a program [7, 40]. Agrawal

et al. [7] introduced the notion of *potential dependence* to capture the potential effects of some branch statements. If these branch statements are evaluated differently, some variables may be re-defined and the slicing criterion may be affected. Based on the notion of potential dependence, Agrawal et al. first presented a relevant slicing algorithm for software debugging. This algorithm works by *backward* traversal of the execution trace. Later, Gyimóthy et al. re-used the definition of potential dependence, and proposed a relevant slicing algorithm which works by *forward* traversal of the execution [40]. However, both of the two relevant slicing algorithms have their inherent limitations, compared with our algorithm presented in Chapter 4. We now elaborate the limitations as follows.

Space Efficiency Neither of the two relevant slicing algorithm is space efficient, and hence they may not be used to analyze realistic programs. More specifically,

- The algorithm in [7] relies on the huge dynamic dependence graph, and [116] has shown that it is not realistic to build the raw dynamic dependence graph for real programs.

- The forward relevant slicing algorithm in [40] avoids using the dynamic dependence graph. However, such a forward traversal based slicing algorithm will compute many redundant dependencies since it is not goal directed.

In Chapter 4, we present a relevant slicing algorithm which works *directly* on our compact representation of execution traces (as presented in Chapter 3). The costly decompression is not required during slicing.

Accuracy The two relevant slicing algorithms are less accurate than ours. In particular,

- The approach in [7] may wrongly ignore some important statements. This is because, if b is a branch statement with which statements in the slice have potential

dependencies, [7] only computes the closure of data and potential dependencies of b. In other words, control dependencies are ignored w.r.t. statements on which b is data dependent.

- The algorithm in [40] may include some superfluous statement into the relevant slice. This is because, while computing the dependencies of a later occurrence of certain branch statements (those which appear in the slice due to potential dependencies), the algorithm also includes statements which affect an early occurrence of the same branch statement.

In this thesis, we define a relevant slice over the *Extended Dynamic Dependence Graph* (EDDG), and it is more accurate than previous ones. Detail comparison of the accuracy appears in Section 4.2.

7.1.3 Hierarchical Exploration

The program analysis literature typically focuses on the analysis algorithms to efficiently produce useful results. Hierarchical exploration is a good way to use these results. In this approach, a programmer can gradually explore the analysis results in a hierarchical fashion, according to the program structure. Hierarchical exploration naturally helps suppress and ignore useless part of the analysis result, so that the burden of the developer can be reduced. Most importantly, the result is logically integrated with the program and presented to the developer, and the developer can understand the result more easily. Several researchers have conducted research on this topic.

Hierarchical Exploration of the Dependence Graph Balmas [13] proposed hierarchical exploration of *static program dependence graphs*. This approach was later extended for hierarchical visualization of dynamic data dependencies [14]. Their

approaches first construct the static/dynamic dependence graph. Nodes in the dependence graph are then hierarchically grouped as "super-nodes", according to the program structure. The programmer can then explore the dependence graph in a hierarchical fashion.

In fact, Balmas approaches present a good way to visualize the dependency graph, whereas we interleave the dependence computation and comprehension steps. Indeed this is the main contribution of our hierarchical dynamic slicing – we feel that *program comprehension cannot be left as a post-mortem activity, and should be used to guide dependence computation.* This is because such an interleaving may reduce the amount of human intervention during the exploration. Let us consider the situation where a programmer wants to explore the details of a "super-node" n. Balmas approaches will report all nodes and dependencies which are grouped together as this super-node n. On the other hand, our approach can prune some irrelevant dependencies guided by the human comprehension, although these dependencies and nodes can be reached from the slicing criterion.

In addition, we have proposed a phase division method which helps identify and structure the exact feedback needed from the programmer in general — the programmer needs to select one from among a given set of inter-phase dependencies. Last, but certainly not the least, we have conducted detailed experiments to show that our approach has the potential to make dynamic slicing more useful for software debugging.

The idea of using the phases of an execution trace for debugging also appears in earlier works. Miller and Choi [74] proposed to do so by: hierarchically dividing the execution trace into phases and presenting the dynamic dependence graph of each phase to the user. This is effectively exposing the dynamic dependence chains inside the phase completely to the programmer, thereby burdening him/her with

lot of redundant information! Our hierarchical dynamic slicing is exactly the *reverse* — we seek to hide the dynamic dependence chains inside a phase. Instead we summarize a phase via its "input" and "output" variables, which is gleaned from the inputs/outputs of the program as well as those of the preceding and succeeding phases.

Finally, we note that our hierarchical dynamic slicing is very different from the recently proposed Hierarchical Delta Debugging method [75]. This work seeks to simplify the program input that causes a program to fail. In this endeavor, it exploits the hierarchy present in the program input (*e.g.*, if the program input is an XML or HTML file). Our hierarchical dynamic slicing approach, on the other hand, seeks to hierarchically detect and explore the control/data dependence chains in a program.

Algorithmic Debugging and Program Slicing Algorithmic debugging presents an interactive debugging process [93]. Like our hierarchical dynamic slicing, algorithmic debugging also automatically divides the execution into phases in a hierarchical fashion, and gradually exposes these phases to the developer. The developer then inspects input/output variables of each phase, in order to determine the correctness of each phase and find out the suspicious phase. The suspicious phase is then further divided and examined, until the error is located. However, algorithmic debugging requires the developer to examine *all* input/output variables of each phase; while hierarchical dynamic slicing only requires the developer to examine *some* input/output variables which are related to the observable error. This means that hierarchical dynamic slicing requires less human intervention and is more productive than algorithmic debugging.

The works of [34, 54] combine algorithmic debugging and dynamic slicing to alleviate this problem, since dynamic slicing can certainly prune some irrelevant variables for the examination. These works bear some similarity to our work, since they also

rely on summarizing the behaviors of execution phases. However, to summarize a phase (say corresponding to a procedure call), they rely on a static summary of the procedure itself. In particular, they summarize the variable definitions of a procedure, which in the context of Java programs will require static points-to analysis. In contrast, the hierarchical dynamic slicing method only seeks to identify the inter-phase *dynamic* dependencies which can proceed efficiently without any points-to analysis.

7.2 Test Based Fault Localization

Because dynamic slicing techniques are often expensive, there has been a lot of interest in test based fault localization techniques. These techniques often collect limited control/data flow information of the execution run, and compare successful and failing runs of the buggy program. The difference is summarized as a bug report. The developer can then use the bug report for fault localization, since such a bug report often contains only a few statements and may pinpoint the error. We now discuss previous works in this research area.

Compare Execution Runs The literature has proposed many different test based fault localization techniques [12, 22, 38, 51, 83, 86, 87, 110]. These techniques often differ in *which characteristic of execution runs* is used for comparison.

Reps et al. [87] proposed to collect and compare two sets of acyclic paths for the purpose of debugging. One set contains the acyclic paths of successful runs, and another set contains the acyclic paths of failing runs. The bug report is defined as the difference between the sets. This is because, the acyclic paths of successful runs are considered as representative correct behaviors. If a path appears in the failing runs, but not the successful runs, this path is certainly suspicious and may be related to the error.

Pytlik et al. [83] proposed to use sets of potential invariants, instead of acyclic

paths, to detect the key difference between successful runs and failing runs. The potential invariants are dynamically discovered from execution runs, by using the Daikon [30] tool. The invariants are regarded as informal specifications to describe the behaviors of execution runs. Later, Brun and Ernst presented another way to use potential invariants for fault localization [17]. First, potential invariants were discovered from representative execution runs. In this step, machine learning techniques were used to prune some invariants which are unlikely to represent program properties. Next, these invariants were applied to the failing execution runs, and violated invariants were reported for fault localization.

Jones et al. [51] proposed to combine testing and test based fault localization techniques together for the purpose of software debugging. Their approach assigns a score to every statement *stmt*, to indicate the likelihood what the statement *stmt* is the error. The score is computed according to relative percentage of successful runs that execute the statement *stmt*, to failing runs. A lower score means that the statement *stmt* is executed primarily in failing runs, and should be highly suspicious as being faulty. Recently, [52] conducted empirical evaluation to show that this approach is quite productive for software debugging. Ruthruff et al. [90] used the idea of prioritizing statements according to the likelihood that the statement is faulty.

Liblit et al. proposed to use sampling techniques to randomly collect some data at certain points of the program [68, 69]. Their technique monitors branch coverage, return values, and invariant information. The collected data are then represented as predicates. The statistical theory is then applied to analyze and report the relationship between the predicates and the observable error, thereby help the developer debug the program.

Software fault localization via model checking has also been studied [19, 37]. These works seek to explain the counter-example produced by model checking by invoking

an optimization problem. The optimization generates a successful run which is "closest" to the counter-example; this is typically accomplished by an external constraint solver. Note that for these approaches, either the program model needs to closely reflect the behaviors of the actual program, or the approaches risk generating a spurious successful run (not corresponding to any program execution) which necessitates further refinement of the optimization problem.

Obtain Successful Runs Previous research along this line focuses on various ways to characterize program behaviors and compare successful and failing execution runs to generate accurate bug reports. However, these works do not discuss how a successful run is obtained. This is the main topic in Chapter 6. In this thesis, we have presented a control flow based difference metric, and shown how to use the difference metric to obtain successful runs.

The work of Renieris and Reiss [86] is related to ours. They have demonstrated through empirical evidence that a successful run which is "closest" to the failing run can be more helpful for error localization than a randomly selected successful run. However, [86] measures the proximity of two runs by comparing the *set* of basic blocks[1] executed in each run. Thus, they cannot distinguish between runs which execute exactly the same statements but in different order — consider the program `for (....){ if (...)  S1 else S2 }` and the two execution runs $\langle$`S1`, `S2`$\rangle$, $\langle$`S2`, `S1`$\rangle$. We consider the *sequence* of statements executed in each run for determining proximity between two runs. Clearly, even if for a faulty run, the programmer has a number of successful runs at his/her disposal (*i.e.* automated generation of one successful run is unnecessary), our sequence based distance metric can be used for accurately comparing the control flow of two runs. Additionally, the technique in [86] cannot be used to *generate* the closest successful run from a failing run.

[1] Actually a sorted sequence of the basic blocks based on execution counts is used; this is different from the execution sequence of the basic blocks in the failing run.

Our difference metric bears similarities to the notion of proximity between runs proposed by Zeller et al. in [22, 110]. Their approach compares program states with similar contexts for fault localization at some control locations. Through a series of binary search over the program state and re-executing (part of) the program from "mixed" states, a set of variables which may be responsible for the bug are mined and reported. However, these "mixed" states may be infeasible. Furthermore, it may be quite costly to compare program states and to re-execute the program several times.

The *delta debugging* in [111] automatically simplifies the erroneous input by removing part of this input. The reduced input usually corresponds to a shorter execution, which may be easier to debug. This approach may also generate a successful program input. However, the approach is more suitable for debugging language/text processing programs like compilers or web-browsers where we can get program inputs by deleting parts of a program input. For programs with integer inputs this approach may be problematic *e.g.* consider the situation where the failing input of a program is `i=2` and the only successful input is `i=3`.

There has also been intense research on the topic of input/test case generation based on various coverage criteria. The aim of these methods is to expose more program behaviors for the purpose of testing. This is somewhat different from the goal of path generation method, since we generate a program input/execution-run for localizing the error cause in a *specific* failing run.

CHAPTER 8

CONCLUSION

This chapter concludes the thesis. Section 8.1 summarizes the contribution of this thesis and Section 8.2 discusses some future directions.

8.1 Summary of the Thesis

With the increasing complexity of computer software, it is desirable that the burden of the debugging can be shifted from programmers to debugging tools. In this thesis, we investigate start-of-art automatic debugging techniques which may reduce the burden of programmers.

Dynamic slicing techniques identify parts of the program which are irrelevant to the observable error, so that programmers can focus on relevant parts of the program that need debugging, instead of the entire program. This is achieved by analyzing the dynamic control/data dependencies of the program execution, and capturing the statements which contribute to the computation of the observable error.

In this thesis, we study in detail the issues related to applying dynamic slicing for Java programs, and present a slicing infrastructure for Java. We also develop a dynamic slicing tool *JSlice* to be used by the research/development community. JSlice does not simply monitor and collect information about a program execution. Most importantly, JSlice analyzes the control and data dependencies in an execution for understanding why a test case failed. Because the JSlice tool supports the entire Java programming language and can be used for any Java program, more than *80 users* have downloaded it for their research and development. Following are the potential uses of our tool, almost all of which seem to be exercised by our current user base.

1. **Usage in software development:** Our tool can be used to highlight causes of an observable error while developing Java programs. Since it is tuned to a widely used programming language like Java, this gives the tool potential for wide applicability. In particular, our tool can be integrated with software testing frameworks to explain failed test cases for a software being developed.

2. **Usage in teaching:** Our tool can be used for teaching software/system engineering in Singapore and overseas.

3. **Usage in software engineering research:** Many researchers are currently using our tool for research in software reliability and comprehension.

4. **Using parts of the tool for purposes other than software debugging:** Since JSlice is available as an open-source tool, users can take parts of it and use them for other problems. In particular, the instrumentation and trace compression part of the tool can be used for efficient program profiling.

In our research work leading to the JSlice tool, we have made the following general contributions to the field of dynamic slicing.

First, dynamic slicing often requires the traces of execution runs. Because of the huge sizes of execution traces, we have developed a space efficient scheme for compactly representing bytecode traces of Java programs. The major space savings in our method come from the optimized representation of (a) data addresses used as operands by memory reference bytecodes, and (b) instruction addresses used as operands by control transfer bytecodes. We present a dynamic slicing algorithm which can directly traverse our compact bytecode traces without resorting to costly decompression. For our subject programs (drawn from standard suites such as the Java Grande benchmark suite or the SPECjvm suite) we obtain compression in varying amounts ranging from $5 - 5000$ times. We show that the time overheads for constructing this representation on-the-fly during program execution are tolerable.

Second, traditional dynamic slicing algorithms only analyze dynamic control/data dependencies which *actually* happen during the execution. However, these algorithms do not consider *potential* dependencies, where the execution of some statements may be wrongly omitted. In this thesis, we extend our dynamic slicing algorithm to perform "relevant slicing", by capturing execution omission errors. We show that our definition of relevant slicing is more accurate and helpful for software debugging than previously proposed notions of relevant slices. Additionally, our experimental results indicate that the additional capability of relevant slices comes at the cost of modest additional overheads in terms of computation time or slice sizes.

Third, the dynamic slice, *i.e.* the result of dynamic slicing, is often too large for human comprehension. We have proposed *Hierarchical Dynamic Slicing*, where a programmer is gradually guided through complex program dependence chains. This is as opposed to the arduous task of understanding a full dynamic slice, where all of the comprehension is left to the programmer. We have conducted detailed experiments on well-known subject programs written in Java, to evaluate the effectiveness of this approach. Our experiments show a substantial reduction in program understanding effort for our subject programs.

Dynamic slicing techniques have been proven useful in the last decades. However, people have found that the resultant dynamic slices often contain lots of false positives, *i.e.* the slice has lots of statements which are correct. This is because dynamic slicing techniques only analyze the failing execution run to produce the dynamic slices, where the failing run can only tell what is wrong, and cannot describe what is correct. Because of this disadvantage of dynamic slicing, researchers have proposed *test based fault localization* techniques. The fundamental observation of these techniques is that: the buggy program often has both the failing run, and some successful runs, and the successful runs show the expected behaviors of the program. By using failing and successful runs together, we may produce more meaningful bug reports, which can

help the developers.

In this thesis, we focus on the availability of a successful run for comparison. We have proposed two approaches for this purpose. The first approach is to automatically generate a successful execution π_s close to the failing execution π_f, and the second approach is to choose a suitable successful run π_s from a pool of successful runs. We then compare π_f and π_s to discover the likely defects from the buggy program. Through this comparison, we highlight the sequence of branches in the failing run which are evaluated differently in the successful run. Our approach does not require the user to provide successful executions for debugging as in previous approaches.

8.2 Future Work

In the future, the research can be continued in the following directions.

8.2.1 Future Extensions of our Slicing Tool

In this thesis, we have made our Java dynamic slicing tool *JSlice* available for use by researchers and developers. The dynamic slicing tool JSlice supports most features of the Java programming languages, such as object, field, inheritance, polymorphism, etc. In future, we can enhance JSlice to support more features of the Java programming language. In particular, exceptions, reflection and multi-threading are widely used features of the Java programming language. We can extend our dynamic slicing tool to handle these features in the following manner.

Exceptions When a program violates any semantic constraint of the Java programming language, the Java virtual machine throws an exception to signal this error [53]. This exception will cause a non-local transfer of control from the point where the exception occurred to the exception handler which can be specified by the programmer. It is necessary to store this non-local transfer of control during trace collection, so that we can reconstruct such a control transfer during the backward traversal for

dynamic slicing.

JSlice maintains traces of each bytecode separately in the trace tables for the program, as discussed in Section 3.1. Thus, the non-local transfer of control should be stored in the traces of the first bytecode of the exception handler. Note that this control transfer will cause the Java virtual machine to change the call stack, in the process of looking for an appropriate exception handler. In particular, the virtual machine will pop method invocations from the call stack up to the method invocation *invo_excep* (which the exception handler belongs to), and then execute the exception handler. For each invocation *invo* which is popped from or revised in the call stack, we need to record the following (assume that *invo* is an invocation of method *meth*):

- the class name of *meth*, and
- the method name of *meth*, and
- the signature of *meth*, and
- the id/address of last executed bytecode of *invo*, and
- the size of the operand stack of *invo* before *invo* is popped or revised.

When the first bytecode b of an exception handler is encountered during the backward traversal for slicing, the dynamic slicing algorithm should retrieve information from the traces of b, and reconstruct the call stack, so that the backward traversal can continue.

Exception handling also introduces extra dependencies into the program: the dynamic control dependence between (a) the bytecode occurrence which throws the exception and (b) the exception handler which catches the exception [95]. This means that, when any bytecode instance in the exception handler is included into the dynamic slice, the bytecode occurrence which throws the exception should also be included into the slice.

For Java programs, exception handlers often come with a `finally` block, where the Java Virtual Machine ensures that the finally block is always executed even if an unexpected exception occurs. However, the usage of the finally block complicates the construction of the control flow graph of a Java method, as discussed in the following. During the execution of a Java method, a finally block is always entered by executing a `JSR` bytecode. The semantics of the `JSR` bytecode is very similar to that of the `goto` bytecode. However, when a `JSR` bytecode b is executed, the address of the bytecode b' which immediately follows b is stored into the operand stack. When the finally block finishes execution, the saved address of the bytecode b' is retrieved and the execution continues from the bytecode b'. In other words, the bytecode b', which is executed after the finally block, is not represented as an operand in the last bytecode of the finally block. As a result, it is not clear which bytecodes may be executed after a finally block.

In order to discover this kind of information, the algorithm in Figure 8.1 is used. Given a method *meth*, the algorithm in Figure 8.1 returns an array *succ*, where for every bytecode *exit* which is the last bytecode of a finally block, $succ[exit]$ represents the set of bytecodes which may be executed after the bytecode *exit*. The algorithm proceeds by traversing the bytecode sequence of the method *meth* twice. During the first traversal, we mark the entry bytecode *entry* of each finally block, and maintain $next[entry]$, the set of bytecodes which may be executed after the finally block. During the second traversal, for every *entry* bytecode of a finally block, we detect the corresponding *exit* bytecode which exits the finally block. Additional, we set $succ[exit]$ to $next[entry]$, so that we can get the control flow information w.r.t. these *exit* bytecodes and construct the control flow graph as usual. The stack *entryStack* is required here because of nested finally blocks.

```
1   findNext (meth: a Java method)
2       initialize each element of the array next and succ to ∅;
3       initialize entryStack to null;
4       for (each bytecode b of the method meth)
5           if (the bytecode b is a JSR bytecode)
6               entry = the operand bytecode of the JSR bytecode b;
7               mark the bytecode entry as an entry of a finally block;
8               b' = the bytecode which immediately follows the JSR bytecode b;
9               next[entry] = next[entry] ∪ {b'};
10      for (each bytecode b of the method meth)
11          if (the bytecode b is an entry of a finally block)
12              push(entryStack, b);
13          if (the bytecode b is the last bytecode of a finally block)
14              entry = pop(entryStack);
15              exit = b;
16              succ[exit] = next[entry];
17      return succ;
```

Figure 8.1: The algorithm to find the bytecodes which may be executed after each `finally` block.

Reflection Reflection gives the Java code access to internal information of classes in the Java Virtual Machine, and allows the code to work with classes selected during execution, not in the source code. The main difficulty to support reflection for slicing lies in the fact that many reflection methods are implemented as native methods, and JSlice cannot trace details of native methods. Of all the native reflection methods, the following two kinds of methods are particularly important for dynamic slicing.

- *Methods which invoke a Java method*, such as the java.lang.reflect.Method.invoke method. Clearly, there exists a control transfer from the native method to the callee Java method. This control transfer is important for dynamic slicing, since we need to traverse the callee Java method for dynamic data dependence analysis. Here, we have to explicitly record the control transfer. The class name, method name, and signature of the callee method should be recorded.

- *Methods which read/write fields or arrays*, where we can deduce which variables are accessed according to the parameters and the invoking objects. For

example, field access methods in the java.lang.reflect.Field class fall into this category. These native methods are also essential for dynamic data dependence analysis. Note that these methods behave similarly with field/array access bytecodes, we can trace and analyze these methods in a similar way as corresponding bytecodes. That is, we trace the address (or identity) corresponding to the object/array, and the field name or the index of the array element. During dynamic slicing, such information is retrieved to detect dynamic data dependencies.

Multi-threading We plan to extend JSlice to support multi-threaded Java programs. The trace representation for a multi-threaded Java program could be similar to that for a single-threaded Java program. That is, each method has one trace table, and each row of the table maintains the control and data flow traces of a specific bytecode (see Section 3.1 for the trace representation). However, Java threads often communicate with each other through inter-thread events, such as shared variable access events, wait/notify events. The order of these events is required for dynamic slicing, because such an order is essential to reason/detect inter-thread dynamic dependencies. Levrouw et al. have proposed an efficient mechanism which can be used to trace the order of these inter-thread events [65]. We now briefly describe this approach in the following.

Levrouw's approach is based on the Lamport Clocks [61]. During the execution, each thread t_i has a scalar clock c_t^i, and each object o also maintains a clock c_o. These clocks are initialized to 0. Whenever there is an inter-thread event e where the thread t_i accesses the object o, this event is recorded with a time stamp $c_e = max(c_t^i, c_o) + 1$. The function max returns the maximum value of the two inputs. Additionally, c_t^i and c_o are updated to c_e. These recorded time stamps actually impose an partial order on all inter-thread events. Levrouw et al. show that we can replay the original execution and re-construct the dynamic dependencies, by enforcing inter-thread events following

the partial order.

There is one practical problem in employing the above scheme for tracing multi-threaded Java programs — all objects can be accessed by different threads, and it is often non-trivial to know which objects are shared before the execution. As a result, every access to an object should be considered as an inter-thread event. However, if we trace the time stamp for every object access, the trace size may explode. Fortunately, Levrouw et al. show that it is not necessary to trace all time stamps to record the partial order. In particular, for an inter-thread event e where the thread t_i accesses the object o, let c_t^i be the time stamp of t_i, and c_o of be the time stamp of o. We only need to trace the increment of c_t^i before and after the event e, if $c_t^i < c_o$. The reader is referred to [88, 65] for details. Note that the tracing scheme given here will work for multi-threaded Java programs running on multi-processor platforms as well.

The dynamic slicing algorithm for multi-threaded programs is similar to that for single-threaded programs (see Section 3.2). However, the algorithm should now maintain several operand stacks and call stacks, each of which corresponds to one thread. At any specific time, only one operand stack and one call stack are active. When we encounter an inter-thread event during the backward traversal, we pause the traversal along this thread until we have traversed all inter-thread events with a bigger time stamp. In addition, besides dynamic control and data dependencies, the slicing algorithm should also consider inter-thread dependencies, such as the dependencies introduced by wait-notify operations.

8.2.2 Other Research Directions

Application of the Compact Trace Besides dynamic slicing, our compact bytecode traces may also be useful for many other applications in code optimization and program visualization. First, the trace contains the sequence of target addresses for each conditional branch bytecode. We can obtain the most likely taken target

addresses from these sequences to merge basic blocks into a superblock [48]. Secondly, the operand sequences of bytecodes to invoke virtual/interface methods describe which methods are most likely to be invoked; this information is helpful in inlining methods for optimization [21]. Finally, note that by recording addresses of objects that each bytecode creates, our trace provides information about memory allocations. This can be used to understand the memory behavior via visualization, as discussed in [84].

Application of Hierarchical Exploration In this thesis, we have proposed the concept of *Hierarchical Exploration* and successfully applied it to dynamic slicing. It will be valuable to employ our idea of hierarchical dependence chain exploration to dynamic analysis methods other than slicing. For example, test based fault localization techniques generate a bug report by comparing execution runs. The programmer can then use the bug report to locate the real bug. However, when the bug report does not contain the actual error, the programmer can try to look at the control/data dependencies of the statements in the bug report in an attempt to localize the bug. The concept of hierarchical exploration can be employed in this context and makes the test based fault localization techniques more applicable.

Applying program debugging methods to modeling languages In the early stages of software development, modeling languages are often used to formally describe the software specifications and requirements. When any error is detected in the specifications, the error should be fixed before implementing the software. Some modeling languages, such as Live Sequence Charts (LSCs) [23], are executable and can be considered as abstract programs. It would be valuable to extend existing methods proposed for imperative programming languages to locate errors in specifications described by executable modeling languages.

REFERENCES

[1] "Apache JMeter." website: `http://jakarta.apache.org/jmeter/`.

[2] "The GNU project debugger." website: `http://www.gnu.org/software/gdb/gdb.html`.

[3] "The Kaffe Java virtual machine." website: `http://www.kaffe.org`.

[4] Agrawal, H., *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

[5] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "Debugging with dynamic slicing and backtracking," *Software - Practice and Experience (SPE)*, vol. 23, no. 6, pp. 589–616, 1993.

[6] Agrawal, H. and Horgan, J., "Dynamic program slicing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 246–256, 1990.

[7] Agrawal, H., Horgan, J., Krauser, E., and London, S., "Incremental regression testing," in *International Conference on Software Maintenance (ICSM)*, pp. 348–357, 1993.

[8] Akgul, T., Mooney, V., and Pande, S., "A fast assembly level reverse execution method via dynamic slicing," in *International Conference on Software Engineering (ICSE)*, pp. 522–531, 2004.

[9] Andersen, L. O., *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[10] Anderson, P. and Teitelbaum., T., "Software inspection using Codesurfer," in *the 1st Workshop on Inspection in Software Engineering*, 2001.

[11] Ball, T. and Larus, J. R., "Efficient path profiling," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 46 – 57, 1996.

[12] Ball, T., Naik, M., and Rajamani, S. K., "From symptom to cause: localizing errors in counterexample traces," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pp. 97–105, 2003.

[13] Balmas, F., "Displaying dependence graphs: a hierarchical approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 3, pp. 151–185, 2004.

[14] Balmas, F., Wertz, H., and Chaabane, R., "DDgraph: a tool to visualize dynamic dependences," in *Workshop on Program Comprehension through Dynamic Analysis*, 2005.

[15] Berk, E. J. and Ananian, C. S., "A lexical analyzer generator for Java." website: `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[16] Briand, L. C., Labiche, Y., and Sun, H., "Investigating the use of analysis contracts to support fault isolation in object oriented code," in *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 70–80, 2002.

[17] Brun, Y. and Ernst, M. D., "Finding latent code errors via machine learning over program executions," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 480–490, 2004.

[18] Bush, W., Pincus, J., and Sielaff, D., "A static analyzer for finding dynamic programming errors," *Software - Practice and Experience (SPE)*, vol. 30, no. 7, pp. 775–802, 2000.

[19] Chaki, S., Groce, A., and Strichman, O., "Explaining abstract counterexamples," in *ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE)*, pp. 73–82, 2004.

[20] Choi, J.-D. and Zeller, A., "Isolating failure-inducing thread schedules," in *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 210–220, 2002.

[21] Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M., "Practicing JUDO: Java under dynamic optimizations," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 13–26, 2000.

[22] Cleve, H. and Zeller, A., "Locating causes of program failures," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 342–351, 2005.

[23] Damm, W. and Harel, D., "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[24] Demsky, B., Cadar, C., Roy, D., and Rinard, M., "Efficient specification-assisted error localization," in *International Workshop on Dynamic Analysis (WODA)*, pp. 60–67, 2004.

[25] Detlefs, D., Leino, R., Nelson, G., and Saxe, J., "Extended static checking," tech. rep., December 1998.

[26] Detlefs, D., Nelson, G., and Saxe, J., "Simplify: A theorem prover for program checking," tech. rep., HP Labs, Palo Alto, CA, 2003. `http://research.compaq.com/SRC/esc/Simplify.html`.

[27] Dhamdhere, D., Gururaja, K., and Ganu, P., "A compact execution history for dynamic slicing," *Information Processing Letters*, vol. 85, no. 3, pp. 145–152, 2003.

[28] Dhodapkar, A. S. and Smith, J. E., "Comparing program phase detection techniques," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 217– 227, 2003.

[29] Do, H., Elbaum, S. G., and Rothermel, G., "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005. `http://www.cse.unl.edu/~galileo/sir`.

[30] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, 2001.

[31] Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[32] Flanagan, C. and Freund, S. N., "Type-based race detection for Java," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 219–232, 2000.

[33] Foster, J. S., Terauchi, T., and Aiken, A., "Flow-sensitive type qualifiers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–12, 2002.

[34] Fritzson, P., Gyimothy, T., Kamkar, M., and Shahmehri, N., "Generalized algorithmic debugging and testing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 317–326, 1991.

[35] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[36] Goel, A., Roychoudhury, A., and Mitra, T., "Compactly representing parallel program executions," in *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 191–202, 2003.

[37] Groce, A., "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 108–122, 2004.

[38] Groce, A. and Visser, W., "What went wrong: Explaining counterexamples," in *SPIN Workshop on Model Checking of Software*, pp. 121–135, 2003.

[39] Guo, L., Roychoudhury, A., and Wang, T., "Accurately choosing execution runs for software fault localization," in *Compiler Construction (CC)*, pp. 80–95, 2006.

[40] Gyimóthy, T., Beszédes, A., and Forgács, I., "An efficient relevant slicing method for debugging," in *7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 303–321, 1999.

[41] Hailpern, B. and Santhanam, P., "Software debugging, testing, and verification," *IBM system Journal*, vol. 41, no. 1, 2002.

[42] Harrold, M. J., Rothermel, G., Wu, R., and Yi, L., "An empirical investigation of program spectra," in *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 83–90, 1998.

[43] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G., "Software verification with Blast," in *SPIN Workshop on Model Checking of Software (SPIN)*, pp. 235–239, 2003.

[44] Horwitz, S. and Reps, T., "The use of program dependence graphs in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 392–411, 1992.

[45] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[46] Hunt, J. W. and Szymanski, T. G., "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[47] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 191–200, 1994.

[48] Hwu, W. W. and others, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, 1993.

[49] Jackson, D. and Vaziri, M., "Finding bugs with a constraint solver," in *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 14–25, 2000.

[50] JGF, "The Java Grande Forum Benchmark Suite." website: `http://www.epcc.ed.ac.uk/javagrande/seq/contents.html`.

[51] Jones, J. A., Harrold, M. J., and Stasko, J., "Visualization of test information to assist fault localization," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 467–477, 2002.

[52] Jones, J. A. and Harrold., M., "Empirical evaluation of the Tarantula automatic fault-localization technique," in *IEEE International Conference on Automated Software Engineering (ASE)*, pp. 273–282, 2005.

[53] Joy, B., Steele, G., Gosling, J., and Bracha, G., *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co, 2000.

[54] Kamkar, M., "Application of program slicing in algorithmic debugging," *Information and Software Technology*, vol. 40, no. 11-12, pp. 637–645, 1998.

[55] Katsoff, H., "Sdb: Symbolic debugger," *Unix Programmer's Manual*, 1979.

[56] Knuth, D. E., *The Art of Comuter Programming 4: searching and sorting*. Addison-Wesley Pub Co, 1973.

[57] Korel, B. and Laski, J. W., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[58] Korel, B. and Rilling, J., "Application of dynamic slicing in program debugging," in *International Workshop on Automatic Debugging*, 1997.

[59] Kremenek, T., Ashcraft, K., Yang, J., and Engler, D., "Correlation exploitation in error ranking," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp. 83–93, 2004.

[60] Krinke, J., "Static slicing of threaded programs," in *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 35–42, 1998.

[61] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[62] Larsen, L. and Harrold, M., "Slicing object-oriented software," in *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 495–505, 1996.

[63] Larus, J. R., "Whole program paths," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 259–269, 1999.

[64] Larus, J., "Abstract execution: A technique for efficiently tracing programs," *Software - Practice and Experience (SPE)*, vol. 20, pp. 1241–1258, 1990.

[65] Levrouw, L. J., Audenaert, K. M. R., and Campenhout, J. M., "A new trace and replay system for shared memory programs based on Lamport clocks," in *Euromicro Workshop on Parallel and Distributed Processing*, pp. 471–478, 1994.

[66] Lhoták, O., "Spark: A flexible points-to analysis framework for Java," Master's thesis, McGill University, December 2002.

[67] Lhoták, O. and Hendren, L. J., "Scaling Java points-to analysis using Spark," in *International Conference on Compiler Construction (CC)*, pp. 153–169, 2003.

[68] Liblit, B., Aiken, A., Zheng, A., and Jordan, M. I., "Bug isolation via remote program sampling," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 141–154, 2003.

[69] Liblit, B., Naik, M., Zheng, A., Aiken, A., and Jordan, M., "Scalable statistical bug isolation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 15–26, 2005.

[70] Lindholm, T. and Yellin, F., *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub Co, 1999.

[71] Lucia, A. D., "Program slicing: Methods and applications," in *IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 142–149, 2001.

[72] Majumdar, R. and Jhala, R., "Path slicing," in *International Conference on Programming Language Design and Implementation (PLDI)*, pp. 38–47, 2005.

[73] Maranzano, J. F. and Bourne, S. R., "A tutorial introduction to GDB," *Unix Programmer's Manual*, 1979.

[74] Miller, B. P. and Choi, J. D., "A mechanism for efficient debugging of parallel programs," in *ACM SIGPLAN conference on Programming Language design and Implementation (PLDI)*, pp. 135–144, 1988.

[75] Misherghi, G. and Su, Z., "HDD: Hierarchical delta debugging," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 142–151, 2006.

[76] Murray, D. J. and Parson, D. E., "Automated debugging in Java using OCL and JDI," in *International Symposium on Automated and Analysis-Driven Debugging (AADEBUG)*, 2000.

[77] Nagpurkar, P. and Krintz, C., "Visualization and analysis of phased behavior in Java programs," in *ACM International Conference on principles and practice of programming in Java*, pp. 27–33, 2004.

[78] Nevill-Manning, C. G. and Witten, I. H., "Linear-time, incremental hierarchy inference for compression," in *Data Commpression Conference (DCC)*, pp. 3–11, 1997.

[79] Nishimatsu, A., Jihira, M., Kusumoto, S., and Inoue, K., "Call-mark slicing: An efficient and economical way of reducing slice," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 422–431, 1999.

[80] Ottenstein, K. J. and Ottenstein, L. M., "The program dependence graph in a software development environment," in *ACM Software Engineering Symposium on Practical Software Development Environments*, pp. 177–184, 1984.

[81] Park, D. Y. W., Stern, U., Sakkebaek, J. U., and Dill, D. L., "Java model checking," in *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, pp. 253–256, 2000.

[82] Pleszkun, A. R., "Techniques for compressing programm address traces," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 32–39, 1994.

[83] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. P., "Automated fault localization using potential invariants," *CoRR*, vol. cs.SE/0310040, Oct, 2003.

[84] Reiss, S. P. and Renieris, M., "Generating Java trace data," in *ACM Java Grande Conference*, pp. 71–77, 2000.

[85] Reiss, S. P. and Renieris, M., "Encoding program executions," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 221–230, 2001.

[86] Renieris, M. and Reiss, S. P., "Fault localization with nearest neighbor queries," in *Automated Software Engineering (ASE)*, pp. 30–39, 2003.

[87] Reps, T. W., Ball, T., Das, M., and Larus, J. R., "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE)*, pp. 432–449, 1997.

[88] Ronsse, M. and Bosschere, K. D., "Recplay: a fully integrated practical record/replay system," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 133–152, 1999.

[89] Rothermel, G. and Harrold, M. J., "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, 1998.

[90] Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., II, M. F., and Main, M., "End-user software visualizations for fault localization," in *ACM Symposium on Software Visualization*, pp. 123–132, 2003.

[91] Sazeides, Y., "Instruction-isomorphism in program execution," in *Annual Value Prediction Workshop (affiliated with ISCA-30)*, pp. 47–54, 2003.

[92] Scheemaecker, M. D., "NanoXML." website: `http://nanoxml.sourceforge.net/orig/`.

[93] Shapiro, E., *Algorithmic program debugging*. PhD thesis, MIT Press, 1982.

[94] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically characterizing large scale program behavior," in *International Conference on Architectural Support for Programming Languages and Operating System*, pp. 45–57, 2002.

[95] Sinha, S. and Harrold, M. J., "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[96] SPECjvm98, "Spec JVM98 benchmarks." website: `http://www.specbench.org/osg/jvm98/`.

[97] Steensgaard, B., "Points-to analysis in almost linear time," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 32–41, 1996.

[98] Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[99] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V., "Soot - a Java bytecode optimization framework," in *Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, p. 13, 1999.

[100] Venkatesh, G. A., "Experimental results from dynamic slicing of C programs," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 197–216, 1995.

[101] Wang, T. and Roychoudhury, A., "*Jslice*: A dynamic slicing tool for Java programs." National University of Singapore, `http://jslice.sourceforge.net`.

[102] Wang, T. and Roychoudhury, A., "Using compressed bytecode traces for slicing Java programs," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 512–521, 2004.

[103] Wang, T. and Roychoudhury, A., "Automated path generation for software fault localization," in *ACM/IEEE International Conference on Automated Software Engineering (short paper)*, pp. 347–351, 2005.

[104] Wang, T. and Roychoudhury, A., "Dynamic slicing on Java bytecode traces," *ACM Transactions on Programming Languages and Systems (TOPLAS), to appear*, 2007.

[105] Wang, T. and Roychoudhury, A., "Hierarchical dynamic slicing," in *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pp. 228–238, 2007.

[106] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357.

[107] Xie, Y. and Aiken, A., "Scalable error detection using boolean satisfiability," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pp. 351–363, 2005.

[108] Xie, Y., Chou, A., and Engler, D. R., "Archer: using symbolic, path-sensitive analysis to detect memory access errors," in *Symposium on Foundations of Software Engineering held jointly with European Software Engineering Conference, (ESEC/FSE)*, pp. 327–336, 2003.

[109] Xu, B., Chen, Z., and Yang, H., "Dynamic slicing object-oriented programs for debugging," in *IEEE International Workshop on Source Code Analysis and Manipulation*, p. 115, 2002.

[110] Zeller, A., "Isolating cause-effect chains from computer programs," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp. 1–10, 2002.

[111] Zeller, A. and Hildebrandt, R., "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[112] Zhang, X., Gupta, N., and Gupta, R., "Locating faults through automated predicate switching," in *IEEE/ACM International Conference on Software Engineering*, pp. 272–281, 2006.

[113] Zhang, X., Gupta, N., and Gupta, R., "Pruning dynamic slices with confidence," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 169–180, 2006.

[114] Zhang, X. and Gupta, R., "Cost effective dynamic program slicing," in *ACM SIGPLAN conference on Programming language design and implementation*, pp. 94–106, 2004.

[115] Zhang, X. and Gupta, R., "Whole execution traces," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 105–116, 2004.

[116] Zhang, X., Gupta, R., and Zhang, Y., "Precise dynamic slicing algorithms," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 319–329, 2003.

[117] Zhang, X., Gupta, R., and Zhang, Y., "Effective forward computation of dynamic slices using Reduced Ordered Binary Decision Diagrams," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 502–511, 2004.

[118] Zhang, X., He, H., Gupta, N., and Gupta, R., "Experimental evaluation of using dynamic slices for fault location," in *International Symposium on Automated and Analysis-Driven Debugging (AADEBUG)*, pp. 33–42, 2005.

[119] Zhang, X., Tallam, S., and Gupta, R., "Dynamic slicing long running programs through execution fast forwarding," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp. 81–91, 2006.

[120] Zhang, Y. and Gupta, R., "Timestamped whole program path representation and its applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 180–190, 2001.

[121] Zilles, C. B. and Sohi, G., "Understanding the backward slices of performance degrading instructions," in *International Symposium on Computer Architecture (ISCA)*, pp. 172–181, 2000.

[122] Ziv, J. and Lempel, A., "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–349, 1977.

APPENDIX A

PROOFS AND ANALYSIS FOR DYNAMIC SLICING ALRITHM

A.1 Complexity Analysis of the RLESe Algorithm

In this appendix, we prove that the RLESe compression algorithm described in Section 3.1.3 is linear in both space and time.

A.1.1 Properties preserved by RLESe algorithm

Recall that RLESe constructs a context free grammar to represent a sequence, by preserving three properties w.r.t. the grammar: (1) no contiguous repeated symbols property, (2) digram uniqueness property, and (3) rule utility property (details can be found in Section 3.1.3). Figure A.1 presents the algorithm. The RLESe algorithm proceeds by iteratively reading a symbol from the input sequence (line 2 of Figure A.1), appending a node $\langle sym : 1 \rangle$ to the end of start rule (line 3 of Figure A.1), and re-structuring the grammar by preserving the above three properties (line 4-18 of Figure A.1). When the algorithm checks whether any property is violated (lines 5, 7, and 15 of Figure A.1), it is sufficient to examine changed nodes or digrams, instead of going through the entire grammar. For example, when the algorithm looks for continuous repeated nodes (line 5 of Figure A.1), only nodes which have just been inserted into the grammar are necessary to check. This is particularly important for the efficiency of the algorithm. We explain how to re-construct the grammar when any one of the three properties of RLESe are violated.

The first property (*i.e.* no contiguous repeated symbols property) is preserved by

line 5 and 6 of Figure A.1. If two nodes $\langle sym : n \rangle$ and $\langle sym : n' \rangle$ are adjacent in the grammar, line 6 merges the two nodes. That is we delete node $\langle sym : n' \rangle$, and change node $\langle sym : n \rangle$ to $\langle sym : n + n' \rangle$. Clearly, this merge operation can save one node in the grammar size.

Lines 7-14 of Figure A.1 preserve the second property of RLESe (*i.e.*, the digram uniqueness property). Recall that two digrams are *similar* if their nodes contain the same pair of symbols. Two digrams are *identical* if they have the same pairs of symbols and counters. Line 7 checks whether there are similar digrams in the grammar. If so, the algorithm can obtain two identical digrams, and replace both identical digrams with a non-terminal node for a rule (possibly already in existence) that has the identical digram as its right side. Note that, when there are two similar digrams $\langle sym_1 : n_1, sym_2 : n_2 \rangle$, and $\langle sym_1 : n_1', sym_2 : n_2' \rangle$, it may require splitting nodes (line 9 of Figure A.1) to obtain identical digrams as $\langle sym_1 : min(n_1, n_1'), sym_2 : min(n_2, n_2') \rangle$, where $min(n_1, n_1')$ is the minimum of n_1 and n_1'. Splitting one node will introduce one more node into the grammar. Line 9 of Figure A.1 checks whether one of the two identical digrams is exactly the right side of an existing rule. If so, the algorithm replaces another identical digram with a non-terminal node for the rule (line 11 of Figure A.1). We change the grammar rules

$$A \rightarrow \cdots sym_1 : n_1, sym_2 : n_2, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

to

$$A \rightarrow \cdots B : 1, \cdots \qquad B \rightarrow sym_1 : n_1, sym_2 : n_2$$

This can save one node in the resultant grammar. If not, a new rule is introduced, and the algorithm replaces both identical digrams with a non-terminal node for the new rule (line 13-14 of Figure A.1). That is, we change the grammar rule

$$A \rightarrow \cdots sym_1 : n_1, sym_2 : n_2, \cdots sym_1 : n_1, sym_2 : n_2, \cdots$$

to

$$A \rightarrow \cdots B:1, \cdots B:1, \cdots \quad B \rightarrow sym_1:n_1, sym_2:n_2$$

This operation does not introduce more nodes nor save any node, but introduces one more rule in the resultant grammar.

Lines 15-17 of Figure A.1 preserve the third property — the rule utility property. RLESe eliminates a rule referenced only once by replacing the reference with the right side of the rule. That is, we change the grammar rules

$$A \rightarrow \cdots B:1, \cdots \quad B \rightarrow sym_1:n_1, sym_2:n_2$$

to

$$A \rightarrow \cdots sym_1:n_1, sym_2:n_2, \cdots$$

This operation can save one node in the resultant grammar.

A.1.2 Operations in the RLESe algorithm

We proceed to prove that the complexity of the RLESe algorithm in Figure A.1 is linear in both space and time w.r.t. the length of input sequence. The proof is similar to the proof for SEQUITUR in [78], where we do not put a bound on each operation to re-construct the grammar. Instead we calculate the amortized costs, that is, we obtain a bound on the *total* amount of work done re-constructing the grammar. Our analysis of RLESe also uses an assumption made in SEQUITUR's analysis in [78] — given a digram, the average time to look for its similar digram is bounded by a constant. This can be achieved by indexing digrams with a hash table [56].

Table A.1 shows variables which will be used later for complexity analysis, as well as descriptions of these variables. The third column shows line number of the RLESe algorithm in Figure A.1 in which each variable is used; the last column shows corresponding operations for each line. Clearly, the time to perform each operation in Table A.1 is constant. For example, it needs constant time to check whether a specific

```
1   while (input sequence is not empty)
2     sym= read next symbol from input;
3     append node ⟨sym : 1⟩ to the end of start rule s
4     do
5       if (there are continuous repeated nodes)
6         merge the two nodes;
7       if (there are two similar digrams)
8         if (the two digrams are not identical)
9           split nodes to get two identical digrams;
10        if (one of the identical digrams is a complete rule)
11          replace another digram with a non-terminal node for the rule
12        else
13          create a new rule, where the right side of the rule is the identical digram;
14          replace both digrams with a non-terminal node for the new rule;
15      if (rule R is referenced only once)
16        replace the use of R with the right side of R;
17        remove the rule R;
18    while (any of the three properties is violated)
```

Figure A.1: The RLESe compression algorithm

node has the same symbol with its neighboring nodes (line 5 of Figure A.1). Thus, the total time cost to execute line 5 is proportional to m_2, the number of times to check the first property. Some lines of the compression algorithm are always executed together (*e.g.* lines 13 and 14 of Figure A.1). They are considered as one operation during complexity analysis, so they are put in the same entry in this table. The size of the RLESe grammar (*i.e.* m) denotes the nodes in the right-hand side of grammar rules, because nodes in the left-hand side of grammar rules can be recreated according to the order in which these rules appear.

A.1.3 Space Complexity

We derive an equation describing the size of the final grammar. Operations 3,7,10 in Table A.1 refer to merging nodes, using an existing rule and removing a rule; these operations save the number of nodes in the grammar. On the other hand, operation 6 (splitting a grammar node), increases the grammar size. Thus, we get the following equation relating the grammar size (m) and the length of the input sequence (n).

Var.	Description	Line in Fig. A.1	Operation
n	the size of the input sequence		
m	# of nodes in the final grammar		
r	# of rules in the final grammar		
m_1	# of times to read a symbol from input	2, 3	1: read
m_2	# of times to check the first property	5	2: check the 1st property
m_3	# of times to merge two nodes	6	3: merge
m_4	# of times to check the second property	7	4: check the 2nd property
m_5	# of times two similar digrams are found	8, 10	5: check digrams
m_6	# of split nodes	9	6: split
m_7	# of times an existing rule is used	11	7: use an existing rule
m_8	# of times a new rule is introduced	13, 14	8: introduce a new rule
m_9	# of times to check the third property	15	9: check the 3rd property
m_{10}	# of times a rule is removed	16, 17	10: remove a rule

Table A.1: Operations in the RLESe algorithm

$$n - m = m_3 + m_7 + m_{10} - m_6 \tag{A.1}$$

In addition, we can only split nodes which have been merged. Note that if a node was not produced by any merging, its run-length must be 1, so the question of splitting does not arise. We get:

$$m_6 \leq m_3 \tag{A.2}$$

From the two formulas, we can conclude that

$$n - m \geq m_7 + m_{10} > 0 \tag{A.3}$$

This shows that the size of the final grammar (*i.e.* the variable m) is bound by the length of the input sequence (*i.e.* the variable n), and the algorithm is linear in space.

A.1.4 Time Complexity

Next, we study the time complexity of the RLESe algorithm. When the algorithm checks the "no contiguous repeated symbols property", the first property of RLESe (operation 2), it is sufficient to look at the nodes inserted by operations 1, 7, 8, and 10 (refer Table A.1). That is,

$$m_2 = m_1 + m_7 + 2m_8 + 2m_{10} \tag{A.4}$$

The coefficient 2 is used because both operations 8 and 10 insert two nodes.

Operations 1, 7, 8, and 10 introduce new digrams which necessitate check of the digram uniqueness property, the second property of RLESe. We have

$$m_4 = m_1 + 2m_7 + 4m_8 + 2m_{10} \tag{A.5}$$

When a new rule is introduced and two identical diagrams are removed (operation 8), the number of references to a rule may be reduced, and the third property of RLESe (the rule utility property) is checked. Therefore,

$$m_9 = m_8 \tag{A.6}$$

In addition, the following formulas hold according to the structure of the algorithm,

$$m_3 \leq m_2 \quad \text{and} \quad m_5 \leq m_4$$

Now, let us look at the total time overhead for the compression algorithm in Figure A.1, which is sum of time cost for each operation. The expression for the total time overhead can be simplified as:

$$\begin{aligned} & m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 + m_{10} \qquad \text{(A.7)} \\ \leq\ & m_1 + 3m_2 + 2m_4 + m_7 + 2m_8 + m_{10} \\ \leq\ & m_1 + 3(m_1 + m_7 + 2m_8 + 2m_{10}) + 2(m_1 + 2m_7 + 4m_8 + 2m_{10}) + \\ & m_7 + 2m_8 + m_{10} \\ =\ & 6n + 8m_7 + 16m_8 + 11m_{10} \end{aligned}$$

For the number of rules r in the grammar, we have $r = m_8 - m_{10}$ since operation 8 introduces new rules, and operation 10 removes rules. Also, note the right side of each rule has at least two nodes. Thus $r < m$.

The expression for the total time overhead in Formula A.7 can then be simplified

as

$$
\begin{aligned}
& m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8 + m_9 + m_{10} \qquad \text{(A.8)} \\
= \;& 6n + 8m_7 + 16(r + m_{10}) + 11m_{10} \\
< \;& 6n + 16m + 27(m_7 + m_{10})
\end{aligned}
$$

Recall from Formula A.3

$$0 < m_7 + m_{10} \leq n - m < n \qquad \text{(A.9)}$$

So the time complexity for the RLESe algorithm in Figure A.1 is $O(n)$.

A.2 Analysis of the Dynamic Slicing Algorithm

In this appendix, we prove the lemmas used in the proof of Theorem 3.1, which proves the correctness of the dynamic slicing algorithm in Figure 3.2.

Lemma A.1. *Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 3.2. Then $\forall i, j,\ 0 < i < j \Rightarrow \varphi_i \subseteq \varphi_j$.*

Proof. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm, φ_i=φ_{i-1} or φ_i=$\varphi_{i-1} \cup \{\beta\}$. Thus, for all i we have $\varphi_{i-1} \subseteq \varphi_i$, and the lemma holds. □

Lemma A.2. *Let φ_i be the φ set, and $fram_i$ be the $fram$ after i loop iterations of the dynamic slicing algorithm in Figure 3.2. Let $fram_i^j$ represents a method invocation in $fram_i$. Then $\forall \beta'$, $\exists fram_i^j \in fram_i$, $\beta' \in fram_i^j.\gamma$, iff. $\beta' \in \varphi_i$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations.*

Proof. Let $\Gamma_i = \cup_j fram_i^j.\gamma$, *i.e.* the union of γ sets of all method invocations in $fram_i$, after i loop iterations of the dynamic slicing algorithm in Figure 3.2.

To prove this lemma, it is equivalent to prove: $\forall\beta' \in \Gamma_i$, iff. $\beta' \in \varphi_i$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations. Next we prove this by induction on loop iterations of the slicing algorithm.

Base : Initially, φ_0 and Γ_0 are both empty, so the lemma holds.

Induction : Assume $\forall\beta'' \in \Gamma_{i-1}$ iff. $\beta'' \in \varphi_{i-1}$ and the algorithm has not found the bytecode occurrence which β'' is dynamically control dependent on after $i-1$ loop iterations. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm in Figure 3.2, $\Gamma_i = (\Gamma_{i-1} - \mathcal{C}) \cup \mathcal{O}$, where,

- $\mathcal{C}$ is the set of bytecode occurrences in Γ_{i-1} which are dynamically control dependent on β. Note that if β is a method invocation bytecode occurrence, $\mathcal{C}= last\_fram.\gamma$ (line 14 in Figure 3.2). If β is a branch bytecode occurrence, $\mathcal{C}= BC$ (line 23 in Figure 3.2).

- $\mathcal{O} = \{\beta\}$ iff. $\beta \in \varphi_i$, and $\mathcal{O} = \emptyset$ iff. $\beta \notin \varphi_i$ (lines 32 and 33 in Figure 3.2).

We first prove the only if part of the lemma. For any $\beta' \in fram_i$,

1. if $\beta' \in \Gamma_{i-1} - \mathcal{C} \subseteq \Gamma_{i-1}$, $\beta' \in \varphi_{i-1}$ and the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after $i-1$ loop iterations according to the assumption. Lemma A.1 shows $\varphi_{i-1} \subseteq \varphi_i$, so $\beta' \in \varphi_i$. Since $\beta' \notin \mathcal{C}$, β' is not dynamically control dependent on β. This means that the algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations.

2. if $\beta' \in \mathcal{O}$ and $\mathcal{O} \neq \emptyset$, then $\beta' = \beta \in \varphi_i$. Clearly, the slicing algorithm has not found the bytecode occurrence $\overline{\beta}$ which β is dynamically control dependent on, because backward traversal has not encountered $\overline{\beta}$, which appears earlier than β during trace collection.

Next, we prove the if part of the lemma. Note that $\varphi_i = \varphi_{i-1}$ or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$ according to the slicing algorithm. For any $\beta' \in \varphi_i$ s.t. the slicing algorithm has not found the bytecode occurrence which β' is dynamically control dependent on after i loop iterations, we need to show that $\beta' \in \Gamma_i$. The following are the two possibilities.

1. if $\beta' \in \varphi_{i-1}$, then $\beta' \in \Gamma_{i-1}$ according to assumption. Since β' is not dynamically control dependent on β, $\beta' \notin \mathcal{C}$ and $\beta' \in \Gamma_i$.

2. if $\beta' = \beta$, then $\beta \in \varphi_i$ and $\mathcal{O} = \{\beta\}$. So $\beta' \in \Gamma_i$.

This completes the proof. □

Lemma A.3. *Let φ_i be the φ set, and δ_i be the δ set after i loop iterations of the dynamic slicing algorithm in Figure 3.2. Then $\forall v$, $v \in \delta_i$ iff. variable v is used by a bytecode occurrence in φ_i and the slicing algorithm has not found any assignment to v after i loop iterations.*

Proof. We prove the lemma by induction on loop iterations of the slicing algorithm.

Base : Initially, φ_0 and δ_0 are both empty, so the lemma holds.

Induction : Assume that $\forall v'$, $v' \in \delta_{i-1}$ iff. variable v' is used by a bytecode occurrence in φ_{i-1} and the algorithm has not found any assignment to v' after $i-1$ loop iterations. Let β be the bytecode occurrence encountered at the ith loop iteration. According to the algorithm, $\delta_i = (\delta_{i-1} - def\_vars) \cup use\_vars$, where

- def_vars is the set of variables assigned by β (lines 28 and 29 in Figure 3.2).

- use_vars is the set of variables used by β iff. $\beta \in \varphi_i$, and use_vars=$\emptyset$ iff. $\beta \notin \varphi_i$. (lines 20, 25, 30, 32 and 34 in Figure 3.2)

We first prove the only if part of the lemma. For any $v \in \delta_i$,

1. if $v \in \delta_{i-1} - def\_vars \subseteq \delta_{i-1}$, v is used by a bytecode occurrence in φ_{i-1} and the algorithm has not found any assignment to v after $i-1$ loop iterations according

to the assumption. Lemma A.1 shows $\varphi_{i-1} \subseteq \varphi_i$. So, v is used by a bytecode occurrence in φ_i. Since $v \notin def\_vars$, v is not defined by β. We can infer that the algorithm has not found any assignment to v after i loop iterations.

2. if $v \in use\_vars$ and $use\_vars \neq \emptyset$, then v is used by bytecode occurrence β and $\beta \in \varphi_i$. Clearly, the slicing algorithm has not found any assignment to the variable v after i loop iterations, because backward traversal has not encountered these assignments, which appear earlier than β during trace collection.

Next, we prove the if part of the lemma. Note that $\varphi_i = \varphi_{i-1}$ or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$ according to the slicing algorithm. Consider a variable v which is used by a bytecode occurrence in φ_i, and the slicing algorithm has not found any assignment to v after i loop iterations. For such a variable, we have the following two cases.

1. if v is used by a bytecode occurrence in φ_{i-1}, then $v \in \delta_{i-1}$ according to assumption. Since v is not defined by β, then $v \notin def\_vars$ and $v \in \delta_i$.

2. if v is used by bytecode occurrence β and $\beta \in \varphi_i$, then $v \in use\_vars$ and $use\_vars \subseteq \delta_i$. Thus, $v \in \delta_i$.

In both cases, we show that $v \in \delta_i$. This completes the proof. □

Lemma A.4. *During dynamic slicing according to the algorithm in Figure 3.2, a bytecode occurrence β pops an entry from op_stack, which is pushed to op_stack by bytecode occurrence β', iff. β' uses an operand in the operand stack defined by β during trace collection.*

Proof. The *op_stack* for slicing is a reverse simulation of the operand stack for computation during trace collection. That is, for every bytecode occurrence β'' encountered during slicing, the slicing algorithm pops entries from (pushes entries to) the *op_stack* iff. β'' pushes operands to (pops operands from) the operand stack during trace collection — as shown in the `updateOpStack` method in Figure 3.6. Consequently, a

bytecode occurrence β pops an entry from *op_stack*, and this entry is pushed to *op_stack* by bytecode occurrence β' during slicing, iff. β defines an operand in the operand stack, and β' uses the operand during trace collection. □

Lemma A.5. *Let φ_i be the φ set after i loop iterations of the dynamic slicing algorithm in Figure 3.2, and β be the bytecode occurrence encountered at the ith loop iteration. Then $\beta \in \varphi_i - \varphi_{i-1}$ iff. (1) β belongs to the slicing criterion, or, (2) $\exists\beta' \in \varphi_{i-1}$, β' is dynamically control or data dependent on β.*

Proof. Note that $\beta \notin \varphi_{i-1}$. According to the slicing algorithm, $\beta \in \varphi_i - \varphi_{i-1}$ iff. any of lines 19, 22 and 27 in Figure 3.2 is evaluated true so that any of lines 21, 26, and 31 in Figure 3.2 is executed. We next prove that any of lines 19, 22 and 27 in Figure 3.2 is evaluated true iff. (1) β belongs to the slicing criterion, or, (2) $\exists\beta' \in \varphi_{i-1}$, β' is dynamically control or data dependent on β.

First, line 19 in Figure 3.2 is evaluated to true iff. β belongs to the slicing criterion.

Next, we prove that line 22 in Figure 3.2 is evaluated to true iff. $\exists\beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β. According to the slicing algorithm, the check *computeControlDependence*(b_β, $curr\_fram$, $last\_fram$) in line 22 of the dynamic slicing algorithm (see Figure 3.2) returns true iff:

- β is a branch bytecode occurrence, and $\exists\beta' \in curr\_fram.\gamma$, $curr\_fram \in fram_{i-1}$ β' is dynamically control on β, or
- β is a method invocation bytecode occurrence, and $\exists\beta' \in last\_fram.\gamma$, $last\_fram \in fram_{i-1}$, β' is dynamically control on β,

According to Lemma A.2, $\forall\beta'$, $\exists fram^j_{i-1} \in fram_{i-1}$, $\beta' \in fram^j_{i-1}.\gamma$ only if $\beta' \in \varphi_{i-1}$. So, line 22 returns true only if $\exists\beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β.

On the other hand, if $\exists\beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β, then the algorithm has not found the bytecode occurrence which β' is dynamically control

dependent on after $i-1$ loop iterations, because every bytecode occurrence is dynamically control dependent on exactly one bytecode occurrence. So, $\exists fram^{j}_{i-1} \in fram_{i-1}$ $\beta' \in fram^{j}_{i-1}.\gamma$, according to Lemma A.2. If β is a branch bytecode occurrence, then $\beta' \in curr\_fram.\gamma$, $curr\_fram \in fram_{i-1}$, since β and β' should belong to the same method invocation. If β is a method invocation bytecode occurrence, then $\beta' \in last\_fram.\gamma$, $last\_fram \in fram_{i-1}$, since β' should belong to last method invocation, which is called by β. So line 22 in Figure 3.2 returns true if $\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β.

Finally, we prove that line 27 in Figure 3.2 is evaluated to true iff $\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β. Note that line 27 invokes the *computeDataDependence* method defined in Figure 3.7 to check dynamic data dependence. The check *computeDataDependence*(β, b_β) returns true iff either of the following conditions holds:

- if β defines a variable in δ_{i-1} (line 9 of Figure 3.7), where δ_{i-1} represents the δ set after $i-1$ loop iterations.
- if one of the top $def\_op(b_\beta)$ entries of the *op_stack* is pushed by a bytecode occurrence $\beta' \in \varphi_{i-1}$ (line 12 of Figure 3.7), where $def\_op(b_\beta)$ is the number of operands defined by bytecode b_β of occurrence β during trace collection.

When the `computeDataDependence` method returns true: (a) if β defines a variable $v \in \delta_{i-1}$, then $\exists \beta' \in \varphi_{i-1}$, v is used by β' and the algorithm has not found any assignment to v after $i-1$ loop iterations according to Lemma A.3. So β' is dynamically data dependent on β. (b) if one of the top $def\_op(b_\beta)$ entries of the *op_stack* is pushed by a bytecode occurrence $\beta' \in \varphi_{i-1}$. Because all the top $def\_op(b_\beta)$ entries of the *op_stack* will be popped by β (lines 2 and 3 of method `updateOpStack` in Figure 3.6), β' uses an operand in the operand stack defined by β during trace collection according to Lemma A.4. Consequently, β' is dynamically data dependent on β. This proves that line 27 in Figure 3.2 is evaluated to true, only if $\exists \beta' \in \varphi_{i-1}$,

β' is dynamically data dependent on β.

On the other hand, if $\exists\beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β, then either (a) $\exists v, \beta'$, $\beta' \in \varphi_{i-1}$, v is used by β' and v is defined by β. According to Lemma A.3, $v \in \delta_{i-1}$; so the `computeDataDependence` method returns true and line 27 in Figure 3.2 is evaluated to true. (b) $\exists\beta'$, $\beta' \in \varphi_{i-1}$, β' uses an operand in the operand stack defined by β during trace collection. According to Lemma A.4, β should pop an entry from *op_stack*, which is pushed into *op_stack* by β'. Since β pops top $def\_op(b_\beta)$ entries from the *op_stack*, line 12 in Figure 3.7 is evaluated to true, and the `computeDataDependence` method returns true. This proves that line 27 in Figure 3.2 is evaluated to true, if $\exists\beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β. □

A.3 Analysis of the Relevant Slicing Algorithm

In this appendix, we prove the correctness of the relevant slicing algorithm in Figure 4.7.

Lemma A.6. *Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 4.7. Then $\forall i, j,\ 0 < i < j \Rightarrow \varphi_i \subseteq \varphi_j$.*

Proof. Proof of this lemma is the same as the proof of Lemma A.1 in Appendix A.2, for the dynamic slicing algorithm. □

Lemma A.7. *Let φ_i be the φ set, and $fram_i$ be the $fram$ set after i loop iterations of the relevant slicing algorithm in Figure 4.7. Let $fram_i^j$ represents a method invocation in $fram_i$. Then $\forall\beta$, $\exists fram_i^j \in fram_i$, $\beta' \in fram_i^j.\gamma$ iff. (1) $\beta \in \varphi_i$, and (2) β belongs to slicing criterion or $\exists\beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (3) the algorithm has not found the bytecode occurrence which β is dynamically control dependent on after i loop iterations.*

Proof. Proof of this lemma is the similar to the proof of Lemma A.2 in Appendix A.2, for the dynamic slicing algorithm. □

Lemma A.8. *Let φ_i be the φ set, and δ_i be the δ set after i loop iterations of the relevant slicing algorithm in Figure 4.7. Then $\forall v$, $v \in \delta_i$ iff. (1) variable v is used by a bytecode occurrence $\beta \in \varphi_i$ s.t. (a) β belongs to slicing criterion, or (b) $\exists \beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (2) the algorithm has not found any assignment to v after i loop iterations.*

Proof. Proof of this lemma is the similar to the proof of Lemma A.3 in Appendix A.2, for the dynamic slicing algorithm. □

Lemma A.9. *Let φ_i be the φ set, and θ_i be the θ set after i loop iterations of the relevant slicing algorithm in Figure 4.7. Then $\forall v$, $\exists prop, v \in prop$, and $\langle \_\beta', prop \rangle \in \theta_i$ iff. (1) variable v is used by a bytecode occurrence $\beta \in \varphi_i$, where (a) β does not belong to slicing criterion, and (b) there is no $\beta' \in \varphi_i$ s.t. β' is dynamically control/data dependent on β, and (2) the algorithm has not found any assignment to v after i loop iterations.*

Proof. The proof of this lemma is similar to the proof of Lemma A.8. □

Indeed, the δ set (in Lemma A.8) includes variables used by bytecode occurrences β s.t. β is added into φ when (1) β belongs to the slicing criterion, or (2) there is any bytecode occurrence in φ which is *dynamically control/data dependent* on β. On the other hand, the *prop* sets of θ (in Lemma A.9) includes variables used by bytecode occurrences β s.t. β is added into φ when (1) there is any bytecode occurrence in φ which is *potentially dependent* on β, and (2) β does not belong to slicing criterion, and no bytecode occurrence in φ is dynamically control/data dependent on β.

Lemma A.10. *During relevant slicing according to the algorithm in Figure 4.7, a bytecode occurrence β pops an entry from op_stack, which is pushed to op_stack by*

bytecode occurrence β, iff. β uses an operand in the operand stack defined by β during trace collection.

Proof. Proof of this lemma is the same as the proof of Lemma A.4 in Appendix A.2, for the dynamic slicing algorithm. □

Lemma A.11. *Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 4.7, and β be the bytecode occurrence encountered at the ith loop iteration. Then $\beta \in \varphi_i - \varphi_{i-1}$ iff.*

1. *β belongs to the slicing criterion, or,*
2. *$\exists \beta' \in \varphi_{i-1}$, β' is dynamically control dependent on β, and β' was not introduced into the relevant slice φ because of potential dependencies.*[1]
3. *$\exists \beta' \in \varphi_{i-1}$, β' is dynamically data dependent on β, or*
4. *none of above three conditions is satisfied, and $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β.*

Proof. Note that $\beta \notin \varphi_{i-1}$. According to the slicing algorithm, $\beta \in \varphi_i - \varphi_{i-1}$ iff. any of lines 21, 26, 31 and 39 in Figure 4.7 is executed. Further,

I. line 21 in Figure 4.7 is executed iff. condition (1) in this lemma holds, which checks the slicing criterion.

II. line 26 in Figure 4.7 is executed iff. condition (2) in this lemma holds, which checks dynamic control dependencies.

III. line 31 in Figure 4.7 is executed iff. condition (3) in this lemma is satisfied, which checks dynamic data dependencies.

[1] *In other words, either there exists a bytecode β" $\in \varphi_{i-1}$ which is dynamically data/control dependent on β', or β' belongs to the slicing criterion.*

IV. line 39 in Figure 4.7 is executed iff. condition (4) in this lemma is satisfied, which checks potential dependencies.

Proofs of I, II and III are similar to proof of Lemma A.5 in Appendix A.2, for the dynamic slicing algorithm.

Next, we prove IV., that is line 39 in Figure 4.7 is executed iff. condition (4) in this lemma is satisfied. According to the slicing algorithm, line 39 in Figure 4.7 is executed iff. line 34 in Figure 4.7 is evaluated to false and line 38 in Figure 4.7 is evaluated to true. Note that line 34 in Figure 4.7 is evaluated to false iff. lines 19, 22, and 27 are all evaluated to false, which are equivalent to that none of conditions (1) (2) and (3) of this lemma holds. Note that line 38 invokes the *computePotentialDependence* method defined in Figure 4.8 to check potential dependencies. The check *computePotentialDependence*(β, b_β) returns true iff. either of following conditions holds:

1. line 6 in Figure 4.8 is evaluated to true, or
2. line 9 in Figure 4.8 is evaluated to true.

We first prove that the *computePotentialDependence* method returns true only if $\exists\beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 4.7 is evaluated to false. We have the following two cases:

1. there exists $v \in \delta_{i-1}$ which may be defined by evaluating the branch bytecode occurrence β differently. The β refers to the bytecode occurrence encountered at the ith loop iteration of the relevant slicing algorithm. According to Lemma A.8, $\exists\beta' \in \varphi_{i-1}$, v is used by β'. So, β' is potentially dependent on β.
2. there exists $v, prop''$, $v \in prop''$, $\exists\langle \_\beta'', prop''\rangle \in \theta_{i-1}$, and v may be defined by evaluating the branch bytecode occurrence β differently. According to Lemma A.9, $\exists\beta' \in \varphi_{i-1}$, v is used by β'. So, β' is potentially dependent on β.

In both cases, there exists one bytecode occurrence in φ_{i-1} which is potentially dependent on β.

Now we prove that the *computePotentialDependence* method returns true if $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 4.7 is evaluated to false. The following are two possibilities:

1. there exists v used by a bytecode occurrence $\beta' \in \varphi_{i-1}$, where β' was not introduced into the relevant slice φ because of potential dependencies. According to Lemma A.8, $v \in \delta_{i-1}$. So line 7 of Figure 4.8 is executed and the *computePotentialDependence* method returns true.

2. there exists v used by a bytecode occurrence $\beta' \in \varphi_{i-1}$, where β' was introduced into the relevant slice φ because of potential dependencies. According to Lemma A.9, $\exists prop''$, $v \in prop''$, and $\exists \langle \beta'', prop'' \rangle \in \theta_{i-1}$. According to the algorithm, β' is (transitively) dynamically control dependent on β'', so β'' is not dynamically control dependent on β. Thus, line 10 of Figure 4.8 is executed and the *computePotentialDependence* method returns true.

The completes our proof that the *computePotentialDependence* method returns true if $\exists \beta' \in \varphi_{i-1}$, β' is potentially dependent on β, assuming that line 34 in Figure 4.7 is evaluated to false. Consequently, line 39 in Figure 4.7 is executed iff. condition (4) in this lemma is satisfied.

In all cases, we have shown that any of lines 21, 26, 31 and 39 in Figure 4.7 is executed iff. any of the four conditions in the lemma is satisfied. Consequently, the lemma holds. □

Finally, we prove the correctness of the relevant slicing in Figure 4.7. Note that the relevant slice defined in Definition 4.2 is based on the *Extended Dynamic Dependence Graph* (EDDG). In the EDDG, two nodes in the graph may refer to the same bytecode occurrence. In the following, we use $nn(\beta)$ to represent the *non-dummy node* for

bytecode occurrence β in the EDDG, and $dn(\beta)$ to represent corresponding *dummy node* for bytecode occurrence β. Two nodes of the same bytecode occurrence do not contribute to relevant slice together. This is because in the EDDG, non-dummy nodes only have incoming edges representing dynamic control/data dependencies, and dummy nodes only have incoming edges representing potential dependencies. Further, the relevant slicing algorithm includes a bytecode occurrence β into the slice φ when $\exists\beta' \in \varphi$ s.t. β' is dependent on β for *any* of dynamic control, dynamic data and potential dependencies.

Theorem A.1. *Given a slicing criterion, the relevant slicing algorithm in Figure 4.7 returns relevant slice defined in Definition 4.2.*

Proof. Let φ_i be the φ set after i loop iterations of the relevant slicing algorithm in Figure 4.7, φ_* be the resultant φ set when the algorithm finishes, and β be the bytecode occurrence encountered at the ith loop iteration. As mentioned in the above, there may be two nodes $nn(\beta')$ and $dn(\beta')$ for a bytecode occurrence β' in the EDDG. So, we will prove this lemma by showing: φ_*=$\{\beta' | nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG$\}$.

We first prove the soundness of the algorithm, *i.e.* for any $\beta', \beta' \in \varphi_*$, only if either $nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG. In particular, we prove that: $\forall\beta' \in \varphi_*$, (a) if β' is added into φ_* because of slicing criterion or dynamic control/data dependencies, then $nn(\beta')$ is reachable from the slicing criterion in the EDDG, and (b) if β' is added into φ_* because of potential dependencies, then $dn(\beta')$ is reachable from the slicing criterion in the EDDG. We prove this by induction on loop iterations of the slicing algorithm. Initially, $\varphi_0 = \emptyset$, so the base case holds.

Induction : Assume that for any $\beta'' \in \varphi_{i-1}$, (a) if β'' is added into φ_{i-1} because of slicing criterion or dynamic control/data dependencies, then $nn(\beta'')$ is reachable from the slicing criterion in the EDDG, and (b) if β'' is added into φ_{i-1} because of potential dependencies, then $dn(\beta'')$ is reachable from the slicing criterion in the EDDG.

Note that $\varphi_i = \varphi_{i-1}$, or $\varphi_i = \varphi_{i-1} \cup \{\beta\}$. Then, $\forall \beta' \in \varphi_i$, we have two cases:

1. if $\beta' \in \varphi_{i-1}$, the induction hypothesis still hosts, since $\varphi_{i-1} \subseteq \varphi_i$ according to Lemma A.6.

2. if $\beta' = \beta$, where β is the bytecode occurrence encountered at the ith loop iteration of the slicing algorithm, then $\beta \in \varphi_i - \varphi_{i-1}$. According to Lemma A.11, we have following four possibilities to add β into φ_i:

   I. if β belongs to the slicing criterion,then clearly $nn(\beta)$ belongs to slicing criterion,

   II. if $\exists \beta'' \in \varphi_{i-1}$, β'' is dynamically control dependent on β, and β'' was not added into the relevant slice because of potential dependencies, then $nn(\beta'')$ is reachable from the slicing criterion in the EDDG according to the induction hypothesis. In addition, there is an dynamic control dependence edge from $nn(\beta'')$ to $nn(\beta)$ in the EDDG. Thus, $nn(\beta)$ can be reached from the slicing criterion.

   III. $\exists \beta'' \in \varphi_{i-1}$, β'' is dynamically data dependent on β, then either $nn(\beta'')$ or $dn(\beta'')$ is reachable from the slicing criterion according to the induction hypothesis. In the EDDG, there are dynamic data dependence edges from $nn(\beta'')$ to $nn(\beta)$, and from $dn(\beta'')$ to $nn(\beta)$. Thus, $nn(\beta)$ can be reached from the slicing criterion.

   IV. $\exists \beta'' \in \varphi_{i-1}$, β'' is potentially dependent on β, then either $nn(\beta'')$ or $dn(\beta'')$ can be reached from the slicing criterion according to the induction hypothesis. In the EDDG, there are potential dependence edges from $nn(\beta'')$ to $dn(\beta)$, and from $dn(\beta'')$ to $dn(\beta)$. Thus, $dn(\beta)$ can be reached from the slicing criterion.

In all four cases, we show that (a) if β is added into φ_i because of slicing

criterion or dynamic control/data dependencies, then $nn(\beta)$ is reachable from the slicing criterion in the EDDG, and (b) if β is added into φ_i because of potential dependencies, then $dn(\beta)$ is reachable from the slicing criterion in the EDDG.

Next, we prove the completeness of the slicing algorithm, *i.e.* for any $\beta', \beta' \in \varphi_*$, if either $nn(\beta')$ or $dn(\beta')$ is reachable from the slicing criterion in the EDDG. Note that there is no cycle in the EDDG, so we prove the completeness by induction on structure of the EDDG.

Base : Consider a bytecode occurrence β' where β' belongs to the slicing criterion. Clearly, $nn(\beta')$ is reachable from the slicing criterion in the EDDG. Let β' be encountered at the ith loop iteration of the slicing algorithm. By Lemma A.11 & A.6, $\beta' \in \varphi_i \subseteq \varphi_*$.

Induction : Assume that a set of bytecode occurrences $\beta'' \in \varphi_*$, which satisfy (1) if $nn(\beta'')$ is reachable from the slicing criterion in the EDDG, β'' is added into the relevant slice φ_* because of slicing criterion or dynamic control/data dependencies, and (2) if $nn(\beta'')$ is not reachable and $dn(\beta'')$ is reachable from the slicing criterion, then β'' is added into the relevant slice φ_* because of potential dependencies.

Consider a bytecode occurrence β, which can be reached from the slicing criterion by traversing only nodes of bytecode occurrences in φ_*. Clearly, $\exists \beta' \in \varphi_*$, β is dynamically control, or dynamically data, or potentially dependent on β'. Let β be encountered at the ith loop iteration of the algorithm, and β' be encountered at the jth loop iteration of the algorithm. Because β appears earlier than β' during trace collection, backward traversal of the trace will encounter β after β', *i.e.* $j < i$. Thus, $\beta' \in \varphi_j \subseteq \varphi_{i-1}$ according to Lemma A.6. We now show that $\beta \in \varphi_i$ according to the relevant slicing algorithm. In particular, (1) if $nn(\beta)$ is reachable from the slicing criterion in the EDDG, then β is added into the slice because of slicing criterion, or dynamic control/data dependencies, and (2) if $nn(\beta)$ is not reachable and $dn(\beta)$ is

reachable from the slicing criterion, then β is added into the slice because of potential dependencies. Note that the relevant slicing algorithm check dynamic control/data, and potential dependencies in order. The following are three possibilities:

I. if (1) there is a dynamic control dependence edge from $nn(\beta')$ to $nn(\beta)$, and (b) $nn(\beta')$ is reachable from the slicing criterion, then β' is added into the relevant slice φ_* because of slicing criterion or dynamic control/data dependencies, according to the induction hypothesis. Thus, $\beta \in \varphi_i$ and β is added into the relevant slice φ_* because of dynamic control dependencies, since condition (2) of Lemma A.11 is satisfied.

II. if (a) condition of case I does not hold, and (b) there is a dynamic data dependence edge from either $nn(\beta')$ ($dn(\beta')$) to $nn(\beta)$, and (c) $nn(\beta')$ ($dn(\beta')$) is reachable from the slicing criterion. Note that β' is in the slice. So $\beta \in \varphi_i$ and β is added into the relevant slice φ_* because of dynamic data dependencies, since condition (3) of Lemma A.11 is satisfied.

III. if (a) conditions of cases I-II do not hold, and (2) there is a potential dependence edge from $nn(\beta')$ ($dn(\beta')$) to $dn(\beta)$, and (c) $nn(\beta')$ ($dn(\beta')$) is reachable from the slicing criterion. Note that β' is in the slice, so:

- $nn(\beta)$ is not reachable (due to the conditions for cases I-II not being true) and $dn(\beta)$ is reachable from slicing criterion
- β is added into the relevant slice φ_* because of potential dependencies, and $\beta \in \varphi_i$ since condition (4) of Lemma A.11 is satisfied.

In all possible cases, (1) if $nn(\beta)$ is reachable from the slicing criterion in the EDDG, then β is added into the slice because of slicing criterion, or dynamic control/data dependencies, and (2) if $nn(\beta)$ is not reachable and $dn(\beta)$ is reachable from

the slicing criterion, then β is added into the slice because of potential dependencies. Consequently, $\beta \in \varphi_i \subseteq \varphi_*$. This completes the proof. $\square$