Abstract of “A Research Framework for Software-Fault Localization Tools” by Emmanuel Renieris, Ph.D., Brown University, May 2005

Once a program run exposes a bug, the programmers’ first task is fault localization: identifying the portions of code they need to change to correct the bug. Help for this task takes the form of fault-localization tools, which vary from static code analyzers to debuggers. This dissertation develops a research framework for fault-localization tools that require only a set of runs, some of which fail, to produce their output.

The first part of the framework is an architecture for fault-localization tools. The architecture prescribes five phases: data collection for each run, data abstraction for each run, modeling of sets of runs to produce models of correct and failing runs, contrasting the two models, and finally mapping the difference back to source code. This architecture also describes a number of existing tools.

The second part of the framework provides an evaluation method for fault-localization tools, based on the effort an ideal user must invest to interpret a tool’s results. This effort is measured as the proportion of code that lies between the report and the faulty locations on a graph representing the structure of source locations.

The dissertation presents and evaluates a number of fault-localization tools developed within this framework that are built around program profiles and collections of dynamically discovered invariants. Notable contributions include the nearest-neighbor idea, where failing runs are compared with their most similar successful run, and the discrete comparison and differencing of program profiles based on permutation distances. The combination of these two contributions achieved the best results of the tools we developed, as computed by the evaluation method.

Finally, this dissertation presents and quantifies elided events, a cause for caution in the interpretation of program profile data. The problem is that the execution of a part of code may have no effect on the run’s correctness. As a result, code that should be suspicious is considered safe, with adverse effects for debugging.

Abstract of “A Research Framework for Software-Fault Localization Tools” by Emmanuel Renieris, Ph.D., Brown University, May 2005

Once a program run exposes a bug, the programmers’ first task is fault localization: identifying the portions of code they need to change to correct the bug. Help for this task takes the form of fault-localization tools, which vary from static code analyzers to debuggers. This dissertation develops a research framework for fault-localization tools that require only a set of runs, some of which fail, to produce their output.

The first part of the framework is an architecture for fault-localization tools. The architecture prescribes five phases: data collection for each run, data abstraction for each run, modeling of sets of runs to produce models of correct and failing runs, contrasting the two models, and finally mapping the difference back to source code. This architecture also describes a number of existing tools.

The second part of the framework provides an evaluation method for fault-localization tools, based on the effort an ideal user must invest to interpret a tool’s results. This effort is measured as the proportion of code that lies between the report and the faulty locations on a graph representing the structure of source locations.

The dissertation presents and evaluates a number of fault-localization tools developed within this framework that are built around program profiles and collections of dynamically discovered invariants. Notable contributions include the nearest-neighbor idea, where failing runs are compared with their most similar successful run, and the discrete comparison and differencing of program profiles based on permutation distances. The combination of these two contributions achieved the best results of the tools we developed, as computed by the evaluation method.

Finally, this dissertation presents and quantifies elided events, a cause for caution in the interpretation of program profile data. The problem is that the execution of a part of code may have no effect on the run’s correctness. As a result, code that should be suspicious is considered safe, with adverse effects for debugging.

A Research Framework for Software-Fault Localization Tools

by

Emmanuel Renieris

Dipl. Engr., National Technical University of Athens, Greece, 1997

Sc. M., Brown University, 2000

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2005

This dissertation by Emmanuel Renieris is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ____
Steven P. Reiss, Director

Recommended to the Graduate Council

Date ______________ ____
Pascal Van Hentenryck, Reader

Date ______________ ____
Philip N. Klein, Reader

Approved by the Graduate Council

Date ______________ ____
Karen Newman
Dean of the Graduate School

Vita

Emmanuel (Manos) Renieris was born in 1973 in a small clinic on the same street as the National Technical University of Athens, Greece. He followed a roundabout way taking 18 years to enter that university. Five years later he graduated and went to Providence, Rhode Island, where the first impressive thing was the lack of mountains and how big the sky was. The second impressive thing was that cold, white, watery stuff falling from the sky, which he is still not used to.

Acknowledgments

I wish to thank the people who saw this dissertation through: my advisor, Steve Reiss, and my committee members, Pascal Van Hentenryck and Phil Klein.

I gratefully acknowledge the help of the people who worked with me on parts of this dissertation: Shriram Krishnamurthi, Brock Pytlik, Sébastien Chan-Tin, and Haley Allen. Trina Avery patiently edited many drafts.

I wish to thank all the people who picked me up when things seemed gloomy, principal among them Vaso Chatzi, Nikos Papadopoulos and Ioannis Tsochantaridis.

I need to thank my parents and brother who have been patient and supportive throughout this long absence.

And I want to thank my fiancee, editor, buttress, and love, Liz Marai. Multumesc.

Contents

⋆ Parts of this document have been published as [Renieris and Reiss, 2003; Pytlik et al., 2003; Renieris et al., 2004].

List of Tables

List of Figures

A Research Framework for Software-Fault Localization Tools

Emmanuel Renieris

May 18, 2005

Chapter 1

Introduction

This dissertation demonstrates the practicality of fault-localization tools when the available sources of information are only the program, a set of failing runs, and a set of successful runs. It introduces a research framework comprising two essential instruments for such a demonstration: an architecture that explains and prescribes the development of fault localization tools and a method to evaluate and compare such tools. In addition, this dissertation introduces a number of such tools and evaluates and compares them experimentally.

1.1 Debugging and Debugging Tools

A *faulty* or *buggy* program is one that, on some inputs, confounds the expectations of its programmer. We can say that the program does not conform to its specification, which need not be formal or even explicit. When a bug is exposed, the programmer must identify the part of the program to be changed so that the program meets expectations. This process is called *fault localization* and is often the hardest part of the *debugging* process, which also involves fixing the bug. Debugging tools aim to help the programmers locate and fix the bugs. Most debugging tools are actually fault-localization tools, since fixing the bugs remains firmly the responsibility of humans.

Probably the earliest debugging tools were the EDSAC debugging routines. The EDSAC (Electronic Delay Storage Automatic Computer), designed by Maurice Wilkes at Cambridge University and put into operation in May 1949, was one of the first stored-program computers. The first three programs run on the EDSAC worked flawlessly: they computed squares, primes, and squares and their differences. The fourth was a more ambitious venture: it was meant to calculate numerically the integral of a specific differential equation. Wilkes describes his early experience with this program in his memoirs [Wilkes, 1985]: "I was trying to get working my first non-trivial program . . . the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." Soon thereafter, Gill [1951] created a set of subroutines to help monitor the operation of EDSAC, including many that a modern programmer would recognize as essential parts of a debugger: breakpoints, watchpoints, single-stepping, memory dumps and even tracing under a virtual machine.

The purpose of both the EDSAC subroutines and modern debuggers is to help locate bugs, i.e. identify the portions of code that must change for the program to be correct. Debuggers help programmers by allowing them to examine the execution in detail until they discover a discrepancy between the program actions or the machine's state and their expectations about these actions and states.

Certain kinds of bugs appear again and again in different programs. Debugging programs with such bugs with a debugger is arduous; instead, for some of these bugs programmers can use tools and languages that provide prevention or quick detection mechanisms. An eminent example is type errors, accessing of data as a different type than it is. Type systems can prevent such accesses, either (if checked at compile time) by not allowing the program to execute at all or (if checked dynamically) by checking every access and flagging the erroneous ones. A second example is out-of-bounds array accesses, i.e. accessing memory past the end of an array as part of the array. Here, a tool can check every array access and ensure that the index used is within the array bounds.

If the programmers have inklings about bugs that are not covered in the language or available tools and they can express their suspicions in terms of the program's state at certain points, they can insert assertions at those points, i.e. code that at run time flags the execution of catastrophic operations rather than executing them.

If no language feature, no tool, no assertion catches a bug, the programmer must resort to debuggers and examine the execution closely. In recent years, a number of debugging tools have appeared in the literature [Agrawal et al., 1998; Chen and Cheung, 1997; Cleve and Zeller, 2005; Pan and Spafford, 1992; Reps et al., 1997; Whalley, 1994; Zeller, 2002; Zeller and Hildebrandt, 2002] that minimize programmer effort by letting them contrast successful and failing runs. A major idea here is that it is much easier for the programmer (or even the final user) to provide macroscopic evaluations of correctness ("was this run successful?") than microscopic evaluations ("is the machine state correct?"). This assumption is valid partly because microscopic evaluations require more information and partly because macroscopic evaluations require only knowledge of what the program is supposed to do, rather than how the program is designed to do it. These tools are commonly called *automated fault-localization* tools, since they require minimal input from the programmer. However, when there is no danger of confusion, such tools are also called simply *fault-localization tools.*

1.2 Philosophy and Overview

This dissertation focuses on automated fault-localization tools. The first contribution of this thesis, described in Chapter 3, is the exposition, in relational terms, of an architecture fault-localization tools can and often do follow. The basic idea of the architecture is not restricted to automated fault-localization tools. All the tools we have discussed (debuggers, language features, automated fault-localization tools) operate on the same principle. They employ a model of correctness for the program runs. When a particular run violates it, the how and the when of this violation can provide the programmer with clues on the location of the bug. For debugging tools, the model of correctness is in the programmer's head; for language features, it is in what the language allows ("a correct run accesses each piece of memory as one type only"). Assertions embody such decisions in their code ("the first argument must always be larger than zero"); for automated fault-localization tools, the correct model is discovered, automatically, in the correct runs. Chapter 2 discusses a number of debugging tools from the perspective of the correctness model they employ.

This idea of a correctness model is the foundation of Chapter 3. An automated fault-localization tool collects certain data from each program run, builds a model of correctness from the correct runs, contrasts it with the failing run, and maps the differences back to the source code to give the programmer a hint about the location of the bug. The latter part of Chapter 3 maps a number of research tools to this architecture.

The second contribution of this thesis is an evaluation method for debugging tools. Automated debugging tools often exploit some specific characteristic of the faulty program or its operating environment. As a result, they are applicable only in restricted domains, and as an unfortunate side effect there is no way to compare these tools to one another; indeed, such tools are often validated through case studies. I rectify this situation in Chapter 4, where I design a general evaluation function. The idea is that once a tool reports some portion of code as potentially faulty, we can assign a score to the report by observing not only the reported features but also their distance from the features that *should* be reported, i.e. the faulty features. The notion of distance can be obtained from structural relations between features, say the system dependence graph if our granularity of feature is basic blocks. If we define this distance, then the evaluation function corresponds to the number of features closer to the report than the actual fault. In the latter part of Chapter 4, I present an implementation of the evaluation function based on the program dependence graph and experiments that show its behavior over a number of example programs.

Armed with a general architecture and an evaluation function, I present in Chapter 5 five prototype fault-localization tools. I then evaluate them in Chapter 6. These tools are based on program profiles, i.e. counts of the executions of lines of code during a run, and potential invariants, i.e. predicates that hold true for a run but not necessarily for all runs of the program [Ernst et al., 2001]. These tools serve as models of more involved tools and highlight their strengths and weaknesses. Chapters 5 and 6 combined showcase a research framework for fault-localization tools, consisting of the general architecture of Chapter 3 and the evaluation method of Chapter 4.

Certain of the tools described in Chapter 5 are novel. In particular, although past research has contrasted runs with close inputs to highlight potential fault locations, I contrast runs that are as close as possible in the domain where the differencing takes place. This dispenses with the need for any external knowledge of the similarity of runs, e.g. by assuming knowledge of the program input's structure. Contrasting profiles directly, however, creates the need to compare counts of events in such a way that differences map cleanly and discretely to lines of code: I achieve this by observing that the ordering of lines by their execution counts contains much of the information in their relative execution frequencies, and then applying a permutation distance function. The combination of using close runs and permutation distances achieves the best results.

Using profiles for fault localization can be misleading. In particular, certain executions of portions of code may have no effect on the program's output. Chapter 7 explains and quantifies how this can happen and discusses the repercussions for any analysis that uses program profiles as its primary source of data.

Finally, Chapter 8 presents my conclusions.

Chapter 2

Related Work

In this chapter I discuss a number of debugging tools, sketching the debugging landscape and setting the background of this dissertation. Bugs have been a fact of life for programmers almost from the beginning of computing, and there is a multitude of tools to help them debug their programs. One can organize such tools according to many possible principles, for example along the historical axis, or according to whether the tools operate on program executions (dynamic tools) or solely on source code (static tools).

This dissertation takes a different and more pertinent approach. All debugging tools are based on the following idea: there is a model of correct program behavior, and the debugging effort seeks how a failing run violates the correctness model. Under this light, a major characteristic of a debugging tool is the provenance of the model. For example, the focus of this dissertation is on a relatively new family of tools that derive a correctness model from a number of correct runs. This discussion follows a classification of debugging tools stemming from the correctness model idea. The list of tools discussed here is in no way exhaustive; instead, it provides a sample of the variety of efforts on the debugging problem.

2.1 Definitions and Overview

In this document I use *bug* and *fault* to mean a deviation of the program's behavior from the desired behavior, and also the portion of the program responsible for the deviation. When necessary, I talk explicitly about the deviant behavior or *manifestation* of the bug.

Debugging is the process of discovering deviant behavior, finding its cause in the code, and fixing it so that the behavior is eliminated. The first step of the process, discovering deviant behavior, is usually done on a running program through software testing. However, in certain cases, it is possible to identify patterns of code that will result in deviant behaviors if executed, and then static tools (i.e., tools that examine only the source code) can be applied and isolate that code before run time. The second step in the debugging process, finding the cause of the deviant behavior in the code, is often called *fault localization*. Most debugging tools are actually fault localization tools: the third step of debugging, fixing the code, remains firmly a human endeavor.

Fault localization tools employ a model of correctness and contrast it with the faulty program or a particular run. Partial correctness models have emerged that formulate whole classes of bugs, and languages and tools have incorporated them to help the programmers deal with bugs from those classes. Examples of such tools are described in section 2.2.

Other classes of bugs are foreseen during the specification process, and the resulting specification incorporates correctness models that exclude those bugs. Tools can then allow the programmers to compare the specification with the code. Yet other bugs are anticipated during coding; for those, the user can include error-checking code around the functionality. Bugs that nobody anticipated, so that their existence in the program does not conflict with any correctness model attached to the program, are the most difficult to find. In such cases, programmers can use debuggers, which let them examine an execution at a very low level and isolate the failure-inducing code, perhaps refining their mental correctness model along the way. A slew of languages, interfaces, and extensions to debuggers allow higher-level examination of executions. I describe such tools in section 2.3.

Using a debugger or writing a specification can be a arduous process. Therefore, in recent years, a new family of tools has emerged that automatically builds a correctness model from program runs that the user has to label as correct. Such tools are described in section 2.4.

2.2 Correctness Models Provided in Tools and Languages

Programming languages already provide correctness models that exclude many faulty programs or runs. The preeminent example is type systems (for an overview, see e.g. [Pierce, 2002]), which among other things ensure that each memory location is accessed as a single data type throughout an execution. When implemented statically (e.g. in ML [Milner et al., 1997]), the compiler rejects code that might result in the violation of this property. When implemented dynamically (e.g. in Scheme [Kelsey et al., 1998]), every memory access is checked for type safety before it is executed. The correctness model prohibits all programs that may result in illegal accesses.

Formulations of correctness models are sometimes immature, expensive to check, or inaccurate, yet checking them can be effective. In those cases, embedding checks for those models in a language is inappropriate, and such checks are instead put into extralingual tools. As the tools mature, these checks can be put in the later versions of the language or the compiler.

2.2.1 Static Tools

Lint [Johnson, 1979] is a tool for the C programming language that warns about a number of static code properties such as unused variables, the use of uninitialized variables, disagreement between functions' formal and actual parameters (function prototype checking), and "strange constructions" (sic). Lint often flagged too many errors, but certain of its checks, such as function prototype checking, were important and were incorporated in the 1989 C language standard.

The standard's rationale states:

> The function prototype mechanism is one of the most useful additions to the C language. The feature, of course, has precedent in many of the Algol-derived languages of the past 25 years. The particular form adopted in the Standard is based in large part upon C++.
>
> Function prototypes provide a powerful translation-time error detection capability. In traditional C practice without prototypes, it is extremely difficult for the translator to detect errors (wrong number or type of arguments) in calls to functions declared in another source file. Detection of such errors has either occurred at runtime, or through the use of auxiliary software tools.

The "auxiliary software tools" include lint. Function prototypes are thus a way to specify a (partial) correctness model that found its way from a language definition (in Algol) to a extralingual tool (Lint) back into a language definition (C, via C++). The bugs addressed by the model are malformed function calls.

Lint's checking of function prototypes had a set of properties that made it particularly suitable for incorporation into the C compiler (no doubt due to its own linguistic heritage): it required only source code for its operation; when it raised a false alarm (flagged a correct function call as incorrect), other relatively simple language features (casts) could be used to silence it explicitly; it raised false alarms rarely; and the correct alarms it raised flagged particularly catastrophic function calls.

Other static tools do not necessarily have all these characteristics, but can still be extremely useful. PREfix [Bush et al., 2000] anticipates a class of illegal memory accesses, such as illegal pointer references, and examines a number of control paths to find potential such references. It does not catch all such references, nor does it guarantee that every reference it catches can actually happen at run time; still, it is reportedly a very useful tool [Larus et al., 2004].

2.2.2 Dynamic Tools

Unlike tools such as Lint and PREfix, which operate on the program's code, tools like Purify [Hastings and Joyce, 1991] and valgrind [Seward and Nethercote, 2005] operate during program execution, therefore requiring a compiled program and a set of test inputs. Then they can monitor all memory accesses and prevent any illegal memory accesses. The correctness model includes properties such as “no memory address shall be read before it is written”.

Such tools can also tackle a set of synchronization bugs, such as deadlocks and data races [Savage et al., 1997]. In fact, much of the work here is devoted to improving the correctness model so that it is easy to check, yet effective.

2.3 Correctness Models Provided by Programmers

Although many bugs are anticipated by language designers or tool writers, such bugs are by necessity “generic” bugs, in the sense that they apply to all programs, and therefore they cannot address correctness properties that should hold only for particular programs. For example, any program with a function call site where the actual arguments do not agree with the formal arguments can be flagged as problematic, but for a particular program it might be an error to call a function with all its arguments equal to zero.

When there is no tool that will check a correctness property that applies to a particular program, the description of correctness has to be provided by the programmer. The simplest way to do this, which requires virtually no tool support, is direct coding of assertions [Hoare, 2003] around sensitive pieces of code. Then the assertions are checked during program execution. With more effort, specifications can be written and checked statically with tools such as LClint [Evans et al., 1994] and Extended Static Checking [Detlefs et al., 1998]. With the advent of software model checking [Holzmann, 1997], a number of tools have appeared that check an abstraction of the program against a formula in temporal logic.

When a bug escapes all correctness models developed during language design, tool design, and program specification, but surfaces during testing or even deployment, the developers need to explore the failing execution with a debugger. Debuggers have a long history, starting with the EDSAC monitoring routines [Gill, 1951]. A debugger is a tool that allows the user to examine an execution closely by following the control flow step by step, examining the values of variables at any time, etc. Since debuggers provide only the means to examine a program execution, but no guidance for what to examine, they remain difficult tools to use; hence the addition of elaborate graphical interfaces [Zeller and Lütkehaus, 1996] and query languages [Linton, 1983; Ducassé, 1991; Ducassé, 1999; Lencevicius et al., 1997]. The idea in these languages is that users, after seeing a bug, will have new-found knowledge about the nature of the bug and therefore will be able to refine their mental model of correctness and construct valuable queries, perhaps akin to the assertions they would have written had they foreseen the bug.

Examining program runs is a slow, tedious task in which the user must step slowly through the execution and verify the correctness of each state. Ehud Shapiro's system [Shapiro, 1983] minimizes the states the user must look at and guides the users to the faulty function call by asking them to verify intermediate results. With binary search, the system minimizes the number of oracle queries. Shapiro's work assumes a stateless language.

Program slicing [Weiser, 1984; Tip, 1995] is based on the idea that the programmer knows of a variable which, at a specific point in the code, has the wrong value. The variable together with the point in the code is called the *slicing criterion.* Slicing finds the part (slice) of the program that affects the slicing criterion. Dynamic slicing [Agrawal and Horgan, 1990; Smith and Korel, 2000; Korel and Laski, 1990], the dynamic extension of slicing, assumes that the user can pinpoint a variable that has the wrong value at a particular program point, and also at a particular time during an execution. It then selects the code portion that had an effect on the variable at that program point during a particular run. Since users must provide the slicing criterion, they must first discover a variable with a wrong value, which presumes a significant amount of work on the user's part.

2.4 Correctness Models Discovered Automatically

A newer class of fault localization tools build the correctness model automatically. In certain cases the model can be discovered by examining existing programs' source, as in Engler et al.'s work [Engler et al., 2001]. The underlying assumption is that frequently occurring patterns in source code stem from implicit rules that encode good programming practices. Therefore, code locations that almost, but not exactly, conform to the patterns are possible sites of bugs. The necessary condition is that those programs are similar to the one we want to debug. This line of research has been successfully applied to, for example, device drivers, which all must obey a common set of rules, those of the operating system.

More common is the building of models from program runs. Ammons et al. [2002] discover rules about the interaction of programs and libraries dynamically. The idea is to define the boundary between programs and libraries and then observe how the programs use the library. Inconsistencies in this use can flag bugs. In further research Ammons et al. [2003] provide tools for improving the inferred specifications.

Contrasting program runs or portions of runs can provide intuition on the location of bugs. A key concept here is *coverage*. If a run executes a particular piece of code, then we say the portion of code was covered during the run.

Pan and Spafford [1992] present a set of ideas on how to contrast program slices with different slicing criteria from a run. The core idea is that, if a variable has the correct value during the run and a different variable has a wrong value, then the fault is in the slice of the second variable but outside the slice of the first. Chen and Cheung [1997] extend the idea to include differences between slices from different runs.

χVue [Agrawal et al., 1998] allows the user to contrast the coverage of programs during runs that execute different user-level features. The goal is to assist the programmer locate these features in the code for program maintenance. $\chi Slice$ [Agrawal et al., 1998] allows the comparison of program slices to facilitate fault localization, implementing some of the ideas in [Pan and Spafford, 1992].

Jones et al. [2002] contrast many failing and successful runs together and employ a SeeSoft-like visualization [Eick et al., 1992] to display a ranking of suspect portions of code. Portions of code that execute rarely in a correct run and frequently in a failing run have high ranking. The tool also displays a confidence judgment using different visual aspects on the same display.

Ball [1999] suggests comparing program runs via concept analysis [Ganter and Wille, 1999].

None of these tools address the problem of selecting the runs or slices to contrast. This problem is often addressed by manipulating program inputs to create two very similar inputs, one that causes the program to succeed and one that causes the program to fail. Assuming that similar inputs result in similar runs, programmers can then contrast the two runs to help locate the fault.

Whalley presents *vpoiso* [Boyd and Whalley, 1993; Whalley, 1994], a tool to localize bugs in a compiler. A bug is detected by comparing the output of a program compiled with the buggy compiler with one compiled with a "correct" compiler. To find bugs in the optimizer, a number of runs of the optimizer are contrasted at the level of minimal improving transformations of the compiler's internal data structures. If an improving transformation is skipped, the compiler will still produce correct, if less efficient, code. If the transformations are $t_1, t_2, \ldots, t_n$, *vpoiso* orders the set $\{\{t_1\}, \{t_1, t_2\}, \ldots, \{t_1, t_2, \ldots, t_n\}\}$, under the usual subset relation. Then, for every such set, it allows the included tranformations to execute and checks the result. *vpoiso* performs a binary search on the set of sets, isolating a fault inducing set $\{t_1, t_2, \ldots, t_m\}$ such that $\{t_1, t_2, \ \ldots, \ t_{m-1}\}$ is not faulty. *vpoiso* assumes the transformations independent, therefore it blames t_m. For non-optimizing bugs, *vpoiso* does not localize the bug, although it isolates a minimal input for which the compiler fails. It orders (arbitrarily) the functions $\{f_1, f_2, \ldots, f_n\}$ of the subject program, and considers the subsets $\{f_1\}$, $\{f_1, f_2\}$, $\ldots$, $\{f_1, f_2, \ldots, f_n\}$. For each one of them, it compiles the set's members with the suspect compiler, and the other functions of the subject program with a trusted compiler. Similar to the optimizing case, a function set $\{f_1, f_2, \ldots, f_m\}$ is isolated such that compiling all of its functions with the suspect compiler reveals the fault, while compiling only $\{f_1, f_2, \ldots, f_{m-1}\}$ with the suspect compiler does not. Compilation of f_m is blamed. Whalley's techniques work under strict rules: suffixes of the sequence of improving transformations can be turned off without jeopardizing the program's correctness, and an error in compiling one function cannot be masked by wrongly compiling another one. Essentially the set of prefixes of $t_1, t_2, \ldots, t_n$ must map to to failure or success monotonically: if a prefix fails, all its longer prefixes fail, and if it succeeds, all longer prefixes succeed.

Zeller and Hildebrandt [2002] extended these minimizing techniques to handle cases where there is no monotonicity, while giving weaker guarantees on the minimality of the input. In particular, Zeller's algorithm *Delta Debugging* is a greedy algorithm that examines potentially a quadratic number of input subsets to find an *1-minimal change* in the input. A 1-minimal change is a change of size one that turns a non-failure-inducing input into a failure inducing input. Delta Debugging is cleverly tuned to make full use of Whalley's assumptions if they hold. The first application of the technique on input was on the input of a web browser. Further applications [Choi and Zeller, 2002; Zeller, 1999] include other debugging domains, such as finding the difference between a thread schedule that makes a multithreaded program fail and one that does not, and isolating the change in a multi-version system that causes the program to fail.

Reps et al. [1997] present DynaDiff, a tool that isolates faults at the level of acyclic, intraprocedural control paths. They target business programs they suspect of the Y2K bug and run them twice, once with the system clock set to the end of 1999 and once with the clock set to the beginning of 2000. Then they inspect the program coverage from both runs, trying to find paths that were not covered in the former case but were in the later. The idea is that paths covered only after the crucial date are suspect.

Hangal and Lam [2002] present Diduce, a system that continuously computes a set of predicates about a program run and reports when any of those predicates do not hold anymore. The predicates record which bits in the program state have remained constants, and which have received both possible values. This allows them to discover constancy, upper bounds, and the sign of numerical values. Diduce uses a simple set of predicates similar to Ernst et al.'s potential invariants [Ernst et al., 2001]. A potential invariant is a predicate on a number of program variables that may hold during an execution. Executing the program on an input will show that some invariants are not true; after more runs, the set of remaining invariants is closer to the set of true invariants. If those runs are correct, then the set of remaining invariants can serve as a model of correctness.

Many bugs are only discovered after deployment; as a result, the program inputs are not directly available to the programmer, and data collection resources are not fully available to the user who discovers the bug. A new generation of tools addresses this problem, by instrumenting programs under the constraint that they will run off-site, and implementing statistical analysis on samples of the program runs [Orso et al., 2002; Liblit et al., 2003].

Most of the tools I have described in this section do not interfere with the program's state. Programmers working with the debugger can often change the program's state, or even manipulate the program itself while it is running. A small number of tools help or automate this process. Critical slicing [Agrawal et al., 1993; DeMillo et al., 1996], by DeMillo et al., starts with a slicing criterion and then deletes lines of code in the slice, re-executes the remaining slice, and observes if the value of a variable remains unchanged. Zeller's more recent work [Zeller, 2002; Cleve and Zeller, 2005] copies values from a successful run to a failing run to isolate the part of the state that is relevant to the failure. These techniques can be viewed as generating arbitrary runs of the program to contrast with failing runs. There arbitrary runs do not obey the semantics of the program, but they have the advantage of being very close to the failing run, so that if they do not themselves fail, they can help pinpoint the fault.

Differencing techiques are not applied only to actual program runs. One of the advantages of software model checking is that, when the program violates a specified property, model checking tools provide the user with a counterexample trace; i.e., a sequence of abstract program states that, if part of an execution, would violate the property. However, such traces can be long and unwieldy, and existing work in model checking [Ball et al., 2003; Groce and Visser, 2003] creates explanations of specification violations by differencing successful and failing traces. Although the approaches are similar to the one I describe in this document, the overall process is different since the errors are against a specification, and the runs are over abstractions of the program.

Most of the tools described in this chapter have been validated through case studies. A direct comparison of these tools is hindered by the fact that they collect different data from running programs, use different models, and different techniques to contrast them with failing runs. In the next two chapters, I present an architecture encompassing many of the tools that build a correctness model from correct runs of the program, and an evaluation technique that allows comparing them.

Chapter 3

Architecture

In this chapter I describe an architecture for automated fault localization systems. The major contribution is a decomposition of such systems into discernible parts, facilitating the design, implementation and study of fault-localization systems. The key separation is the classical one between data and operations.

In essence, the architecture models a particular kind of fault localization system that collects information about correct runs, builds an abstraction representing a "generic" correct run, contrasts this with the information collected about a failing run, and maps the difference to the source code to isolate the possible source of the bug. Therefore, the architecture describes a specific process consisting of specific components with a specific data flow among them. In addition, the architecture prescribes this process, resulting in a blueprint for future systems and allowing us to separate and study independently the various parts of the process and their effect on the performance of the overall system.

In the past few years a number of research debugging tools have appeared that demonstrate the efficacy of automated fault localization. Many of those tools follow the process described in this architecture. On the other hand, these tools draw much of their success from exploiting knowledge about their subject (i.e. faulty) programs or their operating environments. Such knowledge might include that the subject program is a Y2K-sensitive application, that the program is an optimizing compiler with (ideally) independent phases, or that a multitude of failing runs is available. The presentation of these systems often focuses on this knowledge, and it is this focus that hinders comparing, extending, and improving these tools. The architecture described in this chapter aims to alleviate this problem. In addition, elements of the architecture are reused in an evaluation method that is the topic of the next chapter.

In this chapter and elsewhere, we often assume that only a single failing run is available. There are multiple reasons to make that assumption. First, a single failing run may be all that is available. Second, the architecture needs only minor changes in the last phase if we have many failing runs available. Most importantly, two failures caused by different bugs can confuse the debugging process, since we would be searching for two bugs at the same time, each potentially eliding the other. At the same time, a single failing run is likely to fail because of a single bug rather than the interaction of multiple bugs. The underlying issue is known as *fault coupling*, and its low probability of fault coupling has been proven for restricted set of programs [How Tai Wah, 2000]. The intuition behind the low probability of fault coupling is fairly simple: if we already assume that bug manifestations are rare (in other words, many more runs succeed than fail), then a bug manifests itself during runs that meet a restrictive set of conditions. Two bugs manifesting themselves in the same run would imply that the run lies in the intersection of *two* such restrictive sets, a far more restrictive and therefore less likely set.

3.1 An Example

The architecture is built around two basic data types: *spectra*, the internal representations of program runs, and *models*, the internal representations of summaries of program runs. This section aims to provide a grounding and intuition about these terms and the process encompassed by the architecture by applying the process to debugging a simple function.

The function we are going to debug is the `triangleType` function of Figure 3.1. Figure 3.1 also contains the driver function that reads the numbers from a user, passes them to `triangleType` and prints `triangleType`'s return value.

```
                                                4,4,4 4,5,5       3,4,5
 1  triangleType(float a, float b, float c)
 2  {
 3    if (a == b and b == c)                      ●     ●     ●     ●
 4      return Equilateral;                       ●           ●
 5    if (a == b or b == c or a == c)                   ●     ●     ●
 6      return Isosceles;                               ●     ●
 7    return Isosceles; /* Should be Scalene */                     ●     ●
 8  }
 9  main()
10  {
11    read(a,b,c);                                ●     ●     ●     ●
12    tt = triangleType(a,b,c);                   ●     ●     ●     ●
13    print tt;                                   ●     ●     ●     ●
14  }
```

Figure 3.1: Simple fault localization

The function takes three numeric arguments representing the lengths of the sides of a triangle and is supposed to return the type of the triangle: Equilateral, Isosceles, or Scalene. However, the function has an error: it returns Isosceles when it should return Scalene.

If we run the program with input $(4, 4, 4)$, it correctly returns Equilateral. In the process, it executes lines 3,4,11,12, and 13. The first column of the right part of Figure 3.1 shows this fact by including a dot next to each line of code that executes for the $(4, 4, 4)$ input, shown as the header of the column.

The program also produces the correct result (Isosceles) on the input $(4, 5, 5)$. We mark this column 2 of the right part of the figure.

For input $(3, 4, 5)$ the program fails, returning Isosceles instead of Scalene. Column 4 shows the lines executed for that input.

Having executed the program on inputs $(4, 4, 4)$, $(4, 5, 5)$, $(3, 4, 5)$, we discover at the last input that the program has a bug. We can then use the information collected during the correct runs to find where the bug is not, and contrast it with the information from the failing run to find where the bug is. Column 3 summarizes the correct executions by showing a dot for every line of code that executed in any correct run, i.e. showing a dot if there was is a dot in the same row of columns 1 or 2.

Column 5 contains a dot for each line that executed in no correct run but did execute in the failing run, i.e. a dot in every row that has a dot in column 4 but not in 3. The only such line is line 7, which is the faulty line; we have therefore found the bug.

3.2 Discussion

Finding the bug in `triangleType` involved a few steps that are special cases of the process discussed here. First, we collected a set of inputs on which the program succeeds. We ran the program on these inputs and collected information about the inner workings of the each run, as shown in the first two columns of Figure 3.1. We did not keep all the information from the runs, but rather an abstraction; this case we kept only the set of executed lines.

We call the information kept from a run the *spectrum*, a term spectrum introduced in [Reps et al., 1997] to mean the set of intraprocedural acyclic paths executed in the program run. We also collected the same information from a failing run. We call this spectrum the *failing spectrum* and the spectra from successful runs *successful spectra*.

We then summarized the correct runs into a *model* of successful runs. The model is a generic description of the correct runs; it summarizes what we know about them, possibly in a lossy way. In this example the model contains only information about the lines executed in any correct run, but no information about how these lines may interact or even be mutually exclusive.

At the next step we compute the *difference* between the model of success and the failing spectrum. This is the crucial step yielding the information that can help us locate the bug.

In our example, the last step is implicit: we need to *map* the difference to the source code in order to indicate where the bug is. Since our example spectra, model, and difference are all expressed in terms of lines of code, the last step is simply an identity function. In the general case though, a more elaborate computation may be necessary.

The mapping operation is a severe requirement since it implies that the difference can be mapped to the source, effectively excluding certain classes of spectra. For example, it could be that our spectrum includes only the duration of the execution, our model computes the average over all successful runs, and the difference is the quotient of the failing spectrum value over the model value. This would be a perfectly valid definition for spectra, models, and differences only if there is some way to map the quotient to the source code. One could use an arbitrary mapping from ratios to source code, but the quality of the tool would suffer.

3.3 Formalization

In our architecture discussion we discuss a number of artifacts: programs, source locations, events, program runs, spectra, models, and differences. We use sets and relations as our (only) constructs.

The programmer trying to debug a system only cares about a single program at a time. The interactions among different processes might be the root of the problem, but then we can view the system of processes as the program. As a result, we need not address multiple programs in our architecture. At this stage, we also do not care about the static interconnections among portions of the program. Therefore, we can view the program as an single unstructured set L of atomic *source locations* or simply *locations*:

$$\mathrm{L} = l_1, l_2, \ldots, l_n$$

In our example, the set of locations is the set of executable lines of code. Alternatively, the locations could be the functions of a program, particular variables, pairs of lines where one sets and the other uses the same variable, etc.

Once we have defined the set L, the fault-localization problem becomes the problem of selecting the faulty source locations. The fault localizer takes the whole program as an argument: in other words, the type of the fault localizer is

$$\mathrm{FaultLocalizer} : \mathrm{L} \rightarrow \mathrm{L}$$

i.e., a fault localizer is a function from the location set to the location set.

In order to talk about a program run, we need to introduce a notion of time. For our purposes, we can view time as the set of integers $\mathbf{N}$, although any given run will last only for a subset $[0..T]$, for some T.

We model a single execution of the program as a sequence of atomic *events* from an event set E. So as to allow multiple events to happen at the same time (for example, if the program is multithreaded or if events can overlap) and times when no event happens, we model a run as a relation between events and time: $r \subseteq \mathrm{E} \times \mathbf{N}$. We call the set of all runs R. R potentially includes all the relations between events and time, but certain such relations will not represent a run; thus R is a subset of the set of all relations bewteen events and runs:

$$\mathrm{R} \subseteq \mathcal{P}(\mathrm{E} \times \mathbf{N})$$

($\mathcal{P}$ is the powerset operator.) In our example, an event corresponds to the execution of a single line of code. In this case, it is also trivial to define the correspondence between events and locations: it is simply the identity relation. In general, though this need not be the case, and we can provide a MapBack relation that takes us from events to locations:

$$\mathrm{MapBack} \subset \mathrm{E} \times \mathrm{L}$$

We use a strict subset because a mapping that maps all events to all locations is not useful.

Having described in relational terms the basic data a fault-localization system works with, we can now describe its actual operation. The user of a faulty system needs to provide the debugging system with the set $\mathrm{A} \subseteq \mathrm{R}$ of runs. Not all runs need to be available; in fact, most runs probably are not. The user also needs to provide a single bit of information for each run that classifies the run as successful or failed. In other words, the user provides a partition of A into successful (or correct) runs C and failing runs F:

$$\mathrm{F} \cap \mathrm{C} = \emptyset, \mathrm{F} \cup \mathrm{C} = \mathrm{A}$$

What we have defined so far is dependent on the program (L), the program and the debugging system (E, R), the user's expectations, and whether they are met during a particular run (C, F). We have not addressed at all the inner workings of the fault localization system. We do so through two relations: the first, *abstraction*, maps runs to *spectra* from a set S. A spectrum is the result of processing a single run:

$$\text{Abstract} : R \rightarrow S$$

To keep the relational description general, we leave spectra as uninterpreted relations, but familiar examples of spectra include profiles (the results of the aggregational relational query $a \mapsto \mathit{count}(a, .)$) or covers (the results of the query $(a, .) \mapsto a$ over the run relation). For example, a coverage spectrum is a set of the first projection of the pairs of the $\mathrm{E} \times \mathbf{N}$ relation. A profiling spectrum is a relation $\mathrm{E} \times \mathbf{N}$ when the second item of each tuple is the number of occurrences of the first item in the original relation.

The second relation summarizes a set of spectra into *models*. We call the set of possible models M and again, for generality, we leave models as uninterpreted relations. Models encapsulate sets of runs, for example the set C or the set F. The modeling operation maps a set of spectra to the corresponding model:

$$\text{Modeling} : \mathcal{P}S \rightarrow M$$

Sometimes is it useful to let the model of a set of spectra (say, the set of successful spectra) to depend on another set of spectra (say, a set of failing spectra):

$$\text{Modeling}_2 : \mathcal{P}S \times \mathcal{P}S \rightarrow M$$

The Modeling relation can be seen as a special case of the Modeling2 relation with the second spectrum-set empty.

The essential requirement on models is that they can be contrasted to produce a set of interesting events:

$$\text{Contrast} : (M \times M) \rightarrow \mathcal{P}\,\mathrm{E}$$

Then we need to utilize the MapBack relation to gain a set of candidate faulty locations.

If we have available only a single failing run, then a model must be contrasted with a single spectrum:

$$\text{Contrast}_1 : (M \times S) \rightarrow \mathcal{P}\,\text{E}$$

The fault localizer is the composition of the described relations:

$$\text{FaultLocalizer} : \text{F}, \text{C} \mapsto$$
$$\text{MapBack}(\text{Contrast}(\text{Modeling}(\text{Abstract}(\text{F})), \text{Modeling}(\text{Abstract}(\text{C}))))$$

or for a single failing run:

$$\text{FaultLocalizer}_1 : \text{F}, \text{C} \mapsto$$
$$\text{MapBack}(\text{Contrast}(\text{Abstract}(\text{F}), \text{Modeling}(\text{Abstract}(\text{C}))))$$

3.4 Mapping Existing Tools to the Framework

A number of existing tools can be described within this architecture. I begin with five tools described in detail in Chapter 5. The first four of them are collectively called the Whither tools, while the last one is called Carrot. The latter section maps a set of recent research tools to the architecture.

3.4.1 The Whither and Carrot Tools

The first tool, on which the triangle example is based, uses the basic blocks (for our purposes here, equivalent to lines of code) as the set of program locations. The set of events corresponds to the execution of a line of code. The set R corresponds to all sequences of executed lines of code. The MapBack relation maps the event of the execution of line A to line A. The spectrum is a cover, meaning that the Abstract relation maps a run to the set of events that corresponds to the set of lines executed during the run. The Modeling relation is defined as the union operation on sets. The Contrast relation is the set difference operation. The tool highlights code that executes only in the failing runs.

The second tool is identical to the first one except that the Modeling relation is the intersection operation on sets and the Contrast relation is the difference operation with the arguments swapped. This tool highlights code that executes all successful runs but not in the failing runs.

The third tool is also identical to the first, except that the Modeling relation selects a single run to contrast with the failing run. The selection process utilizes a distance function between runs and selects the correct run closest to the failing run. Therefore, this tool uses a Modeling$_2$ relation that highlights code that executes in a failing run but not in a successful run.

The fourth tool is similar to the third tool with respect to the sets $L, \mathrm{E}, \mathrm{R}$. The Abstract relation, however, comprises many steps:

1. It produces a profile, i.e. it counts how many times each event occurs during the run,

2. It sorts the events according to their execution frequency during the run,

3. It discards the frequencies.

The spectrum for this tool is a ranking of all events according to their frequency. The model, as in the third tool, selects a single successful run according to a distance function defined on the event rankings. The Contrast relation is also defined over event rankings, but returns events; details about this tool are given in Chapter 5.

The fifth tool is significantly different. It is based on the notion of potential invariants [Ernst et al., 2001]. The set of locations is the set of entries and exits of functions. The events are function calls and returns, augmented with the values of parameters and local variables. The Abstract relation first confounds all executed events that refer to the same location together and then, for each location, it computes the predicates (from a predicate universe) that hold for all events referring to the location. The set of holding predicates for each location forms the spectrum of the run. Modeling a set of spectra entails computing which predicates exist in all spectra. Contrasting finds the predicates broken in the failing run. Mapping back finds the locations to which the broken predicates correspond.

3.4.2 Other Research Tools

Pan and Spafford [1992] present a set of tools similar to the first and second Whither tool. The significant difference is that instead of program runs, they use slices from the same run, computed according to different slicing criteria. The propose a number of heuristics for processing these slices, including the ones in the first and second Whither tools. χVue works similarly.

Tarantula [Jones et al., 2002] is similar to the first Whither tools. For each line of code x they compute the ratio

$$\frac{\frac{|C(x)|}{|C|}}{\frac{|C(x)|}{|C|} + \frac{|F(x)|}{|F|}}$$

where $|C|$ is the number of correct runs, $|F|$ is the number of failing runs, $|C(x)|$ is the number of correct runs that executed x, and $|F(x)|$ is the number of failing runs that executed x. They also compute the quantity max $\left(\frac{|C(x)|}{|C|}, \frac{|F(x)|}{|F|}\right)$. Then they use a Seesoft [Eick et al., 1992] visualization that shows the source code compactly, and color each line according to the first ratio for the hue component and the second ratio for the brightness component. It is then up to the user to interpret this information. This technique maps as follows to the architecture: The location set, event set, run set, abstraction and spectra are as in the first Whither tool. The modeling process for a set of runs $|R|$ computes the relation $\left(x, \frac{|R(x)|}{|R|}\right)$. The differencing process first computes the two ratios as above. The user's role is part of the differencing process: it is to set implicit thresholds for the hue and brightness ratios, and select only those lines of code where the two ratios exceed the thresholds.

DynaDiff [Reps et al., 1997] is similar to the third Whither tool. Only two runs are compared. Its set of locations is the intraprocedural acyclic paths of the program, an event corresponds to the execution of such a path, and the run is the sequence of executed events. The abstraction process results in spectra that correspond to the set of events that happened during the run. The differencing is directly on the spectra; the modelling process is simply an identity function. The key in this technique is the careful selection of program run to compare to the (potentially) failing run.

vpoiso [Boyd and Whalley, 1993; Whalley, 1994] is similar to the third Whither tool. Eventually, only two runs are compared. Unlike DynaDiff, *vpoiso* does not require that the user select the runs; instead, it constructs a (correct run, failing run) pair to contrasts them. The set of locations for *vpoiso* is the set of transformations;

an event is the execution of a tranformation, a run the sequence of events, and a spectrum the events executed. The tools searches for a run that fails while the minimally shorter run does not. If all of those runs were available from the beginning rather than constructed on demand, *vpoiso* would be performing a nearest-neighbor computation to find the two closest runs one of which succeeds while the other fails.

Some applications of Delta Debugging [Zeller and Hildebrandt, 2002] proceed in a similar way. In the debugging of gcc, the event set corresponds to optimization phases and each run corresponds to a different sets of optimization phases. The runs are constructed externally, by manipulating the compilers flags. Newer applications of Delta Debugging [Zeller, 2002] generate a set of "runs" by grafting parts of the state of a correct run onto the failing one. These are not runs of the program since they can be impossible; i.e., some of the states Delta Debugging examines may not be possible in any run without such grafting. These tools report chains of events and therefore they cannot be said to localize the bugs. The newest application of Delta Debugging [Cleve and Zeller, 2005], on the other hand, reports specific lines of code and therefore fits the architecture better.

Diduce [Hangal and Lam, 2002] is similar to the Carrot tool. The set of source locations is the set of write operations in the program code. Its event set is the set of writes to variables, augmented with the new value (from which we can find the old value for the next write). A run is a sequence of writes. The spectrum is an abstraction over the bits of values: which bits were constant (and which constant), and which bits took both values. The model of a set of runs is the predicates over bits that were always true. The difference is the predicates that are not true in the model of failing runs (which can be a single run) and true for the model of successful runs.

Chapter 4

Evaluation Method

In this chapter I present a quantitative measure of the quality of a fault-localization report. Such a measure is useful in assessing of single fault-localization tools; it becomes indispensable in comparing such tools. The measure imitates an ideal user who systematically explores the program text, starting from the report until encountering a faulty statement. The proportion of code such a user would *not* have to explore is the score of the report. The measure is easy to apply and encourages small, accurate reports without being overly strict. It has already been adopted by other researchers [Groce, 2004; Cleve and Zeller, 2005].

4.1 Motivation

The ultimate judgment on a fault-localization system's quality lies with its users. Therefore, the ideal evaluation of a fault localizer has to rely on experiments with human subjects. Such an approach has several drawbacks, at least during the development phase.

First, human-subject experiments are expensive, since they require training the subjects on an elaborate tool. In addition, the tool needs to be finished, e.g. it must be stable and have a polished user interface. Such systems are hard to develop, and in prototype form the effort to develop the user interface can well exceed that to develop the core tool.

More importantly, comparing tools on the basis of human experiments is error-prone. Human experiments are hard to replicate, since the results need to be calibrated for different sets of users, and the quality of the tool's user interface can strongly affect the tool's evaluation, so that we learn less about the tool's fundamental principles than about its implementation.

Last, one of the higher-level goals of building fault-localization tools is their potential to shed light on what constitutes an interesting feature of a program. In general, it is difficult to define an interesting program run, much more to decide which parts of the program are responsible for the run being interesting. But when a program mostly succeeds, then its failing runs are de facto interesting and there is an identifiable code portion (the bug) responsible for the interesting behavior. A desired side effect of building fault localizers will be discovering and understanding these code portions. If we evaluate fault-localization tools only with user experiments, then the user experience becomes our only goal and this valuable side effect is lost.

In fact, many of these reasons apply in other areas in computer science, notably information retrieval and natural language processing (NLP). The NLP textbook by Manning and Schütze [1999] points out that:

> An important recent development in NLP has been the use of much more rigorous standards for the evaluation of NLP systems. It is generally agreed that the ultimate demonstration of success is showing improved performance at an application task, be that spelling correction, summarizing job advertisements, or whatever. Nevertheless, while developing systems, it is often convenient to assess components of the system, on some artificial performance score (such as perplexity), improvements in which one can expect to be reflected in better performance for the whole system on an application task.

This chapter presents the first (to my knowledge), widely applicable evaluation measure for fault-localization systems. Given a program, the locations of the fault in the program, and the report of a fault localizer, the measure assigns a score to the report. The next section describes an example application of the method.

```
if (c1) {
  y = 9;
  z = 10;
if (c2)
  x = z;
else
  x = y;
}
```

Figure 4.1: A simple program

4.2 An Example

In this section, I present an example application of the measure to a simple fault-localization report. We follow the actions of an ideal user who explores the program starting from the report and proceeds in a methodical way until the faulty code is found. The score of the report is the percentage of the program that our imaginary user does not need to visit.

Figure 4.1 shows a short program fragment. Suppose the report consists of the first line of code, i.e. the condition `c1`. Our ideal user starts exploring the program from the first line. If the first line happens to be faulty, the ideal user stops there and focuses on fixing the fault. The score of the report would then be $1 - 1/6 = 5/6$, since the user does not need to examine five sixths of the code. This is the best attainable score.

If the first line is not faulty, the user must choose which line to explore next. Multiple possibilities exist, of course: the user could examine the next line in the program text, skip the if statement entirely (and therefore the whole fragment), etc. Instead of some other arbitrary choice, we equip our user with a venerable tool of program analysis, the program dependence graph (PDG) [Ferrante et al., 1987].

The program dependence graph contains a node for each maximal expression (including assignments) in the program. It contains an *control edge* from an expression A to an expression B if expression A affects whether statement B is evaluated or not, and a *data edge* from A to B if the value computed by A potentially affects the value computed by B. Figure 4.2 shows the program dependence graph for the program of

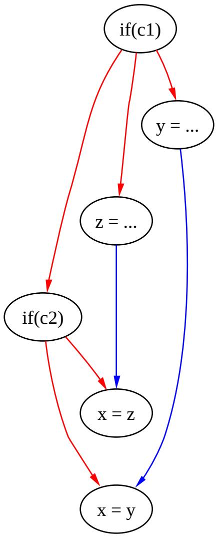


Figure 4.2: The program dependence graph of the program of Figure 4.1

Figure 4.1.

The advantage of using the PDG rather than, say, the text order is that certain arbitrary artifacts of the text order do not appear in the PDG. For example, since lines 2 and 3 of the program in Figure 4.1 have no dependencies on each other, the order in which they appear in the text is arbitrary; the semantics of the program would not change if the two lines were exchanged. This fact is reflected in the PDG by the absence of any edges between the two nodes representing those lines of code.

Given a report as a set of nodes on the PDG, we can split the PDG into layers of nodes of equal distance from the report. Layer n is defined as the nodes that can be reached by a path of length at most n from the starting nodes of the exploration, i.e. the report. The example program has three layers, when starting from the `c1` node: layer 0 contains only the `c1` node, layer 1 contains the assignments to `z` and `y` as well as the second conditional, and layer 2 contains the assignments to `x`. Figure 4.3 shows the layers of the example PDG.

Armed with the PDG, our user explores it, layer by layer, starting from the report. The user examines a whole layer at a time, stopping at the layer that contains a faulty node. In the example program, if one of the assignments to `z` or `y` or the condition `c2` is faulty, the user stops at layer 1. In that case, the score of the report is 2/6, as the

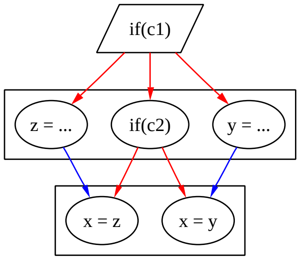


Figure 4.3: The program dependence graph of the program of Figure 4.1, split into layers

user avoids examining only two nodes. Otherwise, if the fault is in the assignments to `x`, the user stops at layer 2, having examined the whole graph, and therefore the score of the report is 0.

This example hints at a few characteristics of the measure. The method encourages small, accurate reports. Its range is $[0, 1)$. A small report which finds the bug gets a larger score (closer to 1) than a large report which finds the bug. A report that finds the bug gets a higher score than a report that does. The measure is applicable under the following assumptions:

- the bug and the fault localizer report are syntactic
- the syntactic description of the bug is known

Furthermore, for the measure to give interesting results, a third assumption is necessary:

- the bug is small

The first assumption is necessary because the measure relies on a syntactic comparison of bug and report. The second assumption limits the applicability of the evaluation measure to cases we already have fixed the bugs; such a requirement is acceptable at the evaluation stage of the fault localization tool. The third assumption states that given a large syntactic fault that would encompass most of the program, the measure does not provide a large range of scores.

4.2.1 Precision and Recall

In this section, I discuss the framework upon which the measure is based. In information retrieval and natural language processing, success is measured by a combination of two measures: precision and recall. Indeed, the adoption of these measures by the natural language community has goals similar to ours. Our discussion of the precision/recall framework is based on Manning and Schütze [1999].

Precision and recall apply to systems whose set of possible answers is the power set of a universe. In other words, the system's answers are sets; the answers are subsets of a known finite set, the universe; elements of the universe can appear in a set independently of other elements. Precision and recall also require that a perfect answer is known.

Precision and recall can be described in terms of the error types in statistical decision theory. Type-I errors are the errors in which an element of the perfect answer does not appear in the system's answer, and type-II errors are those in which the system's answer contains elements that are not in the perfect answer. Let us call A the system's answer and C the perfect (or correct) answer and let us split the elements of the universe into four sets, depending on whether they appear in the perfect answer and whether they appear in the system's answer:

	Perfect (Correct) Answer C	
System Answer A	present	absent
present	in A and in C	In A but not in C (Type II error)
absent	not in A, though in C (Type I error)	not in A, not in C

The recall score of A, given by $R(A) = (A \cap C)/C$, measures how many type-I errors it contains. The precision score of A, given by $P(A) = (C - A)/A$, measures how many type-II errors it contains.

Each of these measures assigns excellent scores to many degenerate cases. For example, a report containing all the elements of the universe has excellent recall, while an empty report has excellent precision. For this reason, precision and recall are usually combined into a single measure, for example the F-measure: $2PR/(R+P)$.

The precision/recall framework incorporates a notion of similarity of reports to the perfect report. With respect to precision, a perfect report contains only the elements of C; removing elements of C does not affect the perfection of the report, but adding elements from $U-C$ does. With respect to recall, a perfect report is one that includes all of C; subtracting elements from that perfect report takes us further from a perfect score, but adding elements does not.

4.2.2 The Measure

While it is possible to evaluate fault localizers using a precision/recall framework, such an evaluation does not take into account the structure of programs. For example, consider the an if statement with two branches: `if(c) then A else B`. Suppose that the condition is incorrect so that for a particular input, the program executes `A` rather than `B`. A fault-localization report that "blames" the conditional has high recall. At the same time, a report that also includes `A` would get a lower precision, and a report that includes only `A` would get low precision and recall. We definitely want the first case (the report that includes the conditional) to receive a high score; however, we would also like the reports that include `A` to receive a high score, since even though they are not exactly where the bug is, they are "close". On the other hand, if the bug is in `A`, but the report includes only `B`, we want the score to be low: the report points at exactly the case where the code is correct.

We would like a measure that embeds a suitable notion of closeness to perfect reports. From our analysis of the precision/recall framework, we can discern several needs for our measure:

- A universe of answers
- A set of perfect reports
- A distance metric that maps an answer to a number

The universe of answers depends on the fault localizer; the perfect report depends on the bug.

It remains to define the distance between reports. Let us recall our desiderata: small reports are better than large reports, reports closer the bug are better than reports further from the bug. We base our distance on a basic distance between the syntactic elements of the program. Syntactic elements of program are typically organized in graphs. In such graphs, the nodes represent syntactic elements, e.g. statements or functions, and the edges represent relations such as control dependencies, data dependencies, or potential function calls. The current implementation of the measure uses the system dependence graph (SDG), which is the interprocedural extension of the PDG, with edges linking actual and formal arguments. However, one could apply a similar process with a call graph, where each node is a function and each edge a potential function call.

We want our evaluation metric to reward tools for reporting few expressions, as close as possible to the faulty one. A bug typically encompasses more than one minimal expression. We do not require that all nodes be included; one suffices.

We now focus on our imperfect reports. The easy case is when the faulty node is identified. In that case, we know that the report is accurate and we penalize only for size.

The harder cases occur when the report does not include a faulty node. Because of program structure, such reports are not always useless. For example, in an if statement, a report might say that the fault is in the condition when it is actually in the body. In other words, the tool might say that the body should not execute. If the body is wrong, then *this* body should not execute, which is helpful and "almost accurate".

This leads to the following idea: if, moving in the SDG from the report, we quickly encounter a faulty node, then we want this report to have a high score. Such a simple scheme contains a loophole for tools: all they need to do is report a central node, that is, a node close to every node. A central node will be close to a faulty node, and the report will get an undeserved high score. We would therefore like to penalize reports that include central nodes; on the other hand, we want to do so only in moderation, since central nodes may actually be faulty.

To solve this problem, we penalize the nodes in the report for every node they are closer to than the faulty nodes. We do this by the following process: We start from the reported nodes and moving in a breadth first manner, we mark all the visited nodes.

We mark each node in the graph with its distance from any node in the report. Then we find the minimum rank of all the faulty nodes. Then we split all the nodes in the graph into three classes:

- those closer to the report than any faulty node
- the faulty nodes
- the nodes that are further from the report than the faulty node closest to the report

The score of the report is the percentage of nodes in that last class.

Certain characteristics at the limit points of the scoring function we can calculate immediately:

- A perfect report achieves a score close to 100%
- A report that includes the whole graph achieves a score of 0%
- The score of a report that includes a faulty node decreases linearly with its size
- a report that does not include a faulty node is is first expanded until it does; after that, its score decreases linearly with its size

Our metric thus bears similarities to both precision, as higher scores are assigned to small reports, and recall, as higher scores are assigned to reports that include (or will include, after a few expansion steps) the faulty node.

Although this is of no consequence in the example of the previous section, we want the ideal user to only follow paths that follow only one edge direction (either along the direction of the edges or the opposite). The reasoning for this is that if we let the ideal user follow paths while ignoring the direction of the edges, then sibling nodes would be at distance equal to two because of their common parent. In that case, blaming the wrong branch of a conditional statement would receive a high score. This would be unfortunate, since the sibling of a faulty branch is exactly the code that does not execute when the faulty code does. On the other hand, we want the search to proceed in both directions: if an expression is suspect, then the expressions that affected it are obviously suspect ("why is this expression wrong?"), but so are the expression it affects ("what does this expression affect that may affect the output?"). For this reason, we define the distance between two nodes as the length of the shortest path between them that includes either edges with their direction intact or edges with the direction inverted.

4.3 Formalization

In this section, I formalize the evaluation method and connect it to the architecture of the previous chapter. Again, the program is a set of locations L. This time, however, we take into account the structure between the locations. This structure is given by a relation $R \subseteq L \times L$, listing the edges of the graph that contains the locations as nodes. If the set of locations is the set of expressions, as in the discussion in the previous sections, then the relation R can be the system dependence graph. In another easy to imagine case, the locations can be the functions of the program and the relation R can represent the static call graph.

The evalution function then needs to be provided with two subsets of L: the set of suspicious locations S and the set of guilty locations G. S is the output of the fault localization system we are evaluating; G is the perfect report.

We first consider the case where the search for G from S can follow only the direction of edges as described by R. Then the evaluation function finds the minimal n such that $R^n[S] \cap G \neq \emptyset$. In other words, if $R^0, R^1, R^2, \dots$ are the identity relation I, R itself, the join of R with itself once $(R * R)$, etc. and $R^0[S], R[S], R^2[S], \dots$ are the images of S under those relations, we are looking for the smallest number of self-joins of R under which the image of S includes at least one node from G. If we view R as a graph, then $R^0[S], R[S], R^2[S], \ldots$ corresponds to the set of nodes reachable from S through zero, one, two, etc. edges, the levels of the initial example. The term $\bigcup_{i=0}^{m} R^i[S]$ corresponds to the set of nodes reachable from S with up to m edges. Once the function has established the minimal n such that $R^n[S] \cap G \neq \emptyset$, the score of the report S is

$$1 - \frac{|\bigcup_{i=0}^{n} (R_i[S])|}{|L|}$$

To allow the search for G from S to happen along directed paths on the graph described by R, we need to consider not only the relation R but also its reflection R^{-1}. (The reflection is not an inverse, since it is not necessarily true that $R * R^{-1} \neq I$.) $R[S] \cup R^{-1}[S]$ includes all the nodes reachable from S with one step, regardless of the direction. The relation $R^2[S] \cup (R^{-1})^2[S]$ includes all the nodes reachable from S with two steps along the same direction (following either R or R^{-1} for both steps, but not R for one step and R^{-1} for the other). Let us define the relations $Q_n = R^n \cup (R^{-1})^n$. The relation Q_n can now play the role R was playing previously, except that now the search proceeds along directed paths on either direction. So the function first searches for the smallest n such that $Q_n[S] \cap G \neq \emptyset$, and the final score assigned to the report S is

$$1 - \frac{|\bigcup_{i=0}^{n} (Q_i[S])|}{|L|}$$

4.4 Experimental Behavior

In this section I discuss the behavior of the evaluation function when applied to the results of various fault localizers and seven different programs. The fault localizers, the programs, and the bugs are described in detail in the following chapters; of interest here is the score of a report with respect to its size. Table 4.1 shows some characteristics of these programs: their size in lines of code, the number of nodes in their SDG, and the pseudodiameter (i.e. the maximum distance between two reachable nodes) of the SDG.

Program	LOC	SDG Nodes	SDG diameter
print_tokens	565	214	13
print_tokens2	510	199	15
replace	563	238	23
schedule	412	191	20
schedule2	307	150	21
tcas	173	88	10
tot_info	406	148	18

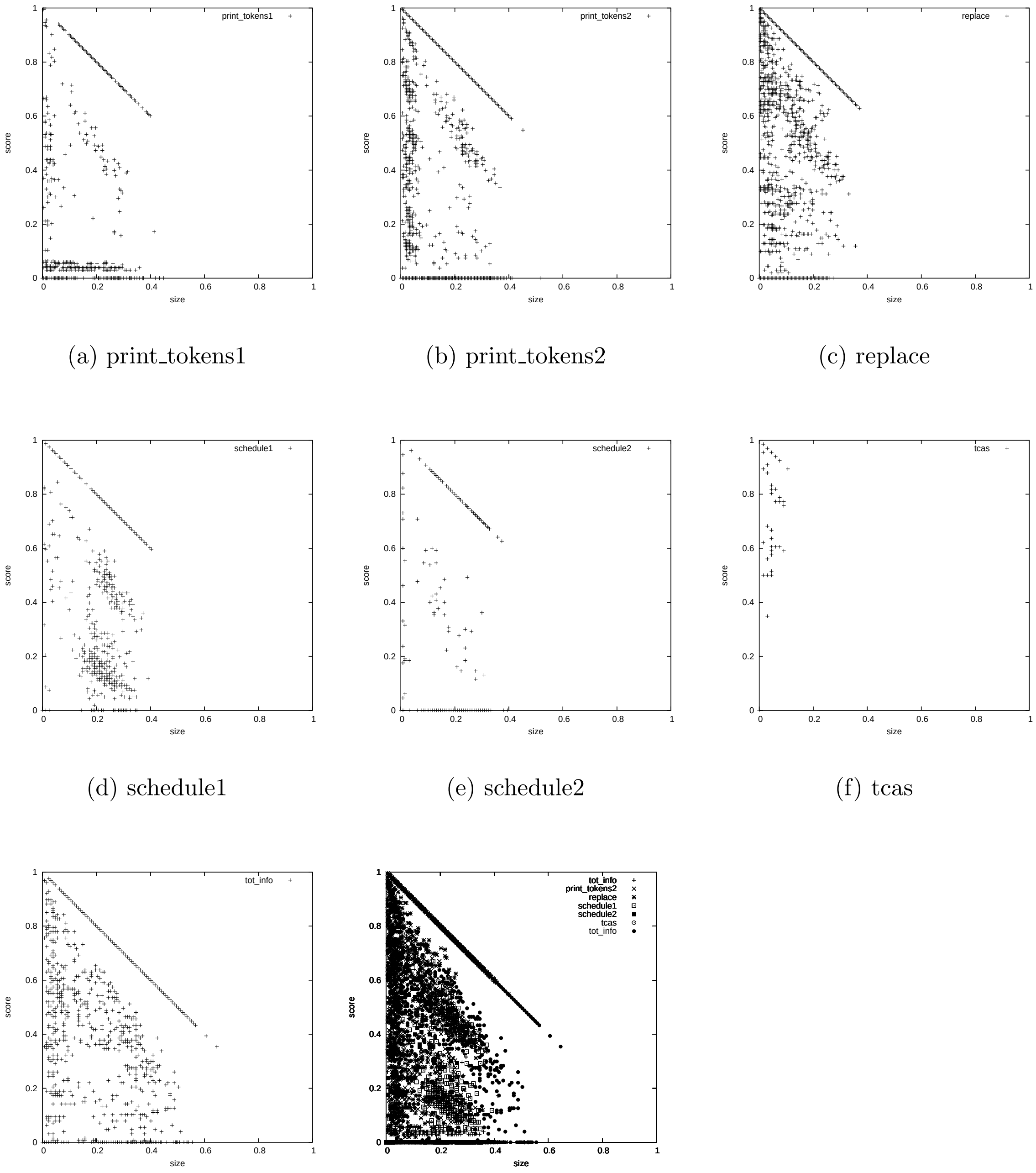
Table 4.1: A number of programs and their size characteristics

Figure 4.4 shows eight plots: the first seven contain data from individual programs, while the last one combines data from all programs. On each plot, each point corresponds to a report. The horizontal coordinate corresponds to the size of the report, and the vertical coordinate corresponds to its score.

Since the score of a report is bounded by its size, all points in these graphs are below the $1-x$ line. The most striking characteristic is the number of reports that lie on the $1-x$ line. Those are reports that include a faulty node; therefore, they only pay a penalty for their size. No report contained more than 70% of the program's nodes, although this is an artifact of the localizers used here. The graphs for certain programs (for example, tcas) include only a few points; tcas is a small program with only 66 nodes, whereas print_tokens1 has 203. As a result, the fault localizers generate fewer reports for tcas than for the other programs. One should also note the discrete nature of the reports. Even though the scores and sizes are normalized with respect to the size of the graph of the relevant program, only certain report sizes and scores are possible.

Overall, the evaluation function covers a large portion of the space. Many reports are assigned low scores, but some get high scores. Also, the evaluation method has managed a balance between the precision and the recall component: high scores are given to small reports, but a small report does not necessarily receive a high score.

The power of the evaluation function to discriminate among fault localizers is examined in Chapter 6, after I present the set of fault localizers.



(a) print_tokens1

(b) print_tokens2

(c) replace

(d) schedule1

(e) schedule2

(f) tcas

(g) tot_info

(h) all programs

Figure 4.4: Fault localization reports: scores for seven programs

4.5 Variations

A number of variations can be implemented for the evaluation method, although they usually cause some loss in the comparability of reports obtained under slightly different circumstances.

First, if we want the possible scores to include 100%, i.e. cover the entire closed interval $[0, 1]$, then it suffices to omit the failing nodes when we compute the sizes of the PDG and the report, since the report should not be penalized for the number of faulty nodes it includes. However, a side effect is that report scores from different versions of the same program are no longer directly comparable any more, since different sets of faulty nodes yield different baselines.

Second, all techniques based on dynamic information have access to coverage information, and it is tempting to express their scores with respect to the amount of executed code the ideal user would not have to examine. The reasoning is that non-executed code cannot contain the bug, and no user would ever examine it. The problem with this modification is that the results from different failing runs would be expressed with a different baseline. For example, suppose we have a failing run that executes most of the code and a failing run that executes only a small portion of the code. Suppose also that we obtain two reports from a fault-localization system, one based on each run, and that both of these reports contain the bug. If we express the scores with respect to the nodes executed in each run, then the report coming from the run executing most of the code will receive a better score than the other, even if it is larger in absolute terms.

Chapter 5

Implementations of the Architecture

This dissertation presents two instruments for effective research in debugging tools: a design architecture and an evaluation method. This chapter and the next describe a pilot study in the use of these instruments: in this chapter we design a set of tools using different program monitoring information, and in the next chapter we evaluate them. This pilot study demonstrates that:

- the architecture can be used effectively to guide the design and description of tools

- the evaluation function can yield measurable differences in the quality of tools

5.1 Tool Overview

Here I present a number of fault-localization tools based on the architecture of Chapter 3. As prescribed by the architecture, are five kinds of data there involved in each tool:

- The trace data: the information we collect from each run

- The spectra: the information we retain about each run

- The models: the information we retain about sets of runs
- The difference: the information we compute as the contrast between models (or spectra) and models
- The source code

This section briefly describes five tools individually; the following sections describe the design of each tool in terms of the architecture.

All tools use a set of successful runs but a single failing run, for the fault-coupling reasons outlined in Chapter 3. The first four systems are implemented in a system called Whither, while the fifth is called Carrot.

The first Whither tool highlights pieces of code that execute only the failing run. The underlying idea is to isolate portions of code that, if executed during a program run, the run will definitely fail, perhaps immediately, perhaps not. If such a piece of code exists and is responsible for the failing run we have, and if all the correct runs of the program were available, then the first tool would successfully isolate the faulty code. Since, in general, some correct runs are unavailable, the tool may isolate portions of code that only execute in the failing run but would execute in some correct run that is unavailable.

The second Whither tool highlights pieces of code that execute in all successful runs but not in the failing run. The idea is that such code performs some necessary or "redeeming" operation, whose absence renders a run a failure.

The third Whither tool introduces the idea of a *nearest-neighbor model*. The tool reports pieces of code that execute in the failing run but do not execute in a very similar successful run, where similarity is high if the two runs executed the same pieces of code. Past research has demonstrated that comparing a failing run with a successful run close to it can point to the location of the bug. However, the definition of "close" has been domain-specific, suggesting that information about the program's input structure is necessary to select the successful run effectively among a set of successful runs. For example, in Reps et al. [1997] part of the program's input is the date, and a significant change in behavior is anticipated when the year changes from

1999 to 2000. Zeller and Hildebrandt [2002] assume that the program's input can be halved (recursively) to generate a failure-inducing input that differs minimally from a non-failure-inducing input. In contrast, the third Whither tool defines a distance operation on the representations of runs (the spectra), and then selects for contrast the correct spectrum that is closest to the failure spectrum. Two past findings indicate comparing spectra directly is a promising avenue. Harrold et al. [2000] performed a large scale experiment on program spectra. The result was that, although an unusual spectrum does not necessarily signal a faulty run, faulty runs tend to have unusual spectra. Further work by Dickinson et al. [2001] applies clustering techniques to observation testing. They organize the profiles of runs in a metric space, and they find that the profiles of failing runs tend to belong in smaller clusters and be further away from their nearest neighbor than successful runs. These results are not surprising. Programs that succeed on a large number of inputs (almost-correct programs) must encompass a lot of knowledge about the typical inputs and consequently typical runs.

The fourth Whither tool introduces count permutations together with the nearest neighbor approach. The underlying idea is that a bug can be located by looking at code that executes too often or too rarely during a run. The definition of "too often", however, changes among runs. For example, suppose the program reads a set of characters from the input and, when operating correctly transforms all those characters, while when it fails it transforms all but one of those characters. The number of transformations is the hint to the bug; but that number is too small only in comparison to the number of characters read. The fourth Whither tool examines such information, highlighting pieces of code that execute more (or less) often with respect to other pieces of code in the failing run than they do in successful runs.

Carrot is based on the idea of potential program invariants [Ernst et al., 2001], predicates over data that may hold at specific points during a run. Carrot isolates invariants that hold over all correct runs but not over the failing run. The program location of the broken invariant is highlighted as the possible location of the bug.

5.2 Data Collection

The first step in the systems described here is data collection from the program executions. The most basic data-collection method is *program coverage*. Program-coverage tools partition the program in executable fragments and tell us, after an execution, which of these fragments executed and which did not. The granularity of the partition can vary: two common examples are functions (subroutines, methods), and basic blocks, defined in Aho et al. [1986] as

> sequences of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Often the kind of system just described is called "node coverage", because the resulting information refers to the fragments themselves; if we are interested in the transitions between pairs of fragments we have "edge coverage". The idea is that the program's fragments are the nodes of a graph, and the possible transitions from one to the other (function calls when the fragments are functions or branches when the fragments are basic blocks) are the edges of the graph. Then an execution is a path through this graph, and node profiling reports the visited nodes while edge profiling reports the traversed edges.

The program instrumentation for coverage is not particularly complex; a probe at the beginning of each fragment or, for edge profiling, at each edge will suffice. Such a probe ensures that a single bit of information is set when the probe executes. This is only marginally less expensive than recording the number of executions of the probe, and indeed this latter form of data collection, *program profiling*, is far more commonly implemented. The same choices about fragment granularity and edge versus node monitoring apply to profiling also. It is easy to obtain coverage data from a profile by simply mapping all nonzero values to true and all zero values to one.

In our implementations, we used gcov, the basic-block level profiler of the GNU compiler suite, to obtain basic-block profiles that we also mapped to basic-block coverage data.

Although most of the tools discussed here used coverage and profile data, i.e. control-flow data, Carrot used the notion of *potential invariants* by Ernst et al. [2001]. Ernst [2000] describes invariants:

> A program invariant is a property that is true at a particular program point or points, such as might be found in an assert statement, a formal specification or a representation invariant.

Discovering potential invariants automatically involves examining the values of certain variables whenever a specific fragment of code is to be executed. At the data-collection stage, debugging tools based on invariants must record those values. The relevant fragments of the program can vary, but of particular interest are function entries and exits and loop heads. Our implementation instruments programs so that the emitted value trace contains the values of function arguments at function entry as well as the return value on exit.

5.3 From Traces to Spectra

In certain cases, the data collected from program instrumentation can readily serve as a spectrum. This is true, for example, for coverage data, which can be seen as subsets of the set of program fragments and represented as binary vectors. Then all set operations are applicable, include operations on binary vectors such as the Hamming distance or set difference.

Profiles are harder to use as spectra, even though they can be viewed as vectors of integers, because operations on such vectors often yield in vectors with non-integer values, which cannot be mapped cleanly to fragments of code. For example, suppose we want to compute the distance and then the difference between two vectors containing basic-block execution counts. The standard element-wise vector difference will not do; long runs can actually be very close to shorter runs (going though a loop 10 times in one example and 10,000 in another) but element-wise difference does not preserve this information. The same is true when two loops in the program execute an equal number of times within each run, but the numbers differ in the two runs. We could normalize the vectors before subtracting them, but then the components that execute the same numbers of times (for example, the first line of the main function in a C program, which executes exactly once) would appear different for any run of even miniscule length differences.

The insight here is that we can keep the relevant execution counts of the runs, but not the counts themselves. We preserve this information by sorting the vectors and discarding the actual counts, thereby representing each run as the sorted sequence of its basic blocks, where the sorting key is the number of times that basic block executed.

Invariant-based spectra are built on the basic idea of dynamically discovered invariants. Here, the spectrum of a program run is the set of invariants that were never invalidated during the run. In our implementation, the potential invariants, in the style of Daikon [Ernst et al., 2001], are drawn from the following relational schemata, instantiated on function entries and exits:

- The *equality* invariant checks whether two variables are always equal
- The *sum* invariant checks whether two variables always sum to the same value
- The *less than* invariant checks whether a variable is always less than another variable
- The *constant equality* invariant checks whether a variable is always equal to a constant

The instantiations of a schema bind each schema metavariable to each of the variables available at the beginning of a function or the return value of the function.

In addition, Carrot generates *value sets*, which record the set of all values bound to a variable. Unlike potential invariants, which are falsified by runs, value sets are initially empty and acquire values over the course of execution. This simple form of value sets, however, can lead to needless inaccuracy in estimating program behavior. Suppose a function f has two formal arguments, x and y. If f is called twice, once on $\langle 1, 3\rangle$ and the second time on $\langle 2, 4\rangle$, the value set for x contains 1 and 2, and the value set for y contains 3 and 4. The cross-product of these sets contains the pair $\langle 1, 4 \rangle$, which incorrectly suggests that the call $f(1, 4)$ occurred in the program's execution. To diminish this form of inaccuracy, we also maintain sets of *pairs of values* that occurred during execution.

5.4 Models

If a spectrum can be viewed as a set, then models representing sets of runs can be built with set operations. For example, consider basic-block coverage: since it is a set, the set union of many such sets gives us the basic blocks that were executed in all the runs represented in the model. Similarly, the set intersection of the sets gives us a different model. When we later want to take the difference between a model of successful runs and the spectrum of a failing one, we can use the set difference operation. For the union model case, we then discover the basic blocks executed only in the failing run; for the intersection model case, we discover the basic blocks executed in all the runs but the failing run.

Invariant-based spectra can be handled in essentially the same way: the model can be built as the set of invariants never invalidated in any correct run; the difference then contains the invariants invalidated in the failing run.

Such models suffer from the fact that they confound elements of different runs. For example, if basic blocks A and B need to execute for a run to fail but each one of them can execute in a successful run, a union model can easily fail to highlight the problem. One solution to this effect is never to confound runs, but to carefully select a single successful run to contrast with the failing one, essentially reducing the model to a single run.

The natural candidate for this single correct run is the nearest neighbor of the failing run according to some distance function. For set-based spectra, we can use the Hamming distance for this purpose. The Hamming distance between two binary vectors is the count of positions at which the two vectors disagree. In this case we can still employ a set difference for the last step of the computation of candidate faulty code fragments. The choice of the Hamming distance (a symmetric operation) and the asymmetric set difference operation for the nearest-neighbor model means that it is possible for a pair of runs to have nonzero distance and an empty difference.

For sorted profiles, we can measure distance between the sequences as the distance between two permutations. Distances between permutations have a long history, starting with Cayley [1849]; they are based on the cost of transforming one to the other given a certain set of operations (for a quick introduction, see Critchlow [1985]). This is equivalent to considering one of the permutations as sorted and counting the number of operations needed to sort the other. If we allow only exchanges operations, the distance is called Cayley's distance. If we allow only adjacent exchanges (as in insertsort or bubblesort), the distance is called Kendall's τ. If we allow arbitrary moves the distance is called Ulam's distance. For example, the permutations a, b, c, d and a, c, d, b are at Ulam distance 1, because we can transform the first to the second by simply moving the b to the end.

Ulam's distance is especially suitable for the distance between spectra based on permutations of basic blocks. Allowing arbitrary moves models the phenomenon of executing the body of one loop more or fewer times than the body of another. Additionally, in computing differences we want to attribute the difference between two spectra to the basic blocks involved in the editing operations; Kendall's and Cayley's methods involve too many. Because of its relation to sorting, we would expect computing Ulam's distance to be inexpensive. Indeed, Ulam's distance between two vectors can be computed in $n \log n$ time: the algorithm is a special case dynamic programming algorithm for the longest common subsequence problem [Hunt and Szymanski, 1977].

An interesting but minor problem appears in using Ulam's distance if two basic blocks execute the same number of times in one run but not in the other. In that case, the basic blocks involved in the transformation depend on the tie-breaking operation used in sorting. For example, say we have four basic blocks a, b, c, d, a run A in which their respective execution counts are $4, 2, 3, 1$, and a run B in which their execution counts are $3, 2, 2, 1$. The permutation corresponding to A is d, b, c, a, but there are two permutations corresponding to B: d, b, c, a, and d, c, b, a, depending on whether we break the tie between b and c as $b < c$ or vice versa. If we compare A with the first B permutation there is no difference: the fact that c executed too few times with respect to b is lost. If, on the other hand, we compare A with the second B permutation, both b and c must move, and they would be highlighed. To solve this problem, we sort all basic block count vectors twice, with opposite tie-breaking rules. Then we compare permutations coming from the same tie-breaking rule, and we blame the basic blocks involved in either transformation.

5.5 Summary of Implementations

We have implemented five tools: a tool based on the basic-block coverage and a union model, a tool based on basic-block coverage and an intersection model, a tool based on basic-block coverage and the nearest-neighbor model (using the Hamming distance), a tool based on basic-block profiles and the nearest-neighbor model (using the Ulam distance), and a tool, Carrot, based on dynamically discovered invariants and the union model. All tools were implemented, for the most part, in Ocaml [Leroy et al., 2004]. The first four tools are part of the same program, with models and spectra encapsulated as modules and a single driving routine choosing among them. Carrot uses Ernst's data-collection tools and its own engine for invariant detection.

Chapter 6

Experiments

In this chapter I present a set of experiments designed to evaluate the implementations described in the previous chapter. The goal is to show how such an evaluation can actually happen, as well as to provide a set of basis data for future comparisons.

Two main results immediately follow from the findings of this chapter:

- It is possible to locate bugs with the simple profiles as spectra
- The nearest neighbor model outperforms all other models

Additionally, the utility of invariant-based fault localization remains doubtful.

More importantly, this chapter shows the successful application of the comparison process to a varying set of implementations of the architecture. Together with other researchers' application of the same evaluation process [Groce, 2004; Cleve and Zeller, 2005], this chapter provides good evidence that the evaluation process and its specific implementation of Chapter 4 are widely applicable.

6.1 Tools

To collect the program profiles, I used gcc, the GNU C compiler, in conjunction with gcov, the GNU coverage testing tool. Gcc can be instructed to instrument each line of code so that at the end of each run the program produces a file containing the

Program	Description	Vers.	LOC	Tests	Failed Tests	Versions with collisions only
print_tokens	lexical analyzer	7	565	4072	6–186	0
print_tokens2	lexical analyzer	10	510	4057	33–518	0
replace	pattern replacement	32	563	5542	3–309	1
schedule	priority scheduler	9	412	2627	7–293	1
schedule2	priority scheduler	10	307	2683	2–65	1
tcas	altitude separation	41	173	1592	1–131	18
tot_info	information measure	23	406	1026	3–251	2

Table 6.1: Overview of the Siemens suite

number of times each line executed. Gcov can then be used to read and process the file. Gcc does not provide instrumentation at the basic-block level, which would be more desirable for our purposes; however, some of the same effect can be achieved by rewriting the program's source code as described in the next section.

The various spectra and models were implemented in the Whither tool, written in OCaml. A separate tool, Carrot, also in OCaml, implements the invariant-based spectrum and model.

To obtain the SDG, we used CodeSurfer [Anderson and Teitelbaum, 2001], a commercial program slicing tool, which exports a SDG. We had to convert the exported SDG to a graph over lines, our level of profiling. We did this by adding a node to the SDG for every line and connecting all line nodes to the SDG nodes representing the line. We gave such edges weight 0. The distance between any two lines is the length of the shortest directed path between them.

6.2 Subject Programs

Our subject faulty programs come from the Georgia Tech version [Rothermel and Harrold, 1998] of the Siemens suite [Hutchins et al., 1994]. The Siemens test suite consists of 132 C programs with injected faults (Table 6.1). Each program is a variant of one of seven "golden" programs, ranging in size from 170 to 560 lines, including comments. Each faulty version has exactly one fault injected, but the faults are not necessarily localized; some of them span multiple lines or even functions. A significant number of the faults are simple code omissions: some of them make some condition in the program stricter, some make a condition laxer, and some omit a statement from a sequence. We made some slight cosmetic modifications to the programs, largely to tune them to the available tools. For example, we joined lines that were part of the same statement. Our most important change was to align the programs with the correct one, so that all bugs were just line modifications as opposed to line insertions and deletions.

Each of the seven program families comes with a test suite that exposes the bugs in each faulty version. To separate the fault-inducing inputs from the non-fault-inducing inputs, we ran the golden version on each of the inputs and compared the result with that of running a faulty program on the same input. An overwhelming number of tests succeed for each version. Column 6 (Failed Tests) in Table 6.1 shows the range of the number of faulty runs for each program family. For example, one version of print_tokens failed on only 6 of 4072 inputs, but another version of the same program failed on 186 inputs. Two programs (version 32 of replace and version 9 of schedule2) gave us no faulty runs. In the first one, the inserted bug was the exchange of a logical and operation for a bitwise and operation, but on the system on which we ran the programs this had no effect. For the second program, the inserted bug exchanges one function call for another, but the intended function is called transitively (and the result discarded). We excluded these two programs from the rest of our experiments. When working with the nearest-neighbor model we need to consider every pair consisting of a successful and a failing run of every specific program version. The number of such pairs per program version ranges from about 1500 to about two million. The total number of such pairs over all programs exceeds 34 million.

While collecting traces, we observed that in some cases, spectra of successful and failing runs collided. That is, the spectra of some failing runs were indistinguishable from those of some successful runs for the same version of the program. We observed 2888 such collisions with the coverage spectrum and 1088 such collisions with the permutation spectrum. Naturally, all the runs that collide in the permutations spectrum also collide for the binary coverage spectrum. We chose to exclude all the failing runs with collisions in the binary coverage spectrum from our experiments, for three reasons. First, when two spectra collide, all the techniques we are concerned with produce empty reports, and therefore those runs provide no comparison points. Second, any score we would assign to empty reports would be arbitrary since empty reports are obviously not good reports, though at least they do not mislead the programmer. Last, we expect collisions to be rare when using more elaborate spectra. Once we exclude all failing runs with collisions, no failing runs were left for some programs. This leaves 109 programs that we actually used in the experiment.

6.3 Results

We are interested in the average behavior of each method. We cannot simply average all the scores that each method achieves for each failing run, because certain programs have many more runs than others and the simple average would reflect those programs more. Instead, we average the scores in stages, obtaining a weighted average in which every program version has the same weight.

Let us focus on a program version P, and let us call F its set of failing runs, S its set of successful runs, and $score_U(f)$ the score of the union method, when it uses a failing run f. Then the score of the union method for P is the average of the scores of the reports based on each failing run of that program:

$$score_U(P) = \frac{\sum_F score_U(f)}{|F|}$$

The formula for the intersection model is similar.

For the random techniques, the score of the technique depends not only on the failing run, but also on the correct run the technique picks to contrast with it. If there are multiple nearest neighbors the technique could uniformly pick any of them. Therefore, to compute the score for the failing run, we average over the scores obtained by selecting each correct run. More formally, if f is a failing run in the set of all failing

Score	Inter.	Union	NN/Cov	NN/Perm
0-10%	108	101	31	19
10-20%	0	0	7	0
20-30%	0	0	15	2
30-40%	0	0	15	7
40-50%	0	0	10	4
50-60%	0	0	8	21
60-70%	0	0	4	15
70-80%	0	1	5	13
80-90%	0	1	9	10
90-100%	1	6	5	18

Table 6.2: Distribution of scores per method

runs F, then the random selection's score is:

$$score_{Random}(f) = \frac{\sum_{S} score(f,n)}{|S|}$$

Then the score for a program is defined as in the union and intersection cases.

The process works similarly for the nearest-neighbor techniques, except we average not over the set of correct runs, but over the set of nearest neighbors to the failing run N_f. The nearest neighbor score for the run is

$$score_{NN}(f) = \frac{\sum_{N_f} score(f,n)}{|N_f|}$$

Then the score for a program is defined as in the union and intersection cases.

6.3.1 Technique Performance

Table 6.2 shows the distribution of scores for the five techniques.

The intersection technique gives an empty report and achieves a zero score for all programs bar one (version 9 of schedule), for which it gives an almost perfect report. The reason for the proliferation of empty reports is the following: our test suites assure program coverage. Therefore, every top-level condition in every program has to be true in some execution and false in some other. When it is true, the parts of the code that are in the false branch are excluded from the intersection. Conversely, when the condition is false, the parts of the code that are in the true branch are excluded from the intersection. Therefore, every line of code that is guarded by a top-level condition cannot appear in the intersection if the non-failure-inducing part of the test suite obtains branch coverage of top-level conditional statements, a highly likely situation. The only parts of the program that could appear in the intersection under top-level coverage are the ones that are not guarded by anything. But those statements are always executed, even in the failing runs we examine. The intersection technique achieves a high score for version 9 of schedule for an interesting reason: the program ends prematurely (with a segmentation fault) producing an empty profile. The bug is actually in the first executable line of the main function. Obviously, all successful runs execute that line; the failing run also executes it, but because of the segmentation fault, it is not reflected in the spectrum.

The union model did succeed in finding some bugs. The interesting thing about it, though, is its almost bimodal behavior: it reports either nothing or things very close to the bug. It depends on almost the reverse of what the intersection depends on: it must be impossible to achieve full code coverage with only successful runs. This means that the bug must be very well localized at the level of abstraction the spectra provide.

The average score for the nearest neighbor model was 56% with the permutation spectrum and 35% with the coverage spectrum. The nearest neighbor models give us consistently better results than the union and intersection models, even when the bugs are not found exactly. Nearest neighbors are not hindered by coverage issues. Having a large number of runs helps, because it is easier to find a close neighbor of a failing run, but it is the existence of a close neighbor, not the number of runs, that matters. This is an important property. For the union and intersection models the set of successful runs must be large enough (to exclude all irrelevant parts of the program) but not too large, because then the successful runs overshadow the bug. This is why previous work based on these models had to use slicing, introducing an a priori relevance of the spectra to the bug. Thanks to the nearest-neighbor models we

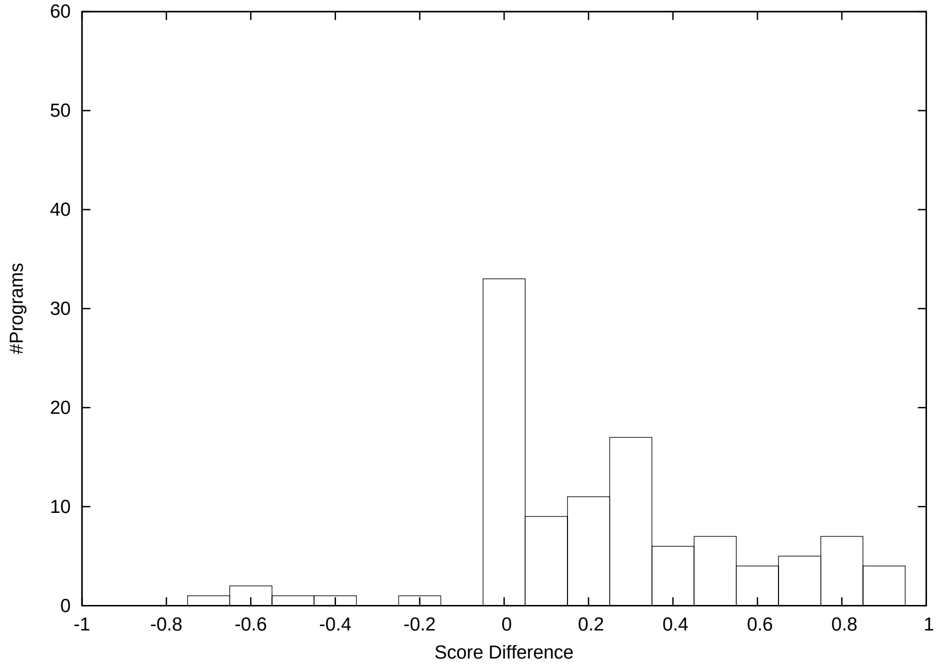


Figure 6.1: Distribution of the difference in scores between the nearest neighbor model (with the coverage spectrum) and the union model for our subject programs

can answer our first question, about the possibility of locating bugs with the given spectra, in the affirmative.

6.3.2 Technique Comparison

The second question is how the nearest-neighbor techniques perform in comparison to the union and intersection techniques. The comparison with the intersection technique is not very interesting; in the one case in which it locates the bug, the bug is also found by the nearest neighbor techniques (for the same reasons). On the other hand, the intersection technique produces mostly empty reports, so at least it does not mislead the user.

The question of the behavior of the nearest neighbors with respect to the union technique is a little harder to answer. The union technique gets more scores above 90% than the nearest-neighbor technique with the coverage spectrum. However, it achieves fewer scores above 80%. The distribution of the difference of scores between the union and the nearest neighbor method is shown in Figure 6.1. The average

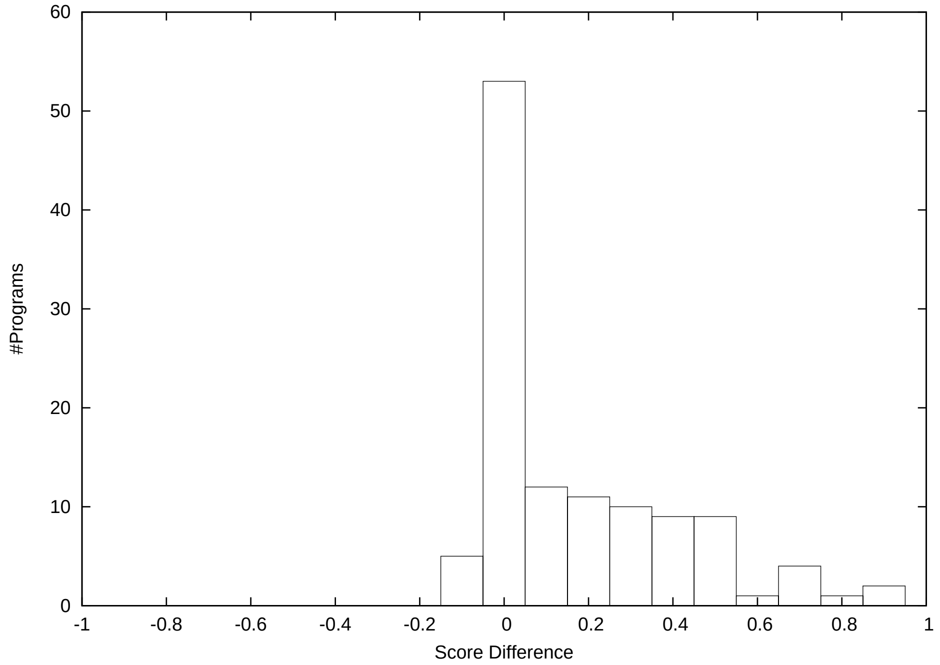


Figure 6.2: Distribution of the difference in scores between the permutation spectrum and the coverage spectrum, with the nearest neighbor model

difference is 27 points. The nearest neighbor does better in all but seven cases. Not surprisingly, six of these cases are the ones in which union gets a score above 90%. However, it is also true that for those cases, the difference in scores is large, which means that the nearest neighbor is not performing well. Note, though, that this is the average nearest neighbor, and that in two of these case there is a particular run for which the nearest neighbor finds the bug.

Figure 6.2 shows a similar graph for the nearest neighbor with the coverage spectrum and the nearest neighbor with the permutation spectrum. The permutation spectrum performs considerably better: on average, it achieves a score 10 points higher. Still, in a few cases the simpler spectrum does better. The reason is that in these cases the more complex spectrum, which is symmetric, gives us a few more nodes than just the faulty ones, and therefore the score is a little lower.

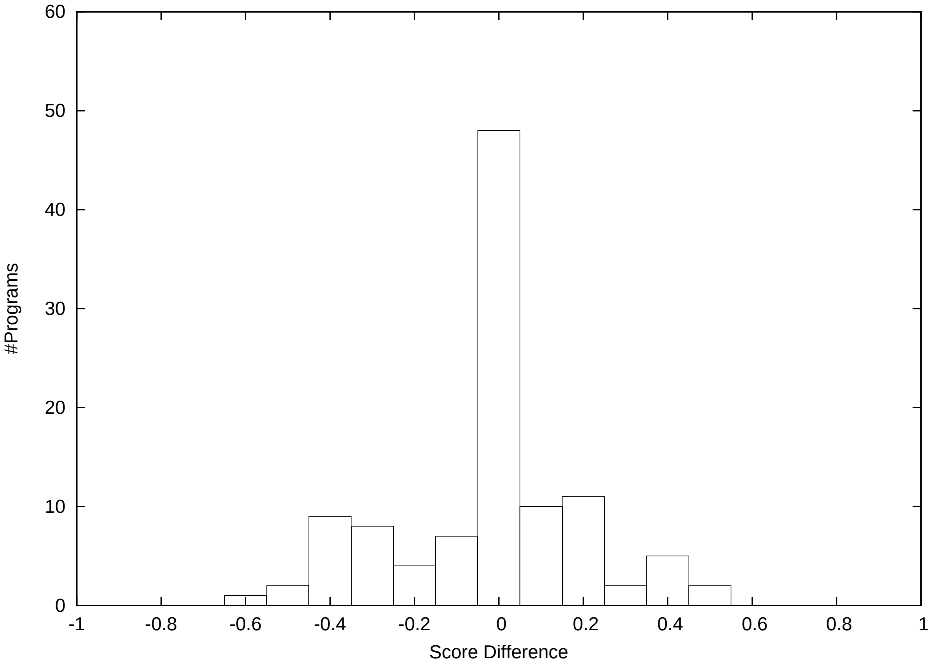


Figure 6.3: Distribution of the difference in scores between choosing the nearest neighbor and choosing any successful run uniformly (coverage spectrum)

6.3.3 Nearest Neighbor vs. Random Selection

In this section, we test whether selecting the nearest neighbor for comparison with the failing run is superior to selecting a correct run randomly. Figure 6.3 shows the distribution of the differences of scores between using the nearest neighbor and using any run. That is, given a failing run, what score would we get if, instead of going to the trouble of choosing the nearest neighbor, we chose some arbitrary successful run?

Surprisingly, in the simple spectrum case, the average run performs a bit better! A simple explanation for this is that the space defined by the coverage spectrum is too compact. Because of this, there is always a run which, when compared with our failing run, will isolate the faulty line. If this intuition is correct, then enlarging the space by using a more complete spectrum, should increase the difference from the average. Indeed Figure 6.4 shows the distribution of the difference between the score of the nearest neighbor using the more complex spectrum and the average score of using any run, instead of the nearest neighbor. This difference is 10 points on average, suggesting that indeed, in more elaborate spaces, selecting the nearest

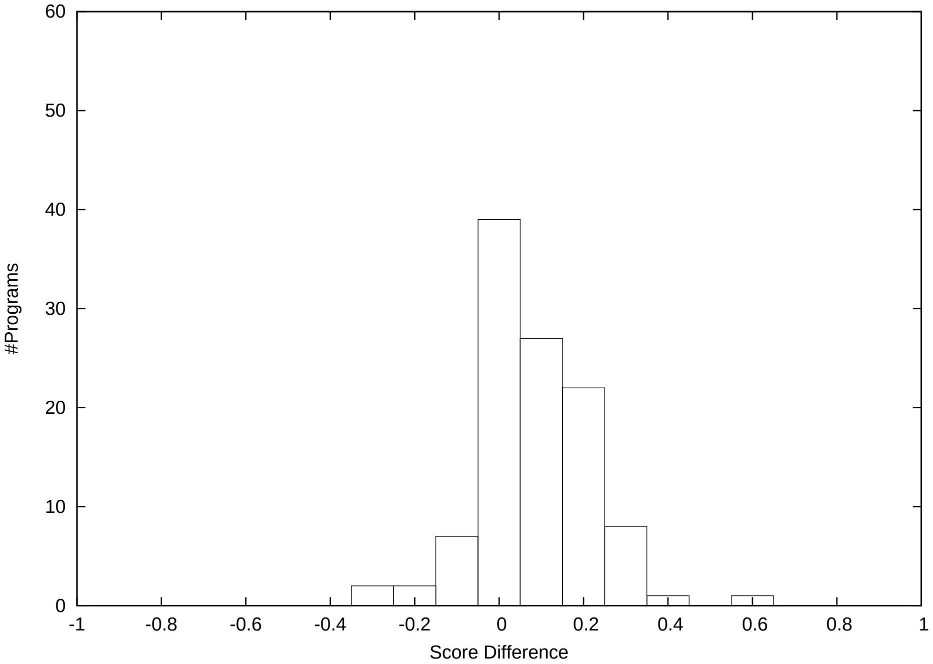


Figure 6.4: Distribution of the difference in scores between choosing the nearest neighbor and choosing any successful run uniformly (permutation spectrum)

neighbor significantly outperforms selecting a random run.

6.3.4 Discussion

There are threats to the validity of our results: the choice of programs, the choice of spectra, and the fact that I used my own evaluation strategy. The programs in the Siemens suite are relatively small and the faults are injected; there is definitely need for more realistic experiments. The spectra I use are simplistic, as witnessed by the number of spectrum collisions, and there is no guarantee that the results will hold for more elaborate spectra. Still, I believe that the use of a single similar run to help isolate faults will hold with more elaborate spectra, as witnessed by previous research that exploited input structure. Perhaps the strongest threat is that the performance evaluation method is our own. However, its simplicity leads us to believe that there are no hidden biases in favor of the nearest-neighbor model.

Our technique assumes the availability of a large number of runs, or some other way of discovering a similar run. This need can be addressed by keeping the results of beta testing by a test case generator [Korel, 1990], or by driving a manual testing process [Horwitz, 2002], perhaps based on the fault-localization report. I have claimed that my technique is independent of the input structure. If the test cases come from a test-case generator, the advantage is smaller: the existence of a test generator assumes knowledge about the input structure, but our technique alleviates the need for a distance metric on inputs and it discharges the assumption that similar inputs lead to similar runs.

6.4 Carrot

To evaluate the invariant-based spectrum and model, we[1] developed a separate tool, Carrot. To evaluate Carrot's effectiveness, we must determine whether

1. the model eventually converges to a "steady state", i.e., additional good runs do not significantly alter it
2. contrasting the model of good runs with the spectrum for a bad run produces anything at all
3. the difference in (2) holds a clue to the actual error.

Comparing a premature model of good runs against a bad run can make the tool report many more potential invariants than comparing with a steady state model. The first item is therefore important in minimizing the number of invariants the programmer needs to examine.

We tested Carrot on two programs from the Siemens suite: tcas and print_tokens. To check the eventual stability of the model, we experimented with the correct versions of tcas and print_tokens. Further experiments with a third program from the Siemens suite, replace, produced similar results. The results differ for value sets and relational invariants. We find that the size of the set of relational invariants decreases rapidly and eventually reaches a steady state. In contrast, even the last five runs of the programs cause hundreds of value-set extensions.

[1] This section is based on work done with Brock Pytlik, Shriram Krishnamurthi and Steve Reiss.

To check the second and third requirements, we examine all faulty versions available for tcas and print_tokens. We found that contrasting the faulty run to the steady-state model does not always result in invalidations. In particular, there is no invalidation for any of the 568 faulty runs of versions of tcas. For print_tokens, out of 484 faulty runs, only one (of version 5) invalidates relational invariants. The two invalidated invariants are not related to the bug.

We then created a new faulty version of print_tokens with the explicit purpose of uncovering the bug in it. print_tokens contains a partial identity function that returns its argument if the argument is in a specific set of values, and terminates the program otherwise. In our version, the function returns its argument if the argument is in the set of values; otherwise (to inject a fault) it returns the largest value of the set. The result is that the function is not an identity function anymore. This bug should be identified by the invariant schemata we implemented.

Our manufactured version of print_tokens exposes its bug on 48 inputs. Each faulty run invalidates the same two invariants that point directly to the bug, except one run which invalidates two more invariants unrelated to the bug.

6.4.1 Discussion

Our very preliminary experience based on this work is naturally negative. We were unable realistically to locate any bugs in the many variants of the Siemens suite. This failure could be caused by any number of shortcomings in our approach: Carrot does not implement a sufficiently rich set of invariants; the style of invariants used by Carrot is mismatched with the programs we are analyzing; or potential invariants are not suitable for debugging. Our experiments are too premature to conclude the last point. On the one hand, the potential for success clearly exists, as the manufactured version of print_tokens suggests and as tools more closely hewn to a particular domain, such as Diduce [Hangal and Lam, 2002], demonstrate. On the other hand, a similar technique used by Groce and Visser [2003] has not been shown to bear fruit.[2]

[2] Visser, personal communication.

Chapter 7

Elision

When you come to a fork in the road, take it.
— Yogi Berra

Chapter 5 discussed a number of implementations based on coverage spectra, i.e. on which parts of code execute during a run. Intuitively, our confidence in a fragment of code increases when the fragment executes (is covered) during a correct run and decreases when it executes during a failing run. This intuition lies at the basis of a number of systems [Pan and Spafford, 1992; Reps et al., 1997; Jones et al., 2002] and is exemplified in its strongest form by the first Whither tool: every line of code that executes in a correct run is deemed correct, every line of code that executes only in a failing run is deemed faulty.

In this chapter I show how this intuition can lead to wrong conclusions, with adverse effects for debugging. Just because a line of code executes in a correct run does not make it correct; it may have had no effect on the output of the program. A well-known manifestation of this absense of effect is that, given a program and a test suite that covers it fully, even if none of the tests fail, we still cannot be sure that the program harbors no mistakes.

In this chapter I examine a special case of this phenomenon in which individual conditional choices give us misleading coverage information. Conditional choices are the basic elements of coverage: coverage is simply an aggregate view of all the control-flow choices made during a program run. Therefore, when individual choices give us misleading information about the correctness of a fragment, the aggregation will be misleading also. In the particular case examined in this chapter, a choice makes no difference to the program's outcome. Thus, the coverage of any program fragment due to this choice provides no evidence on the correctness of the fragment.

7.1 An Example

Consider the program in Figure 7.1. Suppose that the expected outcome of the program is `1` and the value of the complicated expression is `true`. When the `then` clause is executed, coverage provides (correct) evidence that both the condition and the `then` clause are correct.

However, suppose that the `else` clause is replaced with `x:=5;`, as in Figure 7.2, and that there exist two test cases with different values of `c` for which the expected output is `1`. The program succeeds in both cases, and coverage provides no reliable information on the correctness of either the condition or the executed clause. In effect, the conditional statement made no choice, so both the condition and the executed branch could be erroneous. Conversely, if the expected output is not `1` and `c` is equal to `true`, coverage points to the `then` clause as suspect. However, since the condition made no choice, it may well have been wrong, and then the `else` clause could be the erroneous one.

```
c := (* complicated expression *)
if (c)
then x := 3;
else x := 4;
print x mod 2;
```

Figure 7.1: A simple if-then-else

```
c := (* complicated expression *)
if (c)
then x := 3;
else x := 5;
print x mod 2;
```

Figure 7.2: An elided if-then-else

We say that a particular evaluation of a conditional during a run is *elided* if the run has the same outcome regardless of which branch is executed. We also call the conditional itself elided. If all evaluations of a particular conditional during a run are elided, we say that the conditional was *totally elided* during the run.

Elision is a dynamic concept. The elision of a particular evaluation of a conditional is affected by both the program state at the time of the evaluation and the future of the run. In our first example, if instead of `x := 5` the `else` clause read `x := y`, `y`'s value would influence the conditional's elision. Moreover, if the conditional is reached twice, once with `y` odd and once with `y` even, it would be elided only once. The rest of the computation, and how it handles the outcomes of each branch, also determines whether the conditional is elided. In our example, the immediate effects on `x` are discernible, but the conditional is elided by the modulo operation. Therefore, the particular dynamic context affects whether a statement is elided or not.

```
x := 5;
c := y < 10;
d := true;
e := (y < 50) and (z < 10);
if (c orelse d orelse e)
then x = 3;
print x mod 2;
```

Figure 7.3: Elision with boolean operators

At first sight, it seems unlikely that conditional elision is common. After all, every `if` statement in a program is there for good reason, and programs avoid recomputing the same values, as dictated by efficiency concerns and good algorithm design. But these are local and static properties: in full programs, at run time, decisions are often repeated or have no effect on the program's outcome. Programs do not output their internal state; we only have narrow windows through which to discern their behavior. In Figure 7.3 we cannot observe the whole value of `x`, only that of its lowest bit. Second, elision is a dynamic property of the program, and not every conditional statement has a crucial place in all contexts. In Figure 7.3, given that `d` is true, `c` is elided, although used[1]. Lastly, the same conditional check may appear multiple times in different guises: in Figure 7.3 the computations of `c` and `e` both check whether `y < 50`.

To discover if a conditional is elided or not, we need to pose the following question: how does the output of the program change if the condition is inverted? To answer the question experimentally, we forcibly change the value of the condition just before the direction of the branch is decided, and we monitor any differences in the observable behavior of the program. In this chapter, we focus on *single elided conditionals*: in other words, we make a single inversion during a run. We repeat the process for each conditional, starting each time in the original run and inverting the condition immediately following the last condition we inverted (Figure 7.4).

Examining both branches of each `if` is akin to introducing faults into the subject program. We replace every statement of the form `if(c)` with a statement of the form `if(c xor s)`, where `s` is a boolean "slack" variable. Then we allow only one slack variable to be `true`, exactly once through the program run, and we examine its effects. We replicate the process for each slack variable and for each evaluation of the condition during the run. Unlike with other approaches like mutation testing [DeMillo et al., 1978], we need not specify directly how the new conditional differs from the original. This lets us separate all possible mutations of the conditional into two groups: the group that leaves all decisions intact and the group that inverts the decision once. Then we can examine both groups without enumerating them. Therefore, we can implicitly address a large number of potential faults and examine whether a test case

[1] This example also illustrates that elision is not static or dynamic slicing: `c`, `d` and `e` would be in the backward static slice of `x`, and `c` and `d` would be in the dynamic slice.

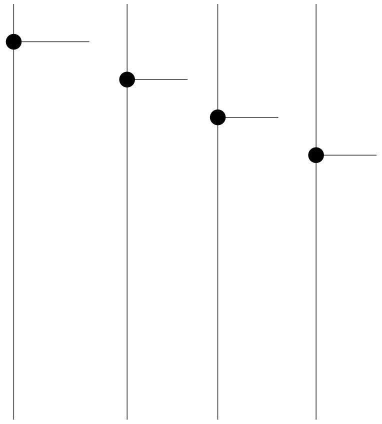


Figure 7.4: The structure of the elision exploration. Each of the vertical lines represents a run identical to the original run; each dot on a vertical line represents the conditional statement we examine; each horizontal line represents a run that deviates from the original in that we forcibly change the value of the condition.

would expose them.

7.2 Implementation

Before we discuss our implementation, we should mention a number of other testing and debugging approaches that modify the program behavior, either statically or at run time. Dynamic mutation testing [Laski et al., 1997] establishes the sensitivity of code by changing function return values. Delta Debugging experiments with splicing parts of the state of a successful execution onto a failing one in order to isolate cause-effect chains [Zeller, 2002]. Our approach is more restrained: we merely change boolean values in conditions. This frees us from the need for an alternate run and allows us to exhaust the space of changes. Critical slicing [DeMillo et al., 1996] selectively removes statements to examine whether they affect a slice. Removing a statement is equivalent to leaving the program state intact. For conditionals, this technique would not always examine both branches. Mutation testing [DeMillo et al., 1978] modifies the subject program; the user has to augment a test suite until it distinguishes all mutants from the original program. Elision can be used in this way: the user would have to build a test suite in which all conditionals matter at least once.

Our experimental framework for discovering elided conditionals has two coupled components:

- An *instrumentation component* that rewrites the source code of the subject program to let us explore both branches of each conditional.
- A *monitoring component* that lets us examine the differences in the program's behavior when we follow the branch of a conditional not dictated by its evaluation context.

Our instrumentation system [Chan-Tin, 2004] is built on top of CIL [Necula et al., 2002], a source-to-source transformation library for the C language. We first apply two of CIL's own transformations to the subject program. The first one transforms all loops into simple conditional statements, as in Figure 7.5. The second transforms all binary boolean operators to sequences of nested `if` statements, as in Figure 7.6. Both transformations insert `goto` statements as necessary.

```
while (i != EOF) {
    // loop body
}
```

BECOMES

```
while (1) {
    while_0_continue:
    if (! (i != EOF)) {
        goto while_0_break;
    }
    // loop body
}
while_0_break:
```

Figure 7.5: Sample CIL loop transformation

After these two transformations, we apply one of our own that transforms all statements of the form `if (c)` into statements of the form `if (split(c))`. The three

```
if (i == ’#’ || i == ’@’) {
    // conditional body
}
```

BECOMES

```
if (i == ’#’) {
    // conditional body
} else {
    if (i == ’@’) {
      // conditional body
    }
}
```

Figure 7.6: Sample CIL boolean transformation

transformations together ensure that we call the `split` function on every branch of the program.

The `split` function, implemented on top of the Unix `fork` system call, is an identity function with an important side effect: it duplicates the running process. Following Unix terminology, we call the newly created process the *child process*. The child process differs in two major ways from the original parent process: the return of `split` is inverted, and future invocations of `split` have no side effects (and return their argument unmodified). In this way, we create a number of processes equal to the number of branches encountered during the program execution (Figure 7.7). This can dramatically increase the running time of a test run.

Our monitoring system is implemented as a modified C library. Our C library behaves differently for the original process, which follows all the branches as if we had not inserted the calls to `split`, and for each of the child processes, which have one inverted decision. For the original process, the monitoring system simply records all output, including writing to files and terminals; it also records the process's total execution time. For the child processes, the monitoring system checks whether the output agrees with that of the original process, and also checks that the running time does not exceed three times the running time of the original process. We stop a child

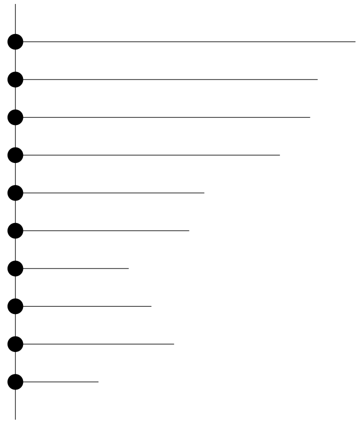


Figure 7.7: Implemented elision exploration, combining the prefixes of processes

process as soon as it (irrevocably) behaves differently from the parent process[2]. In general, any difference that a user might observe should be monitored; however, it is of equal importance not to flag differences that the user cannot observe.

We start each of the child processes in a blocked state. When the parent process finishes, we resume the child processes one at a time and monitor for any observable differences from the parent process.

7.3 Experimental Results

We experimented with three C programs: tcas, print_tokens, and space. Two of these programs, tcas and print_tokens, come from the Siemens suite [Hutchins et al., 1994; Rothermel and Harrold, 1998]; the third is a program written for the European Space Agency [Rothermel et al., 2001; Vokolos and Frankl, 1998]. Each of these programs has a number of versions, all but one of which contain faults. We used the correct version for our experiments.

tcas is an aircraft collision-avoidance system that consumes 12 integers and outputs 0, 1, or 2, depending on whether the airplane should increase, keep, or decrease its altitude. It is essentially a large predicate. Few programs have the same structure

[2] We assume that the program does not undo any of its actions; for example, it does not write into the same position of a file twice.

Program	Description	LOC	#Cond	#Tests
tcas	altitude separation	173	33	1608
print_tokens	lexical analyzer	565	88	4132
space	ADL interpreter	6218	622	13297

Table 7.1: Overview of subject program for elision experiments. Column 3 is the number of lines of code; column 4 is the number of conditionals.

as tcas, but many programs contain complex predicates. print_tokens is a lexical analyzer and space is an interpreter for an array definition language (ADL).

Because of tcas's structure, we expect that most conditionals will be elided. We expect elision to be less extensive for the other two programs, since they output much more information about their inputs.

Program	Boolean Coverage	Boolean Elision	Ratio	Avg. False Coverage
tcas	23615	14923	0.63	9.3
print_tokens	238787	8765	0.04	2.1
space	2769599	366198	0.13	27.5

Table 7.2: Elision statistics over the number of conditionals

Program	Count Coverage	Count Elision	Ratio
tcas	24501	15809	0.65
print_tokens	4291376	533391	0.12
space	63923979	15979909	0.25

Table 7.3: Elision statistics over the number of executions of conditionals

We run each program on a number of tests, as shown in Table 7.1. For each run, we kept information on how many times each conditional statement executed and how many times it was elided. Table 7.2 summarizes our findings. The first column contains the program name. The second column contains the aggregate boolean coverage of conditionals: the number of conditionals evaluated during all runs, when each conditional is counted only the first time it executes during a run. The third column contains the aggregate boolean elision: the number of conditionals elided during all runs, when each conditional is counted once for each run where it was totally elided, i.e. all its evaluations were elided. As we would expect, 63% of conditionals are elided in tcas, 13% in space, but only 4% in print_tokens. The fourth column contains the ratio of columns 2 and 3. Column 5 is the average number of elided conditionals during a run. These conditionals are still executed and thus are reported as covered, but this information is misleading. Nine conditionals are elided on average in the runs of tcas. This is out of 33 conditionals in the entire program, although not all of them execute during all runs. For print_tokens and space the numbers are lower but still significant. One dramatic example appears in tcas. The original, uninstrumented program contains a conditional of the form `if (a && b && c)`. The `c` part of the conditional is executed in 245 runs, yet is totally elided in all of them. As a result, its computation remains untested, after 1608 tests. Figure 7.8 shows a graphical summary of all the tcas conditionals. Another example appears in space, where a nested conditional is elided in 61 of the 75 runs in which it executed.

Table 7.3 repeats the first four columns of Table 7.2, but we count each conditional evaluation individually. For tcas the difference between the second and third columns and the fourth and fifth columns is the same, because any non-elided conditional is non-elided exactly once during the run.

7.4 Discussion and Related Work

Our results here are confirmed by an independent study [Wang et al., 2003], using different tools and subject programs.

Since elided conditionals occur, I believe that future dynamic analyses should take them into account. In general, any information discovered during dynamic analyses that is not true for both branches of each elided conditional in the program should be discarded. Branch and statement coverage should not consider as covered the branches of elided conditionals. In def-use chain tracing, an elided conditional should not constitute a use. Path profiling [Reps et al., 1997] should not include paths that

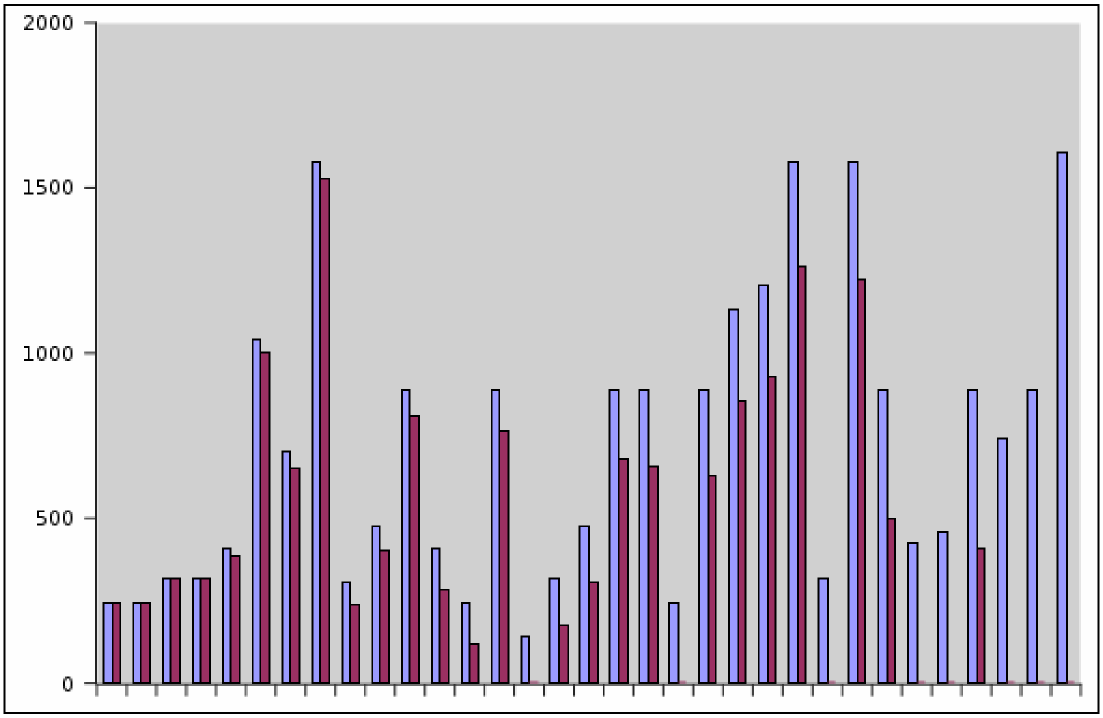


Figure 7.8: tcas profile and elision data per conditional. Each pair of bars corresponds to a conditional and the vertical axis represents number of runs. The blue (light) bars represent the number of runs in which a conditional executed, while the red (dark) bars represent the number of runs in which the conditional was totally elided. The conditionals are ordered by the number of runs in which they were not elided.

include elided conditionals. Invariant detection [Ernst et al., 2001] should not include invariants stemming from branches of elided conditionals. Automated debugging based on coverage data [Jones et al., 2002; Renieris and Reiss, 2003; Reps et al., 1997] should not include elided branches in spectra comparisons.

Elision can be a tool in the search for causal relations of program faults [Groce and Visser, 2003; Zeller, 2002]. By its nature, elision modifies the program in minimal ways, consistent with Lewis's theory of causation [Lewis, 1973]. When a program does not fail even if a conditional is changed, this conditional clearly cannot be the cause of the fault.

In its present implementation, elision discovery is prohibitively expensive. Smarter state sharing, as in the Java Pathfinder [Visser et al., 2003], would facilitate its use. This would also let us experiment with multiple (boolean) faults.

Elided conditionals are but the simplest case of elision. In the more general case, we would have to perturb a single non-boolean variable at every expression. This is much harder: the alternative domain may be much larger and not well defined. Consider an integer used as an index to an array. Its domain, not taking into account its usage, is the full domain of integers. If we take its usage into account, then its domain in an unsafe language like C remains unchanged, but in a safe language like Java it becomes the union of the valid array indexes and a single item that would cause an out-of-bounds error. The situation becomes harder when the variable we want to manipulate is a pointer. Its domain could include every item in the data structure it points to, every item of the same type as the item currently pointed to, or, in unsafe languages, any address in memory. The situation dictates that we develop specific fault models for each language and perhaps each program.

We can view elision as a dynamic information-flow problem [Denning and Denning, 1977]. If we consider the regular input of the program as public and the values of the slack variables (c.f. section 7.1) `s` as private, then the elision question becomes equivalent to the information-flow question whether we can find the values of `s` from the public inputs and the output. However, the enforcement of secure information flow is often local in the code [Sabelfeld and Myers, 2003], and in elision we are interested in remote effects.

Such remote effects are captured by the RELAY model of fault origination and transfer [Thompson, 1991; Thompson et al., 1993]. Assuming a fault from a class of syntactic faults, RELAY constructs a necessary and sufficient condition for the observation of the fault. RELAY targets a relatively simple language (for example, the target language has no pointers), and yet the conditions it needs to construct are complex and sometimes fault-specific. The authors point out that it is impractical to instrument a program to monitor the condition constructed by RELAY. Still, the model provides insight into the complexity of the problem, and could provide local necessary conditions to improve the efficiency of a brute-force method like ours.

Chapter 8

Conclusions

When I started work on automated fault localization, I was hoping that research in the field was as mature as in compiler optimization, with an established set of benchmarks and a well understood architecture. I found out this was not true: much debugging work is disconnected. Once I had the basic ideas of the framework, I kept finding more and more papers that implicitly discussed the same architecture, within the scope of a particular solution to a particular problem. Most of these papers do not reference previous work that implemented the same architecture; terms did not exist to describe the differences among them.

Two things became clear from reading a few of these papers: that the emerging architecture, or at the very least design pattern, for fault-localization systems had not been identified and articulated, and that there was no way to compare different tools. These observations implied that the barrier for new research in this field was extremely high: researchers had to start from specific domains (which means that they had to be domain experts in some other area) and their that progress could not be quantified, and was therefore doomed to be slow and disconnected.

I have tried to remedy these two problems in this dissertation. A multitude of debugging spectra and models are waiting to be discovered. Until now, whole systems had to be invented in one gulp, and my hope for the framework described in this dissertation is that it breaks the problem in smaller, more tractable pieces. At the same time, the evaluation method allows researchers to compare different implementations in a way that was not possible before, and therefore allows us to identify progress in the field far more easily.

Bibliography

Agrawal, Hira, James L. Alberi, Joseph R. Horgan, J. Jenny Li, Saul London, W. Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73 [1998]

Agrawal, Hiralal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616 [1993]

Agrawal, Hiralal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [1990]

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley [1986]

Ammons, Glenn, Rastislav Bodik, and James Larus. Mining specifications. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [2002]

Ammons, Glenn, David Mandelin, Rastislav Bodik, and James Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [2003]

Anderson, Paul and Tim Teitelbaum. Software inspection using Codesurfer. In *Proceedings of the Workshop on Inspection in Software Engineering* [2001]

Ball, Thomas. The concept of dynamic analysis. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* [1999]

Ball, Thomas, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [2003]

Boyd, Mickey R. and David B. Whalley. Isolation and analysis of optimization errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [1993]

Bush, William R., Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802 [2000]

Cayley, Arthur. Note on the theory of permutations. *Philosophical Magazine*, 34:527–529 [1849]

Chan-Tin, Sébastien. *Sandboxing Programs*. Master's thesis, Brown University [2004]

Chen, T. Y. and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Experience*, 9(1):33–46 [1997]

Choi, Jong-Deok and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the International Symposium on Software Testing and Analysis* [2002]

Cleve, Holger and Andreas Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering* [2005]

Critchlow, Douglas E. *Metric Methods for Analyzing Partially Ranked Data*, volume 34 of *Lecture Nodes in Statistics*. Springer-Verlag [1985]

DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41 [1978]

DeMillo, Richard A., Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the International Symposium on Software Testing and Analysis* [1996]

Denning, Dorothy E. and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513 [1977]

Detlefs, David L., K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report #159, Compaq Systems Research Center, Palo Alto, USA. `ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-159.ps.gz` [1998]

Dickinson, William, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* [2001]

Ducassé, M. Abstract views of Prolog executions in Opium. In *Proceedings of the International Symposium on Logic Programming* [1991]

Ducassé, Mireille. Coca: A debugger for C based on fine grained control flow and data events. In *Proceedings of the International Conference on Software Engineering* [1999]

Eick, Stephen G., Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968 [1992]

Engler, Dawson, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Symposium on Operating Systems Principles* [2001]

Ernst, Michael D. *Dynamically Discovering Likely Program Invariants*. Ph.D. thesis, University of Washington [2000]

Ernst, Michael D., Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123 [2001]

Evans, David, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering* [1994]

Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349 [1987]

Ganter, Bernhard and Rudolf Wille. *Formal Concept Analysis.* Springer-Verlag, English edition [1999]

Gill, S. The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society; Series A Mathematical and Physical Sciences*, 206(1087):538–554 [1951]

Groce, Alex. Error explanation with distance metrics. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems* [2004]

Groce, Alex and Willem Visser. What went wrong: Explaining counterexamples. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software* [2003]

Hangal, Sudheendra and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering* [2002]

Harrold, Mary Jean, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194 [2000]

Hastings, Reed and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Usenix Winter 1992 Technical Conference* [1991]

Hoare, C. A. R. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25 [2003]

Holzmann, Gerald J. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295 [1997]

Horwitz, Susan. Tool support for improving test coverage. In *Proceedings of the European Symposium on Programming* [2002]

How Tai Wah, K. S. (Roland). A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45 [2000]

Hunt, James W. and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353 [1977]

Hutchins, Monica, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering* [1994]

Johnson, S. C. Lint : A *C* program checker. In *UNIX Programmer Manual.* BELL Labs., 7th edition [1979]

Jones, James A., Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the International Conference on Software Engineering* [2002]

Kelsey, Richard, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand. [1998]

Korel, B. Automated software test generation. *IEEE Transactions on Software Engingeering.*, 16(8):870–879 [1990]

Korel, Bogdan and Janusz Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195 [1990]

Larus, James R., Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3) [2004]

Laski, Janusz, Wojciech Szermer, and Piotr Luczycki. Dynamic mutation testing in integrated regression analysis. In *Proceedings of the International Conference on Software Engineering* [1997]

Lencevicius, Raimondas, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* [1997]

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system version 3.08. `http://caml.inria.fr/` [2004]

Lewis, David K. *Counterfactuals*. Harvard University Press [1973]

Liblit, Ben, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [2003]

Linton, M. *Queries and Views of Programs Using a Relational Database System*. Ph.D. thesis, Computer Science Department, University of California, Berkeley, California [1983]

Manning, C. D. and H. Schütze. *Foundations of statistical natural language processing*. MIT Press [1999]

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML — Revised*. MIT Press [1997]

Necula, George C., Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction* [2002]

Orso, Alessandro, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis* [2002]

Pan, Hsin and Eugene H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University [1992]

Pierce, Benjamin C. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts [2002]

Pytlik, Brock, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the Workshop on Automated and Algorithmic Debugging* [2003]

Renieris, Manos, Sébastien Chan-Tin, and Steven P. Reiss. Elided conditionals. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering* [2004]

Renieris, Manos and Steven P. Reiss. Fault localization with nearest-neighbor queries. In *Proceedings of the IEEE Conference on Automated Software Engineering*, 30–39 [2003]

Reps, Thomas, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 432–449 [1997]

Rothermel, Gregg and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419 [1998]

Rothermel, Gregg, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948 [2001]

Sabelfeld, Andrei and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19 [2003]

Savage, Stefan, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411 [1997]

Seward, Julian and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference* [2005]

Shapiro, Ehud Y. *Algorithmic Program Debugging*. The ACM Distinguished Dissertation Series. The MIT Press, Cambridge, Massachusetts [1983]

Smith, Raymond and Bogdan Korel. Slicing event traces of large software systems. In *Proceedings of the Fourth International Workshop on Automated Debugging* [2000]

Thompson, Margaret C. *An Investigation of Fault-Based Testing Using the Relay Model*. Ph.D. thesis, University of Massachusetts at Amherst. Available as Technical Report 1991-022, Computer Science, University of Massachussets at Amherst [1991]

Thompson, Margaret C., Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis* [1993]

Tip, Frank. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189 [1995]

Visser, Willem, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232 [2003]

Vokolos, Filippos I. and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance* [1998]

Wang, Nicholas, Michael Fertig, and Sanjay Patel. Y-branches: When you come to a fork in the road, take it. In *International Conference on Parallel Architectures and Compilation Techniques*, 56–67 [2003]

Weiser, Mark. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357 [1984]

Whalley, David B. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16(5):1648–1659 [1994]

Wilkes, Maurice V. *Memoirs of a Computer Pioneer*. MIT Press [1985]

Zeller, Andreas. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 253–267 [1999]

Zeller, Andreas. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering* [2002]

Zeller, Andreas and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200 [2002]

Zeller, Andreas and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *ACM Sigplan Notices*, 31(1):22–27 [1996]