Fault Localization in *In Vivo* Software Testing

by
Del Slane

A Thesis submitted to the Faculty
in partial fulfillment of the requirements for the
BACHELOR OF ARTS

Accepted

Paul Shields, Thesis Advisor

Charles Derr, Second Reader

Mary B. Marcy, Provost and Vice President

Bard College at Simon's Rock
Great Barrington, Massachusetts
2009

Acknowledgements

First and foremost, I would like to thank Paul Shields. He introduced me to the world of Computer Science and offered encouragement as I began exploring the field with no background whatsoever. Without his consistent feedback and perpetual willingness to answer all sorts of questions that arose as I learned, I would not have been able to acquire the necessary competence to enjoy the more advanced nuances of the field. As writing software has turned out to be my career, I owe Paul much credit for helping me make a living in a field I truly enjoy.

A great deal of thanks is also due to my mother for her unwavering support of my academic pursuits as well as her financial contributions that made it possible to concentrate on my studies.

Thanks also to Charlie Derr who was on the thesis committee, and Chris Murphy with whom I worked on topics related to *in vivo* testing.

Contents

List of Figures

Abstract

The growing reliance on computing devices to advance efficiency and functionality in fields from mobile technologies to finance has been accompanied with a need for increasingly complex software in order to provide that functionality. Experience has proven this complexity in software difficult to manage, and most software released to the public today contains numerous defects. The impossibility of exhaustive testing in a pre-release environment suggests that at least some of these latent defects will be found in the much larger exploration of the program's state space after release. However, in this situation it is difficult for the software designers to pinpoint the exact cause of a failure. For this reason the idea of automated fault localization after software deployment in the context of *in vivo* software testing is explored, and the feasibility of defect identification using existing algorithms is demonstrated.

Chapter 1

Introduction

Computer software has become a necessity for managing and interfacing with a variety of devices used in everyday life. Some examples are mobile phones and GPS devices, automobiles, some household appliances, thermostats, in-flight entertainment systems, handheld and console gaming platforms, and of course desktop and laptop computers. The ballooning popularity of internet-based services epitomized by the recent growth of Facebook and the previous establishment of Google has created an entirely new and expanding market for on-demand software products—a drastic shift from store-bought software that was the only major distribution medium prior to the internet.

The benefits of software and powerful hardware on which to run it have been realized across almost every industry through automation of business processes from back office, to manufacturing, to risk management, to coordination. There is also increasing popularity of private high performance computing centers especially in fields like finance, which requires computer models to price complex financial instruments

and analyze risk exposure. It is fair to say that software has even been the main factor behind the scale on which automation and data management are currently possible; without large enterprise software managing everything from ordering to processing and distribution, many businesses could simply not operate as efficiently as they do.

The result is that various software components have been extensively developed with a strong theoretical and practical foundation, and are frequently leveraged across disciplines. Various engineering practices have moved towards sophisticated three dimensional modeling tools and computational simulations; any field that requires gathering of data likely does so in a digital format and stores it within a database management system; and artists use software to create, edit, and distribute their work. The internet has become a medium for publication of and access to everything from independent work to peer-reviewed journals, and has established itself as an ecosystem of its own that composes software systems in new ways, from commerce to social networking and cloud computing.

As consumers and industry alike demand more and more from the software they use, various technologies have been invented to cope with managing the dramatic increase in complexity of the systems delivered. In fact, the field of Software Engineering has come into being to establish methods for controlling software complexity throughout development and until retirement of a system. While common best practices have been established for testing a system against its functionality and behavioral requirements, there remains a serious problem of defects in deployed software systems. This is a direct result of the infeasible computational requirements of comprehensively testing all possible interactions between a system's various compo-

nents, even in light of exponential improvements in the speed of and storage available to modern computers.

Software defects are the cause of myriad problems with computer systems such as data loss and corruption, improper behavior preventing them from fulfilling their intended purpose, reliability issues, and security compromises. However, it is eventually decided that loss of revenue from delaying product releases outweighs the benefits of time spent finding and fixing more defects, and systems are therefore shipped with latent defects. In addition, there is no feasible way to prove the absence of defects in such a complex system, and the severity of possible defects is completely unknown until discovered. From this perspective, the development and use of software has evolved into a risk management scenario concerned with balancing the efficiency and scale gains of using the software against the possibility that its incorrectness might threaten data or other systems (as in a networked environment).

Of course it is desirable to identify and fix these defects before they can cause any damage or be exploited. This is made possible via systems to "patch" the software such as is done with operating system update functionality like the Windows Update system for Microsoft operating systems and Software Update in Apple's OS X. Defects found through continued testing in a development environment are easier to create patches for than those identified through the typical crash reporting supplied with operating systems and applications. This is because in a development environment engineers may inspect the entire internal state of the program as a failure is reproduced and have access to input data that will reproduce the defect whereas neither advantage is typically available otherwise.

In order to help locate the cause of program failures in a deployed software instance, methods for gathering relevant information during execution and periodically running tests and reporting result data are examined in chapter 4, *In Vivo* Fault Localization. Information on the field of software engineering and the process for pre-deployment software testing as established thus far is presented in chapter 2, Pre-deployment SoftwareTesting, which covers various methods for testing along with their associated benefits and tradeoffs. Existing research related to automated fault localization and post-deployment software analysis and testing is presented in chapter 3, Related Research, offering a good introduction to methods as old as 1981 up through current work in the fields.

Chapter 2

Pre-deployment Software Testing

This chapter explains the concepts of software testing phases that are typically implemented prior to deployment of a production software system in order to offer sufficient assurance that it operates to its specifications and is generally well-behaved. Before addressing the specifics of testing, the field of software engineering is introduced in order to put the testing component of software development in the context of a complete and refined software creation process.

2.1 Software Engineering

The complexity of modern software is what necessitates the field known as *software engineering*, leading Van Vliet to claim that "the central theme is mastering complexity" [1, pp. 7]. Software engineering consists of various areas that assist the goal of creating software in an efficient, well-structured manner. In order to create quality software, a number of standard phases are typically completed in various ways. This

leads to what is known as "software processes" or "the software life cycle" [1, 2, 3, pp. 64, pp. 64, pp. 48], which is generally composed of the following components:

- Specification: gathering of requirements from end users of the contracted software system, and their subsequent validation and translation into technical specifications that are presented to the system engineers
- Design and Implementation: decision on the organization of the system, technologies used, the physical process used for development, and actual implementation of the system
- Verification and Validation: various actions taken to assure that the software developed meets the requirements outlined, and that it is sufficiently reliable
- Evolution: change of the deployed system to meet evolving client needs

All areas mentioned have been formalized to some degree. The specification phase identifies two sorts of requirements: functional requirements that describe functionality the system must or must not provide and possible required behavior, and non-functional requirements that can generally be viewed as constraints or limitations on the system imposed by resource, timing and process requirements among other things [2, 3, pp. 119, pp. 103-104]. The problem of gathering sufficiently complete and detailed requirements is also acknowledged, and various techniques for eliciting and validating requirements are mentioned in all three sources as well.

The design and implementation phase has a multitude of well-established paradigms for appropriate completion including a variety of "design patterns" that

outline various system architectures. Each supports different functionality to varying degrees, best practices for code organization and architecture, and also has a unique conceptual process describing the steps in development—waterfall, iterative, agile are some examples [2, pp. 64-88]. The evolution process is addressed in literature in a variety of manners, but descriptions concur on at least two empirically-observed laws of software evolution: the law of continuing change, and the law of increasing complexity. The former states that requirements of the software system will perpetually be evolving, creating a challenge for the architects and engineers to adapt a system to requirements for which it was not originally designed to fulfill. This is a significant driving force for the latter observation, which claims that as software persists and is improved it is made increasingly complex [1, 2, pp. 491, pp. 453].

The remainder of this chapter focuses on the verification and validation step of the software process, which has been thoroughly formalized as outlined below.

2.2 The Need for Software Testing

The name "verification and validation" is typically given to the portion of the software development process that includes software testing. The subtle difference between the two components is clearly delineated by Sommerville, who notes that validation is essentially "are we building the right product?" whose answer involves acceptability of the system, and that verification is "are we building the product right?" whose answer is related to verifying that the software does indeed conform to specifications [2, pp. 516]. It is the latter part that encompasses what is commonly called "software

testing", and it is an integral part of software development since some sort of verification of correctness is required due to the complexity of many modern-day software systems.

The goal of software testing is universally recognized to be the efficient identification of errors within a given software program, and in the larger context of the entire verification and validation process, the remediation of those errors. The prevalence of modern software failures brought to light in a public manner are an indicator that there are great benefits to improving the software testing process. Since it is simply unfeasible to test software under all possible internal state configurations and transitions, it is a very difficult if not impossible problem to prove program correctness through current software testing methodologies. This intractability of exhaustive testing of software behavior has led to a near certainty of defects in all modern software systems [1, 2, 3, 4, pp. 539, pp. 397, pp. 205, pp. 9].

In light of this, it is evident that the net benefit of software testing with regards to a deployed system is not the absence of defects but rather a certain degree of assurance that the software system will behave as intended. This leads to various heuristics that are used throughout the software testing process in order to maximize the assurance that the software is indeed sufficiently correct. A main stumbling block for achieving this goal efficiently is what has been widely recognized as a natural tendency towards psychological mindsets that do not facilitate proper software testing. Both [1, pp. 405-406] and [4, pp. 5-6, 8] warn against the mentality that the purpose of testing is to demonstrate that no errors are present, and link this behavior to inefficiency in test case design.

The need for software testing is perhaps best shown through the dedication of time towards software testing practices in real-world projects and the results of its inadequate presence. It was estimated in [5] that time waiting for tests to run accounted for 10—15% of total time of software development, and [6] cites the results of some studies claiming that more than half of all development effort is put into testing and debugging. Additionally [7] justifies *in vivo* testing by noting that even with existing testing efforts, "40% of companies consider insufficient pre-release testing to be a major cause of later production problems".

2.3 Manual Testing

Manual testing of software via human analysis is an integral part of a software engineering process, but the scope of the problem leads to the necessity of other forms of testing as well; it is not economically sound to develop an exclusively manual process for software testing because of the inability to re-use developed work in an automated fashion. However, it is noted in [4, pp. 23] that manual review of the system's code yields benefits due to exact localization of the fault rather than simply its symptoms as is typical in automated testing, and also that experienced software testers find some classes of problems better suited to human analysis while others appear to be more efficiently identified through automated testing.

A widely used form of manual software testing is known as a "code inspection" or "code review", and involves a team of people who review a particular section of program code as a group. While noted that the number of people involved in a code

inspection is not limited in any way, all suggest a team of about four people that fulfill at least the following roles:

- The original author of the piece of code being inspected to answer questions
- Inspectors to review the code
- A moderator to keep the meeting moving
- A scribe to record a list of the defects discovered during the meeting
- A reader to narrate the code being reviewed

[1, 2, 4, pp. 415, pp. 523, pp. 24]. It is generally suggested that formal documents be provided for the code inspection such as a code inspection checklist, which contains common types of programming errors and spaces to note the occurrence of each if present. Identified defects are not fixed during the inspection, and it is the author's responsibility to address the issues found in time outside the meeting [4, pp. 22]

Sommerville includes individual preparation in the code inspection process. The meeting itself is driven by the person fulfilling the role of the reader who narrates the program code and allows for interruptions from inspectors who require clarification or wish to discuss a possible defect. The maximum time for an efficient code review is cited at around 2 hours as the process can become mentally taxing, and figures between 40% and 60% for the percentage of net discovered defects uncovered via code inspections are typical. It is also noted that the psychological approach taken by the code inspectors plays a crucial role in team dynamics as any personal attacks regarding the author's code can easily be internalized, so it is recommended that comments are

directed at the code itself rather than the author in any manner whatsoever. It is also recognized that even comments regarding defects in the code can have negative impacts on the author if the process is improperly approached [1, 2, 4, pp. 414, pp. 522,526-527, pp. 23,26].

[4] notes other types of manual code inspections including "walkthroughs", which are similar to code inspections in terms of the duration and number of people gathered for the walkthrough. However, unlike code inspections, they focus on manual execution of code stemming from simple test cases in order to further understanding of the program's workings as well as find defects. A "desk check" is a term for an individual effort to debug programs, and it is suggested that a person other than the author of the program perform the check. This process may be somewhat structured, for example by requiring that the examiner fill out a checklist of various types of common mistakes, but is generally less effective than full-fledged, team-based forms of manual testing. Finally, peer review of programs by multiple individuals can offer valuable feedback on factors like style and organization as well as offer a well-rounded opinion of the examined code [1, 4, pp. 414, pp. 38-41].

2.4 Black-box Testing

Black-box testing is the process of verifying that the program behaves correctly while treating it as a "black box" so that all examination of the system is done without knowledge of internal implementation. Therefore this type of testing is typically the process of testing input against requirements for the system's associated output that

were gathered in the specification step of the software process [4, pp. 9], or according to the technical documents following from the original specification if testing is carried out at a finer granularity than at the system level. However, as previously noted it is infeasible to test every possible configuration, so a method called "equivalence partitioning" is used in order to maximize the effectiveness of non-exhaustive testing scenarios.

This process breaks ranges of input into various groupings called "equivalence classes" that are likely to have all items in the group behave in the same manner. Using this technique, a minimal number of members of a given equivalence class can likely represent behavior of a majority (or ideally all) of the values in that class, and hence the number of test cases required for reasonable assurance of correctness of the program can be significantly reduced [1, 2, 3, 4, pp. 403, pp. 553, pp. 213, pp. 52-53].

There are no all-encompassing rules for performing equivalence partitioning properly as it depends on the specific piece of software and requires experience to perform well [2, 4, pp. 554, pp. 54-55]. An example of a common equivalence partition would be splitting the possible values for integers into negative and non-negative to catch errors resulting from assumptions based on the sign of a given value. Once these equivalence classes have been established, the technique of "boundary value analysis" is used to make effective use of equivalence classes by exploiting the observation that errors are more common on the values at the bounds of equivalence classes and hence should be more thoroughly tested [2, 4, pp. 554, pp. 59].

Using these techniques is helpful both for finding defects and for assuring that changes to a system do not result in breaking some related functionality. There are

a wide number of tools available to automate the running of tests [2, 4, pp. 563, pp. 120-121], and they are necessary to facilitate the practice of "regression testing", in which tests are re-run to ensure their exercised functionality is not compromised due to changes made to the system. Since running all available tests for every change made to the system may be impractical a subset of tests may be re-run instead, selected by the likelihood of the change impacting the functionality exercised by the tests [1, 4, pp. 411, pp. 147].

2.5 White-box Testing

While black-box testing as described in §2.4 is useful for testing, higher assurance can be achieved by also performing white-box testing (also called "structural testing"). In this case, the software is not treated as an opaque unit, but various analysis techniques requiring access to the program's source code may be carried out in order to test the system more thoroughly. Access to the program's source code during testing allows for expansion on black-box testing by establishing either new or more specific equivalence classes to be used during testing [2, pp. 557].

Various metrics can be used to measure the completeness of a testing infrastructure, and many focus on the extent to which various parts of the code are exercised during testing. The simplest of these is the metric of statement coverage, which measures the number of unique source code statements executed during a full run of tests. While it can give some measure regarding the completeness of a program's testing, it is a relatively weak indicator of program correctness since the construction of trivial

test cases offering full statement coverage that do not expose errors is relatively simple [1, pp. 42]. This is because statement coverage is unable to detect two important classes of errors: those stemming from incorrect values of data, and errors of omission where required functionality is simply not present. In fact, Myers claims "the statement coverage criterion is so weak that it generally is useless" [4, pp. 45]. While this may be true from a defect identification standpoint, it can be used to ensure that more robust testing methods are actually examining a reasonable portion of the supplied program code.

The weakness of simple statement coverage as an indicator of defects can be made more robust by performing "path testing" or "branch coverage" testing, which requires that each possible conditional branch must be be tested so that it is examined as both succeeding and failing [1, 2, 3, 4, pp. 421, pp. 559, pp. 257-260, pp. 46-47]. This is not nearly as strong as the criterion that every possible path through the program should be examined, but the combinatoric explosion of possible cases renders that method unfeasible for any reasonably complex program. Branch coverage can in turn be extended to what is known as "condition/decision coverage", which requires that each part of a boolean expression evaluate to both false and true throughout the course of testing (and likewise not that all possible combinations of predicate values be examined) [1, 4, pp. 422, pp. 49].

While the above are basic concepts behind white-box testing, there are inevitably many other techniques. Chapter 3 describes various methods for automated fault detection and localization that rely on access to program source code in order to function and are hence a form of white-box testing. However the relatively small col-

lection of techniques overviewed above is the common denominator in various books on software engineering and testing, indicating that the field has only been established on a basic level.

2.6 Integration Testing

Abstraction methods allowing compartmentalization of functionality as well as code re-use have made the complexity of modern software somewhat manageable, but problems can arise when various pieces of a system are integrated even if individual units have passed rigorous testing in isolation. This sort of testing is required for any substantial software product, as its assurance of meeting requirements must ultimately be at the system level. There are multiple approaches to integration testing including the well-known strategies of "bottom-up", "top-down", and "big-bang" that are described below along with their various trade-offs.

Before addressing the different styles of integration testing, it is necessary to explain the place of "stubs" and "drivers" in the testing process. In order to test a module, test cases and the associated data must be passed along to it by a driver. Automating the testing process to facilitate regression testing as mentioned in §2.4 is useful when considering integration of components by developing a driver that can be re-used in all stages of testing. In order to thoroughly test functionality of individual modules in isolation it can also be necessary to create small toy implementations of other modules with which the module being tested interfaces. Such implementations are stubs, and the effort expended in creating stubs varies depending on the

integration technique used.

The big-bang method for integration testing is the simultaneous integration of all individual modules into a cohesive unit for testing. While this technique is very straightforward, it can greatly increase the overhead of localizing the cause of defects because of the large number of untested interactions that could be the root cause of the failure [3, 4, pp. 263, pp. 107-108]. Bottom-up integration deals with combining the most basic modules (those not referencing other modules) first and eventually working up towards more and more top-level modules (those not reference d by other modules), whereas top-down integration is the opposite. These two methods are known as "incremental" integration methods since modules are added to the testing base incrementally, making the problem of fault localization that complicates the big-bang approach more manageable.

Myers claims that the big-bang integration style unconditionally requires more overall work, and Sage and Palmer do not treat it as a viable style but rather suggest that the bottom-up integration style typically results in a big-bang style integration strategy in practice [3, 4, pp. 263, pp. 107] (possibly stemming from a dated view of bottom-up integration). However, the details of this disagreement are rendered less important by the fact that some form of incremental method is generally used in practice (Sommerville and Van Vliet only cite examples using incremental techniques)—Sommerville considers the problem of fault localization so significant that it is his main rationale for discussing incremental techniques exclusively, and work as far back as that of Sage and Palmer claims "[big-bang integration] is a very costly and usually unsuccessful approach" [1, 2, 3, 4, pp. 438-439, pp. 541-543, pp. 216,

pp. 109].

Using the top-down approach has the benefit of offering an early preview of the fully-integrated software modules as it begins with top-level modules that are not called by other modules, and works down towards core modules. The specific benefits are varied and include the testing of what are likely the most frequently used module interfaces, allows demonstration to the client, shows architectural flaws earlier, and can be used as a morale boost for the development team since it demonstrates a working system [3, 4, pp. 264, pp. 114]. However, it is apparent that this integration method requires stubs to be created as each module is tested—when a new module is added, it replaces a stub and all stubs it requires to function properly are created. The reliance on stubs can be difficult in testing for many reasons outlined by Myers including the need for multiple stubs per replaced subtree in the module hierarchy to facilitate multiple test cases, as well as the possibility of a test case spanning the functionality multiple stubs [1, 4, pp. 439, pp. 110-112].

Myers makes a specific point of noting that the bottom-up integration scheme's need for drivers presents less of a challenge than the stubs required by the top-down scheme because only a single driver should be needed per module instead of multiple stubs, and that drivers are generally less complex and therefore easier to write. In addition bottom-up integration has the benefit of being able to include I/O modules immediately to facilitate testing, although a top-down integration scheme can be designed to integrate I/O modules early in the process [4, pp. 113, 116-117]. The benefits of bottom-up integration sorely miss the benefits of a functioning system mock-up, and for this reason a combination of the two incremental styles is almost

always used in practice; because of the testing of both bottom and top parts of the module hierarchy at an early stage, this hybrid method is frequently called something similar to "sandwich integration" [1, 2, 3, pp. 439, pp. 541, pp. 265].

2.7 System Testing

System testing can be a somewhat ambiguous term as Sommerville has it encompassing integration testing while the two are considered separately by Myers and Van Vliet, [1, 2, 4, pp. 439, pp. 541, pp. 130]. But here it is viewed as by Myers: a completely disjoint form of testing that focuses solely on aspects of the fully-assembled system in order to make sure it meets the necessary constraints and specifications. Since functional tests for expected behavior are done on inputs and verified with outputs by other testing methods, system testing focuses on differences between system design and implementation and the description of and intent for the system as contracted. From this context, it encompasses a variety of criteria enumerated at length by Myers, with many mentioned by other sources although none have as comprehensive a collection. However, some are redundant in the face of other testing methods, so they are enumerated in a modified, abbreviated form below:

- Volume testing: examination of program behavior while processing relatively high volumes of data in order to overrun possible assumed size boundaries where applicable

- Stress testing: examination of program behavior on high rates of data for processing to discover problems with simultaneous actions, their relative ordering,

etc.

- Usability testing: finding out whether or not the program and its interface are sufficiently useful to end users as a tool for data manipulation

- Security testing: examination of the program under atypical conditions, with atypical input etc. as well as attempts to illicit incorrect behavior using known exploitation techniques

- Performance testing: verification that the program does indeed meet the performance criteria outlined in its specification

[4, pp. 130-142].

Chapter 3

Related Research

Useful software systems produce some sort of output, and it is required that the output be correct to the extent dictated by the application domain. Programs that can produce some sort of incorrect output are deemed defective or "buggy" as they must contain some defects or "bugs" that cause improper output to be generated.

The problem of assessing whether or not an arbitrary software program is buggy is undecidable, as a trivial case where the program does not ever terminate due to its own logic does not produce any output and is an instance of "the halting problem", which has been proven undecidable. Therefore, in order to have reasonable assurance that a software system fulfills its requirements it is necessary to carry out as much testing of the software as allowed before the utility of shipping the software outweighs the perceived risk of remaining defects. The following sections explore some work related to two concepts underlying the work on post-deployment fault localization as elaborated in chapter 4.

3.1 Assisted/Automated Fault Localization

3.1.1 Source Code Analysis

Slicing

Perhaps one of the first techniques proposed for automated analysis of programs was called "program slicing" or more recently "static slicing" to contrast with the closely related concept of "dynamic slicing", and had the goal of automating the debugging process. The idea of program slicing was first described in [8] by Weiser in 1981, and both it and its extensions are explored to the present day. Program slicing requires a "slicing criterion" consisting of a variable value and a line number in the target program's source code, and produces a subset of the original program that contains all statements influencing the value of the slicing criterion. As noted by Weiser, this process is likely similar to that carried out by professional programmers when debugging and altering complex software systems in order to facilitate changes to the software without requiring consideration of an unwieldy amount of program code.

Slicing is implemented by applying data flow analysis to the target program in order to determine exactly which statements can influence the value of the slicing criterion. While readily inferred, two desirable properties of program slices noted explicitly by Weiser are that

1. The slice must be obtained from the original program by statement deletion

2. The behavior of the slice must correspond with the behavior of the original program from the perspective of the slicing criterion when the program terminates

The requirement that the program terminate in the second point above is required for cases such as below in which the behavior of the program cannot possibly be sliced on the value of x at line 7 to properly correspond with its original behavior.

```
1  read x
2  if x is 0
3      then while true
4                 do ▷ no-op
5             x = 1
6      else   x = 2
7  ...
```

Figure 3.1: Illustration of the requirement for program termination in program slicing

The analysis and output of a semantically valid slice of the original program is performed by using its representation as a "flowgraph" consisting of a set of nodes N, a set of edges $E \in N \times N$, and an initial node representing the starting point of the program. This structure can be created by making non-control flow program statements correspond with nodes, making each of their related statements nodes as well, and creating edges from a node a to a node b in order to indicate that the statement represented by b can logically follow the statement represented by a. Weiser uses this graph structure to assist in describing the creation of semantically valid slices as well as to formally define procedures for computing slices. Specifically, deletion of statements during the process of creating a slice corresponds to deleting their nodes in the associated flowgraph and only leaves the program semantically valid when the deleted nodes have a single successor. His algorithm for creating slices is not elaborated here at length, but essentially backtracks from the location of the slicing criterion by including all statements modifying variables that contribute to

the value of the criterion as well as all control flow statements influencing execution of those statements [8].

In contrast to static slicing, dynamic slicing performs similar tasks although on a specific execution instance of a program instead of static analysis on the program source code. Precursors to full-fledged dynamic slicing were explored as early as 1973 in [9], but were formalized by Korel and Laski in 1990 [10] and promptly explored more in [11]. The idea of dynamic program slicing is to output a program slice that more closely corresponds with the minimal slice for the given criterion by computing additional data dependencies that can prune nodes from the graph of the resulting slice by using data from the actual execution instance of the target program.

Using the terminology from [10], the requirement to include any statements possibly contributing to the value of a variable is extended to keeping track of a specific history in a program's execution called a "trajectory". The trajectory of a program is represented as a sequence of nodes in the corresponding flowgraph for the target program as with static slicing, but it is also necessary to differentiate between various executions of the same statement since each can possibly have different data dependencies. Therefore the trajectory must store not only nodes in the graph, but *specific instances* of nodes in the graph corresponding with a specific execution—[11] notes that improvements may be made on the amount of storage actually required to store an execution trajectory, but that improvements are limited because there is no way around keeping at least one state for each unique node and its related dependencies. There is also the requirement to specify external inputs in order to make the program execution instance well-defined, so these values are given along

with the original criterion defined for static slicing.

This execution-dependent information allows for reasonable reduction in the size of a generated slice, but much more additional computation is required to properly prune extraneous nodes. Many of the gains of dynamic slicing arise from the use of conditional and looping constructs available in most programming languages, and are therefore widely applicable. Consider the following example and explanation from [11]

```
 1  read x
 2  if x < 0
       then
 3            y = f1(x)
 4            z = g1(x)
       else
 5            if x is 0
                then
 6                    y = f2(x)
 7                    z = g2(x)
                else
 8                    y = f3(x)
 9                    z = g3(x)
10  write y
11  write z
```

Figure 3.2: Program for showing differences between static and dynamic slices

The static slice of variable y at statement 10 of figure 3.2 can be calculated as follows:

1. Find all possible definitions of the variable y affecting its value at statement 10 (3, 6 and 8)

2. Add all nodes that can lead to these definitions (1, 2 and 5)

so the static slice is $\{1, 2, 3, 5, 6, 8\}$. The dynamic slice for $X = -1$ and the same

location of y is smaller because the only assignment statement actually executed is statement 3, so the dynamic slice is then only $\{1, 2, 3\}$. This shows the gains of dynamic slicing in terms of slice size, but computation can be much more strenuous since the program must actually be executed for the given inputs (long loops magnify this problem) and complexities regarding the storage and processing of data dependencies can arise.

Since the introduction of program slicing there have been many variations and uses related to fault localization outside the obvious applications to highlighting all possible responsible code for an error during debugging. Another type of program slice is specifically targeted towards interoperation with a testing suite by finding all statements labeled as "critical" to a slicing critereon: those leading to the same failure location via mutations of the original program code [6]. There has also been work more closely related to software engineering that attempts to use program slicing combined with requirements documents in order to more effectively localize the most important program faults [12]. These are just two examples of some ways in which program slicing can be applied, and there are certainly many more.

State and Control Flow Analysis

An alternative to program slicing for fault localization deals with analyzing a program's state during execution and/or how that state was achieved, and is closer to the approach as described in chapter 4 for fault localization. The practice of "delta debugging" as developed by Zeller in [13] exercises examination of program state with the goal of increased automation of the debugging process as well.

In this method, differences in variable states between at least one failing and one successful execution are compared and systematically examined in order to locate the cause of failure that must internally stem from some difference between the two runs. Generating these differences depends on creation of a "memory graph": a data structure holding complete information on program state, including pointer values to handle analysis of variable aliasing. Both the failing and successful executions have snapshots taken, and their memory graphs are compared in order to generate a list of differences between the two by using deep comparisons made possible by leveraging type information. Then an increasingly small number of differing variable values are mutated until a subset of the difference is found where no error state in the program is induced.

This process is iterated, and eventually finds the smallest set of changes possible that lead to a failure by closing the gap between minimally-sized known-bad states and maximally-sized known-good states. An example of a known error in the GNU C compiler being identified was presented along with the concept in [13], and pointed to the location of failure but not the direct cause. While it produced very useful information about the defect, it required 44 executions of the program with various state mutations in order to produce the results.

Zeller expands the delta debugging methodology by combining its information about where the failure manifested with information regarding when the state of the program actually became erroneous in [14]. He does this by focusing on what he calls "cause transitions", where the invalid value of a variable effectively taints the value of other variables. He argues that leveraging this method will lead to faster

bug fixes during debugging because it focuses on "good locations for fixes", even though it was still fundamentally unable to locate the defect in the GNU C compiler used in [13] because no comparable state existed between successful and failing runs. This suggests that while similar in spirit to the method outlined in chapter 4, delta debugging is not comparable because of its inability to identify the underlying causes unique to failing executions.

A few other methods encountered while exploring related work on fault localization analyzed the behavior of conditional statements in order to gain information regarding program behavior related to a fault. One even focuses on generating a successful run from an unsuccessful run with the hope that it offers a similar execution, and relaxes the requirement for a similar successful run to make Zeller's work useful [15]. Research described in [16] builds on previous work using special program slices as in [17] to more effectively localize the cause of software failure by switching the values of conditional predicates during execution to glean more information regarding where the error occurred. However, this method is unable to indicate any errors due to data values, which may be detected by Zeller's work as well as the *in vivo* fault localization method outlined in chapter 4.

While the localization of faults is partially automated in [13, 14, 16], all maintain that significant human reasoning (although reasonably reduced) is still necessary to locate the root cause of a failure. The extent to which this can be avoided still remains limited, but there are also drawbacks such as the inability to detect various types of errors. This suggests that these methods are generally less useful than others described in the following sections.

3.1.2 Anomaly Detection

Promising work applying techniques in anomaly detection, the attempt to identify when a program behaves abnormally, looks to be of assistance in fault localization by aiding efforts to tell when and even possibly where errors occur in a program's execution. Both [18] and [19] use a probabilistic model known as a Markov model in order to develop a model of typical program behavior. One proposal for enhancing test suite quality is identifying when new program functionality is exercised while automating the process of changing the model along with the test suite, and another uses anomaly detection in order to identify when a program instance executing in the field runs awry [18, 19].

The concept of a classifier is used in these approaches–a way of separating executions into distinct classes and labeling them as belonging to that class as appropriate. Both methods use first-order Markov models, which create a classifier that essentially claims that the probability of arriving in the current state of the program is only dependent on the state immediately previous. This sort of probabilistic model can be represented as an $n \times n$ matrix for states S having size $|S| = n$ where each row and each column represents a unique state in S, and the entry p_{ij} corresponds to the likelihood that the next state is that represented by j given that the current state is that represented by i. Both methods also use the program's control flow as an indicator of behavior by examining the frequency of each branch taken during execution. In this way it can be considered a dynamic variation on the work in the second part of §3.1.1. However, this approach was put in a separate section as it does

not attempt to use control flow information to estimate *how* the program arrived at a given state.

In [18] Bowring *et al.* create the initial classifier with a customized technique. First, each training execution is associated with its own Markov model, and the individual models are iteratively combined or "clustered" based on their similarity score assessed by a chosen metric; in the case of [18], the scores were simply the magnitude of differences in corresponding positions in a matrix of thresholded entries in the Markov models. For this analysis, each instance is given the label for an execution class that is not necessarily limited to the testing-centric values of "pass" and "fail".

Since the classifier itself is a cluster of Markov models, the probability of each internal model producing the given execution is calculated. If the probability is below a certain threshold the execution instance is labeled as anomalous, and is otherwise given the label corresponding to the most appropriate model in the cluster. The anomalous executions can then be used to re-train the existing clustered model of executions in order to include the previously anomalous behavior if desired. According to [18], one possible application of this method is to improve the efficiency of test suites by identifying which new test cases actually exercise previously unknown program behavior, and can also improve the assurance of test quality by indicating the increasing difficulty with which sufficiently anomalous test cases can be created.

Baah *et al.* [19] generate an initial Markov model from all execution instances based on the value of observed boolean predicates in the program or the fact that they have not been observed, and then proceed to cluster the individual states of the

model based on grouping by either their corresponding line number or the method in which they reside. Finally, an extra state is added to the resulting Markov model that corresponds to all predicate states not included in the generated model. This state corresponds with anomalous behavior, and all of the predicate states falling into this state's representation are all given a uniform, miniscule probability weighting.

Then, a numeric threshold value is defined as

$$\min\left(\frac{P(O_t^{s_i})}{P(O_{t-1}^{s_j})}\right)$$

where $P(O_T^S)$ represents the probability that the current state of predicates O is emitted by the model with the final state S being emitted at time T (this ratio is not necessarily the same value as the raw transition between states s_i and s_j in the corresponding model). Using this value, any execution passing between two states having the execution-time ratio evaluate to a value below the threshold is labeled as an anomalous instance since the model predicts a sufficiently low likelihood that a typical execution instance would cause such a state transition.

This second technique was successfully applied to various versions of a program with known bugs: it correctly identified all executions corresponding to buggy behavior as anomalous in 72% of test cases, and had accuracy of at least 80% in 84% of test cases. Findings also noted that no benefit was observed when comparing the accuracy of method clustering to line number clustering of nodes except for a nearly eleven-fold increase in the time required to build a model using line clustering. It was also claimed but not shown that since this model actively evaluates values by comparing them to a given threshold at the time of program execution, it should facilitate

fault localization. Of course since this analysis method is based solely on control flow of the given program, it is unable to identify anomalous behavior stemming from calculated values unless they significantly influence control flow.

3.1.3 Statistical Debugging

Liblit's work developing the statistical technique of Cooperative Bug Isolation (CBI) was first proposed in his PhD dissertation [20] (which is concisely summarized in [21]), and attempts to apply mathematical analysis to widespread sampling of executing programs with statistical rigor. This sampling is carried out via the insertion of automatically generated boolean predicates as well as the tracking of associated values that can be inferred to form a set of predicates used in statistical analysis. The data collected during execution is simply the number of times a predicate is observed as true during execution during a run of the program along with whether or not the run failed. Using this information, the end goal is to select a small subset of the observed predicates that are statistically likely to be evaluated as true in the occasion of a failure in order to help understand its cause.

Instrumentation is carried out by inserting boolean predicates into code and observing their values during execution. Like *in vivo* testing, this method requires that action be taken during field execution and aims to distribute overhead across many execution instances in order to gather sufficient data regarding defects–this is done by probabilistically executing instrumentation code during execution. However, unlike the implementation of the Invite framework used in [22], Liblit's method addresses optimizations of the instrumentation method extensively.

CBI takes great care to ensure sufficiently random sampling of predicates in a way that minimizes overhead. It is figured that the probability of instrumentation code actually executing in a real-world deployment of the technique would be between $\frac{1}{100}$ and $\frac{1}{1000}$, and therefore the likelihood that any predicate is not assessed for a given run is 99%. Another interesting related statistic is that for n predicates in a given section of code, the probability that none of them executes is $\left(\frac{99}{100}\right)^n$ so that even when $n = 10$ there is more than a 90% chance that no predicate will be evaluated. For this reason, Liblit notes that the predicate evaluations are bernoulli trials and that the number of predicates not evaluated can be represented using a geometrically distributed random number and viewed as a countdown until the next predicate evaluation.

By analyzing a graph representing the control flow of the original program, a weight can be given to each acyclic region (ie. non-recursive functions without loops) representing the maximum number of potential predicate evaluations for that region. When used with the countdown until the next instrumentation execution, the instrumented code can instead skip all instrumentation statements in the acyclic region if the assigned weight is less than the number of encountered predicates until the next evaluation. In this manner Liblit allows for accurate statistical sampling while maintaining performance of the instrumented program.

Analysis of gathered execution data begins with filtering of results by excluding predicates meeting a subset of the following criteria identified by Liblitt:

1. Elimination by universal falsehood: disregard any predicate never observed true

in all runs as it likely represents an impossible condition

2. Elimination by lack of failing coverage: disregard all predicates for an instrumentation site if none of them were encountered since the instrumentation site was likely not explored during execution

3. Elimination by lack of failing example: disregard any predicate not observed true in any failing runs as its value being true is not a likely indicator of failure

4. Elimination by successful counterexample: disregard any predicate observed true in a successful program execution as it can be true without indicating a program failure

What predicates remain after the filtering are ranked based on the discovery of binary classifiers that indicate whether or not a given run will fail, thus suggesting the cause of failure.

While this is essentially solving the same general problem as that considered in the fault localization method described in chapter 4, it is carried out in a different way than the more descriptive but computationally more expensive process for creating classification trees described in §4.3.2. Instead the log likelihood of the predicates' presence in a failing run is found, and the top values are inspected for causes of failure. Given data on program executions $(x_1, y_1), \ldots, (x_M, y_M)$ for $x_i \in \mathbf{R}^n$ being the vector of predicate true-value counts and $y_i \in \{0, 1\}$ denoting the success or failure of the run, the vanilla log likelihood is defined as

$$\sum_{i=1}^{M} \left[y_i \log Pr(Y = 1|x) + (1 - y_i) \log \left(1 - Pr(Y = 1|x)\right) \right]$$

In practice, a weighted version is used in order to force most features towards 0 as there are likely only a few contributing to any given failure. More information on the details are available in [20].

While this method definitely identifies predicates influencing a failure, it is explicitly noted that it only identifies behavior predicting failures rather than a cause for the failure since the outputs are simply the predicates with the highest indication of failure, but not what caused them to have a specific value. For this reason, others have used CBI not as a full-scale solution but rather built upon it in order to assist in fault localization. An example of this is [23], which combines CBI with control flow analysis and machine learning techniques. This is a hybrid process, combining the method of this section and source code analysis, that also uses the following techniques for fault localization:

- Support vector machines (iterative classification models) and Random forests (many decision trees as described in §4.3.2, each for a partition of the observed predicates) [24] to identify predicates critical to execution outcome
- Clustering techniques to identify correlations between predicates in the same executions by using a distance metric to assess differences
- Analysis of the program's control flow graph in order to capture the path of execution leading to the predicate values indicating failure

The addition of control flow analysis appeared to help with the identification of race conditions, but the remainder of the section on the empirical results of the method is rather sparse and similar to results achieved by Liblit.

3.2 Post-deployment Software Testing

This section describes work towards application of software testing techniques to real-world deployed software, and analyzes the feasibility of various proposals. This is the domain of the work described in chapter 4, and includes discussion of the *in vivo* testing concept [22] as well as related projects that address issues in post-deployment testing such as Skoll [25] and Gamma [26].

3.2.1 General Work

Since the task of exercising a program's validity for all possible internal states is infeasible, equivalence partitions of input are leveraged in pre-deployment testing to efficiently exercise a large amount of critical program code. For systems requiring a specific level of assurance of proper operation, the decision to deploy is based on the assumption that the "residual" parts of the program not exercised in existing tests do not contribute in any significant manner to the behavior of the system.

The earliest publication explored in this section relates to work in testing activities of deployed software systems, and focuses on monitoring execution of this residual code. Pavlopoulou and Young [27] apply techniques of software analysis to post-deployment systems in order to gather information regarding the residue of a deployed system, and use the data as a metric of the testing quality of the pre-deployment testing framework. The main concern regarding their implementation is the overhead induced by instrumentation, and they demonstrate progress in its reduction by dynamically reducing the number of instrumentation probes during execution

to only those necessary to detect exploration of residual code. The implementation as discussed is implemented in Java, and is achieved by modifying the bytecode for the target program by inserting the appropriate instrumentation at the entrance to every basic block of code in the program, since all code defaults to being labeled as unexamined.

In order to track coverage, a table mapping uniquely-assigned identifiers to corresponding instrumented code blocks is maintained (and remains consistent across re-compilation) and a second table is also kept to represent the basic blocks not yet executed. Finally, a table mapping the second table onto the first is used to associate the data. This is necessary because the table representing blocks not yet executed changes during execution, and reduces the size of the second table as fewer and fewer blocks remain unexamined in an execution instance as it runs. As measured, this instrumentation adds significant initial overhead—upwards of 100% in code with frequent and/or long loops because the savings of removing instrumentation probes are delayed, but in other types of code full instrumentation overhead can be reduced to around 15%. In both cases, two iterations of testing and re-instrumentation are sufficient to bring overhead down to levels below 2%.

A more engineering-focused approach is proposed in [28], which describes an architecture for verifying that field executions of deployed software conform to its specifications. The proposed system is based on a publish-subscribe model where a centralized (but possibly internally designed as a distributed system) event notification service processes subscription to various data streams called "events" that are published by various execution instances, and handles exporting the event data to

subscribed instances. In this way, the task of verification is distributed over multiple execution instances. Instrumentation of the source code is performed using a custom class loader that can appropriately intercept calls during execution and is driven by an automatic parser of UML descriptions of the system as given in its specification.

While a good deal of work explored so far focuses on the simple feasibility of various techniques, [29] describes an application of random forests to classify program executions as either successful or failing as this information is useful in many remote program analysis techniques [29]. It is immediately obvious that this work is somewhat exploratory, as it specifically aims to address the three following questions:

1. Can we reliably classify program outcomes using execution data?

2. If so, what kinds of execution data should we collect?

3. How can we reduce the runtime data collection overhead?

Experimentation with the method proved that such classification is indeed possible, and examined various aspects of program executions as potential predictors of program behavior.

In fact, the initial attempt at classification proved successful using statement counts (the number of times a block of code is executed) as a valid indicator as they produced error rates of classification near 0%–this suggested that coarsely-collected data could be successfully used for such classification. Along similar lines, it was found that method counts, the same technique applied to methods instead of code blocks, were as good an indicator as statement counts and also had the benefit of being even more coarse hence resulting in less overhead. As the authors were testing Java code,

statistics on throw/catch execution counts were also explored as potential predictors and found to be successful for only one of nineteen faulty programs examined.

3.2.2 Post-deployment Testing

This subsection transitions from automated testing methodologies and early work on distributed testing of post-deployment software systems to the origins and explanation of the *in vivo* approach to software testing in §3.2.3, which is the foundation for the work described in chapter 4. The information below begins with examining software tomography followed by a discussion of work on fundamental properties of distributed post-deployment systems.

In [30], Bowring *et al.* explore the concept of software tomography as it relates to the challenge of distributed software analysis. Software tomography is essentially a technique that is used to divide up a task among many executing software instances, and the authors note that it provides essential services to a distributed system. However, they were not aware of prior work such as [27] when making the claim that work on residual monitoring did not provide functionality for reducing instrumentation sites during execution, a topic that was indeed extensively explored prior to publication. The three main tasks of software tomography as outlined in [30] are as follows:

1. Dividing the desired task into subtasks requiring minimal instrumentation

2. Assigning the subtasks effectively to different instances of the software

3. Integrating the information returned by those instances for comprehensive anal-

ysis

The main contribution of the work is exploration of various "topographic refinement" policies that can be used to effectively distribute instrumentation overhead across many execution instances while eliminating instrumentation sites already exercised from a pool of sites remaining to be examined.

Three techniques for distribution and dynamic reduction of instrumentation code are explored:

- **Simple Refinement**: branches in code are evenly distributed among all instances; each instance is responsible for executing and testing one branch
- **Round-robin Refinement**: instrumentation of covered branches is removed, and all instances instrumenting the removed branches are re-assigned in a round-robin fashion to the yet-uncovered branches; each instance is responsible for one branch
- **Aggregation Refinement**: allows multiple branches to be tested per instance, and is activated when the "refinement ratio" $\frac{\text{branch execution count / time}}{\text{remaining branches}}$ becomes sufficiently low, and is otherwise like Round-robin Refinement

It was found that the problem as explored equated to a reasonable trade-off between completeness/speed of achieving completeness of instrumentation and the amount of interaction between software instances required to successfully carry out each instrumentation method. The **aggregation refinement** method generally performed the best in terms of rapidly approaching full instrumentation with four times the inter-instance communication cost when compared to **round-robin refinement**, which

was slower to gain complete coverage if at all (depending on space partitioning for instances). This method for distribution was cited as core functionality for Gamma System as outlined in [26], which is designed for dynamic, minimally-invasive continuous testing of deployed software in a distributed manner.

Another project with the goal of leveraging distributed deployed software instances is Skoll [25], which focuses on quality assurance testing. Among motivations for using deployed instances in this manner are pressures on production timeframes, and an "explosion of the software configuration space"; the description of the Skoll project asserts that "when increasing configuration space is coupled with shrinking software development resources, it becomes infeasible to handle all QA in-house". In order to accomplish this, it was necessary to develop a way of modeling the configuration space. This was done by pairing every possible configuration option with every possible value except those disallowed by explicit constraints, which could be arbitrarily complex boolean conditions on the configuration parameters' values.

The architecture relies on an "intelligent steering agent" to synchronize the distributed QA work, and offers the ability to hook into the system by providing custom instructions for future behavior based on incoming information. It also offers a great deal of automation to deal with the potentially overwhelming amount of collected information including the identification and analysis of related failures via distance metrics in the configuration space as well as automatic classification of data using classification trees (see §4.3.2). The feasibility and usefulness of Skoll were examined in [25] and many real-world defects in both compilation and execution stemming from its automated testing of possible configurations.

3.2.3 *In Vivo* Testing

The technique of *in vivo* software testing [22, 7] allows adaptation of an existing test suite for execution in a deployed environment in order to assess program correctness in the larger state space examined by field instances of the software as opposed to the smaller state space examined in a pre-deployment testing environment. Both [22] and [7] mention Gamma and Skoll projects as related work, and mention that *in vivo* testing focuses on distribution in order to limit overhead on each individual software instance. They describe *in vivo* testing as the practice of running unit tests during execution of deployed software without modifying the state of the system as a side-effect.

For this reason it is necessary for the system to modify copies of any information used in the test rather than the actual program data itself. In the implementation of the Invite *in vivo* testing framework the copy-on-write characteristics inherent in the forking of new processes in Linux are leveraged to accomplish the separation between the test's state and the state of the executing program; this is implemented in their Java code by using the Java Native Interface (JNI, `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`) to call C's `fork` routine [7].

Tests are specified by prepending "test" to the name of a method to test, and [7] notes that this convention required that methods with multiple tests must internally dispatch a request to run a test to one of the test routines for that method. It is also noted that certain types of unit tests are inherently not applicable for the *in vivo* testing domain, including tests that require a certain state of an object in

order to perform their task. An example of a unit test for a dynamically-sized data structure is given where a unit test begins with an empty data structure, adds an element, removes an element, and finally checks to make sure that the data structure is in fact empty. These tests are then run with a certain probability as specified in a configuration file, which can also be overridden by specified limits on maximum execution/space overhead, etc.

[7] outlines the following five particular classes of defects targeted by *in vivo* testing:

1. Defects remaining in software because unit tests use a reasonably unmodified object for testing purposes

2. Defects arising from the inability to test all valid software configurations before deployment

3. Defects that occur due to legal user action that was not exercised in previous testing

4. Defects from unanticipated (but legal) user actions

5. Defects that occur rarely due to the complex conditions necessary to induce them

The authors take particular interest in the final defect class, and show that this class can readily be identified by the *in vivo* testing methodology. Both [22] and [7] focus heavily on assessing overhead as well as on the ability of the technique to identify various defects, and find that sufficiently low levels of overhead could be achieved

with the probability of running an *in vivo* test at or below 5% while still allowing for extensive testing in medium to large deployment bases.

Directions for further research in the area of *in vivo* testing mentioned in [22, 7] note many opportunities for exploration. First, it is noted that the load distribution mechanisms of Gamma System as described in [26, 30] (and discussed in §3.2.2) could be applied to more effectively distribute the testing task than the specification of a manual, universal probability/overhead metric as currently implemented in Invite. The benefits of integration with the "DejaView" tool described in [31, 32] are also addressed, and it is suggested that the tool can be leveraged to provide robust sandboxing of the *in vivo* test environment as well as maximizing efficiency of filesystem-dependent tests, and possibly network activity and database access if supported by a future version of the tool. Finally, this thesis explores the possibility of automated fault localization in the context of *in vivo* testing by implementing a rudimentary system as elaborated in chapter 4.

3.3 Summary

While techniques for various forms of automated software testing such as program slicing are well-established, their application to a post-deployment environment is suggested to be difficult by the eclectic mix of research and the fundamental nature of the issues addressed in research such as that described in §3.2.2. So far it appears that distributed techniques are the focus of post-deployment research, and that they are driven by the increasing ubiquity of internet connectivity and the widespread dis-

tribution of software applications themselves. This discussion also serves to provide background for work in chapter 4, which attempts to address the problem of fault localization in a post-deployment situation in which failures may be infeasible to reproduce or impossible to debug due to dependence on large amounts of data affecting the program state.

Chapter 4

In Vivo Fault Localization

The complexity of modern software applications combined with existing information on both the quantity and severity of bugs found after deployment of past software systems practically guarantees that almost all software will contain defects at the time it is deployed. For this reason the application of algorithms used in data mining and machine learning to identify the causes of software failures occurring after deployment is explored—a problem where in which it is infeasible for the engineers of the system to carefully inspect the state of the failing execution instance. Initial studies show that with proper instrumentation the cause of multiple types of deterministic defects can be detected: those stemming directly from incorrect changes in variable values as well as those dependent on multiple items occurring in the execution history prior to failure. Furthermore, benefits of the developed system for localizing faults in software system failures may assist in automating the localization of errors before deployment as well.

4.1 Approach

The approach developed to address the problem was specifically designed to work with technologies leveraged in the implementation of the Invite distributed *in vivo* testing framework described in [22] and examined earlier in §3.2.3. The instrumentation for gathering relevant information during program execution is carried out through the use of the AspectJ project for aspect-oriented programming in Java (`http://www.eclipse.org/aspectj/`). Information required for the described implementation of this bug isolation approach is the change in variable values induced by instrumented methods as well as method signatures. This information is currently gathered by tracking the changes in variable values caused by execution of a member method of a class that contains instrumented fields, and storing a record of value differences along with a method name in a list containing as many items as allowed by a configurable bounding length.

Using the standard Invite paradigm, tests are specified by pre-pending a method name with a special prefix that is looked up during program execution to see if a test exists for the currently executing method. These testing methods return a boolean value of `true` if the test failed, or `false` otherwise. After execution of an *in vivo* test, the information collected regarding the tracked state changes of variables is transmitted to a database server with a specific schema and table name as defined by the listing in §A.9.

When a sufficient quantity of execution data has been collected, an automated script pulls information from the database and translates it into (currently a some-

what inefficient representation in) the `arff` data format used by the machine learning toolkit Weka (http://www.cs.waikato.ac.nz/ml/weka/). This file is then passed though Weka's implementation of machine learning algorithms to locate the cause of failure. The two Weka utilities explored were J48 classification trees (based on the C4.5 algorithm, discussed in §4.3.2 [33]) and the Apriori algorithm (discussed in §4.3.3) [34] for their straightforward generation of production rules that can clearly indicate the cause of a failure. The results section (§4.5) only references the Apriori algorithm's output as it was clearer than examining the J48 classification trees, although the trees could have easily been converted into association rules [33] and produced intuitive output.

4.2 Motivating Example

Consider the Java code in figure 4.1 that removes instances of an object x from an internal data structure *ds* starting from the supplied index *start*, and removing any of the next *count* objects if they are equal to x. There is a potential problem with the code if `getVal1() + getVal2(x,y)` overflows the value of the **int** type, resulting in a negative number so the loop never executes, and hence preventing objects' removal from an internal data structure *ds* as intended. This type of defect would be difficult to test comprehensively because it would require a very careful code inspection. This is because the calculated value depends on two other functions, `getVal1` and `getVal2`, that can each be arbitrarily complex and therefore may be difficult to coerce into producing violating output themselves. Furthermore, it is more than plausible that

the cause of such a defect would require more time to correct than a more obvious error, making it a relatively costly fix without automation since programmer time is generally orders of magnitude more expensive than computer time.

```
public int removeFrom(Object x, int start)
{
  ...
  int count = getVal1() + getVal2(x,y);
  ...
  for (int i = start; i < count+start; i++)
        if ( ds.get(i).equals(x) )
                ds.remove(i);
  ...
}
```

Figure 4.1: Motivating example code for *in vivo* fault localization

Assuming a proper implementation of an analogous `containsFrom` method, the following *in vivo* test could identify the defect

```
/**
 * @return true on failure, false otherwise
 */
public boolean testDoRemove(Object x, int start)
{
    if ( containsFrom(x, start) )
    {
        removeFrom(x, start);
        return containsFrom(x, start);
    }
    return false;
}
```

Figure 4.2: *In vivo* test that can be used to identify overflow errors in figure 4.1

Upon completion of the *in vivo* test, tracked information regarding execution history would be transmitted to a central server. If changes to the member variable *ds* are

tracked along with the signatures of methods performing the changes to state, the pattern of failure could be more easily recognized by machine learning algorithms due to the distinct lack of calls to the `ds.remove` method that contrast a failed invocation with a successful one. Extensions to the initial method described here could also be added to track values of loop boundaries and other interesting values to allow for even more accurate fault localization. Finally, in the context of [22, 7], this example defect can be interpreted as belonging to either the first or final class of defects targeted by *in vivo* testing.

4.3 Tools and Algorithms

The various tools and algorithms leveraged in the implementation of the described fault localization technique are described in this section so that they may be adequately explained to facilitate a comprehensive understanding of the method. However, coverage of the topics remain somewhat brief as only the core concepts and uses are explained ([2] dedicates an entire chapter to aspect-oriented programming).

4.3.1 Aspect-oriented Programming

Aspect-oriented programming offers an abstraction called an "aspect" to elegantly handle "cross-cutting concerns"—functionality that is required in multiple places, but either not belonging to or even not able to function in any existing abstraction model. Sommerville notes two specific undesirable characteristics of code that implements cross-cutting concerns in a conventional manner: tangling and scattering. Tangling is

simply the mixing of two pieces of functionality (such as synchronization of operations on a primitive), and scattering is the widespread use of similar code throughout various logical sections of a system such as code present for logging purposes [2].

There are many straightforward examples of plausible functionality that is well-suited for aspect-oriented implementation, and it generally deals with a sequence of actions that are both required and pervasive but not logical to put in a function, subroutine, or method of its own. For example, consider a class in the object-oriented paradigm that manages an object such as an account holding an internal value or balance as well as other necessary properties. Suppose that adherence to legal regulations requires that certain conditions are always maintained, perhaps regarding the status of the account holder relative to the status of the account itself. In this case, proper encapsulation practices can contain the necessary code to the account class, but the checks must still be scattered and tangled with various methods implementing transactions on the account in order to verify that no transaction causes the regulatory conditions to be violated.

Without aspect-oriented programming, two problems arise since every method implementing an account transaction must perform certain checks:

1. Altering the nature of the checks may require updates in all places the checks take place

2. Addition of new checks requires performing the work of finding all necessary points at which to insert the new check

These both lead to increases in the complexity of the class' code and introduce de-

pendencies that render the software system less flexible in the face of future changes. However, the proper definition of one aspect can contain the code implementing the required functionality in a single location and hence facilitate modifications to the cross-cutting code in a substantially simplified manner.

Aspect-oriented tools can be implemented in a variety of ways including both dynamically at runtime [35] as well as at compilation time as with AspectJ. Tools like AspectJ can also possess the same desirable qualities as various other constructs for creating software systems such as type safety, and can also offer the power of pattern-matching to alter code at very specific points and interact with parameters and return values; these qualities were recognized early in the evolution of the paradigm [36]. Due to the fact that aspect-oriented code can be properly "woven" with existing code at compile time and that checks can generally be performed quickly because of the ability to maintain type safety, aspect-oriented functionality can be implemented with somewhat minimal overhead.

Terminology related to aspect-oriented programming includes the following:

- Concern: a cross-cutting piece of functionality to implement with aspect-oriented code
- Advice: the code implementing a concern
- Aspect: a unit of abstraction that contains implementation for various concerns
- Join Point: a point in the execution of a program at which advice is to be inserted/executed

- Pointcut: a statement that assists in specifying a join point

[2].

In AspectJ, pointcuts can be specified on various events including method calls, object construction, field access and mutation, in control flow of specified constructs and more. In addition, AspectJ offers the ability to reference the context of a pointcut such as the type of object receiving a message as well as choosing when advice is executed relative to a pointcut. To simplify the semantic specification of join points, pointcuts can be declared, re-used and composed using the boolean conjunction, disjunction and negation operations. Finally, AspectJ offers basic pattern-matching functionality in order to make it easier to specify join points. Information on the full functionality of AspectJ can be found in the online manual at http://www.eclipse.org/aspectj/doc/released/progguide/index.html or in books on AspectJ programming such as [37], and examples of specific AspectJ syntax used in the implementation of the proposed fault localization method can be seen in §4.4.1 as well as in the aspect-oriented code generated for instrumentation in §A.5.

An example of early criticism of aspect-oriented programming is found in [38], where it is claimed that AspectJ does not provide logical modularization for interdependencies relating to integration and event-based programming. However no indisputable argument is formed, and it rather appears that a problematic architecture is chosen to illustrate a point. In fact, the third proposed alternative architecture to the problematic original can be used in conjunction with AspectJ tools to elegantly fix

its stated problem of redundant synchronization by making it aspect-oriented itself as suggested by Sommerville.

Both this and the space given to the topic by Sommerville suggest that the promise of production-grade aspect-oriented code is more plausible now through better understanding and refinement of aspect-oriented concepts. In fact, [39] specifically sets out to show the scale at which aspect-oriented programming can be applied. However, technology such as the AspectJ project as used in this thesis is still relatively immature, as significant concerns are prominently addressed in the posted FAQ page for the project at the time of writing (`http://www.eclipse.org/aspectj/doc/released/faq.php#q:productplans`).

4.3.2 The C4.5 Algorithm

The construction of J48 classification trees was the first method explored in attempting to derive the source of failure from gathered data with *in vivo* fault localization method, and is based on the C4.5 algorithm. This algorithm is known as a "classifier" since its purpose is to divide items in its data set accurately into groups or "classes" depending on the data's characteristics, and does so by creating a "classification tree" or "decision tree" defined by Quinlan as follows: a structure that is either

- a *leaf*, indicating a class, or
- a *decision node* that specifies some test to be carried out on a single attribute value [of the data], with one branch and corresponding subtree for each possible outcome of the test

(intending of course that subtrees are recursively defined to be either a leaf or a decision node)—all information regarding this algorithm is taken from [33]. In order to examine which of the considered data fields account for a specific instance being classified as belonging to a given class, one can trace the path from the leaf node representing the final classification back to the root of the classification tree, noting the decision made at each decision node. The following information is a summary of the algorithm's mathematical basis.

Quinlan mentions the same fundamentals from the field of information underlying his ID3 algorithm in his description of C4.5: that the amount of information conveyed by identifying a piece of data as part of a class for class C_j and set of data S is

$$-log_2\left(\frac{frequency(C_j, S)}{|S|}\right) \text{ bits}$$

He notes this quantity in the measure of what information theory terms "entropy"

$$info(S) = -\sum_{j=1}^{k}\left[\frac{frequency(C_j, S)}{|S|} \times log_2\left(\frac{frequency(C_j, S)}{|S|}\right)\right] \text{ bits}$$

and states that when applied to a set of training data used to construct the decision tree T this quantity represents the average amount of information needed to identify the class corresponding to an element of T. He then uses the expected value of the training data partitioned using decision test X

$$info_X(T) = \sum_{i=1}^{n}\frac{|T_i|}{|T|} \times info(T_i)$$

to define a term measuring the amount of information gained due to partitioning by X

$$gain(X) = info(T) - info_X(T)$$

Improvement on the above measure for utility in classification as used in the ID3 algorithm adjusts for a bias towards sets with a high number of individual outcomes, as the case of a unique identifier for each item of test data has a maximal gain metric because $info_X(T) = 0$. Instead, the C4.5 algorithm takes the value

$$\textit{split info}(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \log_2 \left(\frac{|T_i|}{|T|} \right)$$

into account, which represents the amount of information potentially generated by partitioning testing data T into partitions T_i using decision test X. Finally, the amount of information useful for classification generated by the partitioning of T by X is

$$\textit{gain ratio}(X) = \frac{\textit{gain}(X)}{\textit{split info(X)}}$$

Since finding the simplest decision tree for a set of training data is NP-complete, the above method is typically used as a heuristic in a greedy manner to construct a decision tree that has a reasonable likelihood of being satisfactorily optimal.

4.3.3 The Apriori Algorithm

The Apriori algorithm is designed to process large quantities of data where items are grouped together by an association called a transaction—a test result in the case of fault localization for *in vivo* testing. The output of the algorithm is a list of association rules of the form $X \Longrightarrow Y$ (defined formally below) and associated metrics calculated for relative ranking of the association rules. The original paper on the algorithm is the source for all information in this section [34].

Formally, given a collection of distinct data items $\mathcal{X} = \{i_1, i_2, \ldots, i_m\}$ and a database of transactions under consideration $\mathcal{D}$ where each transaction $\mathcal{T} \in \mathcal{D}$ is a set of items such that $\mathcal{T} \subseteq \mathcal{X}$, the output of the algorithm is a set of association rules of the form $X \Longrightarrow Y$ where $X \subset \mathcal{X}$, $Y \subset \mathcal{X}$ and $X \cap Y = \emptyset$. In order to efficiently compute relevant association rules, two metrics are defined: support and confidence. The support of an association rule is the percentage of transactions in $\mathcal{D}$ that contain $X \cup Y$, and the confidence of an association rule is $support(X \cup Y)/support(X)$, or the percent of transactions in $\mathcal{D}$ containing X that also contain Y. While somewhat similar, these metrics are definitely distinct. It may help to think of confidence as the strength that the presence of X indicates the presence of Y, and the support to be a measure of the overall prevalence of association rules that subsume the given rule.

The idea of the algorithm is to generate all association rules meeting a user-supplied minimum value constraint for support (*minsup*), and threshold sets of items that are labeled as candidates for inclusion in results by minimum values for other metrics like confidence (*minconf*). This is accomplished by making a number of passes over the data, each time enumerating sets of items ("itemsets") with increasing cardinality that meet the criteria established by *minsup* ("large itemsets") and other parameters like *minconf*. More specifically, the benefit of the Apriori algorithm over alternatives is that it efficiently prunes the state space considered in future iterations based on the realization that all subsets of a large item set are large itemsets as well since they trivially meet the *minsup* requirement (observation 1).

Pseudocode for the algorithm is quite concise and is given in figure 4.3, with n-itemsets being sets of items with n elements, and line 11 possibly augmented by

```
APRIORI(D)
L_1 = {large 1-itemsets}
for (k = 2; L_{k-1} ≠ ∅; k++)
      do
          C_k = APRIORI-GEN(L_{k-1}) ▷ new candidates
          for transaction t ∈ D
                do
                    C_t = SUBSET(C_k, t)
                    for candidate c ∈ C_t ▷ candidate items in t
                          do
                              c.count++
          L_k = {c ∈ C_k | c.count ≥ minsup}
return ⋃_k L_k
```

Figure 4.3: Pseudocode for the Apriori Algorithm (see also figure 4.4)

```
APRIORI-GEN(L_{k-1})
insert into C_k
select p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}
from L_{k-1} p, L_{k-1} q
where p.item_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}
for itemset c ∈ C_k
      do
          for k − 1-subsets s of c
                do
                    if s ∉ L_{k-1}
                       then delete c from C_k
return C_k
```

Figure 4.4: Subroutine used in the Apriori Algorithm

additional constraints such as the *minconf* constraint, etc. This algorithm uses a subset enumeration routine SUBSET (not listed) as well as a routine for generating candidate large k-itemsets, APRIORI-GEN, as shown in figure 4.4. The latter generates large itemsets of cardinality k (L_k) by first getting supersets of L_{k-1} by joining it with itself, and then eliminating all itemsets not having all subsets absent from L_{k-1} as

they cannot possibly be large k-itemsets by observation 1. In other words, "the join is equivalent to extending L_{k-1} with all possible items and then deleting those itemsets for which the $(k-1)$-itemset obtained by deleting the $(k-1)$th item is not in L_{k-1}". Because the Apriori algorithm begins with all large 1-itemsets and all large itemsets of cardinality $k-1$ are present before calling the APRIORI-GEN routine, all large itemsets are eventually generated.

4.4 Implementation

The following sections describe in detail how the fault localization method was implemented. The first topic addressed is how aspect-oriented code is generated given fields of classes to instrument, and is followed by a discussion of how the data is collected in the database and later extracted and processed to indicate the cause of failure.

4.4.1 Code Generation

Concept

The code in the `generation/` folder of the associated source code as listed in appendix A contains the code for automatically generating the AspectJ instrumentation code for the system given the variables to be instrumented. `generation/ajgen-noinvite.pl` is a perl script that will generate the code on execution. The sections of string data used in the script are in a separate file `generation/stringdata.pm`. The `generation/ajgen-noinvite.pl` script allows the user to specify the location for

the generated file, and will ideally allow specification of values to instrument when invoked (values are currently hard-coded, see §4.7 for other thoughts on other future improvements).

Interesting functionality of this step includes automatic insertion of static data fields for instrumentation into those classes to be instrumented, as well as the appropriate generation of pointcuts and advice to instrument the specified fields. All of this is accomplished with the AspectJ implementation of the aspect-oriented programming paradigm. The current code is able to handle the specification of arbitrary class and member names, and little additional work should be required to accept classes/fields to track from command line arguments or a configuration file.

The generated code assumes it will be placed in the `com.invite.core` package of a deployment and also assumes that simple Java classes for tracking value changes exists in the `com.invite.changetracking` package. This instrumentation code consists of the following classes:

- `DiffValue.java`: stores differences between two versions of a value
- `DiffBundle.java`: a collection of `DiffValue`s that are collected in a `DiffChain`
- `DiffChain.java`: a chain of `DiffBundle`s associated with a method invocation

Currently the generated code only instruments primitive types, and objects in a generic manner. The generated code is aspect-oriented, so it is woven with the code being instrumented by the AspectJ compiler and carries out its instrumentation as the program executes, sending data to a central database along with the results of *in vivo* tests.

Detail/Example

The following are examples of code generated for the `IntFields` class instrumented to gather data, and are used to give the reader an idea of the structure of the generated code. Source for the `IntFields` class can be found in §A.4.

The aspect-oriented "pointcuts" below are generated once in the instrumentation code:

- A pointcut matching all methods not named following the Invite convention for test functions

  ```
  pointcut notTestMethod() :
  !call( boolean *.inviteTest () );
  ```

- A pointcut matching all code that does not assist with the instrumentation implementation

  ```
  pointcut notInvite() :
  !within( com.invite.core.* )
  &&  !within( com.invite.changetracking.* );
  ```

These pointcuts are used in the generated code so that the advice used to instrument the target source code is not itself advised, causing infinite recursion. The generated AspectJ code has the following fields per instrumented class:

- A static vector of `DiffChain` objects with each entry corresponding to an object being monitored

- A static, incrementing ID field used to track objects via *O(1)* lookup by its correspondence with an index in the previously mentioned vector of `DiffChain` objects

- A field to hold the value of the ID assigned to a given object instance

These fields are leveraged in the implementation so that every time an object of some type that is to be tracked is generated, it is assigned an ID corresponding to an unused index in the static vector for its type class, and its initial state after creation is noted for use in the storage of difference values on subsequent changes.

The following aspect-oriented pointcuts are generated for each instrumented class, with the examples here generated for the `IntFields` class as listed in §A.4:

- A pointcut describing initialization routines to intercept:

```
pointcut objectCreation( IntFields x ) :
initialization( new( .. ) )
&& target( x )
&& !cflow( execution( * IntFields.clone () ) );
```

- A pointcut describing method calls to intercept (see §4.7 for issues relating to completeness of instrumentation):

```
pointcut IntFieldsCall( IntFields x ) :
call( * * ( .. ) )
&& !call( * clone () )
&& !call( static * * ( .. ) )
&& target( x );
```

As stated previously, the initialization routine takes action adding the new object instance to the static vector inserted into its type class and setting its initial values for difference tracking. The reason these pointcuts exclude the `clone` method is because it is used internally to create copies of objects so the originals are not modified during the *in vivo* test—making them advised by code in Invite would cause infinite recursion. Also, the focus in this simple proof-of-concept test focused solely on member functions in order to work with Invite. As a result static methods were excluded in the `IntFieldsCall` pointcut even though action could easily have been taken on a modification of a field using existing functionality in AspectJ.

Finally, code is dynamically generated to implement the tracking of variable values inside an object instance (the below example is for a primitive type with the name "field1"):

```
// generate these cases based on members
//   being tracked
if ( newValue.field1 != oldValue.field1 )
  diffVals.add( new DiffValue( oldValue.field1 ,
                               newValue.field1 ,
                               ``field1'' ,
                               DiffValue.INT_TYPE ) );
```

Figure 4.5: AspectJ code for updating values in a `DiffChain`

Integration with the Invite framework initially seemed immensely beneficial to this project as it would demonstrate integration with a tool already targeted towards post-deployment testing. It turned out that the version of Invite code originally used in the project leveraged the Java Native Interface (JNI) for making calls to C code from within Java programs in order to do things like set processor affinity to assess parallel performance. This would have been great functionality to retain and would have allowed for interesting measurements regarding the overhead on multi-core processors, but unfortunately it had to be removed because the JNI code prevented the use of network connections (which seemed quite odd for a piece of software that was targeted towards distributed testing).

4.4.2 Data Gathering and Analysis

The gathering of execution data is relatively simple. It requires an active and accessible database server as specified in the connection information variables at the top

of the generated instrumentation AspectJ code, and the AspectJ compiler to weave the aspect-oriented code with standard Java code. The resulting bytecode can be executed with any Java 1.5-compliant JVM, and relevant information regarding the instrumented fields of classes during the test run is automatically transmitted to the database.

Data gathered is extracted from the database for analysis by the Weka machine learning toolkit using the `make-arff.pl` script, which requires the name of the output file and the length of difference chain desired for extraction as parameters. One main issue with the use of classification trees and production rule systems for localization is that when the unaltered, "vanilla" versions of the algorithms are used, the algorithm may pick up that simply using the boolean field indicating whether or not a program failed to be a sufficiently accurate approach rather than actually discriminating based on variable values or method names during classification or rule generation. This could be avoided by modifying the algorithms to avoid such patterns or by formatting the data in another manner, but this has not been implemented due to time constraints and limited personal experience with machine learning algorithms.

Instead the `make-arff.pl` script is currently set up to generate a file with information that is perfectly balanced with respect to failed and successful test data. This ensures that failed cases are included in the generated file so that any useful classifier is easily identified, rather than the analysis algorithm simply using whether or not a test failed as the sole classifier because of a potentially large fraction of cases where tests pass.

The most meaningful results so far are from applying the Apriori algorithm

to collected data. More information regarding how the Apriori algorithm was of assistance in analyzing the data is in §4.5.

4.5 Initial Results

The following sections describe initial results of work on two toy cases that demonstrate the initial feasibility of the approach.

4.5.1 Toy Case 1

This first toy case was a simple check of a boolean member variable *testBool* of an instrumentation class called `IntFields`. The class consists solely of integer fields, accessors and mutators and has a lone *in vivo* test that re-sets the failure variable *testBool* before actually returning a value indicating the result of the test so that it can be placed in a tight loop for automated testing.

```
public boolean inviteTestDecrement() {
        if ( testBool ) {
                testBool = false;
                return true;
        }
        return false;
}
```

Figure 4.6: *In vivo* test for Toy Case 1

This test was exercised by putting calls to an `IntFields` instance inside a loop that probabilistically set the *testBool* variable using the `setTestBool` method to induce a failure of the *in vivo* test, and then called either increment or decrement

with probability $\frac{1}{2}$. When the data was passed through the Apriori algorithm various production rules were produced that suggest both success and failure cases. All useful rules appeared in the top 10 results ranked by confidence, with the ones capturing the more comprehensive rules having the maximum confidence value of 1—ties were broken with the support metric. The highest-ranking association rule was

```
method2_name=IntFields.setTestBool(boolean) ==> succeeded=false
```

indicating that the test failed every time the `testBool` method was called immediately prior.

4.5.2 Toy Case 2

The second of the toy cases was designed to see if the method could correctly identify a multi-cause failure. The two tests in the code listing of figure 4.7 were implemented in the `IntFields` class. These tests are such that a failure would be reported if the `decrement` method was called two or more times directly prior to the execution of the *in vivo* test. Once again, an `IntFields` instance had its `increment` and `decrement` methods called with equal probability inside a tight loop.

When the data was extracted from the database and passed though Weka's Apriori implementation, many rules with a confidence metric of 1 were returned in the top 10 results. Examination of rules with prior knowledge regarding the cause of the failure indeed shows they make sense, but the most useful was the ninth result:

```
method1_name=IntFields.increment(), method2_name=IntFields.decrement()
    ==> succeeded=true
```

Additional time likely would have shown other ways to observe the failure such as tracking the values of *field1* directly.

```
public void increment() {
        oldField1 = field1;
        field1++;
}

// should fail if decrement was called
//     at least twice in a row just prior
//     to failure
public boolean inviteTestDecrement() {
         if ( oldField1 - field1 >= 2 )
                return true;
         return false;
}

public void decrement() {
        field1--;
}
```

Figure 4.7: Code from `IntFields` class for Toy Case 2

4.5.3 Instrumentation Overhead

Since the implementation was originally designed to work with the Invite *in vivo* testing framework [22], it was hoped that the overhead in terms of perceived execution time of the program would be made acceptable by running *in vivo* tests as well as the result reporting on a separate, more available processor core than the majority of the main application. Unfortunately, this impact was not measured as the project became stand-alone after problems relating to network connectivity arose when integrating with the code for Invite. However initial measurement of both aspect-oriented and reflection-based instrumentation was carried out at the beginning of the experimentation phase.

Table 4.1 summarizes the results of various forms of data manipulation being performed in a tight 100,000 iteration loop in which an instance of the `IntFields`

Table 4.1: Execution time for various instrumentation methods

Method	Average time
No instrumentation, plain	0.005s
No instrumentation, copying	0.226s
Instrumented, copying	0.433s
Instrumented, diffing	0.673s
Instrumented, reflection	7.281s

class (source for a version of this class may be found in §A.4) was altered at each iteration. The statistics indicated are measured in seconds and are averages over four trial runs. Instrumented versions are those that included a form of variable value history gathered either via reflection or aspect-oriented code instrumentation. The case using reflection is indicated as such, and the other two cases of instrumented observation are simple copying of information and calculation of differences that are stored in the `DiffChain` container as described in §4.4.1. All of these methods take significantly longer to complete the loop's execution than versions of the loop lacking any instrumentation. In addition to the described instrumentation methods and no instrumentation, there are also measurements for plain calls to copying methods that do not store data in a list structure.

As expected prior to testing, instrumentation methods using the standard Java reflection API were an order of magnitude slower than any other method, and three orders of magnitude slower than performing the loop without any instrumentation

whatsoever. As can be seen, copying of the altered information in the `IntFields` instance every loop iteration increased the cost of the loop by 45 times. However, this is essentially a worst-case scenario because the time to copy an entire object is much larger than that required to increment an integer field, and these actions comprised the majority of execution time and therefore magnified this factor. Furthermore the overhead is essentially minimal in the sense that *in vivo* tests must be completely isolated from the original program to avoid side effects, which necessitates copying.

The overhead of 91% between the two copying instrumentations can be attributed to insertion into a linked list data structure, necessitating even more allocation of memory-resident data. Finally, the differencing operation and storage in the `DiffChain` data structure roughly doubled this overhead. This data indicates that efficiency is indeed an issue that needs more exploration in order to facilitate *in vivo* fault localization that is acceptable to users. But as suggested in [22] a relatively small user base is required in order to execute a sufficient number of tests and bring the amortized overhead of tests over all instances down to an acceptable level. This suggests that even the 50% increase in overhead as measured for a single instance could be distributed across a large installation base of the software.

4.5.4 Apache POI

Code from the Apache POI project was configured for instrumentation using the described method. Code changes were minimal, and only included the implementation of member functions *clone* and *equals* inherited from class `Object`. Due to time restrictions, full results gathering and analysis was not completed during the imple-

mentation phase. Future work can examine the same class of bugs in more complex systems to ascertain the effectiveness of this method as applied to real-world software scenarios. However, the ease with which the code was analyzed and augmented for instrumentation suggests that design for this type of testing would indeed be trivial since it only required a few simple utility functions to implement.

4.6 Discussion

4.6.1 Technique Analysis

Results based on the toy systems seem promising as the Apriori algorithm successfully produced clear association rules showing the exact cause of the defect within the top ten results for each implementation. However, the lower position of the revealing association rule in the second toy example suggests that metrics other than support may potentially be used more effectively as a secondary indicator of rule relevance.

The first obvious extrapolation of this method is to explore instrumentation for values in code, other than member variables, that may be strong indicators of defect presence and may not even require extensive tracking of internal program state (but rather only temporary values calculated from it such as loop iterators). Secondly, no tests have yet been carried out to measure the effectiveness of the described localization method in the presence of multiple failure cases, but it is hypothesized that this would simply lower the confidence and support metrics for defects while still revealing patterns for failed tests.

Although the current implementation is stand-alone, it can easily be integrated

with the Invite *in vivo* testing framework from [22] because it uses the same concepts and technologies. There is currently no parallelism in the model due to complications with using the Invite framework as provided, but the theoretical overhead of this method is that of Invite plus the cost of maintaining a `DiffChain` of object states when a test is run. This cost can vary greatly depending on the quantity of underlying data actually representing a snapshotted member, but grows linearly with member size and test frequency. This is because each field of primitive type in an object is copied and then compared against the value in the most recently stored copy of the object during each *in vivo* test.

Despite the linear growth, the fact that action is taken for every method modifying the object should make the overhead noticeable. However if only some information on instrumented classes is required the cost can be reduced, and the ability to parallelize the problem can take advantage of multi-core processors since *in vivo* tests are strictly isolated from program execution and therefore do not have any external data dependencies. Recent industry moves towards multiple processing cores suggest that the practice of spending time storing a chain of differences of tracked fields at the time of snapshot is preferable to storing a chain of copies, since the overall power available for parallel tasks may quickly become more than can currently be utilized.

The cost in space for the implementation is configurable–currently users may specify a maximum length for `DiffChains` that hold an object's modification history, and this is carried out in such a way that it could be changed dynamically during execution in response to system load if desired in future work. Additionally, the

ability to store full object copies with the goal of saving computation time is another of many possibilities for adjusting resource usage to a given execution environment.

4.6.2 Relation to Other work

While the overhead of this *in vivo* fault localization method is significant and may inhibit its use in time-critical situations, it appears reasonable when compared to Zeller's work on delta debugging[13, 40] which, of all related work explored in chapter 3, achieves the most similar results. This is because the delta debugging technique amounts to a binary search in program executions, which requires a standard case of a logarithmic bound on the *number of (possibly altered) program executions* rather than a constant amount of overhead per tracked item per test pair during execution as with the method described here.

The amount of information collected also gives the benefit of easing the search for the root cause of failure. The strategy in [16] relies only on values observed at branching points in execution and therefore cannot identify the cause of faults that are a result of any specific data values of variables, but rather only by their influence on branching behavior. Zeller's work [13, 14] has a similar problem with localizing faults because states of succeeding and failing executions may simply not be directly comparable and therefore not yield any results.

Finally, Liblitt's technique [20] has much lower overhead, and can essentially be viewed as an alternate version of this technique insomuch as that it trades a good amount of localization ability for speed of instrumentation and a higher proportion of offline processing time to yield results; Liblitt explicitly notes that "we do not

give strict causes and effects". Another significant difference between the approach described here and Liblitt's work is automation of testing. Whereas this method still relies on instrumentation constructed by humans (even if possibly re-used from standard pre-deployment testing practices), Liblitt's method is more automated. Liblitt also notes that in his technique "sampled data is terribly incomplete", suggesting that human intuition may offer a great deal of assistance in tackling this problem.

4.7 Future Work

Parse tree analysis of the code being instrumented would lead to both more comprehensive and more efficient implementation of the fault localization process described. The following optimizations could be implemented using information encoded in the parse tree of the code to instrument:

- Only transmit information relating to variables referenced in the function
- Handle access of public/protected, non-final variables (ie. in classes other than those indicated for instrumentation) as well as the appropriate static member functions if aspect-oriented applications were unable to handle such constructs
- Only execute instrumentation code for methods altering instrumented variables or in special user-defined cases to allow finer control over processing overhead and information gathering

The final method can almost certainly produce noticeable speedups in the proposed fault localization method as it currently looks for an *in vivo* test to run on every

method call, and hence each method call, whether instrumented or not, incurs overhead. This can be remedied by dynamically generating the pointcut criteria for the running of *in vivo* tests based on parse tree analysis so that all overhead of looking for a test method to run is completely eliminated.

Another improvement on the implemented methodology is with regard to the tracking of changed values. The current implementation only handles changes in primitive types, and translates a difference between objects (due to a `false` return value from the `equals` method) into a constant numeric difference. This limitation can be overcome by implementing a facility for specifying new methods for comparison when complex types need to be tracked, as well as by a standard interface for defining how to perform the differencing of internal state.

It is also worth noting that the current implementation of `arff` file generation only extracts method names and associated failure values from the database. This is because of limited time for implementation, and lossless translation into `arff` format is slightly more complex (although feasible). Expanding this functionality would likely result in identification of applications for this localization technique extending to the values of variables and other indicators of internal program state.

In vivo fault localization could also be used in conjunction with a technology called PODs [31, 32], which facilitates efficient snapshotting of a running system as well as replay of execution and branching. By helping to locate the cause of a failure in the presence of otherwise unmanageable quantities of captured execution/error data, *in vivo* fault localization would likely be indispensable in many cases. Finally, there is also the issue of privacy that has been neglected throughout the exploration

of this topic. Privacy must be taken into account for any real-world implementation of this concept since values that may hold sensitive information are transmitted to a centralized database store, but satisfying this requirement is beyond the scope of this thesis.

Bibliography

[1] H. van Vliet, *Software Engineering: Principles and Practice*, 2nd ed. John Wiley & Sons, Inc., 2000.

[2] I. Sommerville, *Software Engineering*, 8th ed. Pearson Education Limited, 2007.

[3] A. P. Sage and J. D. Palmer, *Software Systems Engineering.* John Wiley & Sons, Inc., 1990.

[4] G. J. Myers, *The Art of Software Testing.* John Wiley & Sons, Inc., 2004.

[5] D. Saff and M. Ernst, "Reducing wasted development time via continuous testing," *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pp. 281–292, Nov. 2003.

[6] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis.* New York, NY, USA: ACM, 1996, pp. 121–134.

[7] C. Murphy, G. Kaiser, and M. Chu, "The in vivo approach to testing software applications," *Columbia University Dept. of Computer Science*

Tech Report, 2008. [Online]. Available: http://www.columbia.edu/~cdm6/in-vivo-testing-ISSTA.pdf

[8] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[9] B. O. Aygun, "Dynamic-analysis of execution: possibilities, techniques and problems." Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 1974.

[10] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, vol. 13, no. 3, pp. 187 – 195, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-48TD2NC-8F/2/13c7ee43c4f0611a08bab6f8be792804

[11] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1990, pp. 246–256.

[12] J. Sun, Z. Li, J. Ni, and F. Yin, "Software fault localization based on testing requirement and program slice," *Networking, Architecture, and Storage, 2007. NAS 2007. International Conference on*, pp. 168–176, July 2007.

[13] A. Zeller, "Isolating cause-effect chains from computer programs," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[14] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE '05: Proceedings of the 27th international conference on Software engineering.* New York, NY, USA: ACM, 2005, pp. 342–351.

[15] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* New York, NY, USA: ACM, 2005, pp. 347–351.

[16] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE '06: Proceedings of the 28th international conference on Software engineering.* New York, NY, USA: ACM, 2006, pp. 272–281.

[17] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* New York, NY, USA: ACM, 2005, pp. 263–272.

[18] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis.* New York, NY, USA: ACM, 2004, pp. 195–205.

[19] G. K. Baah, A. Gray, and M. J. Harrold, "On-line anomaly detection of deployed software: a statistical machine learning approach," in *SOQUA '06: Proceedings*

of the 3rd international workshop on Software quality assurance. New York, NY, USA: ACM, 2006, pp. 70–77.

[20] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California, Berkeley, 2004.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.* New York, NY, USA: ACM, 2003, pp. 141–154.

[22] M. Chu, C. Murphy, and G. Kaiser, "Distributed in vivo testing of software applications," *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 509–512, April 2008.

[23] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* New York, NY, USA: ACM, 2007, pp. 184–193.

[24] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[25] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan, "Skoll: distributed continuous quality assurance," *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pp. 459–468, May 2004.

[26] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: continuous evolution of software after deployment," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2002, pp. 65–69.

[27] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pp. 277–284, 1999.

[28] L. Naslavsky, R. S. Filho, C. de Souza, M. Dias, D. Richardson, and D. Redmiles, "Distributed expectation-driven residual testing," *IEE Seminar Digests*, vol. 915, pp. 45–49, 2004.

[29] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying classification techniques to remotely-collected program execution data," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 146–155.

[30] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 1, pp. 2–9, 2003.

[31] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh, "Dejaview: a personal virtual computer recorder," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 279–292.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: a system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 361–376, 2002.

[33] R. J. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[34] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Research Report RJ 9839.* San Jose, California: IBM Almaden Research Center, June 1994.

[35] J. Baker and W. Hsieh, "Runtime aspect weaving through metaprogramming," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2002, pp. 86–95.

[36] R. Lämmel, "A semantical approach to method-call interception," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2002, pp. 41–55.

[37] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, 2003.

[38] K. Sullivan, L. Gu, and Y. Cai, "Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2002, pp. 19–26.

[39] U. Hohenstein, R. Meunier, and C. Schwanninger, "An aspect-oriented implementation of the ejb3.0 persistence concept," in *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software.* New York, NY, USA: ACM, 2007, p. 4.

[40] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE '05: Proceedings of the 27th international conference on Software engineering.* New York, NY, USA: ACM, 2005, pp. 342–351.

Appendix A

Source Code

A.1 com/invite/changetracking/DiffValue.java

```
package com.invite.changetracking;

public class DiffValue {
        public static final int NO_TYPE                 = -1;
        public static final int BYTE_TYPE               = 0;
        public static final int SHORT_TYPE              = 1;
        public static final int INT_TYPE                = 2;
        public static final int LONG_TYPE               = 3;
        public static final int FLOAT_TYPE              = 4;
        public static final int DOUBLE_TYPE             = 5;
        public static final int BOOLEAN_TYPE            = 6;
        public static final int CHAR_TYPE               = 7;

        protected int type;
        protected Object oldValue;
        protected Object newValue;
        protected String fieldName;

        public DiffValue( Object oldValue, Object newValue, String fieldName, int type ) {
                this.oldValue = oldValue;
                this.newValue = newValue;
                this.fieldName = fieldName;
                this.type = type;
        }

        public int getType() {
                return type;
        }

        public Object getNewValue() {
                return newValue;
        }

        public Object getOldValue() {
                return oldValue;
        }

        public String getFieldName() {
                return fieldName;
        }

        public String toString() {
                String typeName = "";
                switch ( type ) {
                        case BYTE_TYPE: typeName = "Byte"; break;
                        case SHORT_TYPE: typeName = "Short"; break;
                        case INT_TYPE: typeName = "Integer"; break;
                        case LONG_TYPE: typeName = "Long"; break;
                        case FLOAT_TYPE: typeName = "Float"; break;
                        case DOUBLE_TYPE: typeName = "Double"; break;
                        case BOOLEAN_TYPE: typeName = "Boolean"; break;
                        case CHAR_TYPE: typeName = "Character"; break;
                        default: assert( false );
                }
                return fieldName + "(" + typeName + "): " + oldValue.toString() + "->" + newValue.
                    toString();
        }
}
```

A.2 com/invite/changetracking/DiffBundle.java

```
package com.invite.changetracking;

public class DiffBundle {
        String methodName;
        DiffValue[] diffVals;

        DiffBundle( String methodName, DiffValue[] diffVals ) {
                this.methodName = methodName;
                this.diffVals = diffVals;
        }

        public String getMethodName() {
                return methodName;
        }

        public DiffValue[] getDiffVals() {
                return diffVals;
        }
}
```

A.3 com/invite/changetracking/DiffChain.java

```
package com.invite.changetracking;

import java.util.LinkedList;

public class DiffChain<T> {
        private T currentVersion;
        private LinkedList<DiffBundle> diffHistory;

        public DiffChain( T o ) {
                currentVersion = o;
                diffHistory = new LinkedList<DiffBundle>();
        }

        // d is the array of changes made by the current method call
        //   maxSize is passed in so all generated code is contained to the AspectJ aspect,
        //   perhaps allow it to change based on system resources
        public void update( String methodName, T newVersion, DiffValue[] d, int maxSize )
        {
                currentVersion = newVersion;
                diffHistory.addFirst( new DiffBundle( methodName, d ) );
                while ( diffHistory.size() > maxSize )
                        diffHistory.removeLast();
        }

        public T getCurrentVersion() {
                return currentVersion;
        }

        public LinkedList<DiffBundle> getDiffHistory() {
                return (LinkedList<DiffBundle>)( diffHistory.clone() );
        }

        public void deleteHistory() {
                 diffHistory.clear();
                 currentVersion = null;
        }

        public String toString() {
                StringBuffer retBuf = new StringBuffer( currentVersion.toString() + ": < " );

                int i = 0;
                for ( DiffBundle bundle : diffHistory ) {
                        retBuf.append( "{" );
                        for ( int j = 0; j < bundle.diffVals.length; j++ ) {
                                if ( j != bundle.diffVals.length - 1 )
                                        retBuf.append( bundle.diffVals[ j ].toString() + ", " );
                                else
                                        retBuf.append( bundle.diffVals[ j ].toString() + " " );
                        }

                        if ( i != diffHistory.size() - 1 )
                                retBuf.append( "}, " );
                        else
                                retBuf.append( "} >" );
                        i++;
                }

                return retBuf.toString();
        }
}
```

A.4 com/invite/drivers/IntFields.java

This is the version as used to carry out the tasks as described in §4.5.2

```java
package com.invite.drivers;

public class IntFields {
        private int field1;
        private int field2;
        private int field3;
        private int field4;

        private int oldField1;

        private Integer field5;
        private Integer field6;

        private boolean testBool;

        public IntFields( int a, int b, int c, int d ) {
                field1 = a;
                field2 = b;
                field3 = c;
                field4 = d;

                field5 = new Integer( a );
                field6 = new Integer( b );

                testBool = false;
        }

        public void increment() {
                oldField1 = field1;
                field1++;
                field2++;
                field3++;
                field4++;
        }

        // should fail if decrement was called at least twice in a row
        public boolean inviteTestDecrement() {
                 if ( oldField1 - field1 >= 2 )
                         return true;
                 return false;
        }

        public void decrement() {
                field1--;
                field2--;
                field3--;
                field4--;
        }

        public void setTestBool( boolean b ) {
                testBool = b;
        }

        public static String myStaticMethod() {
                return "static string";
        }

        public IntFields clone() {
                return new IntFields( field1, field2, field3, field4 );
        }

        public String toString() {
                return "[ " + Integer.toString(field1) + ", " + Integer.toString(field2) + ", " +
                                Integer.toString(field3) + ", " + Integer.toString( field4 ) + "]"
                                        ;
        }
}
```

A.5 com/invite/core/Invite.aj (generated for IntFields)

```
/*
 * Invite.aj
 * Generated on Wed Nov  5 07:49:36 2008
 *
 * Copyright Columbia University 2008
 * Generator written by Del Slane - djs2160
 */

package com.invite.core;


import java.sql.*;
import java.util.*;
import java.lang.reflect.*;

import com.invite.changetracking.*;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

import java.sql.*;
import java.util.*;

import com.invite.changetracking.*;
import com.invite.drivers.*;


public privileged aspect Invite {

        static final String dbURL = "jdbc:mysql://localhost/invite";
        static final String dbUser = "root";
        static final String dbPassword = "rootpass";

        // stores the names of all methods that don't have a corresponding unit test
        static TreeSet noTest = new TreeSet();

        public static final int DEFAULT_OBJ_VEC_SIZE = 50;// allocate this size vector per type
        public static final int MAX_CHAIN_LEN = 30; // max size of diff chains

        pointcut notTestMethod() : !call( boolean *.inviteTest () );
        pointcut notInvite() :
                        !within( com.invite.core.* )
                &&      !within( com.invite.changetracking.* );
        pointcut notInit() :
                        !initialization( *.new ( .. ) )
                &&      !preinitialization( *.new ( .. ) )
                &&      !handler( * );
/********   INSTRUMENTATION FOR CLASS IntFields   ********/

        // Fields for class IntFields
        private static int IntFields.nextId = 0; // set on object creation, is index in vector
            below
        private static Vector<DiffChain<IntFields>> IntFields.trackedObjects =
                        new Vector<DiffChain<IntFields>>( DEFAULT_OBJ_VEC_SIZE );
        private int IntFields.tracerAspectObjId; // add the ID field to each class instance
        // Pointcuts for class IntFields
        pointcut objectCreation( IntFields x ) :
                initialization( new( .. ) )
        &&      target( x )
        &&      !cflow( execution( * IntFields.copy () ) );

        pointcut IntFieldsCall( IntFields x ) :
                call( * * ( .. ) )
        &&      !call( * copy () )
        &&      !call( * toString () ) // remove with print statments
        &&      !call( static * * ( .. ) ) // don't need to track static method calls
        &&      target( x );

        // Joinpoints for class IntFields
        after ( IntFields x ) : objectCreation( x ) && notInvite() && notTestMethod()
        {
                x.tracerAspectObjId = IntFields.nextId;
                IntFields.nextId++;
                IntFields.trackedObjects.add( x.tracerAspectObjId, new DiffChain<IntFields>( x.
                    copy() ) );
        }

        after( IntFields newValue ) : IntFieldsCall( newValue ) && notInvite() && notTestMethod()
        {
                Vector<DiffValue> diffVals = new Vector<DiffValue>( 10, 5 );
                DiffChain<IntFields> trackingChain = IntFields.trackedObjects.get( newValue.
                    tracerAspectObjId );
                IntFields oldValue = trackingChain.getCurrentVersion();

                // generate these cases based on members
```

```
                if ( newValue.field1 != oldValue.field1 )
                        diffVals.add( new DiffValue( oldValue.field1, newValue.field1, "field1",
                             DiffValue.INT_TYPE ) );

                if ( newValue.field3 != oldValue.field3 )
                        diffVals.add( new DiffValue( oldValue.field3, newValue.field3, "field3",
                             DiffValue.INT_TYPE ) );


                DiffValue[] a = new DiffValue[ diffVals.size() ];
                trackingChain.update( thisJoinPointStaticPart.getSignature().toString(), newValue.
                     copy(), diffVals.toArray( a ), MAX_CHAIN_LEN );

                //System.out.println( "after '" + thisJoinPointStaticPart.getSignature() + "': " )
                     ;
                //System.out.println( "\t" + trackingChain );
                //System.out.println();
        }

        // automatically generate pointcuts in other files that access package/public members

        // instruments methods returning objects
        Object around() :
                        notInvite()
                &&      notTestMethod()
                &&      !cflow( within( Invite ) && adviceexecution() )
                &&      !execution( static * * (..) )
                &&      execution( * * (..) )
        {
                // fixme: make this parallelizable
                Object ret = proceed();
                Object target = thisJoinPoint.getTarget();

                String invokedName = thisJoinPoint.getSignature().getName();
                String fullInvokedName = target.getClass().getName() + "." + invokedName;

                if ( noTest.contains( fullInvokedName ) )
                        return ret;

                String firstChar = invokedName.substring(0, 1).toUpperCase();
                String rest = invokedName.substring(1, invokedName.length());
                String testMethodName = "inviteTest" + firstChar + rest;

                try {
                        Method testMethod = target.getClass().getMethod( testMethodName, null );
                        boolean testPassed  = ((Boolean)testMethod.invoke( target, null ) ).
                             booleanValue();
                        reportFailedTest( target, testPassed );
                }
                catch ( NoSuchMethodException e ) {
                        noTest.add( fullInvokedName );
                }
                catch ( Exception e ) {
                        e.printStackTrace();
                        System.exit( 0 );
                }
                finally {
                        return ret;
                }
        }

        static void reportFailedTest( Object target, boolean failed )
        {
                try {
                        Field trackedObjectsField = target.getClass().getField( "
                             ajc$interField$com_invite_core_Invite$trackedObjects" );
                        Field tracerObjIdField = target.getClass().getField( "
                             ajc$interField$com_invite_core_Invite$tracerAspectObjId" );

                        if ( !trackedObjectsField.isAccessible() )
                                trackedObjectsField.setAccessible( true );
                        if ( !tracerObjIdField.isAccessible() )
                                tracerObjIdField.setAccessible( true );

                        Vector<DiffChain<?>> trackedObjects = (Vector<DiffChain<?>>)
                                ( trackedObjectsField.get( target ) );
                        int objId = ((Integer)tracerObjIdField.get( target )).intValue();

                        if ( objId < trackedObjects.size() ) // compensate for calling joinpoint
                             executing before existence
                                sendInfoToDb( trackedObjects.get( objId ), failed );
                }
                catch ( NoSuchFieldException e ) {
                        System.out.println( "Test failure report failed: " + e.getMessage() );
                        e.printStackTrace();
                }
                catch ( SecurityException e ) {
                        System.out.println( "Test failure report failed: " + e.getMessage() );
                        e.printStackTrace();
                }
                catch ( IllegalArgumentException e ) {
                        System.out.println( "Test failure report failed: " + e.getMessage() );
```

```
                        e.printStackTrace();
                }
                catch ( IllegalAccessException e ) {
                        System.out.println( "Test failure report failed: " + e.getMessage() );
                        e.printStackTrace();
                }
        }

        static private void sendInfoToDb( DiffChain<?> chain, boolean failed ) {
                String objType = chain.getCurrentVersion().getClass().toString();

                try {
                        // set up database connection, can make this a pool later
                        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        Connection dbConnection = DriverManager.getConnection( dbURL, dbUser,
                            dbPassword );

                        // use transactions
                        dbConnection.setAutoCommit( false );

                        PreparedStatement getLastId = dbConnection.prepareStatement( "SELECT
                            LAST_INSERT_ID()" );
                        PreparedStatement infoInsert = dbConnection.prepareStatement( "INSERT INTO
                             diffinfo " +
                                        "(tid, object_type, method_name) VALUES (?, ?, ?)" );
                        PreparedStatement valsInsert = dbConnection.prepareStatement( "INSERT INTO
                             diffvals " +
                                        "(field_name, field_value, field_change, did) VALUES (?,
                                            ?, ?, ?)" );

                        PreparedStatement makeTrial = dbConnection.prepareStatement( "INSERT INTO
                            trials (failed) VALUES (?)" );
                        if ( failed )
                                makeTrial.setInt( 1, 1 );
                        else
                                makeTrial.setInt( 1, 0 );
                        makeTrial.executeUpdate();

                        ResultSet rs = getLastId.executeQuery();
                        if ( !rs.next() )
                                throw new SQLException( "No element in result set for last ID" );
                        int tid = rs.getInt( 1 );

                        for ( DiffBundle bundle : chain.getDiffHistory() ) {
                                infoInsert.setInt( 1, tid );
                                infoInsert.setString( 2, objType );
                                infoInsert.setString( 3, bundle.getMethodName() );
                                infoInsert.executeUpdate();

                                rs = getLastId.executeQuery();
                                if ( !rs.next() )
                                        throw new SQLException( "No element in result set" );
                                int did = rs.getInt( 1 );

                                for ( DiffValue value : bundle.getDiffVals() ) {
                                        valsInsert.setString( 1, value.getFieldName() );
                                        valsInsert.setInt( 4, did );
                                        setFinalValInsertParams( value, valsInsert );
                                        valsInsert.executeUpdate();
                                }
                        }

                        dbConnection.commit(); // commit as a transaction
                }
                catch ( SQLException e ) {
                        System.out.println( "SQLException in sendInfoToDb: " + e.getMessage() );
                }
                catch ( Exception e ) {
                        System.out.println( "Error loading MySQL JDBC driver: " + e.getMessage() )
                            ;
                }
        }

        static private void setFinalValInsertParams( DiffValue value, PreparedStatement valsInsert
            )
                        throws SQLException
        {
                switch ( value.getType() ) {
                        case DiffValue.BYTE_TYPE:
                                Byte oldValue = (Byte)value.getOldValue();
                                Byte newValue = (Byte)value.getNewValue();
                                valsInsert.setDouble( 2, newValue.doubleValue() );
                                valsInsert.setDouble( 3, newValue.doubleValue() - oldValue.
                                    doubleValue() );
                                break;
                        case DiffValue.SHORT_TYPE:
                                Short oldValueS = (Short)value.getOldValue();
                                Short newValueS = (Short)value.getNewValue();
                                valsInsert.setDouble( 2, newValueS.doubleValue() );
                                valsInsert.setDouble( 3, newValueS.doubleValue() - oldValueS.
                                    doubleValue() );
                                break;
```

```
                        case DiffValue.INT_TYPE:
                            Integer oldValueI = (Integer)value.getOldValue();
                            Integer newValueI = (Integer)value.getNewValue();
                            valsInsert.setDouble( 2, newValueI.doubleValue() );
                            valsInsert.setDouble( 3, newValueI.doubleValue() - oldValueI.
                                doubleValue() );
                            break;
                        case DiffValue.LONG_TYPE:
                            Long oldValueL = (Long)value.getOldValue();
                            Long newValueL = (Long)value.getNewValue();
                            valsInsert.setDouble( 2, newValueL.doubleValue() );
                            valsInsert.setDouble( 3, newValueL.doubleValue() - oldValueL.
                                doubleValue() );
                            break;
                        case DiffValue.FLOAT_TYPE:
                            Float oldValueF = (Float)value.getOldValue();
                            Float newValueF = (Float)value.getNewValue();
                            valsInsert.setDouble( 2, newValueF.doubleValue() );
                            valsInsert.setDouble( 3, newValueF.doubleValue() - oldValueF.
                                doubleValue() );
                            break;
                        case DiffValue.DOUBLE_TYPE:
                            Double oldValueD = (Double)value.getOldValue();
                            Double newValueD = (Double)value.getNewValue();
                            valsInsert.setDouble( 2, newValueD.doubleValue() );
                            valsInsert.setDouble( 3, newValueD.doubleValue() - oldValueD.
                                doubleValue() );
                            break;
                        case DiffValue.BOOLEAN_TYPE:
                            Boolean oldValueB = (Boolean)value.getOldValue();
                            Boolean newValueB = (Boolean)value.getNewValue();
                            if ( oldValueB.booleanValue() ) {
                                    valsInsert.setDouble( 2, 1.0 );
                                    if ( newValueB.booleanValue() )
                                            valsInsert.setDouble( 3, 0.0 );
                                    else
                                            valsInsert.setDouble( 3, 1.0 );
                            }
                            else {
                                    valsInsert.setDouble( 2, 0.0 );
                                    if ( newValueB.booleanValue() )
                                            valsInsert.setDouble( 3, 1.0 );
                                    else
                                            valsInsert.setDouble( 3, 0.0 );
                            }
                            break;
                        case DiffValue.CHAR_TYPE:
                            Integer oldValueC = new Integer( Character.getNumericValue( (
                                Character)value.getOldValue() ) );
                            Integer newValueC = new Integer( Character.getNumericValue( (
                                Character)value.getNewValue() ) );
                            valsInsert.setDouble( 2, newValueC.doubleValue() );
                            valsInsert.setDouble( 3, newValueC.doubleValue() - oldValueC.
                                doubleValue() );
                            break;
                        default:
                            assert ( false );
                    }
            }

    }
```

A.6 generation/ajgen.pl

```
#!/usr/bin/perl

require 'stringdata.pm';

use Data::Dumper;

if ( $#ARGV == 0 ) {
        die "no classes to snapshot\n";
}
elsif ( $#ARGV == -1 ) {
        die "Usage: aj-gen2 <outfile> <class to trace> ..";
}

open OUTFILE, ">", $ARGV[ 0 ] or die "Could not open output file '" . $ARGV[ 0 ] . "' for writing\
    n";

open INVITE, "<", "../com/invite/core/InsertTest.java.template" or die "Cannot find required file
    InsertTest.java.template\n";

#make these set-able via switches
my $default_obj_vec_size = 50;
my $config_file = "/home/del/Desktop/thesis/tracing/com/invite/core/config";
my $max_chain_len = 30;

my $class_info = get_class_info();

print OUTFILE "package com.invite.core;\n\n";
print OUTFILE "import com.invite.changetracking.*;\n";

# get rid of this and figure out a permantent system for putting instrumented code here
print OUTFILE "import com.invite.drivers.*;\n";


# -- THE BELOW WILL BE IMPLEMENTED WHEN CLASSES ARE STORED AS FULLY-QUALIFIED
#    (also get info for methods accessing members directly)
#print import statements for all instrumented classes
#for my $class ( keys %$class_info ) {
#       print OUTFILE "import $class.*;\n";
#}


while ( <INVITE> ) {
        #s/configFile = ".*"/configFile = "$config_file"/;
        s/Object around(): demoExecs()/Object around(): demoExecs() && notInvite()/;
        s/static final int MAX_CHAIN_LEN  .*/static final int MAX_CHAIN_LEN = $max_chain_len;/;

        print OUTFILE $_;

        if ( /static final void println/ ) {
                print OUTFILE sprintf(
                                        $Stringdata::general_use,
                                        $default_obj_vec_size
                                    );
                print_tracer_aspects( $class_info );
        }

        if ( /\/\*\* insert failure reporting here \*\// ) {
                insert_tracing_logic();
        }
}

sub get_class_info {
        my $i;
        my $classes = {
                IntFields => { field1 => 'INT_TYPE', field3 => 'INT_TYPE', testBool => '
                    BOOLEAN_TYPE' },
        };

        return $classes;
}

#call with a hash of data on what to track as outlined in a previous comment
sub print_tracer_aspects {
        my $class_info = shift;

        foreach my $class ( keys %$class_info ) {
                print OUTFILE "/********   INSTRUMENTATION FOR CLASS $class   ********/\n\n";
                print OUTFILE "\t// Fields for class $class\n";
                print OUTFILE sprintf(
                                        $Stringdata::fields_per_class,
                                        $class,
                                        $class,
                                        $class,
                                        $class,
                                        $class
                                    );
```

```
                print OUTFILE "\t// Pointcuts for class $class\n";
                print OUTFILE sprintf(
                                $Stringdata::pointcuts_per_class,
                                $class,
                                $class,
                                $class,
                                $class
                              );

                my @members;
                while ( my ( $field, $type ) = each ( %{ $class_info->{$class} } ) ) {
                        push @members, sprintf(
                                        $Stringdata::field_checks_per_class,
                                        $field,
                                        $field,
                                        $field,
                                        $field,
                                        $field,
                                        $type
                                       );
                }

                my $member_equality_checks =  join( "\n", @members );

                print OUTFILE "\n\t// Joinpoints for class $class\n";
                print OUTFILE sprintf(
                                $Stringdata::joinpoints_per_class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $class,
                                $member_equality_checks
                              );
        }
}

sub insert_tracing_logic {
        my $tracing_logic = "\t\t/** Huzzah, tracing logic abound! **/\n";
        print OUTFILE $tracing_logic;
}
```

A.7 generation/stringdata.pm

```
#!/usr/bin/perl

package Stringdata;


our $header = <<ML;
/*
 * Invite.aj
 * Generated on %s
 *
 * Copyright Columbia University 2008
 * Generator written by Del Slane - djs2160
 */

package com.invite.core;


import java.sql.*;
import java.util.*;
import java.lang.reflect.*;

import com.invite.changetracking.*;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.CodeSignature;

%s

public privileged aspect Invite {

        static final String dbURL = "jdbc:mysql://localhost/invite";
        static final String dbUser = "root";
        static final String dbPassword = "rootpass";

        // stores the names of all methods that don't have a corresponding unit test
        static TreeSet noTest = new TreeSet();

        public static final int DEFAULT_OBJ_VEC_SIZE = %s;// allocate this size vector per type
        public static final int MAX_CHAIN_LEN = %s; // max size of diff chains

        pointcut notTestMethod() : !call( boolean *.inviteTest () );
        pointcut notInvite() :
                        !within( com.invite.core.* )
                &&      !within( com.invite.changetracking.* );
        pointcut notInit() :
                        !initialization( *.new ( .. ) )
                &&      !preinitialization( *.new ( .. ) )
                &&      !handler( * );
ML

our $fields_per_class = <<ML;
        private static int %s.nextId = 0; // set on object creation, is index in vector below
        private static Vector<DiffChain<%s>> %s.trackedObjects =
                        new Vector<DiffChain<%s>>( DEFAULT_OBJ_VEC_SIZE );
        private int %s.tracerAspectObjId; // add the ID field to each class instance
ML

our $generic_object_tracking_field = <<ML;
        private int %s.%sValue = 0; // add field to track object values as type GENERIC_OBJECT
ML

# Final pointcut is to avoid tracking while in the testing.  The nature of the final pointcut
#   means the user must manually call copy methods etc. inside the test to not disturb instance
#   variables - make code easy/clear, and can be warned via static analysis later
our $pointcuts_per_class = <<ML;
        pointcut objectCreation( %s x ) :
                initialization( new( .. ) )
        &&      target( x )
        &&      !cflow( execution( * %s.clone () ) );

        pointcut %sCall( %s x ) :
                call( * * ( .. ) )
        &&      !call( * clone () )
        &&      !call( * toString () ) // remove with print statments
        &&      !call( static * * ( .. ) ) // don't need to track static method calls
        &&      target( x );
ML


our $joinpoints_per_class = <<ML;
        after ( %s x ) : objectCreation( x ) && notInvite() && notTestMethod()
        {
                x.tracerAspectObjId = %s.nextId;
                %s.nextId++;
                %s.trackedObjects.add( x.tracerAspectObjId, new DiffChain<%s>( x.clone() ) );
        }

        after( %s newValue ) : %sCall( newValue ) && notInvite() && notTestMethod()
```

```
        {
                Vector<DiffValue> diffVals = new Vector<DiffValue>( 10, 5 );
                DiffChain<%s> trackingChain = %s.trackedObjects.get( newValue.tracerAspectObjId );
                %s oldValue = trackingChain.getCurrentVersion();

                // these cases based on members specified
%s

                DiffValue[] a = new DiffValue[ diffVals.size() ];
                trackingChain.update( thisJoinPointStaticPart.getSignature().toString(), newValue.
                    clone(), diffVals.toArray( a ), MAX_CHAIN_LEN );

                //System.out.println( "after '" + thisJoinPointStaticPart.getSignature() + "': " )
                    ;
                //System.out.println( "\\t" + trackingChain );
                //System.out.println();
        }

        // automatically generate pointcuts in other files that access package/public members
ML


# inserted per field to track for each class being processed, adds changed values to a diff value
our $field_checks_per_class = <<ML;
                if ( newValue.%s != oldValue.%s )
                        diffVals.add( new DiffValue( oldValue.%s, newValue.%s, "%s", DiffValue.%s
                            ) );
ML

our $object_checks_per_class = <<ML;
                if ( ! newValue.equals( oldValue ) )
                        diffVals.add( new DiffValue( %s.%sValue, ++%s.%sValue, "%s", DiffValue.
                            INT_TYPE ) );
ML

# the code for failure reporting and the aspect's closing brace
our $footer = <<ML;

        // instruments methods returning objects
        Object around() :
                        notInvite()
                &&      notTestMethod()
                &&      !cflow( within( Invite ) && adviceexecution() )
                &&      !execution( static * * (..) )
                &&      execution( * * (.. ) )
        {
                // fixme: make this parallelizable
                Object ret = proceed();
                Object target = thisJoinPoint.getTarget();

                String invokedName = thisJoinPoint.getSignature().getName();
                String fullInvokedName = target.getClass().getName() + "." + invokedName;

                if ( noTest.contains( fullInvokedName ) )
                        return ret;

                String firstChar = invokedName.substring(0, 1).toUpperCase();
                String rest = invokedName.substring(1, invokedName.length());
                String testMethodName = "inviteTest" + firstChar + rest;

                try {
                        Method testMethod = target.getClass().getMethod( testMethodName, null );
                        boolean testPassed  = ((Boolean)testMethod.invoke( target, null ) ).
                            booleanValue();
                        reportFailedTest( target, testPassed );
                }
                catch ( NoSuchMethodException e ) {
                        noTest.add( fullInvokedName );
                }
                catch ( Exception e ) {
                        e.printStackTrace();
                        System.exit( 0 );
                }
                finally {
                        return ret;
                }
        }

        static void reportFailedTest( Object target, boolean failed )
        {
                try {
                        Field trackedObjectsField = target.getClass().getField( "ajc\$interField\
                            $com_invite_core_Invite\$trackedObjects" );
                        Field tracerObjIdField = target.getClass().getField( "ajc\$interField\
                            $com_invite_core_Invite\$tracerAspectObjId" );

                        if ( !trackedObjectsField.isAccessible() )
                                trackedObjectsField.setAccessible( true );
                        if ( !tracerObjIdField.isAccessible() )
                                tracerObjIdField.setAccessible( true );

                        Vector<DiffChain<?>> trackedObjects = (Vector<DiffChain<?>>)
```

```
                        ( trackedObjectsField.get( target ) );
                int objId = ((Integer)tracerObjIdField.get( target )).intValue();

                if ( objId < trackedObjects.size() ) // compensate for calling joinpoint
                    executing before existence
                        sendInfoToDb( trackedObjects.get( objId ), failed );
            }
            catch ( NoSuchFieldException e ) {
                System.out.println( "Test failure report failed: " + e.getMessage() );
                e.printStackTrace();
            }
            catch ( SecurityException e ) {
                System.out.println( "Test failure report failed: " + e.getMessage() );
                e.printStackTrace();
            }
            catch ( IllegalArgumentException e ) {
                System.out.println( "Test failure report failed: " + e.getMessage() );
                e.printStackTrace();
            }
            catch ( IllegalAccessException e ) {
                System.out.println( "Test failure report failed: " + e.getMessage() );
                e.printStackTrace();
            }
        }

        static private void sendInfoToDb( DiffChain<?> chain, boolean failed ) {
            String objType = chain.getCurrentVersion().getClass().toString();

            try {
                // set up database connection, can make this a pool later
                Class.forName("com.mysql.jdbc.Driver").newInstance();
                Connection dbConnection = DriverManager.getConnection( dbURL, dbUser,
                    dbPassword );

                // use transactions
                dbConnection.setAutoCommit( false );

                PreparedStatement getLastId = dbConnection.prepareStatement( "SELECT
                    LAST_INSERT_ID()" );
                PreparedStatement infoInsert = dbConnection.prepareStatement( "INSERT INTO
                    diffinfo " +
                                "(tid, object_type, method_name) VALUES (?, ?, ?)" );
                PreparedStatement valsInsert = dbConnection.prepareStatement( "INSERT INTO
                    diffvals " +
                                "(field_name, field_value, field_change, did) VALUES (?,
                                    ?, ?, ?)" );

                PreparedStatement makeTrial = dbConnection.prepareStatement( "INSERT INTO
                    trials (failed) VALUES (?)" );
                if ( failed )
                    makeTrial.setInt( 1, 1 );
                else
                    makeTrial.setInt( 1, 0 );
                makeTrial.executeUpdate();

                ResultSet rs = getLastId.executeQuery();
                if ( !rs.next() )
                    throw new SQLException( "No element in result set for last ID" );
                int tid = rs.getInt( 1 );

                for ( DiffBundle bundle : chain.getDiffHistory() ) {
                    infoInsert.setInt( 1, tid );
                    infoInsert.setString( 2, objType );
                    infoInsert.setString( 3, bundle.getMethodName() );
                    infoInsert.executeUpdate();

                    rs = getLastId.executeQuery();
                    if ( !rs.next() )
                        throw new SQLException( "No element in result set" );
                    int did = rs.getInt( 1 );

                    for ( DiffValue value : bundle.getDiffVals() ) {
                        valsInsert.setString( 1, value.getFieldName() );
                        valsInsert.setInt( 4, did );
                        setFinalValInsertParams( value, valsInsert );
                        valsInsert.executeUpdate();
                    }
                }

                dbConnection.commit(); // commit as a transaction
            }
            catch ( SQLException e ) {
                System.out.println( "SQLException in sendInfoToDb: " + e.getMessage() );
            }
            catch ( Exception e ) {
                System.out.println( "Error loading MySQL JDBC driver: " + e.getMessage() )
                    ;
            }
        }

        static private void setFinalValInsertParams( DiffValue value, PreparedStatement valsInsert
            )
```

```
                        throws SQLException
        {
                switch ( value.getType() ) {
                        case DiffValue.BYTE_TYPE:
                                Byte oldValue = (Byte)value.getOldValue();
                                Byte newValue = (Byte)value.getNewValue();
                                valsInsert.setDouble( 2, newValue.doubleValue() );
                                valsInsert.setDouble( 3, newValue.doubleValue() - oldValue.
                                    doubleValue() );
                                break;
                        case DiffValue.SHORT_TYPE:
                                Short oldValueS = (Short)value.getOldValue();
                                Short newValueS = (Short)value.getNewValue();
                                valsInsert.setDouble( 2, newValueS.doubleValue() );
                                valsInsert.setDouble( 3, newValueS.doubleValue() - oldValueS.
                                    doubleValue() );
                                break;
                        case DiffValue.INT_TYPE:
                                Integer oldValueI = (Integer)value.getOldValue();
                                Integer newValueI = (Integer)value.getNewValue();
                                valsInsert.setDouble( 2, newValueI.doubleValue() );
                                valsInsert.setDouble( 3, newValueI.doubleValue() - oldValueI.
                                    doubleValue() );
                                break;
                        case DiffValue.LONG_TYPE:
                                Long oldValueL = (Long)value.getOldValue();
                                Long newValueL = (Long)value.getNewValue();
                                valsInsert.setDouble( 2, newValueL.doubleValue() );
                                valsInsert.setDouble( 3, newValueL.doubleValue() - oldValueL.
                                    doubleValue() );
                                break;
                        case DiffValue.FLOAT_TYPE:
                                Float oldValueF = (Float)value.getOldValue();
                                Float newValueF = (Float)value.getNewValue();
                                valsInsert.setDouble( 2, newValueF.doubleValue() );
                                valsInsert.setDouble( 3, newValueF.doubleValue() - oldValueF.
                                    doubleValue() );
                                break;
                        case DiffValue.DOUBLE_TYPE:
                                Double oldValueD = (Double)value.getOldValue();
                                Double newValueD = (Double)value.getNewValue();
                                valsInsert.setDouble( 2, newValueD.doubleValue() );
                                valsInsert.setDouble( 3, newValueD.doubleValue() - oldValueD.
                                    doubleValue() );
                                break;
                        case DiffValue.BOOLEAN_TYPE:
                                Boolean oldValueB = (Boolean)value.getOldValue();
                                Boolean newValueB = (Boolean)value.getNewValue();
                                if ( oldValueB.booleanValue() ) {
                                        valsInsert.setDouble( 2, 1.0 );
                                        if ( newValueB.booleanValue() )
                                                valsInsert.setDouble( 3, 0.0 );
                                        else
                                                valsInsert.setDouble( 3, 1.0 );
                                }
                                else {
                                        valsInsert.setDouble( 2, 0.0 );
                                        if ( newValueB.booleanValue() )
                                                valsInsert.setDouble( 3, 1.0 );
                                        else
                                                valsInsert.setDouble( 3, 0.0 );
                                }
                                break;
                        case DiffValue.CHAR_TYPE:
                                Integer oldValueC = new Integer( Character.getNumericValue( (
                                    Character)value.getOldValue() ) );
                                Integer newValueC = new Integer( Character.getNumericValue( (
                                    Character)value.getNewValue() ) );
                                valsInsert.setDouble( 2, newValueC.doubleValue() );
                                valsInsert.setDouble( 3, newValueC.doubleValue() - oldValueC.
                                    doubleValue() );
                                break;
                        default:
                                assert ( false );
                }
        }

}
ML

1;
```

A.8 make-arff.pl

```
#!/usr/bin/perl

use warnings;
use strict;

use DBI;
use Getopt::Std;
use Data::Dumper;

die "Usage: ./command <outfile name> <history length>\n" unless $#ARGV + 1 == 2;

open OUTFILE, '>', $ARGV[ 0 ] or die "Cannot open output file: $!";
my $history_len = $ARGV[ 1 ];
my $dbConnection = DBI->connect( 'DBI:mysql:invite:localhost', 'root', 'rootpass' );
my $statement;



# use this as a format string for printing the header,
# formatted strings must be single-quoted, comma-separated strings

my $header_format = <<ML;
\@relation DiffRelation

\@attribute succeeded {true,false}
ML

#my $header_format = <<ML;
#\@relation DiffRelation
#
#\@attribute failed {true,false}
#\@attribute field_value numeric
#\@attribute field_change numeric
#\@attribute field_name {%s}
#ML

my $history_attribute_format = "\@attribute method%s_name {%s}\n";

my $method_names_sql = <<ML;
        SELECT object_type, method_name
        FROM diffinfo
        GROUP BY object_type, method_name
ML

my $fields_sql = <<ML;
        SELECT i.object_type, v.field_name
        FROM diffinfo i, diffvals v
        WHERE i.did = v.did
        GROUP BY i.object_type, v.field_name
ML

write_header();



#####################################
############# MAIN LOOP #############
#####################################

my $sql = <<ML;
        SELECT t.tid AS tid, i.did AS did, t.failed AS failed, i.object_type AS object_type, i.
            method_name AS method_name
        FROM trials t, diffinfo i
        WHERE t.tid = i.tid
        ORDER BY t.tid, i.did
ML

my $sqlFailCount = "SELECT count(*) FROM trials WHERE failed = 1";
my $sqlSucceedCount = "SELECT count(*) FROM trials WHERE failed = 0";

my $sqlFailStmt = $dbConnection->prepare( $sqlFailCount );
my $sqlSucceedStmt = $dbConnection->prepare( $sqlSucceedCount );

$sqlFailStmt->execute();
$sqlSucceedStmt->execute();

my $netFailCount = ( $sqlFailStmt->fetchrow_array() )[ 0 ];
my $netSuccessCount = ( $sqlSucceedStmt->fetchrow_array() )[ 0 ];

$sqlFailStmt->finish();
$sqlSucceedStmt->finish();

$statement = $dbConnection->prepare( $sql );
$statement->execute();

my $limitingValue = $netFailCount < $netSuccessCount ? $netFailCount : $netSuccessCount;
```

```
my $count = 0; # number of methods written for current TID
my $last_tid = -1;
my $failCount = 0;
my $successCount = 0;

while ( my $row_hashref = $statement->fetchrow_hashref() ) {

#print Dumper( $row_hashref )."\n";

        if ( $failCount >= $limitingValue && $successCount >= $limitingValue ) { last; }

        #test with history_len + 1 because of the true/false field

        if ( $last_tid == -1 ) { $last_tid = $row_hashref->{'tid'}; }

        if ( $row_hashref->{'tid'} != $last_tid ) {
                # mark shorter-than-history_len history as unknown
                if ( $history_len + 1 - $count > 0 ) {
                        my @missing = ("?") x ( $history_len + 1 - $count );
                        print OUTFILE ",";
                        print OUTFILE join( ",", @missing );
                }

                print OUTFILE "\n";
                $count = 0;
                $last_tid = $row_hashref->{'tid'};
        }
        elsif ( $count == $history_len + 1 ) { next; }

        my $areBeginningNewLine = $count == 0;
        my $testFailed = $row_hashref->{'failed'};

        # actually write something

        if ( $areBeginningNewLine && $testFailed && $failCount < $limitingValue - 1 ) {
                print OUTFILE "true";
                $failCount++;
        }
        elsif ( $areBeginningNewLine && !$testFailed && $successCount < $limitingValue - 1 ) {
                print OUTFILE "false";
                $successCount++;
        }

        # beginning a new entry and SHOULD NOT PRINT ANYTHING FOR THIS ENTRY
        elsif ( $areBeginningNewLine ) {
                $count = $history_len + 1;
                next;
        }

        else {
                if ( $count != $history_len + 1 ) { print OUTFILE ","; }

                #print out the method names
                $row_hashref->{'method_name'} =~ s/ /_/g;
                $row_hashref->{'object_type'} =~ s/ /_/g;
                print OUTFILE "'$row_hashref->{'object_type'}.$row_hashref->{'method_name'}'";

        }

        $count++;
}

$statement->finish();
$dbConnection->disconnect();

print OUTFILE "\n";


#######################################
########## HEADER PRINTING #############
#######################################

sub write_header
{
        my @methods = ();

        $statement = $dbConnection->prepare( $method_names_sql );
        $statement->execute();
        while ( my ( $method_object, $method_name ) = $statement->fetchrow_array() ) {
                $method_name =~ s/ /_/g;
                $method_object =~ s/ /_/g;
                push @methods, "'$method_object.$method_name'";
        }

#restore the below when handling fields properly
#
#my @fields = ();
#$statement = $dbConnection->prepare( $fields_sql );
#$statement->execute();
#while ( my ( $field_object, $field_name ) = $statement->fetchrow_array() ) {
#       $field_object =~ s/ /_/g;
#       push @methods, "'$field_object.$field_name'";
```

```
#}

#print OUTFILE sprintf(
#                $header_format,
#                join( ',', @fields )
#        );

        print OUTFILE $header_format;

        # print an attribute line for each member of the history chain up to length
        foreach ( 1 .. $history_len ) {
                print OUTFILE sprintf(
                                $history_attribute_format,
                                $_,
                                join( ',', @methods )
                        );
        }

        print OUTFILE "\n\n\@data\n";
}
```

A.9 tables.sql

```
DROP DATABASE invite;
CREATE DATABASE invite;

USE invite;

DROP TABLE IF EXISTS diffvals;
DROP TABLE IF EXISTS diffinfo;
DROP TABLE IF EXISTS trials;

CREATE TABLE trials (
        tid int NOT NULL AUTO_INCREMENT,
        time timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
        failed int(2) NOT NULL,
        CONSTRAINT errors_pk PRIMARY KEY( tid )
) ENGINE = InnoDB;

CREATE TABLE diffinfo (
        did int NOT NULL AUTO_INCREMENT,
        tid int NOT NULL,
        object_type varchar(60) NOT NULL,
        method_name varchar(120) NOT NULL,
        CONSTRAINT diffinfo_pk PRIMARY KEY( did ),
        CONSTRAINT diffinfo_fk1 FOREIGN KEY( tid ) REFERENCES trials( tid ) ON DELETE CASCADE
) ENGINE = InnoDB;

CREATE TABLE diffvals (
        did int NOT NULL,
        field_name varchar(45) NOT NULL,
        field_value DECIMAL(15,8) NOT NULL,
        field_change DECIMAL(15,8) NOT NULL,
        CONSTRAINT diffvals_pk PRIMARY KEY( did, field_name ),
        CONSTRAINT diffvals_fk1 FOREIGN KEY( did ) REFERENCES diffinfo( did ) ON DELETE CASCADE
) ENGINE = InnoDB;
```

A.10 Makefile (with Invite compilation)

```
JNI_INCLUDE = -I/usr/lib/jvm/java-6-openjdk/include/ -I/usr/java/include/linux
CC_SHARED_FLAGS = -shared -fPIC

PREFIX = /Users/del/Desktop/thesis/
AJC = ajc -source 5 #-inpath $(PREFIX)

CORE = com/invite/core
CORE_JAVA = $(CORE)/Affinity.java $(CORE)/Forker.java $(CORE)/Pipe.java $(CORE)/PipeReader.java \
            $(CORE)/Stats.java $(CORE)/InsertTest.java $(CORE)/Sorter.java

CORE_DEPS = $(CORE_JAVA) $(CORE)/Forker.c $(CORE)/Affinity.c $(CORE)/Pipe.c

DRIVERS = com/invite/drivers
DRIVERS_DEPS = $(DRIVERS)/IntFields.java $(DRIVERS)/Runner.java $(DRIVERS)/TestTracking.java

CHANGE = com/invite/changetracking
CHANGE_DEPS = $(CHANGE)/DiffChain.java $(CHANGE)/DiffBundle.java $(CHANGE)/DiffValue.java

all: $(DRIVERS_DEPS) $(CHANGE_DEPS)
        $(AJC) $(DRIVERS_DEPS) $(CHANGE_DEPS) $(CORE)/Invite.aj

all-old: $(CORE_DEPS) $(DRIVERS_DEPS) $(CHANGE_DEPS)
        $(AJC) $(CORE_JAVA) $(DRIVERS_DEPS) $(CHANGE_DEPS)
        javah -jni -d $(CORE) com.invite.core.Forker
        gcc $(CORE)/Forker.c $(CC_SHARED_FLAGS) -o $(CORE)/libforker.so $(JNI_INCLUDE)
        javah -jni -d $(CORE) com.invite.core.Affinity
        gcc $(CORE)/Affinity.c $(CC_SHARED_FLAGS) -o $(CORE)/libaffinity.so $(JNI_INCLUDE)
        javah -jni -d $(CORE) com.invite.core.Pipe
        gcc $(CORE)/Pipe.c $(CC_SHARED_FLAGS) -o $(CORE)/libpipe.so $(JNI_INCLUDE)

clean-core:
        rm -rf $(CORE)/*.class $(CORE)/*.so

drivers: $(DRIVERS_DEPS)
        $(AJC) $(DRIVERS_DEPS)

clean-drivers:
        rm -rf $(DRIVERS)/*.class

changetracking: $(CHANGE_DEPS)
        $(AJC) $(CHANGE_DEPS)

runner: all
        java com/invite/drivers/Runner

runner-old: all-old
        java com/invite/drivers/Runner

clean-changetracking:
        rm -rf $(CHANGE)/*.class

clean: clean-core clean-drivers clean-changetracking
        rm -rf *.class
```

A.11 test.sh

```
#!/bin/bash

make clean
cd generation
./ajgen-noinvite.pl ../com/invite/core/Invite.aj asdf
cd ..
make runner
```